
freud Documentation

Release 0.10.0

The Regents of the University of Michigan

Aug 27, 2018

Contents

1	Installing freud	3
1.1	Examples	3
1.2	Installation	3
1.3	Modules	5
1.4	Development Guide	71
1.5	References and Citations	79
1.6	License	79
1.7	Credits	80
2	Support and Contribution	85
3	Indices and tables	87
	Bibliography	89
	Python Module Index	91

“Neurosis is the inability to tolerate ambiguity” - Sigmund Freud

Welcome to the documentation for freud, a Python package for analyzing particle simulation trajectories of periodic systems. The library contains a diverse array of analysis routines designed for molecular dynamics and Monte Carlo simulation trajectories. Since any scientific investigation is likely to benefit from a range of analyses, freud is designed to work as part of a larger analysis pipeline. In order to maximize its interoperability with other systems, freud works with and returns [NumPy](#) arrays.

CHAPTER 1

Installing freud

The recommended method of installing `freud` is using `conda` through the `conda-forge` channel. First download and install `miniconda` following [conda's instructions](#). Then, install `freud` from `conda-forge`:

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

Alternatively, `freud` can be installed directly from source.

```
$ git clone https://bitbucket.org/glotzer/freud.git
$ cd freud
$ python setup.py install
```

1.1 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 `Bokeh` [[Bokeh](#)].

1.2 Installation

1.2.1 Installing freud

You can either install `freud` via `conda` or compile it from source.

Install via conda

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

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

Compile from source

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

- [Python](#) (2.7, 3.5, 3.6)
- [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 creates a build directory inside the freud source directory and builds freud:

```
git clone --recurse-submodules https://bitbucket.org/glotzer/freud.git
cd freud
python setup.py install
```

By default, freud installs to the `USER_SITE` directory, which is in `~/.local` on Linux and in `~/Library` on macOS. `USER_SITE` is on the Python search path by default, so there is no need to modify `PYTHONPATH`.

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

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

1.2.2 Unit Tests

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

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

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

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

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

1.2.3 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
```

1.3 Modules

Below is a list of modules in freud. To add your own module, read the [development guide](#).

1.3.1 Bond Module

Overview

<i>freud.bond.BondingAnalysis</i>	Analyze the bond lifetimes and flux present in the system.
<i>freud.bond.BondingR12</i>	Compute bonds in a 2D system using a (r, θ_1, θ_2) coordinate system.
<i>freud.bond.BondingXY2D</i>	Compute bonds in a 2D system using a (x, y) coordinate system.
<i>freud.bond.BondingXYT</i>	Compute bonds in a 2D system using a (x, y, θ) coordinate system.
<i>freud.bond.BondingXYZ</i>	Compute bonds in a 3D system using a (x, y, z) coordinate system.

Details

The bond module allows for the computation of bonds as defined by a map. Depending on the coordinate system desired, either a two or three dimensional array is supplied, with each element containing the bond index mapped to the pair geometry of that element. The user provides a list of indices to track, so that not all bond indices contained in the bond map need to be tracked in computation.

The bond module is designed to take in arrays using the same coordinate systems as the *PMFT Module* in freud.

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 .
-

class freud.bond.**BondingAnalysis** (*num_particles*, *num_bonds*)

Analyze the bond lifetimes and flux present in the system.

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

Parameters

- **num_particles** (*unsigned int*) – Number of particles over which to calculate bonds.
- **num_bonds** (*unsigned int*) – Number of bonds to track.

Variables

- **bond_lifetimes** ($(N_{\text{particles}}, \text{varying})$ `numpy.ndarray`) – Bond lifetimes.
- **overall_lifetimes** ($(N_{\text{particles}}, \text{varying})$ `numpy.ndarray`) – Overall bond lifetimes.
- **transition_matrix** (`numpy.ndarray`) – Transition matrix.
- **num_frames** (*unsigned int*) – Number of frames calculated.
- **num_particles** (*unsigned int*) – Number of tracked particles.
- **num_bonds** (*unsigned int*) – Number of tracked bonds.

compute

Calculates the changes in bonding states from one frame to the next.

Parameters

- **frame_0** ($(N_{\text{particles}}, N_{\text{bonds}})$ `numpy.ndarray`) – Current/previous bonding frame (as output from *BondingR12* modules).
- **frame_1** ($(N_{\text{particles}}, N_{\text{bonds}})$ `numpy.ndarray`) – Next/current bonding frame (as output from *BondingR12* modules).

initialize

Calculates the changes in bonding states from one frame to the next.

Parameters **frame_0** ($(N_{\text{particles}}, N_{\text{bonds}})$ `numpy.ndarray`) – First bonding frame (as output from *BondingR12* modules).

class freud.bond.**BondingR12** (*r_max*, *bond_map*, *bond_list*)

Compute bonds in a 2D system using a (r, θ_1, θ_2) coordinate system.

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

Parameters

- **r_max** (*float*) – Distance to search for bonds.
- **bond_map** (*numpy.ndarray*) – 3D array containing the bond index for each r , θ_2 , θ_1 coordinate.
- **bond_list** (*numpy.ndarray*) – List containing the bond indices to be tracked, `bond_list[i] = bond_index`.

Variables

- **bonds** (*numpy.ndarray*) – Particle bonds.
- **box** (*freud.box.Box*) – Box used in the calculation.
- **list_map** (*dict*) – The dict used to map bond index to list index.
- **rev_list_map** (*dict*) – The dict used to map list idx to bond idx.

compute

Calculates the bonds.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ($(N_{particles}, 3)$ *numpy.ndarray*) – Reference points used to calculate the bonding.
- **ref_orientations** ($(N_{particles}, 1)$ –

:param ($N_{particles}$,) *numpy.ndarray*): Reference orientations as angles to use in computation. :param points: Points used to calculate the bonding. Uses `ref_points`

if not provided or None.

Parameters orientations ($(N_{particles}, 1)$ –

:param ($N_{particles}$,) *numpy.ndarray*, optional): Orientations as angles to use in computation. Uses `ref_orientations` if not provided or None.

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

class `freud.bond.BondingXY2D` (*x_max*, *y_max*, *bond_map*, *bond_list*)

Compute bonds in a 2D system using a (x , y) coordinate system.

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

Parameters

- **x_max** (*float*) – Maximum x distance at which to search for bonds.
- **y_max** (*float*) – Maximum y distance at which to search for bonds.
- **bond_map** (*numpy.ndarray*) – 3D array containing the bond index for each x , y coordinate.
- **bond_list** (*numpy.ndarray*) – List containing the bond indices to be tracked, `bond_list[i] = bond_index`.

Variables

- **bonds** (`numpy.ndarray`) – Particle bonds.
- **box** (`freud.box.Box`) – Box used in the calculation.
- **list_map** (`dict`) – The dict used to map bond index to list index.
- **rev_list_map** (`dict`) – The dict used to map list idx to bond idx.

compute

Calculates the bonds.

Parameters

- **box** (`freud.box.Box`) – Simulation box.
- **ref_points** ($(N_{particles}, 3)$ `numpy.ndarray`) – Reference points used to calculate the bonding.
- **ref_orientations** ($(N_{particles}, 1)$ –

:param ($N_{particles},$) `numpy.ndarray`): Reference orientations as angles to use in computation. :param points: Points used to calculate the bonding. Uses `ref_points`

if not provided or None.

Parameters orientations ($(N_{particles}, 1)$ –

:param ($N_{particles},$) `numpy.ndarray`, optional): Orientations as angles to use in computation. Uses `ref_orientations` if not provided or None.

Parameters nlist (`freud.locality.NeighborList`, optional) – NeighborList to use to find bonds (Default value = None).

class `freud.bond.BondingXYT` (`x_max`, `y_max`, `bond_map`, `bond_list`)

Compute bonds in a 2D system using a (x, y, θ) coordinate system.

For each particle in the system determine which other particles are in which bonding sites.

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

Parameters

- **x_max** (`float`) – Maximum x distance at which to search for bonds.
- **y_max** (`float`) – Maximum y distance at which to search for bonds.
- **bond_map** (`numpy.ndarray`) – 3D array containing the bond index for each x, y coordinate.
- **bond_list** (`numpy.ndarray`) – List containing the bond indices to be tracked, `bond_list[i] = bond_index`.

Variables

- **bonds** (`numpy.ndarray`) – Particle bonds.
- **box** (`freud.box.Box`) – Box used in the calculation.
- **list_map** (`dict`) – The dict used to map bond index to list index.
- **rev_list_map** (`dict`) – The dict used to map list idx to bond idx.

compute

Calculates the bonds.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) *numpy.ndarray*) – Reference points used to calculate the bonding.
- **ref_orientations** (($N_{particles}$, 1) –

:param ($N_{particles}$,) *numpy.ndarray*): Reference orientations as angles to use in computation. :param points: Points used to calculate the bonding. Uses *ref_points*

if not provided or None.

Parameters orientations (($N_{particles}$, 1) –

:param ($N_{particles}$,) *numpy.ndarray*, optional): **Orientations as angles to use in computation.** Uses *ref_orientations* if not provided or None.

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

class *freud.bond.BondingXYZ* (*x_max*, *y_max*, *z_max*, *bond_map*, *bond_list*)

Compute bonds in a 3D system using a (*x*, *y*, *z*) coordinate system.

For each particle in the system determine which other particles are in which bonding sites.

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

Parameters

- **x_max** (*float*) – Maximum *x* distance at which to search for bonds.
- **y_max** (*float*) – Maximum *y* distance at which to search for bonds.
- **z_max** (*float*) – Maximum *z* distance at which to search for bonds.
- **bond_map** (*numpy.ndarray*) – 3D array containing the bond index for each *x*, *y*, *z* coordinate.
- **bond_list** (*numpy.ndarray*) – List containing the bond indices to be tracked, *bond_list*[*i*] = *bond_index*.

Variables

- **bonds** (*numpy.ndarray*) – Particle bonds.
- **box** (*freud.box.Box*) – Box used in the calculation.
- **list_map** (*dict*) – The dict used to map bond index to list index.
- **rev_list_map** (*dict*) – The dict used to map list idx to bond idx.

compute

Calculates the bonds.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** (($N_{particles}$, 3) *numpy.ndarray*) – Reference points used to calculate the bonding.
- **ref_orientations** (($N_{particles}$, 4) *numpy.ndarray*) – Reference orientations as quaternions to use in computation.

- **points** ($(N_{particles}, 3)$ `numpy.ndarray`) – Points used to calculate the bonding. Uses `ref_points` if not provided or `None`.
- **orientations** ($(N_{particles}, 4)$ `numpy.ndarray`) – Orientations as quaternions to use in computation. Uses `ref_orientations` if not provided or `None`.
- **nlist** (`freud.locality.NeighborList`, optional) – NeighborList to use to find bonds (Default value = `None`).

1.3.2 Box Module

Overview

`freud.box.Box`The freud Box class for simulation boxes.

Details

The box module provides the Box class, which defines the geometry of the simulation box. The module natively supports periodicity by providing the fundamental features for wrapping vectors outside the box back into it.

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.
- **Lin** (*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)

getCoordinates**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*

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`

1.3.3 Cluster Module

Overview

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

Details

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

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

computeClusters

Compute the clusters for the given set of points.

Parameters

- **points** ($(N_{\text{particles}}, 3)$ *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).

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 $(N_{\text{clusters}}, 3)$ *numpy.ndarray*.

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

1.3.4 Density Module

Overview

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

Details

The 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 <mspell@umich.edu>

Parameters

- **rmax** (*float*) – Distance over which to calculate.
- **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 counts of each histogram bin.
- **R** ((N_{bins}) *numpy.ndarray*) – The values of bin centers.

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 the correlation function.
- **ref_values** ($(N_{particles})$ *numpy.ndarray*) – Real values used to calculate the correlation function.
- **points** ($(N_{particles}, 3)$ *numpy.ndarray*) – Points used to calculate the correlation function. Uses `ref_points` if not provided or `None`.
- **values** ($(N_{particles})$) – 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** ($(N_{particles}, 3)$ *numpy.ndarray*) – Reference points used to calculate the correlation function.
- **ref_values** ($(N_{particles})$ *numpy.ndarray*) – Real values used to calculate the correlation function.
- **points** ($(N_{particles}, 3)$ *numpy.ndarray*) – Points used to calculate the correlation function. Uses `ref_points` if not provided or `None`.

- **values** ($(N_{particles})$) – 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 <mspell@umich.edu>

Parameters

- **rmax** (`float`) – Distance over which to calculate.
- **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 counts of each histogram bin.
- **R** ((N_{bins}) `numpy.ndarray`) – The values of bin centers.

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 the correlation function.
- **ref_values** ($(N_{particles})$ `numpy.ndarray`) – Complex values used to calculate the correlation function.
- **points** ($(N_{particles}, 3)$ `numpy.ndarray`) – Points used to calculate the correlation function. Uses `ref_points` if not provided or `None`.

- **values** ($(N_{particles})$) – 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** ($(N_{particles}, 3)$ `numpy.ndarray`) – Reference points used to calculate the correlation function.
- **ref_values** ($(N_{particles})$ `numpy.ndarray`) – Complex values used to calculate the correlation function.
- **points** ($(N_{particles}, 3)$ `numpy.ndarray`) – Points used to calculate the correlation function. Uses `ref_points` if not provided or `None`.
- **values** ($(N_{particles})$) – 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 the grid based upon the distance of the grid cell from the center of the Gaussian. 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** ($((w_x, w_y, w_z) \text{ numpy.ndarray})$) – The image grid with the Gaussian density.
- **counts** ($((N_{bins}) \text{ numpy.ndarray})$) – The counts of each histogram bin.
- **R** ($((N_{bins}) \text{ numpy.ndarray})$) – The values of bin centers.

compute

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

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **points** ($((N_{particles}, 3) \text{ numpy.ndarray})$) – 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 to calculate the local density. *volume* is the volume of a single particle. *diameter* is the diameter of the circumsphere of an individual particle.

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** ($((N_{particles}) \text{ numpy.ndarray})$) – Density per particle.
- **num_neighbors** ($((N_{particles}) \text{ numpy.ndarray})$) – Number of neighbors for each particle.

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** ($(N_{particles}, 3)$ *numpy.ndarray*) – Reference points to calculate the local density.
- **points** ($(N_{particles}, 3)$ *numpy.ndarray*) – Points to calculate the local density.
- **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 distance to calculate.
- **dr** (*float*) – Distance between histogram bins.
- **rmin** (*float*) – Minimum distance to calculate, defaults to 0.

Variables

- **box** (*freud.box.Box*) – Box used in the calculation.
- **RDF** ((N_{bins}) *numpy.ndarray*) – Histogram of RDF values.
- **R** ($(N_{bins}, 3)$ *numpy.ndarray*) – The values of bin centers.
- **n_r** ($(N_{bins}, 3)$ *numpy.ndarray*) – Histogram of cumulative RDF values.

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** ($(N_{particles}, 3)$ *numpy.ndarray*) – Reference points used to calculate the RDF.
- **points** ($(N_{particles}, 3)$ *numpy.ndarray*) – 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** ($(N_{particles}, 3)$ *numpy.ndarray*) – Reference points used to calculate the RDF.
- **points** ($(N_{particles}, 3)$ *numpy.ndarray*) – 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.

1.3.5 Environment Module

Overview

<i>freud.environment.BondOrder</i>	Compute the bond order diagram for the system of particles.
<i>freud.environment.LocalDescriptors</i>	Compute a set of descriptors (a numerical “fingerprint”) of a particle’s local environment.
<i>freud.environment.MatchEnv</i>	Clusters particles according to whether their local environments match or not, according to various shape matching metrics.
<i>freud.environment.Pairing2D</i>	Compute pairs for the system of particles.
<i>freud.environment.AngularSeparation</i>	Calculates the minimum angles of separation between particles and references.

Details

The 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.

class *freud.environment.BondOrder* (*rmax, k, n, nBinsT, nBinsP*)

Compute the bond order diagram for the system of particles.

Available modes of calculation:

- If *mode* = 'bod' (Bond Order Diagram, *default*): Create the 2D histogram containing the number of bonds formed through the surface of a unit sphere based on the azimuthal (θ) and polar (ϕ) angles.
- If *mode* = 'lbod' (Local Bond Order Diagram): Create the 2D histogram containing the number of bonds formed, rotated into the local orientation of the central particle, through the surface of a unit sphere based on the azimuthal (θ) and polar (ϕ) angles.
- If *mode* = 'obcd' (Orientation Bond Correlation Diagram): Create the 2D histogram containing the number of bonds formed, rotated by the rotation that takes the orientation of neighboring particle *j* to the ori-

entation of each particle i , through the surface of a unit sphere based on the azimuthal (θ) and polar (ϕ) angles.

- If `mode='oocd'` (Orientation Orientation Correlation Diagram): Create the 2D histogram containing the directors of neighboring particles (\hat{z} rotated by their quaternion), rotated into the local orientation of the central particle, through the surface of a unit sphere based on the azimuthal (θ) and polar (ϕ) angles.

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** ($((N_\phi, N_\theta)$ *numpy.ndarray*) – Bond order.
- **box** (*freud.box.Box*) – Box used in the calculation.
- **theta** ($((N_\theta)$ *numpy.ndarray*) – The values of bin centers for θ .
- **phi** ($((N_\phi)$ *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** ($((N_{particles}, 3)$ –
- **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** ($(N_{\text{particles}}, 3)$ `numpy.ndarray`) – Reference points used to calculate bonds.
- **ref_orientations** ($(N_{\text{particles}}, 4)$ `numpy.ndarray`) – Reference orientations used to calculate bonds.
- **points** ($(N_{\text{particles}}, 3)$ –
- **optional**) – Points used to calculate bonds. Uses `ref_points` if not provided or `None`.
- **orientations** ($(N_{\text{particles}}, 3)$ –
- **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.

class `freud.environment.LocalDescriptors` (*box*, *nNeigh*, *lmax*, *rmax*)

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 $(\text{num_bonds}, \text{num_sphs})$. 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 <mspell@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 look for neighbors.
- **negative_m** (*bool*) – True if we should also calculate Y_{lm} for negative m .

Variables

- **sph** ($(N_{\text{bonds}}, \text{SphWidth})$ `numpy.ndarray`) – A reference to the last computed spherical harmonic array.
- **num_particles** (*unsigned int*) – The number of particles.
- **num_neighbors** (*unsigned int*) – The number of neighbors.
- **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** ($((N_{\text{particles}}, 3) \text{ numpy.ndarray})$) – Source points to calculate the order parameter.
- **points** ($((N_{\text{particles}}, 3) \text{ numpy.ndarray}, \text{optional})$) – Destination points to calculate the order parameter (Default value = None).
- **orientations** ($((N_{\text{particles}}, 4) \text{ numpy.ndarray}, \text{optional})$) – 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** ($((N_{\text{particles}}, 3) \text{ numpy.ndarray})$) – Source points to calculate the order parameter.
- **points** ($((N_{\text{particles}}, 3) \text{ numpy.ndarray}, \text{optional})$) – Destination points to calculate the order parameter (Default value = None).

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** ($((N_{\text{particles}}, N_{\text{neighbors}}, 3) \text{ numpy.ndarray})$) – All environments for all particles.
- **num_particles** (*unsigned int*) – The number of particles.
- **num_clusters** (*unsigned int*) – The number of clusters.
- **clusters** ($((N_{\text{particles}}) \text{ numpy.ndarray})$) – 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.

Note: Pairing2D is deprecated and is replaced with *Bond Module*.

class `freud.environment.Pairing2D` (*rmax*, *k*, *compDotTol*)

Compute pairs for the system of particles.

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

Deprecated since version 0.8.2: Use `freud.bond` instead.

Parameters

- **rmax** (*float*) – Distance over which to calculate.
- **k** (*unsigned int*) – Number of neighbors to search.
- **compDotTol** (*float*) – Value of the dot product below which a pair is determined.

Variables

- **match** ($((N_{\text{particles}})$ *numpy.ndarray*) – The match.
- **pair** ($((N_{\text{particles}})$ *numpy.ndarray*) – The pair.
- **box** (*freud.box.Box*) – Box used in the calculation.

compute

Calculates the correlation function and adds to the current histogram.

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **points** ($((N_{\text{particles}}, 3)$ *numpy.ndarray*) – Reference points to calculate the local density.
- **orientations** ($((N_{\text{particles}}, 4)$ *numpy.ndarray*) – Orientations to use in computation.
- **compOrientations** ($((N_{\text{particles}}, 4)$ *numpy.ndarray*) – Possible orientations to check for bonds.
- **nlist** (*freud.locality.NeighborList*, optional) – NeighborList to use to find bonds (Default value = None).

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

Calculates the minimum angles of separation between particles and references.

Module author: Erin Teich

Module author: Andrew Karas

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** ($((N_{\text{neighbors}},)$ *numpy.ndarray*) – The neighbor angles in radians.
- **global_angles** ($((N_{\text{particles}}, N_{\text{global}})$ *numpy.ndarray*) – The global angles in radians.

computeGlobal

Calculates the minimum angles of separation between `global_ors` and `ors`, checking for underlying symmetry as encoded in `equiv_quats`.

Parameters

- **ors** ($(N_{\text{particles}}, 3)$ `numpy.ndarray`) – Orientations to calculate the order parameter.
- **global_ors** ($(N_{\text{particles}}, 4)$ `numpy.ndarray`) – Reference orientations to calculate the order parameter.
- **equiv_quats** ($(N_{\text{particles}}, 4)$ `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`.

Parameters

- **box** (`freud.box.Box`) – Simulation box.
- **ref_ors** ($(N_{\text{particles}}, 4)$ `numpy.ndarray`) – Reference orientations used to calculate the order parameter.
- **ors** ($(N_{\text{particles}}, 3)$ `numpy.ndarray`) – Orientations used to calculate the order parameter.
- **ref_points** ($(N_{\text{particles}}, 3)$ `numpy.ndarray`) – Reference points used to calculate the order parameter.
- **points** ($(N_{\text{particles}}, 3)$ `numpy.ndarray`) – Points used to calculate the order parameter.
- **equiv_quats** ($(N_{\text{particles}}, 4)$ `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).

1.3.6 Index Module

Overview

<code>freud.index.Index2D</code>	freud-style indexer for flat arrays.
<code>freud.index.Index3D</code>	freud-style indexer for flat arrays.

Details

The 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.

```
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

getNumElements

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

`getNumElements`

1.3.7 Interface Module

Overview

`freud.interface.InterfaceMeasure`

Measures the interface between two sets of points.

Details

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

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

Measures the interface between two sets of points.

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

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).

1.3.8 KSpace Module

Overview

<code>freud.kspace.meshgrid2</code>	Computes an n-dimensional meshgrid.
<code>freud.kspace.SFactor3DPoints</code>	Compute the full 3D structure factor of a given set of points.
<code>freud.kspace.AnalyzeSFactor3D</code>	Analyze the peaks in a 3D structure factor.
<code>freud.kspace.SingleCell3D</code>	SingleCell3D objects manage data structures necessary to call the Fourier Transform functions that evaluate FTs for given form factors at a list of K points.
<code>freud.kspace.FTfactory</code>	Factory to return an FT object of the requested type.
<code>freud.kspace.FTbase</code>	Base class for FT calculation classes.
<code>freud.kspace.FTdelta</code>	Fourier transform a list of delta functions.
<code>freud.kspace.FTsphere</code>	Fourier transform for sphere.
<code>freud.kspace.FTpolyhedron</code>	Fourier Transform for polyhedra.
<code>freud.kspace.FTconvexPolyhedron</code>	Fourier Transform for convex polyhedra.
<code>freud.kspace.rotate</code>	Axis-angle rotation.
<code>freud.kspace.quatrot</code>	Apply a rotation quaternion.
<code>freud.kspace.Constraint</code>	Constraint base class.
<code>freud.kspace.AlignedBoxConstraint</code>	Axis-aligned Box constraint.
<code>freud.kspace.constrainedLatticePoints</code>	Generate a list of points satisfying a constraint.
<code>freud.kspace.reciprocalLattice3D</code>	Calculate reciprocal lattice vectors.
<code>freud.kspace.DeltaSpot</code>	Base class for drawing diffraction spots on a 2D grid.
<code>freud.kspace.GaussianSpot</code>	Draw diffraction spot as a Gaussian blur.

Details

Modules for calculating quantities in reciprocal space, including Fourier transforms of shapes and diffraction pattern generation.

Structure Factor

class `freud.kspace.SFactor3DPoints` (*box*, *g*)

Compute the full 3D structure factor of a given set of points.

Given a set of points \vec{r}_i , `SFactor3DPoints` computes the static structure factor $S(\vec{q}) = C_0 \left| \sum_{m=1}^N e^{i\vec{q} \cdot \vec{r}_i} \right|^2$.

In this expression, C_0 is a scaling constant chosen so that $S(0) = 1$, and N is the number of particles.

S is evaluated on a grid of q -values $\vec{q} = h \frac{2\pi}{L_x} \hat{i} + k \frac{2\pi}{L_y} \hat{j} + l \frac{2\pi}{L_z} \hat{k}$ for integer $h, k, l : [-g, g]$ and L_x, L_y, L_z are the box lengths in each direction.

After calling `compute()`, access the q values with `getQ()`, the static structure factor values with `getS()`, and (if needed) the un-squared complex version of S with `getSComplex()`. All values are stored in 3D `numpy.ndarray` structures. They are indexed by a, b, c where $a = h + g, b = k + g, c = l + g$.

Note: Due to the way that numpy arrays are indexed, access the returned S array as $S[c, b, a]$ to get the value at $q = (qx[a], qy[b], qz[c])$.

Parameters

- **box** (`freud.box.Box`) – The simulation box.
- **g** (`int`) – The number of grid points for q in each direction is $2*g+1$.

compute

Compute the static structure factor of a given set of points.

After calling `compute()`, you can access the results with `getS()`, `getSComplex()`, and the grid with `getQ()`.

Parameters **points** ($(N_{particles}, 3)$ `numpy.ndarray`) – Points used to compute the static structure factor.

getQ

Get the q values at each point.

The structure factor $S[c, b, a]$ is evaluated at the vector $q = (qx[a], qy[b], qz[c])$.

Returns (`qx`, `qy`, `qz`).

Return type `tuple`

getS

Get the computed static structure factor.

Returns The computed static structure factor as a copy.

Return type (`X`, `Y`) `numpy.ndarray`

getSComplex

Get the computed complex structure factor (if you need the phase information).

Returns The computed static structure factor, as a copy, without taking the magnitude squared.

Return type (`X`, `Y`) `numpy.ndarray`

class `freud.kspace.AnalyzeSFactor3D` (S)

Analyze the peaks in a 3D structure factor.

Given a structure factor $S(q)$ computed by classes such as `SFactor3DPoints`, `AnalyzeSFactor3D` performs a variety of analysis tasks.

- Identifies peaks.
- Provides a list of peaks and the vector \vec{q} positions at which they occur.
- Provides a list of peaks grouped by q^2
- Provides a full list of $S(|q|)$ values vs q^2 suitable for plotting the 1D analog of the structure factor.
- Scans through the full 3D peaks and reconstructs the Bravais lattice.

Note: All of these operations work in an indexed integer q -space h, k, l . Any peak position values returned must be multiplied by $2\pi/L$ to get to real q values in simulation units.

Parameters `S` (`numpy.ndarray`) – Static structure factor to analyze.

getPeakDegeneracy

Get a dictionary of peaks indexed by q^2 .

Parameters `cut` (`numpy.ndarray`) – All $S(q)$ values greater than cut will be counted as peaks.

Returns A dictionary with keys q^2 and a list of peaks for the corresponding values.

Return type `dict`

getPeakList

Get a list of peaks in the structure factor.

Parameters `cut` (`float`) – All $S(q)$ values greater than cut will be counted as peaks.

Returns peaks, q as lists.

Return type `list`

getSvsQ

Get a list of all $S(|q|)$ values vs q^2 .

Returns `S`, `qsquared`.

Return type `numpy.ndarray`

class `freud.kspace.SingleCell3D` (`k`, `ndiv`, `dK`, `boxMatrix`)

`SingleCell3D` objects manage data structures necessary to call the Fourier Transform functions that evaluate FTs for given form factors at a list of K points. `SingleCell3D` provides an interface to helper functions to calculate K points for a desired grid from the reciprocal lattice vectors calculated from an input `boxMatrix`. State is maintained as `set_` and `update_` functions invalidate internal data structures and as fresh data is restored with `update_` function calls. This should facilitate management with a higher-level UI such as a GUI with an event queue.

I'm not sure what sort of error checking would be most useful, so I'm mostly allowing `ValueErrors` and such exceptions to just occur and then propagate up through the calling functions to be dealt with by the user.

Parameters

- `ndiv` (`int`) – The resolution of the diffraction image grid.
- `k` (`float`) – The angular wave number of the plane wave probe (Currently unused).
- `dK` (`float`) – The k-space unit associated with the diffraction image grid spacing.

- **boxMatrix** ($(N_{particles}, 3)$ `numpy.ndarray`) – The unit cell lattice vectors as columns in a 3x3 matrix.
- **scale** (`float`) – nm per unit length (default 1.0).

Note:

- The `set_` functions take a single parameter and cause other internal data structures to become invalid.
 - The `update_` and `calculate` functions restore the validity of these structures using internal data.
 - The functions are separate to make it easier to avoid unnecessary computation such as when changing multiple parameters before seeking output or when wrapping the code with an interface with an event queue.
-

add_ptype

Create internal data structures for new particle type by name.

Particle type is inactive when added because parameters must be set before FT can be performed.

Parameters **name** (`str`) – particle name

calculate

Calculate FT. The details and arguments will vary depending on the form factor chosen for the particles.

For any particle type-dependent parameters passed as keyword arguments, the parameter must be passed as a list of length $\max(p_type) + 1$ with indices corresponding to the particle types defined. In other words, type-dependent parameters are optional (depending on the set of form factors being calculated), but if included must be defined for all particle types.

get_form_factors

Get form factor names and indices.

Returns List of factor names and indices.

Return type `list`

get_ptypes

Get ordered list of particle names.

Returns List of particle names.

Return type `list`

remove_ptype

Remove internal data structures associated with ptype name.

Parameters **name** (`str`) – Particle type to remove.

Note: This shouldn't usually be necessary, since particle types may be set inactive or have any of their properties updated through `set_` methods.

set_active

Set particle type active.

Parameters **name** (`str`) – Particle name.

set_box

Set box matrix.

Parameters **boxMatrix** ($((3, 3)$ `numpy.ndarray`) – Unit cell box matrix.

set_dK

Set grid spacing in diffraction image.

Parameters **dK** (*float*) – Difference in K vector between two adjacent diffraction image grid points.

set_form_factor

Set scattering form factor.

Parameters

- **name** (*str*) – Particle type name.
- **ff** (*str*) – Scattering form factor named in `get_form_factors()`.

set_inactive

Set particle type inactive.

Parameters **name** (*str*) – Particle name.

set_k

Set angular wave number of plane wave probe.

Parameters **k** (*float*) – $|k_0|$.

set_ndiv

Set number of grid divisions in diffraction image.

Parameters **ndiv** (*int*) – Define diffraction image as ndiv x ndiv grid.

set_param

Set named parameter for named particle.

Parameters

- **particle** (*str*) – Particle name.
- **param** (*str*) – Parameter name.
- **value** (*float*) – Parameter value.

set_rq

Set positions and orientations for a particle type.

To best maintain valid state in the event of changing numbers of particles, position and orientation are updated in a single method.

Parameters

- **name** (*str*) – Particle type name.
- **position** ((N,3) *numpy.ndarray*) – Array of particle positions.
- **orientation** ((N,4) *numpy.ndarray*) – Array of particle quaternions.

set_scale

Set scale factor. Store global value and set for each particle type.

Parameters **scale** (*float*) – nm per unit for input file coordinates.

update_K_constraint

Recalculate constraint used to select K values.

The constraint used is a slab of epsilon thickness in a plane perpendicular to the k_0 propagation, intended to provide easy emulation of TEM or relatively high-energy scattering.

update_Kpoints

Update K points at which to evaluate FT.

Note: If the diffraction image dimensions change relative to the reciprocal lattice, the K points need to be recalculated.

update_bases

Update the direct and reciprocal space lattice vectors.

Note: If scale or boxMatrix is updated, the lattice vectors in direct and reciprocal space need to be recalculated.

class freud.kspace.FTfactory

Factory to return an FT object of the requested type.

addFT

Add an FT class to the factory.

Parameters

- **name** (*str*) – Identifying string to be returned by getFTlist().
- **constructor** (*str*) – Class / function name to be used to create new FT objects.
- **args** (*list*) – Set default argument object to be used to construct FT objects.

getFTlist

Get an ordered list of named FT types.

Returns List of FT names.

Return type *list*

getFTobject

Get a new instance of an FT type from list returned by getFTlist().

Parameters

- **i** (*int*) – Index into list returned by getFTlist().
- **args** – Argument object used to initialize FT, overriding default set at addFT().

class freud.kspace.FTbase

Base class for FT calculation classes.

getFT

Return Fourier Transform.

Returns Fourier Transform.

Return type *numpy.ndarray*

get_density

Get density.

Returns Density.

Return type *numpy.complex64*

get_parambyname

Get named parameter for object.

Parameters **name** (*str*) – Parameter name. Must exist in list returned by `get_params()`.

Returns Parameter value.

Return type *float*

get_params

Get the parameter names accessible with `set_parambyname()`.

Returns Parameter names.

Return type *list*

get_scale

Get scale.

Returns Scale.

Return type *float*

set_K

Set K points to be evaluated.

Parameters **K** (*numpy.ndarray*) – List of K vectors at which to evaluate FT.

set_density

Set density.

Parameters **density** (*numpy.complex64*) – Density.

set_parambyname

Set named parameter for object.

Parameters

- **name** (*str*) – Parameter name. Must exist in list returned by `get_params()`.
- **value** (*float*) – Parameter value to set.

set_rq

Set r, q values.

Parameters

- **r** (*float*) – r .
- **q** (*float*) – q .

set_scale

Set scale.

Parameters **scale** (*float*) – Scale.

class `freud.kspace.FTdelta`

Fourier transform a list of delta functions.

compute

Compute FT.

Calculate $S = \sum_{\alpha} e^{-i\mathbf{K} \cdot \mathbf{r}_{\alpha}}$.

set_K

Set K points to be evaluated.

Parameters **K** (*numpy.ndarray*) – List of K vectors at which to evaluate FT.

set_density

Set density.

Parameters `density` (`numpy.complex64`) – density

set_rq

Set r , q values.

Parameters

- `r` (`float`) – r .
- `q` (`float`) – q .

set_scale

Set scale.

Parameters `scale` (`float`) – Scale.

Note: For a scale factor, λ , affecting the scattering density $\rho(r)$, $S_\lambda(k) == \lambda^3 * S(\lambda * k)$

class `freud.kspace.FTsphere`

Fourier transform for sphere.

Calculate $S = \sum_{\alpha} e^{-i\mathbf{K} \cdot \mathbf{r}_{\alpha}}$.

get_radius

Get radius parameter.

If appropriate, return value should be scaled by `get_parambyname('scale')` for interpretation.

Returns Unscaled radius.

Return type `float`

set_radius

Set radius parameter.

Parameters `radius` (`float`) – Sphere radius will be stored as given, but scaled by `scale` parameter when used by methods.

class `freud.kspace.FTpolyhedron`

Fourier Transform for polyhedra.

compute

Compute FT.

Calculate $S = \sum_{\alpha} e^{-i\mathbf{K} \cdot \mathbf{r}_{\alpha}}$.

get_radius

Get radius parameter.

If appropriate, return value should be scaled by `get_parambyname('scale')` for interpretation.

Returns Unscaled radius.

Return type `float`

set_K

Set K points to be evaluated.

Parameters `K` (`numpy.ndarray`) – List of K vectors at which to evaluate FT.

set_density

Set density.

Parameters `density` (`numpy.complex64`) – Density.

set_params

Construct list of facet offsets.

Parameters

- **verts** $((N_{particles}, 3) \text{ numpy.ndarray})$ – Vertex coordinates.
- **facets** $((N_{facets}, 3) \text{ numpy.ndarray})$ – Facet vertex indices.
- **norms** $((N_{facets}, 3) \text{ numpy.ndarray})$ – Facet normals.
- **d** $((N_{facets} - 1) \text{ numpy.ndarray})$ – Facet distances.
- **area** $((N_{facets} - 1) \text{ numpy.ndarray})$ – Facet areas.
- **volume** (*float*) – Polyhedron volume.

set_radius

Set radius of in-sphere.

Parameters **radius** (*float*) – Radius of inscribed sphere without scale applied.

set_rq

Set r , q values.

Parameters

- **r** (*float*) – r .
- **q** (*float*) – q .

class freud.kspace.**FTconvexPolyhedron**

Fourier Transform for convex polyhedra.

Parameters **hull** $((N_{verts}, 3) \text{ numpy.ndarray})$ – Convex hull object.

Spoly2D

Calculate Fourier transform of polygon.

Parameters

- **i** (*float*) – Face index into self.hull simplex list.
- **k** (*numpy.ndarray*) – Angular wave vector at which to calculate $S(i)$.

Spoly3D

Calculate Fourier transform of polyhedron.

Parameters **k** (*int*) – Angular wave vector at which to calculate $S(i)$.

compute_py

Compute FT.

Calculate $P = F * S$:

- $S = \sum_{\alpha} e^{-i\mathbf{K} \cdot \mathbf{r}_{\alpha}}$.
- F is the analytical form factor for a polyhedron, computed with `Spoly3D()`.

get_radius

Get radius parameter.

If appropriate, return value should be scaled by `get_parambyname('scale')` for interpretation.

Returns Unscaled radius.

Return type *float*

set_radius

Set radius of in-sphere.

Parameters **radius** (*float*) – Radius of inscribed sphere without scale applied.

Diffraction Patterns

class freud.kspace.DeltaSpot

Base class for drawing diffraction spots on a 2D grid.

Based on the dimensions of a grid, determines which grid points need to be modified to represent a diffraction spot and generates the values in that subgrid. Spot is a single pixel at the closest grid point.

Parameters

- **shape** – Number of grid points in each dimension.
- **extent** – Range of x,y values associated with grid points.

get_gridPoints

Get indices of sub-grid.

Based on the type of spot and its center, return the grid mask of points containing the spot.

makeSpot

Generate intensity value(s) at sub-grid points.

Parameters **cval** (`numpy.complex64`) – Complex valued amplitude used to generate spot intensity.

set_xy

Set *x, y* values of spot center.

Parameters

- **x** (*float*) – x value of spot center.
- **y** (*float*) – y value of spot center.

class freud.kspace.GaussianSpot

Draw diffraction spot as a Gaussian blur.

Grid points filled according to Gaussian at spot center.

Parameters

- **shape** – Number of grid points in each dimension.
- **extent** – Range of x, y values associated with grid points.

makeSpot

Generate intensity value(s) at sub-grid points.

Parameters **cval** (`numpy.complex64`) – Complex valued amplitude used to generate spot intensity.

set_sigma

Define Gaussian.

Parameters **sigma** (*float*) – Width of the Gaussian spot.

set_xy

Set *x, y* values of spot center.

Parameters

- **x** (*float*) – x value of spot center.
- **y** (*float*) – y value of spot center.

Utilities

class `freud.kspace.Constraint`

Constraint base class.

Base class for constraints on vectors to define the API. All constraints should have a ‘radius’ defining a bounding sphere and a ‘satisfies’ method to determine whether an input vector satisfies the constraint.

satisfies

Constraint test.

Parameters **v** – Vector to test against constraint.

class `freud.kspace.AlignedBoxConstraint`

Axis-aligned Box constraint.

Tetragonal box aligned with the coordinate system. Consider using a small z dimension to serve as a plane plus or minus some epsilon. Set $R < L$ for a cylinder.

satisfies

Constraint test.

Parameters **v** – Vector to test against constraint.

`freud.kspace.constrainedLatticePoints()`

`constrainedLatticePoints(v1, v2, v3, constraint)` Generate a list of points satisfying a constraint.

Args:

v1 (`numpy.ndarray`): Lattice vector 1 along which to test.

v2 (`numpy.ndarray`): Lattice vector 2 along which to test.

v3 (`numpy.ndarray`): Lattice vector 3 along which to test.

constraint (*Constraint*): Constraint object to test lattice points against.

`freud.kspace.reciprocalLattice3D()`

`reciprocalLattice3D(a1, a2, a3)` Calculate reciprocal lattice vectors.

3D reciprocal lattice vectors with magnitude equal to angular wave number.

Args: **a1** (`numpy.ndarray`): Real space lattice vector 1. **a2** (`numpy.ndarray`): Real space lattice vector 2. **a3** (`numpy.ndarray`): real space lattice vector 3.

Returns: list: Reciprocal space vectors.

Note:

For unit test, `dot(g[i], a[j]) = 2 * pi * diracDelta(i, j)`: list of reciprocal lattice vectors

`freud.kspace.meshgrid2(*args)`

Computes an n-dimensional meshgrid.

source: <http://stackoverflow.com/questions/1827489/numpy-meshgrid-in-3d>

Parameters **args** (*list*) – Arrays to meshgrid.

Returns Tuple of arrays.

Return type `tuple`

1.3.9 Locality Module

Overview

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

Details

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

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 <mspell@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.array*) – Array of integers corresponding to indices in the set of reference points.
- **index_j** (*np.array*) – Array of integers corresponding to indices in the set of target points.
- **weights** (*np.array, optional*) – Array of per-bond weights (if `None` is given, use a value of 1 for each weight) (Default value = `None`).

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()`.
- **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.

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.computeCellList(box, positions)
for i in range(positions.shape[0]):
    # Cell containing particle i
    cell = lc.getCell(positions[i])
    # 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) – True if pairs of points with identical indices should be excluded; if None, is set to True if points is None or the same object as ref_points (Default value = None).

computeCellList

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) – True if pairs of points with identical indices should be excluded; if None, is set to True if points is None or the same object as ref_points (Default value = 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 ($N_{\text{neighbors}}$) `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`

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`** ($(N_{\text{particles}})$ *numpy.ndarray*) – The wrapped vectors padded with -1 for empty neighbors.
- **`r_sq_list`** ($(N_{\text{particles}}, N_{\text{neighbors}})$ *numpy.ndarray*) – The R_{sq} 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) – True if pairs of points with identical indices should be excluded; if None, is set to True if points is None or the same object as *ref_points* (Default value = None).

getNeighborList

Return the entire neighbor list.

Returns Neighbor List.

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.

getRsqr

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*

setCutMode

Set mode to handle *rmax* by Nearest Neighbors.

- **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: *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.

Parameters **strict_cut** (*bool*) – Whether to use a strict *rmax* or allow for automatic expansion.

1.3.10 Order Module

Overview

freud.order.CubicOrderParameter

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

Continued on next page

Table 10 – continued from previous page

<code>freud.order.NematicOrderParameter</code>	Compute the nematic order parameter for a system of particles.
<code>freud.order.HexOrderParameter</code>	Calculates the k -atic order parameter for each particle in the system.
<code>freud.order.TransOrderParameter</code>	Compute the translational order parameter for each particle.
<code>freud.order.LocalQl</code>	Compute the local Steinhardt [Steinhardt1983] rotationally invariant Q_l order parameter for a set of points.
<code>freud.order.LocalQlNear</code>	A variant of the <code>LocalQl</code> class that performs its average over nearest neighbor particles as determined by an instance of <code>freud.locality.NeighborList</code> .
<code>freud.order.LocalWl</code>	Compute the local Steinhardt [Steinhardt1983] rotationally invariant W_l order parameter for a set of points.
<code>freud.order.LocalWlNear</code>	A variant of the <code>LocalWl</code> class that performs its average over nearest neighbor particles as determined by an instance of <code>freud.locality.NeighborList</code> .
<code>freud.order.SolLiq</code>	Computes dot products of Q_{lm} between particles and uses these for clustering.
<code>freud.order.SolLiqNear</code>	A variant of the <code>SolLiq</code> class that performs its average over nearest neighbor particles as determined by an instance of <code>freud.locality.NeighborList</code> .
<code>freud.order.BondOrder</code>	Deprecated Compute the bond order diagram for the system of particles.
<code>freud.order.LocalDescriptors</code>	Deprecated Compute a set of descriptors (a numerical “fingerprint”) of a particle’s local environment.
<code>freud.order.MatchEnv</code>	Deprecated Clusters particles according to whether their local environments match or not, according to various shape matching metrics.
<code>freud.order.Pairing2D</code>	Deprecated Compute pairs for the system of particles.
<code>freud.order.AngularSeparation</code>	Deprecated Calculates the minimum angles of separation between particles and references.

Details

The 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.

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.

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.

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

class `freud.order.TransOrderParameter` (*rmax, k, n*)

Compute the translational order parameter for each particle.

Module author: Michael Engel <engelmm@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** ($(N_{\text{particles}})$ *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** ($(N_{\text{particles}}, 3)$ `numpy.ndarray`) – Points to calculate the order parameter.
- **nlist** (`freud.locality.NeighborList`) – Neighborlist to use to find bonds.

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: $\bar{Q}_{lm}(i) = \frac{1}{N_b} \sum_{j=1}^{N_b} Y_{lm}(\theta(\vec{r}_{ij}), \phi(\vec{r}_{ij}))$. 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: $Q_l(i) =$

$$\sqrt{\frac{4\pi}{2l+1} \sum_{m=-l}^l |\bar{Q}_{lm}|^2}.$$

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 $\bar{Q}_{lm}(i)$ in the original definition by the average value of $\bar{Q}_{lm}(k)$ 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** ($(N_{\text{particles}})$ `numpy.ndarray`) – The last computed Q_l for each particle (filled with NaN for particles with no neighbors).
- **ave_Ql** ($(N_{\text{particles}})$ `numpy.ndarray`) – The last computed \bar{Q}_l for each particle (filled with NaN for particles with no neighbors).
- **norm_Ql** ($(N_{\text{particles}})$ `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** ($(N_{\text{particles}})$ `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** ($(N_{\text{particles}})$ `numpy.ndarray`) – The last computed Q_l for each particle (filled with NaN for particles with no neighbors).
- **ave_Ql** ($(N_{\text{particles}})$ `numpy.ndarray`) – The last computed \bar{Q}_l for each particle (filled with NaN for particles with no neighbors).
- **norm_Ql** ($(N_{\text{particles}})$ `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** ($(N_{\text{particles}})$ `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** ($(N_{\text{particles}}, 3)$ `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** ($(N_{\text{particles}}, 3)$ `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** ($(N_{\text{particles}}, 3)$ `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** ($(N_{\text{particles}}, 3)$ `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.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: $\bar{Q}_{lm}(i) = \frac{1}{N_b} \sum_{j=1}^{N_b} Y_{lm}(\theta(\vec{r}_{ij}), \phi(\vec{r}_{ij}))$. 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 $\bar{Q}_{lm}(i)$ values using Wigner 3j symbols (CG 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 $\bar{Q}_{lm}(i)$ in the original definition by the average value of $\bar{Q}_{lm}(k)$ 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.

class `freud.order.SolLiq` (`box`, `rmax`, `Qthreshold`, `Sthreshold`, `l`)

`SolLiq`(`box`, `rmax`, `Qthreshold`, `Sthreshold`, `l`) Computes dot products of Q_{lm} between particles and uses these for clustering.

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

Args:

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.

Attributes:

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 (unsigned int): 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*)

[SolLiqNear](#)(box, rmax, Qthreshold, Sthreshold, l, kn=12) 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>

Args:

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.

Attributes:

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 (*unsigned int*): 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.

Deprecated Classes

The below functions have all either been deprecated or moved to the *Environment Module* module

Bond Order

class `freud.order.BondOrder` (*rmax, k, n, nBinsT, nBinsP*)

Deprecated Compute the bond order diagram for the system of particles.

Note: This class is only retained for backwards compatibility. Please use `freud.environment.BondOrder` instead.

Deprecated since version 0.8.2: Use `freud.environment.BondOrder` instead.

Local Descriptors

class `freud.order.LocalDescriptors` (*box, nNeigh, lmax, rmax*)

Deprecated Compute a set of descriptors (a numerical “fingerprint”) of a particle’s local environment.

Note: This class is only retained for backwards compatibility. Please use `freud.environment.LocalDescriptors` instead.

Deprecated since version 0.8.2: Use `freud.environment.LocalDescriptors` instead.

Environment Matching

class `freud.order.MatchEnv` (*box, rmax, k*)

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

Note: This class is only retained for backwards compatibility. Please use `freud.environment.MatchEnv` instead.

Deprecated since version 0.8.2: Use `freud.environment.MatchEnv` instead.

Pairing

Note: Pairing2D is deprecated and is replaced with *Bond Module*.

class `freud.order.Pairing2D` (*rmax, k, compDotTol*)

Deprecated Compute pairs for the system of particles. .. note:

This class is only retained for backwards compatibility.
Please use `:py:mod:`freud.bond`` instead.

Deprecated since version 0.8.2: Use `freud.bond` instead.

Angular Separation

class `freud.order.AngularSeparation` (*box, rmax, n*)

Deprecated Calculates the minimum angles of separation between particles and references.

Note: This class is only retained for backwards compatibility. Please use `freud.environment.AngularSeparation` instead.

Deprecated since version 0.8.2: Use `freud.environment.AngularSeparation` instead.

1.3.11 Parallel Module

Overview

<code>freud.parallel.NumThreads</code>	Context manager for managing the number of threads to use.
<code>freud.parallel.setNumThreads</code>	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.

1.3.12 PMFT Module

Overview

<code>freud.pmft.PMFTR12</code>	Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in a 2D system described by r, θ_1, θ_2 .
<code>freud.pmft.PMFTXYT</code>	Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] for systems described by coordinates x, y, θ listed in the x, y , and t arrays.
<code>freud.pmft.PMFTXY2D</code>	Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in coordinates x, y listed in the x and y arrays.
<code>freud.pmft.PMFTXYZ</code>	Computes the PMFT [vanAndersKlotsa2014] [vanAndersAhmed2014] in coordinates x, y, z , listed in the x, y , and z arrays.

Details

The 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 PMFT is defined as the negative algorithm of positional correlation function (PCF). A given set of reference points is given around which the PCF is computed and averaged in a sea of data points. The resulting values are accumulated in a PCF array listing the value of the PCF at a discrete set of points. The specific points are determined by the particular coordinate system used to represent the system.

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 .
-

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 $t1$.

- **n_t2** (*unsigned int*) – Number of bins in t2.

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 T1-values for the PCF histogram.
- **T2** ($((N_{\theta_2})$ *numpy.ndarray*) – The array of T2-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 histogram.
- **n_bins_T1** (*unsigned int*) – The number of bins in the T1-dimension of histogram.
- **n_bins_T2** (*unsigned int*) – The number of bins in the T2-dimension of 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** ($((N_{particles}, 1)$ –

:param ($N_{particles},$) *numpy.ndarray*): Reference orientations as angles used in computation. :param points: ($N_{particles}, 3$:param optional): Points used in computation. Uses `ref_points` if not

provided or None.

Parameters orientations ($((N_{particles}, 1)$ –

:param ($N_{particles},$) *numpy.ndarray*, optional): Orientations as angles used in computation. Uses `ref_orientations` if not provided or None.

Parameters 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** ($(N_{particles}, 3)$) –
- **optional** – Points used in computation. Uses `ref_points` if not provided or `None`.
- **orientations** ($(N_{particles}, 4)$) –
- **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.

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`, `t` to compute the PCF at are controlled by `x_max`, `y_max` and `n_bins_x`, `n_bins_y`, `n_bins_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_bins_x`, `n_bins_y`, `n_bins_t` is the number of bins in `x`, `y`, `t`.

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 t .

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 T -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 histogram.
- **n_bins_y** (`unsigned int`) – The number of bins in the y -dimension of histogram.

- **n_bins_T** (*unsigned int*) – The number of bins in the T-dimension of histogram.

accumulate

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

Parameters

- **box** (*freud.box.Box*) – Simulation box.
- **ref_points** ($(N_{\text{particles}}, 3)$ *numpy.ndarray*) – Reference points used in computation.
- **ref_orientations** ($(N_{\text{particles}}, 1)$ –

:param ($N_{\text{particles}},$) *numpy.ndarray*): Reference orientations as angles used in computation. :param points: :type points: ($N_{\text{particles}}, 3$:param optional): Points used in computation. Uses `ref_points` if not

provided or `None`.

Parameters orientations ($(N_{\text{particles}}, 1)$ –

:param ($N_{\text{particles}},$) *numpy.ndarray*, optional): Orientations as angles used in computation. Uses `ref_orientations` if not provided or `None`.

Parameters 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** ($(N_{\text{particles}}, 3)$ *numpy.ndarray*) – Reference points used in computation.
- **ref_orientations** ($(N_{\text{particles}}, 4)$ *numpy.ndarray*) – Reference orientations as angles used in computation.
- **points** ($(N_{\text{particles}}, 3)$ –
- **optional**) – Points used in computation. Uses `ref_points` if not provided or `None`.
- **orientations** ($(N_{\text{particles}}, 4)$ –
- **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.

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* to compute the PCF at 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 histogram.
- **n_bins_y** (*unsigned int*) – The number of bins in the y-dimension of 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** ($((N_{particles}, 1)$ –

:param $(N_{particles},)$ `numpy.ndarray`): Reference orientations as angles used in computation. :param points: :type points: $(N_{particles}, 3$:param optional): Points used in computation. Uses `ref_points` if not

provided or None.

Parameters orientations $((N_{particles}, 1)$ –

:param $(N_{particles},)$ `numpy.ndarray`, optional): Orientations as angles used in computation. Uses `ref_orientations` if not provided or None.

Parameters 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** ($(N_{particles}, 3)$ –
- **optional**) – Points used in computation. Uses *ref_points* if not provided or *None*.
- **orientations** ($(N_{particles}, 4)$ –
- **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.

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* to compute the PCF at 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** `((Nz, Ny, Nx) numpy.ndarray)` – The positional correlation function.
- **PMFT** `((Nz, Ny, Nx) numpy.ndarray)` – The potential of mean force and torque.
- **r_cut** `(float)` – The cutoff used in the cell list.
- **x** `((Nx) numpy.ndarray)` – The array of x-values for the PCF histogram.
- **y** `((Ny) numpy.ndarray)` – The array of y-values for the PCF histogram.
- **z** `((Nz) 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 histogram.
- **n_bins_y** `(unsigned int)` – The number of bins in the y-dimension of histogram.
- **n_bins_z** `(unsigned int)` – The number of bins in the z-dimension of histogram.

accumulate

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

Parameters

- **box** `(freud.box.Box)` – Simulation box.
- **ref_points** `((Nparticles, 3) numpy.ndarray)` – Reference points used in computation.
- **ref_orientations** `((Nparticles, 4) numpy.ndarray)` – Reference orientations as angles used in computation.
- **points** `((Nparticles, 3) –`
- **optional)** – Points used in computation. Uses `ref_points` if not provided or `None`.
- **orientations** `((Nparticles, 4) –`
- **optional)** – Orientations as angles used in computation. Uses `ref_orientations` if not provided or `None`.
- **face_orientations** `((Nparticles, 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 `(Nf, 4)` or a 3D array of shape `(1, Nf, 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** `((Nparticles, 3) numpy.ndarray)` – Reference points used in computation.
- **ref_orientations** `((Nparticles, 4) numpy.ndarray)` – Reference orientations as angles used in computation.
- **points** `((Nparticles, 3) –`
- **optional)** – Points used in computation. Uses `ref_points` if not provided or `None`.

- **orientations** ($(N_{particles}, 4)$ –
- **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.

1.3.13 Voronoi Module

Overview

<code>freud.voronoi.Voronoi</code>	Compute the Voronoi tessellation of a 2D or 3D system using qhull.
------------------------------------	--

Details

The 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` $\geq L/2$ 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)
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 `freud.voronoi.Voronoi.getVolumes()`.

getNeighborList

Returns a neighbor list object.

In the neighbor list, each neighbor pair has a weight value.

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.

Returns Neighbor list.

Return type `NeighborList`

getNeighbors

Get `numShells` of neighbors for each particle

Must call `computeNeighbors()` before this method.

Parameters `numShells` (`int`) – Number of neighbor shells.

getVolumes

Returns an array of volumes (areas in 2D) corresponding to Voronoi cells.

New in version 0.8.

Must call `freud.voronoi.Voronoi.computeVolumes()` before this method.

If the buffer width is too small, then some polytopes may not be closed (they may have a boundary at infinity), and these polytopes’ volumes/areas are excluded from the list.

The length of the list returned by this method should be the same as the array of positions used in the `freud.voronoi.Voronoi.compute()` method, if all the polytopes are closed. Otherwise try using a larger buffer width.

Returns Voronoi polytope volumes/areas.

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

getVoronoiPolytopes

Returns a list of polytope vertices corresponding to Voronoi cells.

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.

The length of the list returned by this method should be the same as the array of positions used in the `freud.voronoi.Voronoi.compute()` method, if all the polytopes are closed. Otherwise try using a larger buffer width.

Returns List of `numpy.ndarray` containing Voronoi polytope vertices.

Return type `list`

setBox

Reset the simulation box.

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

setBufferWidth

Reset the buffer width.

Parameters **buff** (*float*) – Buffer width.

class `freud.voronoi.VoronoiBuffer` (*box*)

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

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

Parameters **box** (`py:class:freud.box.Box`) – Simulation box.

Variables

- **buffer_particles** (`numpy.ndarray`) – The buffer particles.
- **buffer_ids** (`numpy.ndarray`) – The buffer ids.

compute

Compute the voronoi diagram.

Parameters

- **points** (`((Nparticles, 3) numpy.ndarray)`) – Points to calculate Voronoi diagram for.
- **buffer** (*float*) – Buffer distance within which to look for images.

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; Bitbucket 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
    }
}

// somewhere deep in function_b
for (int i = 0; i < n; i++)
{
    vec3[float] pos_i = position[i];
    for (int j = 0; j < n; j++)
    {
        pos_j = position[j];
        // more calls here
    }
}
```

While it *might* be possible to figure out a way to create a base C++ class all such classes inherit from, run through positions, call a calculation, and return, this would be rather complicated. Additionally, any changes to the internals of the code, and may result in performance penalties, difficulty in debugging, etc. As before, being explicit is better.

However, if you have a class which has a number of methods, each of which requires the calling of a function, this function should be written as its own method (instead of being copy-pasted into each method) as is typical in object-oriented programming.

Python vs. Cython vs. C++

The freud library is meant to leverage the power of C++ code imbued with parallel processing power from TBB with the ease of writing Python code. The bulk of your calculations should take place in C++, as shown in the snippet below:

```
# this is bad
def badHeavyLiftingInPython(positions):
    # check that positions are fine
    for i, pos_i in enumerate(positions):
        for j, pos_j in enumerate(positions):
            if i != j:
                r_ij = pos_j - pos_i
                # ...
                computed_array[i] += some_val
    return computed_array

# this is good
def goodHeavyLiftingInCplusplus(positions):
    # check that positions are fine
    cplusplus_heavy_function(computed_array, positions, len(pos))
    return computed_array
```

In the C++ code, implement the heavy lifting function called above from Python:

```
void cplusplus_heavy_function(float* computed_array,
                             float* positions,
                             int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (i != j)
            {
                r_ij = pos_j - pos_i;
                // ...
                computed_array[i] += some_val;
            }
        }
    }
}
```

Some functions may be necessary to write at the Python level due to a Python library not having an equivalent C++ library, complexity of coding, etc. In this case, the code should be written in Cython and a *reasonable* attempt to optimize the code should be made.

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
};
```

(continues on next page)

(continued from previous page)

```
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,
```

(continues on next page)

(continued from previous page)

```
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) (:py: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)
    {
        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.)
```

(continues on next page)

(continued from previous page)

```
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 someCplusplusFunction(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 2010–2018 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
- Cubatic order parameter
- Hexatic order parameter
- Pairing2D

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 LocalQI development

13. Eric Irrgang

- Authored kspace CPP code

Chrisy Du

- Authored all Steinhardt order parameters

Antonio Osorio

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

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

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
- Nematic order parameter

Benjamin Schultz

- Wrote Voronoi module

Bryan VanSaders

Ryan Marson

Tom Grubb

Yina Geng

- Co-wrote Voronoi neighbor list module
- Add properties for accessing class members

Carolyn Phillips

- Initial design and implementation
- Package name

Ben Swerdlow

James Antonaglia

Mayank Agrawal

- Co-wrote Voronoi neighbor list module

William Zygmunt

Greg van Anders

James Proctor

Rose Cersonsky

Wenbo Shen

Andrew Karas

- Angular separation

Paul Dodd

Tim Moore

- Added optional rmin argument to density.RDF

Michael Engel

- Translational order parameter

Alex Dutton

- BiMap class for 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,
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 [Bitbucket](#) 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

- [Bokeh] Bokeh Development Team (2014). Bokeh: Python library for interactive visualization URL <http://www.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] P.J. Steinhardt, D.R. Nelson, and M. Ronchetti (1983). Bond-orientational order in liquids and glasses. *Phys. Rev. B* 28(784). <https://doi.org/10.1103/PhysRevFluids.2.093301>

f

- `freud.bond`, 5
- `freud.box`, 10
- `freud.cluster`, 13
- `freud.density`, 15
- `freud.environment`, 21
- `freud.index`, 28
- `freud.interface`, 30
- `freud.kspace`, 31
- `freud.locality`, 42
- `freud.order`, 48
- `freud.parallel`, 60
- `freud.pmft`, 61
- `freud.voronoi`, 68

Symbols

`__call__()` (freud.index.Index2D method), 29
`__call__()` (freud.index.Index3D method), 30

A

accumulate (freud.density.ComplexCF attribute), 17
 accumulate (freud.density.FloatCF attribute), 16
 accumulate (freud.density.RDF attribute), 20
 accumulate (freud.environment.BondOrder attribute), 22
 accumulate (freud.pmft.PMFTR12 attribute), 62
 accumulate (freud.pmft.PMFTXY2D attribute), 65
 accumulate (freud.pmft.PMFTXYT attribute), 64
 accumulate (freud.pmft.PMFTXYZ attribute), 67
 add_ptype (freud.kspace.SingleCell3D attribute), 34
 addFT (freud.kspace.FTfactory attribute), 36
 AlignedBoxConstraint (class in freud.kspace), 41
 AnalyzeSFactor3D (class in freud.kspace), 32
 AngularSeparation (class in freud.environment), 27
 AngularSeparation (class in freud.order), 60

B

BondingAnalysis (class in freud.bond), 6
 BondingR12 (class in freud.bond), 6
 BondingXY2D (class in freud.bond), 7
 BondingXYT (class in freud.bond), 8
 BondingXYZ (class in freud.bond), 9
 BondOrder (class in freud.environment), 21
 BondOrder (class in freud.order), 59
 Box (class in freud.box), 10

C

calculate (freud.kspace.SingleCell3D attribute), 34
 Cluster (class in freud.cluster), 13
 cluster (freud.environment.MatchEnv attribute), 24
 ClusterProperties (class in freud.cluster), 14
 ComplexCF (class in freud.density), 17
 compute (freud.bond.BondingAnalysis attribute), 6
 compute (freud.bond.BondingR12 attribute), 7
 compute (freud.bond.BondingXY2D attribute), 8

compute (freud.bond.BondingXYT attribute), 8
 compute (freud.bond.BondingXYZ attribute), 9
 compute (freud.density.ComplexCF attribute), 18
 compute (freud.density.FloatCF attribute), 16
 compute (freud.density.GaussianDensity attribute), 19
 compute (freud.density.LocalDensity attribute), 19
 compute (freud.density.RDF attribute), 21
 compute (freud.environment.BondOrder attribute), 22
 compute (freud.environment.LocalDescriptors attribute), 23
 compute (freud.environment.Pairing2D attribute), 27
 compute (freud.interface.InterfaceMeasure attribute), 31
 compute (freud.kspace.FTdelta attribute), 37
 compute (freud.kspace.FTpolyhedron attribute), 38
 compute (freud.kspace.SFactor3DPoints attribute), 32
 compute (freud.locality.LinkCell attribute), 45
 compute (freud.locality.NearestNeighbors attribute), 46
 compute (freud.order.CubaticOrderParameter attribute), 49
 compute (freud.order.HexOrderParameter attribute), 50
 compute (freud.order.LocalQI attribute), 51
 compute (freud.order.LocalQINear attribute), 53
 compute (freud.order.LocalW1 attribute), 54
 compute (freud.order.LocalWINear attribute), 56
 compute (freud.order.NematicOrderParameter attribute), 49
 compute (freud.order.SolLiq attribute), 57
 compute (freud.order.SolLiqNear attribute), 58
 compute (freud.order.TransOrderParameter attribute), 50
 compute (freud.pmft.PMFTR12 attribute), 62
 compute (freud.pmft.PMFTXY2D attribute), 65
 compute (freud.pmft.PMFTXYT attribute), 64
 compute (freud.pmft.PMFTXYZ attribute), 67
 compute (freud.voronoi.Voronoi attribute), 69
 compute (freud.voronoi.VoronoiBuffer attribute), 71
 compute_py (freud.kspace.FTconvexPolyhedron attribute), 39
 computeAve (freud.order.LocalQI attribute), 52
 computeAve (freud.order.LocalQINear attribute), 53
 computeAve (freud.order.LocalW1 attribute), 54

computeAve (freud.order.LocalWlNear attribute), 56
computeAveNorm (freud.order.LocalQl attribute), 52
computeAveNorm (freud.order.LocalQlNear attribute), 53
computeAveNorm (freud.order.LocalWl attribute), 55
computeAveNorm (freud.order.LocalWlNear attribute), 56
computeCellList (freud.locality.LinkCell attribute), 45
computeClusterMembership (freud.cluster.Cluster attribute), 14
computeClusters (freud.cluster.Cluster attribute), 14
computeGlobal (freud.environment.AngularSeparation attribute), 27
computeNeighbor (freud.environment.AngularSeparation attribute), 28
computeNeighbors (freud.voronoi.Voronoi attribute), 69
computeNList (freud.environment.LocalDescriptors attribute), 24
computeNorm (freud.order.LocalQl attribute), 52
computeNorm (freud.order.LocalQlNear attribute), 53
computeNorm (freud.order.LocalWl attribute), 55
computeNorm (freud.order.LocalWlNear attribute), 56
computeProperties (freud.cluster.ClusterProperties attribute), 15
computeSolLiqNoNorm (freud.order.SolLiq attribute), 57
computeSolLiqNoNorm (freud.order.SolLiqNear attribute), 58
computeSolLiqVariant (freud.order.SolLiq attribute), 57
computeSolLiqVariant (freud.order.SolLiqNear attribute), 58
computeVolumes (freud.voronoi.Voronoi attribute), 69
constrainedLatticePoints() (in module freud.kspace), 41
Constraint (class in freud.kspace), 41
copy (freud.locality.NeighborList attribute), 43
CubicOrderParameter (class in freud.order), 48
cube() (freud.box.Box class method), 11

D

DeltaSpot (class in freud.kspace), 40

F

filter (freud.locality.NeighborList attribute), 43
filter_r (freud.locality.NeighborList attribute), 43
find_first_index (freud.locality.NeighborList attribute), 43
FloatCF (class in freud.density), 15
freud.bond (module), 5
freud.box (module), 10
freud.cluster (module), 13
freud.density (module), 15
freud.environment (module), 21
freud.index (module), 28
freud.interface (module), 30
freud.kspace (module), 31

freud.locality (module), 42
freud.order (module), 48
freud.parallel (module), 60
freud.pmft (module), 61
freud.voronoi (module), 68
from_arrays() (freud.locality.NeighborList class method), 43
from_box() (freud.box.Box class method), 11
from_matrix() (freud.box.Box class method), 11
FTbase (class in freud.kspace), 36
FTconvexPolyhedron (class in freud.kspace), 39
FTdelta (class in freud.kspace), 37
FTfactory (class in freud.kspace), 36
FTpolyhedron (class in freud.kspace), 38
FTsphere (class in freud.kspace), 38

G

GaussianDensity (class in freud.density), 18
GaussianSpot (class in freud.kspace), 40
get_density (freud.kspace.FTbase attribute), 36
get_form_factors (freud.kspace.SingleCell3D attribute), 34
get_gridPoints (freud.kspace.DeltaSpot attribute), 40
get_parambyname (freud.kspace.FTbase attribute), 36
get_params (freud.kspace.FTbase attribute), 37
get_ptypes (freud.kspace.SingleCell3D attribute), 34
get_radius (freud.kspace.FTconvexPolyhedron attribute), 39
get_radius (freud.kspace.FTpolyhedron attribute), 38
get_radius (freud.kspace.FTsphere attribute), 38
get_scale (freud.kspace.FTbase attribute), 37
getCell (freud.locality.LinkCell attribute), 45
getCellNeighbors (freud.locality.LinkCell attribute), 45
getCoordinates (freud.box.Box attribute), 12
getEnvironment (freud.environment.MatchEnv attribute), 25
getFT (freud.kspace.FTbase attribute), 36
getFTlist (freud.kspace.FTfactory attribute), 36
getFTObject (freud.kspace.FTfactory attribute), 36
getImage (freud.box.Box attribute), 12
getLatticeVector (freud.box.Box attribute), 12
getNeighborList (freud.locality.NearestNeighbors attribute), 47
getNeighborList (freud.voronoi.Voronoi attribute), 70
getNeighbors (freud.locality.NearestNeighbors attribute), 47
getNeighbors (freud.voronoi.Voronoi attribute), 70
getNumElements (freud.index.Index2D attribute), 29
getNumElements (freud.index.Index3D attribute), 30
getPeakDegeneracy (freud.kspace.AnalyzeSFactor3D attribute), 33
getPeakList (freud.kspace.AnalyzeSFactor3D attribute), 33
getQ (freud.kspace.SFactor3DPoints attribute), 32

getRsQ (freud.locality.NearestNeighbors attribute), 47
 getS (freud.kspace.SFactor3DPoints attribute), 32
 getSComplex (freud.kspace.SFactor3DPoints attribute), 32
 getSvsQ (freud.kspace.AnalyzeSFactor3D attribute), 33
 getVolumes (freud.voronoi.Voronoi attribute), 70
 getVoronoiPolytopes (freud.voronoi.Voronoi attribute), 70

H

HexOrderParameter (class in freud.order), 49

I

Index2D (class in freud.index), 28
 Index3D (class in freud.index), 29
 initialize (freud.bond.BondingAnalysis attribute), 6
 InterfaceMeasure (class in freud.interface), 30
 is2D (freud.box.Box attribute), 12
 isSimilar (freud.environment.MatchEnv attribute), 25
 IteratorLinkCell (class in freud.locality), 44
 itercell (freud.locality.LinkCell attribute), 46

L

LinkCell (class in freud.locality), 44
 LocalDensity (class in freud.density), 19
 LocalDescriptors (class in freud.environment), 23
 LocalDescriptors (class in freud.order), 59
 LocalQI (class in freud.order), 51
 LocalQINear (class in freud.order), 52
 LocalWI (class in freud.order), 53
 LocalWINear (class in freud.order), 55

M

makeFraction (freud.box.Box attribute), 12
 makeSpot (freud.kspace.DeltaSpot attribute), 40
 makeSpot (freud.kspace.GaussianSpot attribute), 40
 MatchEnv (class in freud.environment), 24
 MatchEnv (class in freud.order), 59
 matchMotif (freud.environment.MatchEnv attribute), 25
 meshgrid2() (in module freud.kspace), 41
 minimizeRMSD (freud.environment.MatchEnv attribute), 26
 minRMSDMotif (freud.environment.MatchEnv attribute), 26

N

NearestNeighbors (class in freud.locality), 46
 NeighborList (class in freud.locality), 42
 NematicOrderParameter (class in freud.order), 49
 next (freud.locality.IteratorLinkCell attribute), 44
 NumThreads (class in freud.parallel), 60

P

Pairing2D (class in freud.environment), 26

Pairing2D (class in freud.order), 59
 PMFTR12 (class in freud.pmft), 61
 PMFTXY2D (class in freud.pmft), 64
 PMFTXYT (class in freud.pmft), 63
 PMFTXYZ (class in freud.pmft), 66

R

RDF (class in freud.density), 20
 reciprocalLattice3D() (in module freud.kspace), 41
 remove_ptype (freud.kspace.SingleCell3D attribute), 34
 reset (freud.density.ComplexCF attribute), 18
 reset (freud.density.FloatCF attribute), 17
 reset (freud.density.RDF attribute), 21
 reset (freud.environment.BondOrder attribute), 23
 reset (freud.pmft.PMFTR12 attribute), 63
 reset (freud.pmft.PMFTXY2D attribute), 66
 reset (freud.pmft.PMFTXYT attribute), 64
 reset (freud.pmft.PMFTXYZ attribute), 68

S

satisfies (freud.kspace.AlignedBoxConstraint attribute), 41
 satisfies (freud.kspace.Constraint attribute), 41
 set_active (freud.kspace.SingleCell3D attribute), 34
 set_box (freud.kspace.SingleCell3D attribute), 34
 set_density (freud.kspace.FTbase attribute), 37
 set_density (freud.kspace.FTdelta attribute), 37
 set_density (freud.kspace.FTpolyhedron attribute), 38
 set_dK (freud.kspace.SingleCell3D attribute), 34
 set_form_factor (freud.kspace.SingleCell3D attribute), 35
 set_inactive (freud.kspace.SingleCell3D attribute), 35
 set_K (freud.kspace.FTbase attribute), 37
 set_K (freud.kspace.FTdelta attribute), 37
 set_K (freud.kspace.FTpolyhedron attribute), 38
 set_k (freud.kspace.SingleCell3D attribute), 35
 set_ndiv (freud.kspace.SingleCell3D attribute), 35
 set_param (freud.kspace.SingleCell3D attribute), 35
 set_parambyname (freud.kspace.FTbase attribute), 37
 set_params (freud.kspace.FTpolyhedron attribute), 38
 set_radius (freud.kspace.FTconvexPolyhedron attribute), 39
 set_radius (freud.kspace.FTpolyhedron attribute), 39
 set_radius (freud.kspace.FTsphere attribute), 38
 set_rq (freud.kspace.FTbase attribute), 37
 set_rq (freud.kspace.FTdelta attribute), 38
 set_rq (freud.kspace.FTpolyhedron attribute), 39
 set_rq (freud.kspace.SingleCell3D attribute), 35
 set_scale (freud.kspace.FTbase attribute), 37
 set_scale (freud.kspace.FTdelta attribute), 38
 set_scale (freud.kspace.SingleCell3D attribute), 35
 set_sigma (freud.kspace.GaussianSpot attribute), 40
 set_xy (freud.kspace.DeltaSpot attribute), 40
 set_xy (freud.kspace.GaussianSpot attribute), 40
 setBox (freud.environment.MatchEnv attribute), 26

setBox (freud.order.LocalQl attribute), 52
setBox (freud.order.LocalQlNear attribute), 53
setBox (freud.order.LocalWl attribute), 55
setBox (freud.order.LocalWlNear attribute), 56
setBox (freud.voronoi.Voronoi attribute), 70
setBufferWidth (freud.voronoi.Voronoi attribute), 70
setCutMode (freud.locality.NearestNeighbors attribute),
47
setNumThreads (in module freud.parallel), 60
setNumThreads() (freud.parallel method), 60
SFactor3DPoints (class in freud.kspace), 32
SingleCell3D (class in freud.kspace), 33
SolLiq (class in freud.order), 56
SolLiqNear (class in freud.order), 57
Spoly2D (freud.kspace.FTconvexPolyhedron attribute),
39
Spoly3D (freud.kspace.FTconvexPolyhedron attribute),
39
square() (freud.box.Box class method), 12

T

to_dict (freud.box.Box attribute), 12
to_matrix (freud.box.Box attribute), 12
to_tuple (freud.box.Box attribute), 12
TransOrderParameter (class in freud.order), 50

U

unwrap (freud.box.Box attribute), 12
update_bases (freud.kspace.SingleCell3D attribute), 36
update_K_constraint (freud.kspace.SingleCell3D attribute), 35
update_Kpoints (freud.kspace.SingleCell3D attribute), 35

V

Voronoi (class in freud.voronoi), 68
VoronoiBuffer (class in freud.voronoi), 71

W

wrap (freud.box.Box attribute), 13