

---

# **freud Documentation**

***Release 0.8.1***

**The Regents of the University of Michigan**

**May 09, 2018**



---

## Contents

---

|          |                                    |            |
|----------|------------------------------------|------------|
| <b>1</b> | <b>Contents</b>                    | <b>3</b>   |
| 1.1      | Examples . . . . .                 | 3          |
| 1.2      | Installation . . . . .             | 3          |
| 1.3      | Modules . . . . .                  | 5          |
| 1.4      | Development Guide . . . . .        | 85         |
| 1.5      | References and Citations . . . . . | 93         |
| 1.6      | License . . . . .                  | 93         |
| 1.7      | Credits . . . . .                  | 94         |
| <b>2</b> | <b>Index</b>                       | <b>99</b>  |
|          | <b>Bibliography</b>                | <b>101</b> |
|          | <b>Python Module Index</b>         | <b>103</b> |



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

The freud library is a Python package meant for the analysis of molecular dynamics and Monte Carlo simulation trajectories. The freud library works with and returns [NumPy](#) arrays.

Please visit our repository on [Bitbucket](#) for the library source code, post issues or bugs to our [issue tracker](#), and ask questions and discuss on our [forum](#).



## 1.1 Examples

Examples are provided as [Jupyter](#) notebooks in a separate [freud-examples](#) repository. These can be run locally with the `jupyter notebook` command. These examples will also be provided as static notebooks on [NBViewer](#) and interactive notebooks on [MyBinder](#).

Visualization of data is done via [Bokeh](#) *[Cit0]*.

## 1.2 Installation

### 1.2.1 Requirements

- [NumPy](#) is **required** to build freud
- [Cython](#) `>= 0.23` is **required** to compile your own `_freud.cpp` file. Cython **is not required** to install freud
- [Boost](#) is **required** to run freud
- [Intel Threading Building Blocks](#) is **required** to run freud

### 1.2.2 Documentation

You may use the online documentation from [ReadTheDocs](#), or you may build the documentation yourself:

#### Building the documentation

The documentation is build with sphinx. To install sphinx, run

```
conda install sphinx
```

or

```
pip install sphinx
```

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

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

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

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

### 1.2.3 Installation

Install freud via [conda](#), [glotzpkgs](#), or compile from source.

#### Install via conda

The code below will enable the glotzer conda channel and install freud.

```
conda config --add channels glotzer
conda install freud
```

#### Install via glotzpkgs

Please refer to the official [glotzpkgs](#) documentation.

First, make sure you have a working glotzpkgs environment.

```
# install from provided binary
gpacman -S freud
# installing your own version
cd /path/to/glotzpkgs/freud
gmakepkg
# tab completion is your friend here
gpacman -U freud-<version>-flux.pkg.tar.gz
# now you can load the binary
module load freud
```

#### Compile from source

It is easiest to install freud with a working conda install of the required packages:

- python (2.7, 3.4, 3.5, 3.6)
- numpy
- boost (2.7, 3.3 provided on flux, 3.4, 3.5)
- icu (requirement of boost)

- cython (not required, but a correct `_freud.cpp` file must be present to compile)
- tbb
- cmake

The code that follows creates a build directory inside the freud source directory and builds freud:

```
mkdir build
cd build
cmake ../
# Use `cmake ../ -DENABLE_CYTHON=ON` to rebuild _freud.cpp
make install -j4
```

By default, freud installs to the `USER_SITE` directory. `USER_SITE` is on the Python search path by default, so there is no need to modify `PYTHONPATH`.

To run out of the build directory, run `make -j4` instead of `make install -j4` and then add the build directory to your `PYTHONPATH`.

---

**Note:** freud makes use of submodules. CMake has been configured to automatically init and update submodules. However, if this does not work, or you would like to do this yourself, please execute:

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

---

## 1.2.4 Unit Tests

Run all unit tests with `nosetests` in the source directory. To add a test, simply add a file to the `tests` directory, and `nosetests` will automatically discover it. Read this [introduction to nosetests](#) for more information.

```
# Install nose
conda install nose
# Run tests from the source directory
nosetests
```

## 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

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 in 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:  $\theta$ ):**
    - $X, Y$
    - $X, Y, \theta_2$
    - $r, \theta_1, \theta_2$
  - **3D particle coordinates:**
    - $X, Y, Z$
- 

## Bonding Analysis

**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** – number of bonds to track

### **bond\_lifetimes**

Return the bond lifetimes.

**compute** (*self*, *frame\_0*, *frame\_1*)

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

### Parameters

- **frame\_0** (*numpy.ndarray*, shape=( $N_{particles}$ ,  $N_{bonds}$ ), dtype= *numpy.uint32*) – current/previous bonding frame (as output from *BondingR12* modules)
- **frame\_1** (*numpy.ndarray*, shape=( $N_{particles}$ ,  $N_{bonds}$ ), dtype= *numpy.uint32*) – next/current bonding frame (as output from *BondingR12* modules)

**getBondLifetimes** (*self*)

Return the bond lifetimes.

**Returns** lifetime of bonds

**Return type** *numpy.ndarray*, shape=( $N_{particles}$ , varying), dtype= *numpy.uint32*

**getNumBonds** (*self*)

Get number of bonds tracked.

**Returns** number of bonds

**Return type** unsigned int

**getNumFrames** (*self*)

Get number of frames calculated.

**Returns** number of frames

**Return type** unsigned int

**getNumParticles** (*self*)

Get number of particles being tracked.

**Returns** number of particles

**Return type** unsigned int

**getOverallLifetimes** (*self*)

Return the overall lifetimes.

**Returns** lifetime of bonds

**Return type** `numpy.ndarray`, shape=( $N_{particles}$ , varying), dtype= `numpy.uint32`

**getTransitionMatrix** (*self*)

Return the transition matrix.

**Returns** transition matrix

**Return type** `numpy.ndarray`

**initialize** (*self*, *frame\_0*)

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

**Parameters** **frame\_0** (`numpy.ndarray`, shape=( $N_{particles}$ ,  $N_{bonds}$ ), dtype= `numpy.uint32`) – first bonding frame (as output from *BondingR12* modules)

**num\_bonds**

Get number of bonds being tracked.

**num\_frames**

Get number of frames calculated.

**num\_particles**

Get number of particles being tracked.

**overall\_lifetimes**

Return the overall lifetimes.

**transition\_matrix**

Return the transition matrix.

### Coordinate System: $x, y$

**class** `freud.bond.BondingXY2D` (*x\_max*, *y\_max*, *bond\_map*, *bond\_list*)

Compute the bonds each particle in the 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`

**bonds**

Return the particle bonds.

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *nlist=None*)

Calculates the correlation function and adds to the current histogram.

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape=( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the bonding
- **ref\_orientations** (*numpy.ndarray*, shape=( $N_{particles}$ ), dtype= *numpy.float32*) – orientations as angles to use in computation
- **points** (*numpy.ndarray*, shape=( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the bonding
- **orientations** (*numpy.ndarray*, shape=( $N_{particles}$ ), dtype= *numpy.float32*) – orientations as angles to use in computation
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**getBonds** (*self*)

Return the particle bonds.

**Returns** particle bonds

**Return type** *numpy.ndarray*

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** *freud.box.Box()*

**getListMap** (*self*)

Get the dict used to map list idx to bond idx.

**Returns** list\_map

**Return type** dict

```
>>> list_idx = list_map[bond_idx]
```

**getRevListMap** (*self*)

Get the dict used to map list idx to bond idx.

**Returns** list\_map

**Return type** dict

```
>>> bond_idx = list_map[list_idx]
```

**list\_map**

Get the dict used to map list idx to bond idx.

**rev\_list\_map**

Get the dict used to map list idx to bond idx.

**Coordinate System:**  $x, y, \theta_2$

**class** `freud.bond.BondingXYT` ( $x\_max, y\_max, bond\_map, bond\_list$ )

Compute the bonds each particle in the 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`

#### bonds

Return the particle bonds.

#### box

Get the box used in the calculation.

**compute** (*self, box, ref\_points, ref\_orientations, points, orientations, nlist=None*)

Calculates the correlation function and adds to the current histogram.

#### Parameters

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the bonding
- **ref\_orientations** (*numpy.ndarray*, `shape=( $N_{particles}$ )`, `dtype= numpy.float32`) – orientations as angles to use in computation
- **points** (*numpy.ndarray*, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the bonding
- **orientations** (*numpy.ndarray*, `shape=( $N_{particles}$ )`, `dtype= numpy.float32`) – orientations as angles to use in computation
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**getBonds** (*self*)

Return the particle bonds.

**Returns** particle bonds

**Return type** *numpy.ndarray*

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** *freud.box.Box()*

**getListMap** (*self*)

Get the dict used to map list idx to bond idx.

**Returns** list\_map

**Return type** dict

```
>>> list_idx = list_map[bond_idx]
```

**getRevListMap** (*self*)

Get the dict used to map list idx to bond idx.

**Returns** list\_map

**Return type** dict

```
>>> bond_idx = list_map[list_idx]
```

**list\_map**

Get the dict used to map list idx to bond idx.

**rev\_list\_map**

Get the dict used to map list idx to bond idx.

### Coordinate System: $r, \theta_1, \theta_2$

**class** freud.bond.**BondingR12** (*r\_max, bond\_map, bond\_list*)

Compute the bonds each particle in the system.

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

*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, t2, t1 coordinate
- **bond\_list** (*numpy.ndarray*) – list containing the bond indices to be tracked, `bond_list[i] = bond_index`

**bonds**

Return the particle bonds.

**box**

Get the box used in the calculation.

**compute** (*self, box, ref\_points, ref\_orientations, points, orientations, nlist=None*)

Calculates the correlation function and adds to the current histogram.

#### Parameters

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the bonding
- **ref\_orientations** (*numpy.ndarray*, `shape=( $N_{particles}$ )`, `dtype= numpy.float32`) – orientations as angles to use in computation
- **points** (*numpy.ndarray*, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the bonding

- **orientations** (`numpy.ndarray`, `shape=( $N_{particles}$ )`, `dtype= numpy.float32`)  
– orientations as angles to use in computation
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**getBonds** (*self*)

Return the particle bonds.

**Returns** particle bonds

**Return type** `numpy.ndarray`

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** `freud.box.Box()`

**getListMap** (*self*)

Get the dict used to map list idx to bond idx.

**Returns** list\_map

**Return type** dict

```
>>> list_idx = list_map[bond_idx]
```

**getRevListMap** (*self*)

Get the dict used to map list idx to bond idx.

**Returns** list\_map

**Return type** dict

```
>>> bond_idx = list_map[list_idx]
```

**list\_map**

Get the dict used to map list idx to bond idx.

**rev\_list\_map**

Get the dict used to map list idx to bond idx.

**Coordinate System:**  $x, y, z$

**class** `freud.bond.BondingXYZ` (*x\_max*, *y\_max*, *z\_max*, *bond\_map*, *bond\_list*)

Compute the bonds each particle in the 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`

**bonds**

Return the particle bonds.

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *nlist=None*)

Calculates the correlation function and adds to the current histogram.

**Parameters**

- **box** (`freud.box.Box`) – simulation box
- **ref\_points** (`numpy.ndarray`, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the bonding
- **ref\_orientations** (`numpy.ndarray`, `shape=( $N_{particles}$ , 4)`, `dtype= numpy.float32`) – orientations as quaternions to use in computation
- **points** (`numpy.ndarray`, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the bonding
- **orientations** (`numpy.ndarray`, `shape=( $N_{particles}$ , 4)`, `dtype= numpy.float32`) – orientations as quaternions to use in computation
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**getBonds** (*self*)

Return the particle bonds.

**Returns** particle bonds

**Return type** `numpy.ndarray`

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** `freud.box.Box()`

**getListMap** (*self*)

Get the dict used to map list idx to bond idx.

**Returns** list\_map

**Return type** `dict`

```
>>> list_idx = list_map[bond_idx]
```

**getRevListMap** (*self*)

Get the dict used to map list idx to bond idx.

**Returns** list\_map

**Return type** `dict`

```
>>> bond_idx = list_map[list_idx]
```

**list\_map**

Get the dict used to map list idx to bond idx.

**rev\_list\_map**

Get the dict used to map list idx to bond idx.

## 1.3.2 Box Module

Contains data structures for simulation boxes.

### Simulation Box

**class** freud.box.Box(\*args, \*\*kwargs)

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

For more information about the definition of the simulation box, 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.

**L**

Return the lengths of the box as a tuple (x, y, z).

**Linv**

Return the inverse lengths of the box (1/Lx, 1/Ly, 1/Lz).

**Returns** dimensions of the box as (1/Lx, 1/Ly, 1/Lz)

**Return type** (*float, float, float*)

**Lx**

Length of the x-dimension of the box.

**Getter** Returns this box's x-dimension length

**Setter** Sets this box's x-dimension length

**Type** *float*

**Ly**

Length of the y-dimension of the box.

**Getter** Returns this box's y-dimension length

**Setter** Sets this box's y-dimension length

**Type** float

**Lz**

Length of the z-dimension of the box.

**Getter** Returns this box's z-dimension length

**Setter** Sets this box's z-dimension length

**Type** float

**classmethod cube** (*L*)

Construct a cubic box with equal lengths.

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

**dimensions**

Number of dimensions of this box (only 2 or 3 are supported).

**Getter** Returns this box's number of dimensions

**Setter** Sets this box's number of dimensions

**Type** int

**classmethod from\_box** (*box*)

Initialize a box instance from another box instance.

**classmethod from\_matrix** (*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>

**getCoordinates** (*self*, *f*)

Alias for *makeCoordinates()*

Deprecated since version 0.8: Use *makeCoordinates()* instead.

**getImage** (*self*, *vec*)

Returns the image corresponding to a wrapped vector.

New in version 0.8.

**Parameters** *vec* (*numpy.ndarray*, *shape=(3)*, *dtype= numpy.float32*) – Coordinates of unwrapped vector

**Returns** Image index vector

**Return type** *numpy.ndarray*, *shape=(3)*, *dtype= numpy.int32*

**getL** (*self*)

Return the lengths of the box as a tuple (x, y, z).

**Returns** dimensions of the box as (x, y, z)

**Return type** (*float*, *float*, *float*)

**getLatticeVector** (*self*, *i*)

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*

**getLinv** (*self*)

Return the inverse lengths of the box (1/Lx, 1/Ly, 1/Lz).

**Returns** dimensions of the box as (1/Lx, 1/Ly, 1/Lz)

**Return type** (float, float, float)

**getLx** (*self*)

Length of the x-dimension of the box.

**Returns** This box's x-dimension length

**Return type** float

**getLy** (*self*)

Length of the y-dimension of the box.

**Returns** This box's y-dimension length

**Return type** float

**getLz** (*self*)

Length of the z-dimension of the box.

**Returns** This box's z-dimension length

**Return type** float

**getPeriodic** (*self*)

Get the box's periodicity in each dimension.

**Returns** Periodic attributes in x, y, z

**Return type** list[bool, bool, bool]

**getPeriodicX** (*self*)

Get the box periodicity in the x direction.

**Returns** True if periodic, False if not

**Return type** bool

**getPeriodicY** (*self*)

Get the box periodicity in the y direction.

**Returns** True if periodic, False if not

**Return type** bool

**getPeriodicZ** (*self*)

Get the box periodicity in the z direction.

**Returns** True if periodic, False if not

**Return type** bool

**getTiltFactorXY** (*self*)

Return the tilt factor xy.

**Returns** xy tilt factor

**Return type** float

**getTiltFactorXZ** (*self*)

Return the tilt factor xz.

**Returns** xz tilt factor

**Return type** `float`

**getTiltFactorYZ** (*self*)

Return the tilt factor yz.

**Returns** yz tilt factor

**Return type** `float`

**getVolume** (*self*)

Return the box volume (area in 2D).

**Returns** box volume

**Return type** `float`

**is2D** (*self*)

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

**Returns** True if 2D, False if 3D

**Return type** `bool`

**makeCoordinates** (*self*, *f*)

Convert fractional coordinates into real coordinates.

**Parameters** **f** (`numpy.ndarray`, shape= (3), dtype= `numpy.float32`) – Fractional coordinates (*x*, *y*, *z*) between 0 and 1 within parallelepipedal box

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

**Return type** `list[float, float, float]`

**makeFraction** (*self*, *vec*)

Convert real coordinates into fractional coordinates.

**Parameters** **vec** (`numpy.ndarray`, shape= (3), dtype= `numpy.float32`) – Real coordinates within parallelepipedal box

**Returns** A fractional coordinate vector

**periodic**

Box periodicity in each dimension.

**Getter** Returns this box's periodicity in each dimension (True if periodic, False if not)

**Setter** Set this box's periodicity in each dimension

**Type** `list[bool, bool, bool]`

**set2D** (*self*, *val*)

Set the dimensionality to 2D (True) or 3D (False).

**Parameters** **val** (`bool`) – 2D=True, 3D=False

**setL** (*self*, *L*)

Set all side lengths of box to L.

**Parameters** **L** (`float`) – Side length of box

**setPeriodic** (*self*, *x*, *y*, *z*)

Set the box's periodicity in each dimension.

**Parameters**

- **x** (`bool`) – True if periodic in x, False if not
- **y** (`bool`) – True if periodic in y, False if not

- **z** (*bool*) – True if periodic in z, False if not

**setPeriodicX** (*self*, *val*)

Set the box periodicity in the x direction.

**Parameters** **val** (*bool*) – True if periodic, False if not

**setPeriodicY** (*self*, *val*)

Set the box periodicity in the y direction.

**Parameters** **val** (*bool*) – True if periodic, False if not

**setPeriodicZ** (*self*, *val*)

Set the box periodicity in the z direction.

**Parameters** **val** (*bool*) – True if periodic, False if not

**classmethod square** (*L*)

Construct a 2-dimensional (square) box with equal lengths.

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

**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** (*self*, *vecs*, *imgs*)

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** (*numpy.ndarray*, shape= (3) or (*N*, 3), dtype= *numpy.float32*) – Single vector or array of *N* vectors
- **imgs** (*numpy.ndarray*, shape= (3) or (*N*, 3), dtype= *numpy.int32*) – Single image index or array of *N* image indices

**Note** vecs are returned in place (nothing returned)

**volume**

Return the box volume (area in 2D).

**Returns** box volume

**Return type** *float*

**wrap** (*self*, *vecs*)

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` (`numpy.ndarray`, `shape= (3)` or `(N,3)`, `dtype= numpy.float32`) – Single vector or array of  $N$  vectors

**Note** `vecs` are returned in place (nothing returned)

**xy**

Tilt factor xy of the box.

**Returns** xy tilt factor

**Return type** `float`

**xz**

Tilt factor xz of the box.

**Returns** xz tilt factor

**Return type** `float`

**yz**

Tilt factor yz of the box.

**Returns** yz tilt factor

**Return type** `float`

## 1.3.3 Cluster Module

### Cluster Functions

**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 labelled from 0 to the number of clusters-1 and an index array is returned where `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](mailto:joaander@umich.edu)>*

#### Parameters

- **box** (`freud.box.Box`) – 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.

---

**box**

Return the stored freud Box.

**cluster\_idx**

Returns 1D array of Cluster idx for each particle.

**cluster\_keys**

Returns the keys contained in each cluster.

**computeClusterMembership** (*self*, *keys*)

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 *getClusterKeys()*.

**Parameters** *keys* (*numpy.ndarray*, shape=( $N_{particles}$ ), dtype= *numpy.uint32*) – Membership keys, one for each particle

**computeClusters** (*self*, *points*, *nlist=None*, *box=None*)

Compute the clusters for the given set of points.

**Parameters**

- **points** (*numpy.ndarray*, shape=( $N_{particles}$ , 3), dtype= *numpy.float32*) – particle coordinates
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds
- **box** (*freud.box.Box*) – simulation box

**getBox** (*self*)

Return the stored freud Box.

**Returns** freud Box

**Return type** *freud.box.Box*

**getClusterIdx** (*self*)

Returns 1D array of Cluster idx for each particle

**Returns** 1D array of cluster idx

**Return type** *numpy.ndarray*, shape=( $N_{particles}$ ), dtype= *numpy.uint32*

**getClusterKeys** (*self*)

Returns the keys contained in each cluster.

**Returns** list of lists of each key contained in clusters

**Return type** *list*

**getNumClusters** (*self*)

Returns the number of clusters.

**Returns** number of clusters

**Return type** *int*

**getNumParticles** (*self*)

Returns the number of particles.

**Returns** number of particles

**Return type** *int*

**num\_clusters**

Returns the number of clusters.

**num\_particles**

Returns the number of particles.

**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 `numpy.ndarray`, shape=  $(N_{clusters}, 3)$ .

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

Module author: Joshua Anderson <[joaander@umich.edu](mailto:joaander@umich.edu)>

**Parameters** `box` (`freud.box.Box`) – simulation box

**box**

Return the stored freud Box.

**cluster\_COM**

Returns the center of mass of the last computed cluster.

**cluster\_G**

Returns the cluster  $G$  tensors computed by the last call to `computeProperties()`. `computeProperties`.

**cluster\_sizes**

Returns the cluster sizes computed by the last call to `computeProperties()`. `computeProperties`.

**computeProperties** (*self*, *points*, *cluster\_idx*, *box=None*)

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** (`numpy.ndarray`, shape= $(N_{particles}, 3)$ , dtype= `numpy.float32`) – Positions of the particles making up the clusters
- **cluster\_idx** (`numpy.ndarray`, shape= $(N_{particles})$ , dtype= `numpy.uint32`) – List of cluster indexes for each particle
- **box** (`freud.box.Box`) – simulation box

**getBox** (*self*)

Return the stored `freud.box.Box` object.

**Returns** freud Box

**Return type** `freud.box.Box`

**getClusterCOM** (*self*)

Returns the center of mass of the last computed cluster.

**Returns** numpy array of cluster center of mass coordinates  $(x, y, z)$

**Return type** `numpy.ndarray`, shape= $(N_{clusters}, 3)$ , dtype= `numpy.float32`

**getClusterG**(*self*)

Returns the cluster  $G$  tensors computed by the last call to `computeProperties()`.

**Returns** list of gyration tensors for each cluster

**Return type** `numpy.ndarray`, shape=( $N_{clusters}$ , 3, 3), dtype= `numpy.float32`

**getClusterSizes**(*self*)

Returns the cluster sizes computed by the last call to `computeProperties()`.

**Returns** sizes of each cluster

**Return type** `numpy.ndarray`, shape=( $N_{clusters}$ ), dtype= `numpy.uint32`

**getNumClusters**(*self*)

Count the number of clusters found in the last call to `computeProperties()`

**Returns** number of clusters

**Return type** `int`

**num\_clusters**

Returns the number of clusters.

### 1.3.4 Density Module

The density module contains functions which deal with the density of the system.

#### Correlation Functions

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

Computes the pairwise correlation function  $\langle p * q \rangle(r)$  between two sets of points with associated values  $p$  and  $q$ .

Two sets of points and two sets of real values associated with those points are given. Computing the correlation function results in an array of the expected (average) product of all values at a given radial distance.

The values of  $r$  to compute the correlation function at are controlled by the *rmax* and *dr* parameters to the constructor. *rmax* determines the maximum  $r$  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.

---

Self-correlation: It is often the case that we wish to compute the correlation function of a set of points with itself. If given the same arrays for both points and `ref_points`, we omit accumulating the self-correlation value in the first bin.

Module author: Matthew Spellings <[mspell@umich.edu](mailto:mspell@umich.edu)>

#### Parameters

- **r\_max** (`float`) – distance over which to calculate
- **dr** (`float`) – bin size

**R**

Bin centers.

**RDF**

Returns the radial distribution function.

**Returns** expected (average) product of all values at a given radial distance

**Return type** `numpy.ndarray`, shape=( $N_{bins}$ ), dtype= `numpy.float64`

**accumulate** (*self*, *box*, *ref\_points*, *refValues*, *points*, *values*, *nlist*=None)

Calculates the correlation function and adds to the current histogram.

**Parameters**

- **box** (`freud.box.Box`) – simulation box
- **ref\_points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – reference points to calculate the local density
- **refValues** (`numpy.ndarray`, shape=( $N_{particles}$ ), dtype= `numpy.float64`) – values to use in computation
- **points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the local density
- **values** (`numpy.ndarray`, shape=( $N_{particles}$ ), dtype= `numpy.float64`) – values to use in computation
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *refValues*, *points*, *values*, *nlist*=None)

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

**Parameters**

- **box** (`freud.box.Box`) – simulation box
- **ref\_points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – reference points to calculate the local density
- **refValues** (`numpy.ndarray`, shape=( $N_{particles}$ ), dtype= `numpy.float64`) – values to use in computation
- **points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the local density
- **values** (`numpy.ndarray`, shape=( $N_{particles}$ ), dtype= `numpy.float64`) – values to use in computation
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**counts**

The counts.

**getBox** (*self*)

Get the box used in the calculation

**Returns** `freud.Box`

**Return type** `freud.box.Box`

**getCounts** (*self*)

**Returns** counts of each histogram bin

**Return type** `numpy.ndarray`, shape=( $N_{bins}$ ), dtype= `numpy.int32`

**getR** (*self*)

**Returns** values of bin centers

**Return type** `numpy.ndarray`, shape=( $N_{bins}$ ), dtype= `numpy.float32`

**getRDF** (*self*)

Returns the radial distribution function.

**Returns** expected (average) product of all values at a given radial distance

**Return type** `numpy.ndarray`, shape=( $N_{bins}$ ), dtype= `numpy.float64`

**reduceCorrelationFunction** (*self*)

Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.density.FloatCF.getRDF()`, `freud.density.FloatCF.getCounts()`.

**resetCorrelationFunction** (*self*)

Resets the values of the correlation function histogram in memory

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

Computes the pairwise correlation function  $\langle p * q \rangle (r)$  between two sets of points with associated values  $p$  and  $q$ .

Two sets of points and two sets of complex values associated with those points are given. Computing the correlation function results in an array of the expected (average) product of all values at a given radial distance.

The values of  $r$  to compute the correlation function at are controlled by the *rmax* and *dr* parameters to the constructor. *rmax* determines the maximum  $r$  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.

---

Self-correlation: It is often the case that we wish to compute the correlation function of a set of points with itself. If given the same arrays for both points and `ref_points`, we omit accumulating the self-correlation value in the first bin.

Module author: Matthew Spellings <[mspellings@umich.edu](mailto:mspellings@umich.edu)>

#### Parameters

- **r\_max** (*float*) – distance over which to calculate
- **dr** (*float*) – bin size

**R**

The value of bin centers.

**RDF**

The RDF.

**accumulate** (*self*, *box*, *ref\_points*, *refValues*, *points*, *values*, *nlist=None*)

Calculates the correlation function and adds to the current histogram.

#### Parameters

- **box** (`freud.box.Box`) – simulation box

- **ref\_points** (`numpy.ndarray`, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – reference points to calculate the local density
- **refValues** (`numpy.ndarray`, `shape=( $N_{particles}$ )`, `dtype= numpy.complex128`) – values to use in computation
- **points** (`numpy.ndarray`, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the local density
- **values** (`numpy.ndarray`, `shape=( $N_{particles}$ )`, `dtype= numpy.complex128`) – values to use in computation
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *refValues*, *points*, *values*, *nlist=None*)

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

**Parameters**

- **box** (`freud.box.Box`) – simulation box
- **ref\_points** (`numpy.ndarray`, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – reference points to calculate the local density
- **refValues** (`numpy.ndarray`, `shape=( $N_{particles}$ )`, `dtype= numpy.complex128`) – values to use in computation
- **points** (`numpy.ndarray`, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the local density
- **values** (`numpy.ndarray`, `shape=( $N_{particles}$ )`, `dtype= numpy.complex128`) – values to use in computation
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**counts**

The counts of each histogram.

**getBox** (*self*)

**Returns** freud Box

**Return type** `freud.box.Box()`

**getCounts** (*self*)

**Returns** counts of each histogram bin

**Return type** `numpy.ndarray`, `shape=( $N_{bins}$ )`, `dtype= numpy.int32`

**getR** (*self*)

The value of bin centers.

**Returns** values of bin centers

**Return type** `numpy.ndarray`, `shape=( $N_{bins}$ )`, `dtype= numpy.float32`

**getRDF** (*self*)

The RDF.

**Returns** expected (average) product of all values at a given radial distance

**Return type** `numpy.ndarray`, `shape=( $N_{bins}$ )`, `dtype= numpy.complex128`

**reduceCorrelationFunction** (*self*)

Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.density.ComplexCF.getRDF()`, `freud.density.ComplexCF.getCounts()`.

**resetCorrelationFunction** (*self*)

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.

Module author: Joshua Anderson <[joaander@umich.edu](mailto: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

### • 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)
```

### **box**

Get the box used in the calculation.

**compute** (*self*, *box*, *points*)

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

### Parameters

- **box** (*freud.box.Box*) – simulation box
- **points** (*numpy.ndarray*, `shape=( $N_{particles}$ , 3)`, `dtype= numpy.float32`) – points to calculate the local density

**gaussian\_density**

The image grid with the Gaussian density.

**getBox** (*self*)

**Returns** `freud.Box`

Return type `freud.box.Box`

**getGaussianDensity** (*self*)

Returns Image (grid) with values of Gaussian

Return type `numpy.ndarray`, shape=( $w_x, w_y, w_z$ ), dtype= `numpy.float32`

**resetDensity** (*self*)

Resets the values of GaussianDensity in memory

## 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](mailto: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

### box

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *points*=None, *nlist*=None)

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

### Parameters

- **box** (`freud.box.Box`) – simulation box
- **ref\_points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – reference points to calculate the local density
- **points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – (optional) points to calculate the local density
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

### density

Density array for each particle.

**getBox** (*self*)

**Returns** freud Box

**Return type** `freud.box.Box`

**getDensity** (*self*)

Get the density array for each particle.

**Returns** Density array for each particle

**Return type** `numpy.ndarray`, shape=( $N_{particles}$ ), dtype= `numpy.float32`

**getNumNeighbors** (*self*)

Return the number of neighbors for each particle.

**Returns** Number of neighbors for each particle

**Return type** `numpy.ndarray`, shape=( $N_{particles}$ ), dtype= `numpy.float32`

**num\_neighbors**

Number of neighbors for each particle.

## 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, default 0

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

### R

Values of bin centers.

### RDF

Histogram of RDF values.

**accumulate** (*self*, *box*, *ref\_points*, *points*, *nlist*=None)

Calculates the RDF and adds to the current RDF histogram.

### Parameters

- **box** (`freud.box.Box`) – simulation box

- **ref\_points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – reference points to calculate the local density
- **points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the local density
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *points*, *nlist=None*)

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

**Parameters**

- **box** (`freud.box.Box`) – simulation box
- **ref\_points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – reference points to calculate the local density
- **points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the local density
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**getBox** (*self*)

**Returns** freud Box

**Return type** `freud.box.Box`

**getNr** (*self*)

Get the histogram of cumulative RDF values.

**Returns** histogram of cumulative RDF values

**Return type** `numpy.ndarray`, shape=( $N_{bins}$ , 3), dtype= `numpy.float32`

**getR** (*self*)

Values of the histogram bin centers.

**Returns** values of the histogram bin centers

**Return type** `numpy.ndarray`, shape=( $N_{bins}$ , 3), dtype= `numpy.float32`

**getRDF** (*self*)

Histogram of RDF values.

**Returns** histogram of RDF values

**Return type** `numpy.ndarray`, shape=( $N_{bins}$ , 3), dtype= `numpy.float32`

**n\_r**

Histogram of cumulative RDF values.

**reduceRDF** (*self*)

Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.density.RDF.getRDF()`, `freud.density.RDF.getNr()`.

**resetRDF** (*self*)

Resets the values of RDF in memory

### 1.3.5 Index Module

The index module exposes the 1-dimensional indexer utilized in freud at the C++ level.

At the C++ level, freud utilizes “flat” arrays, i.e. an  $n$ -dimensional array with  $n_i$  elements in each index is represented as a 1-dimensional array with  $\prod_i n_i$  elements.

#### Index2D

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

freud utilizes “flat” arrays at the C++ level i.e. an  $n$ -dimensional array with  $n_i$  elements in each index is represented as a 1-dimensional array with  $\prod_i n_i$  elements.

---

**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](mailto:joaander@umich.edu)>

#### Parameters

- **w** (*unsigned int*) – width of 2D array (number of columns)
- **h** (*unsigned int*) – height of 2D array (number of rows)
- Constructor Calls:

Initialize with all dimensions identical:

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

Initialize with each dimension specified:

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

**\_\_call\_\_** (*self, i, j*)

#### Parameters

- **i** (*unsigned int*) – column index
- **j** (*unsigned int*) – row index

**Returns** 1-dimensional index in flat array

**Return type** unsigned int

**getNumElements** (*self*)

Get the number of elements in the array :return: number of elements in the array :rtype: unsigned int

**num\_elements**

Number of elements in the array.

## Index3D

**class** `freud.index.Index3D(*args)`  
freud-style indexer for flat arrays.

freud utilizes “flat” arrays at the C++ level i.e. an  $n$ -dimensional array with  $n_i$  elements in each index is represented as a 1-dimensional array with  $\prod_i n_i$  elements.

---

**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](mailto: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)
- Constructor Calls:

Initialize with all dimensions identical:

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

Initialize with each dimension specified:

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

**\_\_call\_\_** (*self, i, j, k*)

### Parameters

- **i** (*unsigned int*) – column index
- **j** (*unsigned int*) – row index
- **k** (*unsigned int*) – frame index

**Returns** 1-dimensional index in flat array

**Return type** unsigned int

**getNumElements** (*self*)

Get the number of elements in the array :return: number of elements in the array :rtype: unsigned int

**num\_elements**

Number of elements in the array.

## 1.3.6 Interface Module

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

## InterfaceMeasure

**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

**compute** (*self*, *ref\_points*, *points*, *nlist=None*)

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

### Parameters

- **ref\_points** (*numpy.ndarray*, shape=( $N_{particles}$ , 3), dtype= *numpy.float32*) – one set of particle positions
- **points** (*numpy.ndarray*, shape=( $N_{particles}$ , 3), dtype= *numpy.float32*) – other set of particle positions
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

## 1.3.7 KSpace Module

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

### Meshgrid

freud.kspace.**meshgrid2** (*\*arrs*)

Computes an n-dimensional meshgrid.

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

**Parameters** *arrs* – Arrays to meshgrid

**Returns** tuple of arrays

**Return type** *tuple*

### Structure Factor

Methods for calculating the structure factor of different systems.

**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  $S(\vec{q}) = C_0 \left| \sum_{m=1}^N \exp i\vec{q} \cdot \vec{r}_i \right|^2$ .

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  $q = (qx[a], qy[b], qz[c])$ .

---

**compute** (*points*)

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** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`)  
– 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** `numpy.ndarray`, shape=(X,Y), dtype= `numpy.float32`

**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** `numpy.ndarray`, shape=(X,Y), dtype= `numpy.complex64`

**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.

---

**getPeakDegeneracy** (*cut*)

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** (`cut`)

Get a list of peaks in the structure factor.

**Parameters** `cut` – 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**  $S$ ,  $q^2$

**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.

**add\_ptype** (`name`)

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** (`*args`, `**kwargs`)

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.

**Parameters**

- **position** (`numpy.ndarray`, `shape=(Nparticles, 3)`, `dtype= numpy.float32`) – array of particle positions in nm
- **orientation** (`numpy.ndarray`, `shape=(Nparticles, 4)`, `dtype= numpy.float32`) – array of orientation quaternions
- **kwargs** – additional keyword arguments passed on to form-factor-specific FT calculator

**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** (*name*)

Remove internal data structures associated with ptype *name*.

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

---

**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** (*name*)

Set particle type active.

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

**set\_box** (*boxMatrix*)

Set box matrix.

**Parameters** **boxMatrix** (`numpy.ndarray`, shape=(3, 3), dtype= `numpy.float32`) – unit cell box matrix

**set\_dK** (*dK*)

Set grid spacing in diffraction image.

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

**set\_form\_factor** (*name*, *ff*)

Set scattering form factor.

**Parameters**

- **name** (*str*) – particle type name
- **ff** (*list*) – scattering form factor named in `get_form_factors()`

**set\_inactive** (*name*)

Set particle type inactive.

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

**set\_k** (*k*)

Set angular wave number of plane wave probe.

**Parameters** **k** (*float*) –  $|k_0|$

**set\_ndiv** (*ndiv*)

Set number of grid divisions in diffraction image.

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

**set\_param** (*particle*, *param*, *value*)

Set named parameter for named particle.

**Parameters**

- **particle** (*str*) – particle name
- **param** (*str*) – parameter name

- **value** (*float*) – parameter value

**set\_rq** (*name, position, orientation*)

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** (*numpy.ndarray*, shape=( $N_{particles}$ , 3), dtype= *numpy.float32*) – (N,3) array of particle positions
- **orientation** (*numpy.ndarray*, shape=( $N_{particles}$ , 4), dtype= *numpy.float32*) – (N,4) array of particle quaternions

**set\_scale** (*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** (*name, constructor, args=None*)

Add an FT class to the factory.

#### Parameters

- **name** (*str*) – identifying string to be returned by getFTlist()
- **constructor** (*object*) – 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** (*i*, *args=None*)

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

**Parameters**

- **i** (*int*) – index into list returned by `getFTlist()`
- **args** (*list*) – 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** (*density*)

Get density.

**Returns** density

**Return type** `numpy.complex64`

**get\_parambyname** (*name*)

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** (*K*)

Set  $K$  points to be evaluated.

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

**set\_density** (*density*)

Set density.

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

**set\_parambyname** (*name*, *value*)

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** (*r*, *q*)

Set *r*, *q* values.

**Parameters**

- **r** (`numpy.ndarray`) – *r*
- **q** (`numpy.ndarray`) – *q*

**set\_scale** (*scale*)

Set scale.

**Parameters** **scale** (`float`) – scale

**class** `freud.kspace.FTdelta`

Fourier transform a list of delta functions.

**compute** (*\*args*, *\*\*kwargs*)

Compute FT.

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

**set\_K** (*K*)

Set *K* points to be evaluated.

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

**set\_density** (*density*)

Set density.

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

**set\_rq** (*r*, *q*)

Set *r*, *q* values.

**Parameters**

- **r** (`numpy.ndarray`) – *r*
- **q** (`numpy.ndarray`) – *q*

**set\_scale** (*scale*)

Set scale.

**Parameters** **scale** (`float`) – scale

---

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

---

**class** `freud.kspace.FTsphere`

Fourier transform for sphere.

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

**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** (*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** (*\*args, \*\*kwargs*)

Compute FT.

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

**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** (*K*)

Set *K* points to be evaluated.

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

**set\_density** (*density*)

Set density.

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

**set\_params** (*verts, facets, norms, d, areas, volume*)

Construct list of facet offsets.

**Parameters**

- **verts** (*numpy.ndarray*, *shape*=(*N<sub>verts</sub>*, 3), *dtype*= *numpy.float32*) – list of vertices
- **facets** (*numpy.ndarray*, *shape*=(*N<sub>facets</sub>*, *N<sub>verts</sub>*), *dtype*= *numpy.float32*) – list of facets
- **norms** (*numpy.ndarray*, *shape*=(*N<sub>facets</sub>*, 3), *dtype*= *numpy.float32*) – list of norms
- **d** (*numpy.ndarray*, *shape*=(*N<sub>facets</sub>*), *dtype*= *numpy.float32*) – list of d values
- **areas** (*numpy.ndarray*, *shape*=(*N<sub>facets</sub>*), *dtype*= *numpy.float32*) – list of areas
- **volumes** (*numpy.ndarray*) – list of volumes

**set\_radius** (*radius*)

Set radius of in-sphere.

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

**set\_rq** (*r, q*)

Set *r, q* values.

**Parameters**

- **r** (*numpy.ndarray*) – *r*
- **q** (*numpy.ndarray*) – *q*

**class** `freud.kspace.FTconvexPolyhedron`

Fourier Transform for convex polyhedra.

**Spoly2D** (*i*, *k*)

Calculate Fourier transform of polygon.

**Parameters**

- **i** (*int*) – face index into self.hull simplex list
- **k** (*int*) – angular wave vector at which to calculate  $S(i)$

**Spoly3D** (*k*)

Calculate Fourier transform of polyhedron.

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

**compute\_py** (\*args, \*\*kwargs)

Compute FT.

Calculate  $P = F * S$ :

- $S = \sum_{\alpha} \exp^{-i\mathbf{K} \cdot \mathbf{r}_{\alpha}}$
- 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** (*radius*)

Set radius of in-sphere.

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

## Diffraction Patterns

Methods for calculating diffraction patterns of various systems.

**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.

**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** (*cval*)

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

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

**set\_xy** (*x*, *y*)

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.

**makeSpot** (*cval*)

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

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

**set\_sigma** (*sigma*)

Define Gaussian.

**Parameters** **sigma** (*float*) – width of the Gaussian spot

**set\_xy** (*x*, *y*)

Set *x*, *y* values of spot center.

**Parameters**

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

## Utilities

Classes and methods used by other kspace modules.

**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** (*v*)

Constraint test.

**Parameters** **v** (`numpy.ndarray`, `shape=(3)`, `dtype= numpy.float32`) – 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** (*v*)

Constraint test.

**Parameters** **v** (`numpy.ndarray`, `shape=(3)`, `dtype= numpy.float32`) – vector to test against constraint

`freud.kspace.constrainedLatticePoints` ()

Generate a list of points satisfying a constraint.

**Parameters**

- **v1** (`numpy.ndarray`, `shape=(3)`, `dtype= numpy.float32`) – lattice vector 1 along which to test points
- **v2** (`numpy.ndarray`, `shape=(3)`, `dtype= numpy.float32`) – lattice vector 2 along which to test points

- **v3** (`numpy.ndarray`, `shape=(3)`, `dtype= numpy.float32`) – lattice vector 3 along which to test points
- **constraint** (*Constraint*) – constraint object to test lattice points against

`freud.kspace.reciprocalLattice3D()`

Calculate reciprocal lattice vectors.

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

#### Parameters

- **a1** (`numpy.ndarray`, `shape=(3)`, `dtype= numpy.float32`) – real space lattice vector 1
- **a2** (`numpy.ndarray`, `shape=(3)`, `dtype= numpy.float32`) – real space lattice vector 2
- **a3** (`numpy.ndarray`, `shape=(3)`, `dtype= numpy.float32`) – real space lattice vector 3

**Returns** list of reciprocal lattice vectors

**Return type** `list`

---

**Note:** For unit test,  $\text{dot}(g[i], a[j]) = 2 * \pi * \text{diracDelta}(i, j)$

---

## 1.3.8 Locality Module

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

### NeighborList

**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  $\log(n_{\text{bonds}})$  time using `find_first_index()`.

*Module author: Matthew Spellings <mspellings@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`.

---

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** (*self*, *other=None*)

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

**filter** (*self*, *filt*)

Removes bonds that satisfy a boolean criterion.

**Parameters** **filt** – Boolean-like array of bonds to keep (True means the bond stays)

---

**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** (*self*, *box*, *ref\_points*, *points*, *float rmax*, *float rmin=0*)

Removes bonds that are outside of a given radius range.

**Parameters**

- **ref\_points** (*numpy.ndarray*, *shape=* ( $N_{points}$ , 3), *dtype=* *numpy.float32*) – reference points to use for filtering
- **points** (*numpy.ndarray*, *shape=* ( $N_{points}$ , 3), *dtype=* *numpy.float32*) – target points to use for filtering
- **rmax** (*float*) – maximum bond distance in the resulting neighbor list
- **rmin** (*float*) – minimum bond distance in the resulting neighbor list

---

**Note:** This method modifies this object in-place.

---

**find\_first\_index** (*self*, *unsigned int i*)

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

**from\_arrays** (*type cls*, *Nref*, *Ntarget*, *index\_i*, *index\_j*, *weights=None*)

Create a NeighborList from a set of bond information arrays.

**Parameters**

- **Nref** (*unsigned int*) – Number of reference points (corresponding to *index\_i*)
- **Ntarget** (*unsigned int*) – Number of target points (corresponding to *index\_j*)
- **index\_i** (Array-like of unsigned ints, *length* *num\_bonds*) – Array of integers corresponding to indices in the set of reference points
- **index\_j** (Array-like of unsigned ints, *length* *num\_bonds*) – Array of integers corresponding to indices in the set of target points

- **weights** (Array-like of floats, length `num_bonds`) – Array of per-bond weights (if None is given, use a value of 1 for each weight)

**index\_i**

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**

The target 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()`.

**neighbor\_counts**

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.

**segments**

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.

**weights**

The per-bond weights from the last set of points this object was evaluated with.

**LinkCell**

**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

---

**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)
```

**box**

freud Box.

**compute** (*self*, *box*, *ref\_points*, *points=None*, *exclude\_ii=None*)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= (*N<sub>refpoints</sub>*, 3), dtype= *numpy.float32*) – reference point coordinates
- **points** (*numpy.ndarray*, shape= (*N<sub>points</sub>*, 3), dtype= *numpy.float32*) – point coordinates
- **exclude\_ii** – 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

**computeCellList** (*self*, *box*, *ref\_points*, *points=None*, *exclude\_ii=None*)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= (*N<sub>refpoints</sub>*, 3), dtype= *numpy.float32*) – reference point coordinates
- **points** (*numpy.ndarray*, shape= (*N<sub>points</sub>*, 3), dtype= *numpy.float32*) – point coordinates
- **exclude\_ii** – 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

**getBox** (*self*)

Get the freud Box.

**Returns** freud Box

**Return type** *freud.box.Box*

**getCell** (*self*, *point*)

Returns the index of the cell containing the given point.

**Parameters** **point** (*numpy.ndarray*, shape= (3), dtype= *numpy.float32*) – point coordinates (*x*, *y*, *z*)

**Returns** cell index

**Return type** unsigned int

**getCellNeighbors** (*self*, *cell*)

Returns the neighboring cell indices of the given cell.

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

**Returns** array of cell neighbors

**Return type** *numpy.ndarray*, shape= (*N<sub>neighbors</sub>*), dtype= *numpy.uint32*

**getNumCells** (*self*)

Get the number of cells in this box.

**Returns** the number of cells in this box

**Return type** unsigned int

**itercell** (*self*, *unsigned int cell*)

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

**nlist**

The neighbor list stored by this object, generated by `compute()`.

**num\_cells**

The number of cells in this box.

## NearestNeighbors

**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](mailto: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`

Example:

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

### UINTMAX

Value of C++ `UINTMAX` used to pad the arrays.

### box

freud Box.

**compute** (*self*, *box*, *ref\_points*, *points*, *exclude\_ii=None*)

Update the data structure for the given set of points.

### Parameters

- **box** (*freud.box.Box*) – simulation box

- **ref\_points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – coordinates of reference points
- **points** (`numpy.ndarray`, shape=( $N_{particles}$ , 3), dtype= `numpy.float32`) – coordinates of points
- **exclude\_ii** – 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

**getBox** (*self*)

Get the freud Box.

**Returns** freud Box

**Return type** `freud.box.Box`

**getNRef** (*self*)

Get the number of particles this object found neighbors of.

**Returns** the number of particles this object found neighbors of

**Return type** unsigned int

**getNeighborList** (*self*)

Return the entire neighbor list.

**Returns** Neighbor List

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ,  $N_{neighbors}$ ), dtype= `numpy.uint32`

**getNeighbors** (*self*, unsigned int *i*)

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

**getNumNeighbors** (*self*)

The number of neighbors this object will find.

**Returns** the number of neighbors this object will find

**Return type** unsigned int

**getRMax** (*self*)

Return the current neighbor search distance guess.

**Returns** nearest neighbors search radius

**Return type** float

**getRsqr** (*self*, unsigned int *i*)

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** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**getRsqrList** (*self*)

Return the entire Rsqr values list.

**Returns** Rsqr list

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ,  $N_{neighbors}$ ), dtype= `numpy.float32`

**getUINTMAX** (*self*)

**Returns** value of C++ UINTMAX used to pad the arrays

**Return type** unsigned int

**getWrappedVectors** (*self*)

Return the wrapped vectors for computed neighbors. Array padded with -1 for empty neighbors.

**Returns** wrapped vectors

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**n\_ref**

The number of particles this object found neighbors of.

**nlist**

Returns the neighbor list stored by this object, generated by `compute()`.

**num\_neighbors**

The number of neighbors this object will find.

**r\_max**

Return the current neighbor search distance guess.

**Returns** nearest neighbors search radius

**Return type** `float`

**r\_sq\_list**

Return the entire Rsq values list.

**Returns** Rsq list

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ,  $N_{neighbors}$ ), dtype= `numpy.float32`

**setCutMode** (*self*, *strict\_cut*)

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

**setRMax** (*self*, *float rmax*)

Update the neighbor search distance guess.

**Parameters** `rmax` (*float*) – nearest neighbors search radius

**wrapped\_vectors**

Return the wrapped vectors for computed neighbors. Array padded with -1 for empty neighbors.

### 1.3.9 Order Module

The order module contains functions which compute order parameters for the whole system or individual particles.

## Bond Order

**class** `freud.order.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 ( $\theta$ ) and polar ( $\phi$ ) 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 ( $\theta$ ) and polar ( $\phi$ ) 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 orientation of each particle *i*, through the surface of a unit sphere based on the azimuthal ( $\theta$ ) and polar ( $\phi$ ) 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 ( $\theta$ ) and polar ( $\phi$ ) 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 theta bins
- **n\_bins\_p** (*unsigned int*) – number of phi bins

**accumulate** (*self, box, ref\_points, ref\_orientations, points, orientations, str mode='bod', nlist=None*)

Calculates the correlation function and adds to the current histogram.

### Parameters

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray, shape=( $N_{particles}$ , 3), dtype= numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray, shape=( $N_{particles}$ , 4), dtype= numpy.float32*) – orientations to use in computation
- **points** (*numpy.ndarray, shape=( $N_{particles}$ , 3), dtype= numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray, shape=( $N_{particles}$ , 4), dtype= numpy.float32*) – orientations to use in computation
- **mode** (*str*) – mode to calc bond order. “bod”, “lbod”, “obcd”, and “oocd”
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**bond\_order**

Bond order.

**box**

Box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *mode*='bod', *nlist*=None)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ , 4), dtype= *numpy.float32*) – orientations to use in computation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ , 4), dtype= *numpy.float32*) – orientations to use in computation
- **mode** (*str*) – mode to calc bond order. “bod”, “lbod”, “obcd”, and “oocd”
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**getBondOrder** (*self*)

Get the bond order.

**Returns** bond order

**Return type** *numpy.ndarray*, shape= ( $N_{\phi}$ ,  $N_{\theta}$ ), dtype= *numpy.float32*

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** *freud.box.Box*

**getNBinsPhi** (*self*)

Get the number of bins in the Phi-dimension of histogram.

**Returns**  $N_{\phi}$

**Return type** unsigned int

**getNBinsTheta** (*self*)

Get the number of bins in the Theta-dimension of histogram.

**Returns**  $N_{\theta}$

**Return type** unsigned int

**getPhi** (*self*)

**Returns** values of bin centers for Phi

**Return type** *numpy.ndarray*, shape= ( $N_{\phi}$ ), dtype= *numpy.float32*

**getTheta** (*self*)

**Returns** values of bin centers for Theta

**Return type** *numpy.ndarray*, shape= ( $N_{\theta}$ ), dtype= *numpy.float32*

**reduceBondOrder** (*self*)

Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.order.BondOrder.getBondOrder()`.

**resetBondOrder** (*self*)

resets the values of the bond order in memory

## Order Parameters

Order parameters take bond order data and interpret it in some way to quantify the degree of order in a system. This is often done through computing spherical harmonics of the bond order diagram, which are the spherical analogue of Fourier Transforms.

### Cubic Order Parameter

**class** `freud.order.CubicOrderParameter` (*t\_initial*, *t\_final*, *scale*, *n\_replicates*, *seed*)

Compute the cubic order parameter [Cit1] 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 used

**compute** (*self*, *orientations*)

Calculates the per-particle and global order parameter.

#### Parameters

- **box** (`freud.box.Box`) – simulation box
- **orientations** (`numpy.ndarray`, *shape=* ( $N_{particles}$ , 4), *dtype=* `numpy.float32`) – orientations to calculate the order parameter

**get\_cubic\_order\_parameter** (*self*)

**Returns** Cubatic order parameter

**Return type** `float`

**get\_cubic\_tensor** (*self*)

**Returns** Rank 4 tensor corresponding to each individual particle orientation

**Return type** `numpy.ndarray`, *shape=* (3, 3, 3, 3), *dtype=* `numpy.float32`

**get\_gen\_r4\_tensor** (*self*)

**Returns** Rank 4 tensor corresponding to each individual particle orientation

**Return type** `numpy.ndarray`, *shape=* (3, 3, 3, 3), *dtype=* `numpy.float32`

**get\_global\_tensor** (*self*)

**Returns** Rank 4 tensor corresponding to each individual particle orientation

**Return type** `numpy.ndarray`, shape= (3, 3, 3, 3), dtype= `numpy.float32`

**get\_orientation** (*self*)

**Returns** orientation of global orientation

**Return type** `numpy.ndarray`, shape= (4), dtype= `numpy.float32`

**get\_particle\_op** (*self*)

**Returns** Cubatic order parameter

**Return type** `float`

**get\_particle\_tensor** (*self*)

**Returns** Rank 4 tensor corresponding to each individual particle orientation

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ , 3, 3, 3, 3), dtype= `numpy.float32`

**get\_scale** (*self*)

**Returns** value of scale

**Return type** `float`

**get\_t\_final** (*self*)

**Returns** value of final temperature

**Return type** `float`

**get\_t\_initial** (*self*)

**Returns** value of initial temperature

**Return type** `float`

### Nematic Order Parameter

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

Compute the nematic order parameter for a system of particles.

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

New in version 0.7.0.

**Parameters** *u* (`numpy.ndarray`, shape= (3), dtype= `numpy.float32`) – The nematic director of a single particle in the reference state (without any rotation applied)

**compute** (*self*, *orientations*)

Calculates the per-particle and global order parameter.

**Parameters** *orientations* (`numpy.ndarray`, shape= ( $N_{particles}$ , 4), dtype= `numpy.float32`) – orientations to calculate the order parameter

**get\_director** (*self*)

The director (eigenvector corresponding to the order parameter).

**Returns** The average nematic director

**Return type** `numpy.ndarray`, shape= (3), dtype= `numpy.float32`

**get\_nematic\_order\_parameter** (*self*)

The nematic order parameter.

**Returns** Nematic order parameter

**Return type** `float`

**get\_nematic\_tensor** (*self*)

The nematic Q tensor.

**Returns** 3x3 matrix corresponding to the average particle orientation

**Return type** `numpy.ndarray`, shape= (3,3), dtype= `numpy.float32`

**get\_particle\_tensor** (*self*)

The full per-particle tensor of orientation information.

**Returns** 3x3 matrix corresponding to each individual particle orientation

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ , 3, 3), dtype= `numpy.float32`

## Hexatic Order Parameter

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

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

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

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

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

---

**Note:** 2D: This calculation is defined for 2D systems only. However, particle positions are still required to be passed in as `[x, y, 0]`.

---

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)

### box

Get the box used in the calculation.

**compute** (*self*, *box*, *points*, *nlist*=None)

Calculates the correlation function and adds to the current histogram.

### Parameters

- **box** (`freud.box.Box`) – simulation box
- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**getBox** (*self*)

Get the box used in the calculation.

**Returns** `freud.Box`

**Return type** `freud.box.Box`

**getK**(*self*)

Get the symmetry of the order parameter.

**Returns** *k*

**Return type** unsigned int

**getNP**(*self*)

Get the number of particles.

**Returns**  $N_{particles}$

**Return type** unsigned int

**getPsi**(*self*)

Get the order parameter.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.complex64`

**k**

Symmetry of the order parameter.

**num\_particles**

Get the number of particles.

**psi**

Order parameter.

## Local Descriptors

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

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

*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  $m$

**compute**(*self, box, unsigned int num\_neighbors, points\_ref, points=None, orientations=None, mode='neighborhood', nlist=None*)

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

### Parameters

- **num\_neighbors** – Number of neighbors to compute with
- **points\_ref** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – source points to calculate the order parameter
- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – destination points to calculate the order parameter
- **orientations** (`numpy.ndarray`, shape= ( $N_{particles}$ , 4), dtype= `numpy.float32` or `None`) – Orientation of each reference point

- **mode** (*str*) – 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
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**computeNList** (*self, box, points\_ref, points=None*)

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

**Parameters**

- **num\_neighbors** – Number of neighbors to compute with
- **points\_ref** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – source points to calculate the order parameter
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – destination points to calculate the order parameter

**getLMax** (*self*)

Get the maximum spherical harmonic  $l$  to calculate for.

**Returns**  $l$

**Return type** unsigned int

**getNP** (*self*)

Get the number of particles.

**Returns**  $N_{particles}$

**Return type** unsigned int

**getNSphs** (*self*)

Get the number of neighbors.

**Returns**  $N_{neighbors}$

**Return type** unsigned int

**getRMax** (*self*)

Get the cutoff radius.

**Returns**  $r$

**Return type** float

**getSph** (*self*)

Get a reference to the last computed spherical harmonic array.

**Returns** order parameter

**Return type** *numpy.ndarray*, shape= ( $N_{bonds}$ , SphWidth), dtype= *numpy.complex64*

**l\_max**

Get the maximum spherical harmonic  $l$  to calculate for.

**num\_neighbors**

Get the number of neighbors.

**num\_particles**

Get the number of particles.

**r\_max**

Get the cutoff radius.

**sph**

A reference to the last computed spherical harmonic array.

### Translational Order Parameter

**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)

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *points*, *nlist*=None)

Calculates the local descriptors.

#### Parameters

- **box** (*freud.box.Box*) – simulation box
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the order parameter
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**d\_r**

Get a reference to the last computed spherical harmonic array.

**getBox** (*self*)

Get the box used in the calculation.

**Returns** *freud.Box*

**Return type** *freud.box.Box*

**getDr** (*self*)

Get a reference to the last computed spherical harmonic array.

**Returns** order parameter

**Return type** *numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.complex64*

**getNP** (*self*)

Get the number of particles.

**Returns**  $N_{particles}$

**Return type** unsigned int

**num\_particles**

Get the number of particles.

## Local $Q_l$

```
class freud.order.LocalQl(box, rmax, l, rmin)
    LocalQl(box, rmax, l, rmin=0)
```

Compute the local Steinhardt rotationally invariant  $Q_l$  [Cit4] 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

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

This is then combined in a rotationally invariant fashion to remove local orientational order as follows:  $Q_l(i) =$

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

For more details see PJ Steinhardt (1983) (DOI: 10.1103/PhysRevB.28.784)

Added first/second shell combined average  $Q_l$  order parameter for a set of points:

- Variation of the Steinhardt  $Q_l$  order parameter
- For a particle  $i$ , we calculate the average  $Q_l$  by summing the spherical harmonics between particle  $i$  and its neighbors  $j$  and the neighbors  $k$  of neighbor  $j$  in a local region

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

### Parameters

- **box** (*freud.box.Box*) – simulation box
- **rmax** (*float*) – Cutoff radius for the local order parameter. Values near first minima 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

### Ql

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

### ave\_Ql

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

### ave\_norm\_Ql

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

### box

Get the box used in the calculation.

```
compute (self, points, nlist=None)
```

Compute the local rotationally invariant  $Q_l$  order parameter.

### Parameters

- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the order parameter
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**computeAve** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeAveNorm** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeNorm** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**getAveQl** (*self*)

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** `freud.box.Box`

**getNP** (*self*)

Get the number of particles.

**Returns**  $N_p$

**Return type** unsigned int

**getQl** (*self*)

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**getQlAveNorm** (*self*)

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**getQlNorm** (*self*)

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**norm\_Ql**

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

**num\_particles**

Get the number of particles.

**setBox** (*self*, *box*)

Reset the simulation box.

**Parameters** **box** (`freud.box.Box`) – simulation box

## Nearest Neighbors Local $Q_l$

**class** `freud.order.LocalQlNear` (*box*, *rmax*, *l*, *kn*)

`LocalQlNear(box, rmax, l, kn=12)`

Compute the local Steinhardt rotationally invariant  $Q_l$  order parameter [Cit4] 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

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

This is then combined in a rotationally invariant fashion to remove local orientational order as follows:  $Q_l(i) =$

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

For more details see PJ Steinhardt (1983) (DOI: 10.1103/PhysRevB.28.784)

Added first/second shell combined average  $Q_l$  order parameter for a set of points:

- Variation of the Steinhardt  $Q_l$  order parameter
- For a particle  $i$ , we calculate the average  $Q_l$  by summing the spherical harmonics between particle  $i$  and its neighbors  $j$  and the neighbors  $k$  of neighbor  $j$  in a local region

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

### Parameters

- **box** (`freud.box.Box`) – simulation box
- **rmax** (`float`) – Cutoff radius for the local order parameter. Values near first minima 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

**compute** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

#### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeAve** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

#### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeAveNorm** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

#### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeNorm** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

#### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

## Local $W_l$

**class** `freud.order.LocalWl` (*box*, *rmax*, *l*)

`LocalWl(box, rmax, l)`

Compute the local Steinhardt rotationally invariant  $W_l$  order parameter [Cit4] for a set of points.

Implements the local rotationally invariant  $W_l$  order parameter described by Steinhardt that can aid in distinguishing between FCC, HCP, and BCC.

For more details see PJ Steinhardt (1983) (DOI: 10.1103/PhysRevB.28.784)

Added first/second shell combined average  $W_l$  order parameter for a set of points:

- Variation of the Steinhardt  $W_l$  order parameter
- For a particle *i*, we calculate the average  $W_l$  by summing the spherical harmonics between particle *i* and its neighbors *j* and the neighbors *k* of neighbor *j* in a local region

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

**Parameters**

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

**Ql**

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

**Wl**

Get a reference to the last computed  $W_l$  for each particle. Returns NaN instead of  $W_l$  for particles with no neighbors.

**ave\_Wl**

Get a reference to the last computed  $W_l$  for each particle. Returns NaN instead of  $W_l$  for particles with no neighbors.

**ave\_norm\_Wl**

Get a reference to the last computed  $W_l$  for each particle. Returns NaN instead of  $W_l$  for particles with no neighbors.

**box**

Get the box used in the calculation.

**compute** (*self, points, nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the order parameter
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**computeAve** (*self, points, nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the order parameter
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**computeAveNorm** (*self, points, nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the order parameter
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**computeNorm** (*self, points, nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**getAveWl** (*self*)

Get a reference to the last computed  $W_l$  for each particle. Returns NaN instead of  $W_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** `freud.box.Box`

**getNP** (*self*)

Get the number of particles.

**Returns**  $N_{particles}$

**Return type** unsigned int

**getQl** (*self*)

Get a reference to the last computed  $Q_l$  for each particle. Returns NaN instead of  $Q_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**getWl** (*self*)

Get a reference to the last computed  $W_l$  for each particle. Returns NaN instead of  $W_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.complex64`

**getWlAveNorm** (*self*)

Get a reference to the last computed  $W_l$  for each particle. Returns NaN instead of  $W_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**getWlNorm** (*self*)

Get a reference to the last computed  $W_l$  for each particle. Returns NaN instead of  $W_l$  for particles with no neighbors.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**norm\_Wl**

Get a reference to the last computed  $W_l$  for each particle. Returns NaN instead of  $W_l$  for particles with no neighbors.

**num\_particles**

Get the number of particles.

**setBox** (*self*, *box*)

Reset the simulation box.

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

## Nearest Neighbors Local $W_l$

**class** *freud.order.LocalWlNear* (*box*, *rmax*, *l*, *kn*)

*LocalWlNear*(*box*, *rmax*, *l*, *kn*=12)

Compute the local Steinhardt rotationally invariant  $W_l$  order parameter [Cit4] for a set of points.

Implements the local rotationally invariant  $W_l$  order parameter described by Steinhardt that can aid in distinguishing between FCC, HCP, and BCC.

For more details see PJ Steinhardt (1983) (DOI: 10.1103/PhysRevB.28.784)

Added first/second shell combined average  $W_l$  order parameter for a set of points:

- Variation of the Steinhardt  $W_l$  order parameter
- For a particle *i*, we calculate the average  $W_l$  by summing the spherical harmonics between particle *i* and its neighbors *j* and the neighbors *k* of neighbor *j* in a local region

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

### Parameters

- **box** (*freud.box.Box*) – simulation box
- **rmax** (*float*) – Cutoff radius for the local order parameter. Values near first minima 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

**compute** (*self*, *points*, *nlist*=None)

Compute the local rotationally invariant  $Q_l$  order parameter.

### Parameters

- **points** (*numpy.ndarray*, shape= (*N<sub>particles</sub>*, 3), dtype= *numpy.float32*) – points to calculate the order parameter
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**computeAve** (*self*, *points*, *nlist*=None)

Compute the local rotationally invariant  $Q_l$  order parameter.

### Parameters

- **points** (*numpy.ndarray*, shape= (*N<sub>particles</sub>*, 3), dtype= *numpy.float32*) – points to calculate the order parameter
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**computeAveNorm** (*self*, *points*, *nlist*=None)

Compute the local rotationally invariant  $Q_l$  order parameter.

### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeNorm** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

#### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

### Solid-Liquid Order Parameter

**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](mailto:newmanrs@umich.edu)>

#### Parameters

- **box** (`freud.box.Box`) – simulation box
- **rmax** (`float`) – Cutoff radius for the local order parameter. Values near first minima 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 generally good for FCC or BCC structures)
- **l** (`unsigned int`) – Choose spherical harmonic  $Q_l$ . Must be positive and even.

**Ql\_dot\_ij**

Get a reference to the number of connections per particle.

**Ql\_mi**

Get a reference to the last computed  $Q_{lmi}$  for each particle.

**box**

Get the box used in the calculation.

**cluster\_sizes**

Return the sizes of all clusters.

**clusters**

Get a reference to the last computed set of solid-like cluster indices for each particle.

**compute** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

#### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeSolLiqNoNorm** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeSolLiqVariant** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** `freud.box.Box`

**getClusterSizes** (*self*)

Return the sizes of all clusters.

**Returns** largest cluster size

**Return type** `numpy.ndarray`, shape= ( $N_{clusters}$ ), dtype= `numpy.uint32`

**getClusters** (*self*)

Get a reference to the last computed set of solid-like cluster indices for each particle.

**Returns** clusters

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.uint32`

**getLargestClusterSize** (*self*)

Returns the largest cluster size. Must call a compute method first.

**Returns** largest cluster size

**Return type** unsigned int

**getNP** (*self*)

Get the number of particles.

**Returns** np

**Return type** unsigned int

**getNumberOfConnections** (*self*)

Get a reference to the number of connections per particle.

**Returns** clusters

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.uint32`

**getQldot\_ij** (*self*)

Get a reference to the qldot\_ij values.

**Returns** largest cluster size

**Return type** `numpy.ndarray`, shape= ( $N_{clusters}$ ), dtype= `numpy.complex64`

**getQlmi** (*self*)

Get a reference to the last computed  $Q_{lmi}$  for each particle.

**Returns** order parameter

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.complex64`

**largest\_cluster\_size**

Returns the largest cluster size. Must call a compute method first.

**num\_connections**

Get a reference to the number of connections per particle.

**num\_particles**

Get the number of particles.

**setBox** (*self*, *box*)

Reset the simulation box.

**Parameters** **box** (`freud.box.Box`) – simulation box

**setClusteringRadius** (*self*, *rcutCluster*)

Reset the clustering radius.

**Parameters** **rcutCluster** (`float`) – radius for the cluster finding

## Nearest Neighbors Solid-Liquid Order Parameter

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

`SolLiqNear(box, rmax, Qthreshold, Sthreshold, l, kn=12)`

Computes dot products of  $Q_{lm}$  between particles and uses these for clustering.

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

### Parameters

- **box** (`freud.box.Box`) – simulation box
- **rmax** (`float`) – Cutoff radius for the local order parameter. Values near first minima 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 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

**compute** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, `shape= (Nparticles, 3)`, `dtype= numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeSolLiqNoNorm** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, `shape= (Nparticles, 3)`, `dtype= numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**computeSolLiqVariant** (*self*, *points*, *nlist=None*)

Compute the local rotationally invariant  $Q_l$  order parameter.

**Parameters**

- **points** (`numpy.ndarray`, `shape= (Nparticles, 3)`, `dtype= numpy.float32`) – points to calculate the order parameter
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

## Environment Matching

**class** `freud.order.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 first minimum of the RDF are recommended.
- **k** (`unsigned int`) – Number of nearest neighbors taken to define the local environment of any given particle.

**cluster** (*self*, *points*, *threshold*, *hard\_r=False*, *registration=False*, *global\_search=False*, *env\_nlist=None*, *nlist=None*)

Determine clusters of particles with matching environments.

**Parameters**

- **points** (`numpy.ndarray`, `shape= (Nparticles, 3)`, `dtype= numpy.float32`) – particle positions
- **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.
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find neighbors of every particle, to compare environments
- **env\_nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find the environment of every particle

**getClusters** (*self*)

Get a reference to the particles, indexed into clusters according to their matching local environments

**Returns** clusters

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.uint32`

**getEnvironment** (*self*, *i*)

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

**Parameters** *i* (*unsigned int*) – environment index

**Returns** the array of vectors

**Return type** `numpy.ndarray`, shape= ( $N_{neighbors}$ , 3), dtype= `numpy.float32`

**getNP** (*self*)

Get the number of particles.

**Returns**  $N_{particles}$

**Return type** unsigned int

**getNumClusters** (*self*)

Get the number of clusters.

**Returns**  $N_{clusters}$

**Return type** unsigned int

**getTotEnvironment** (*self*)

Returns the entire  $m\_Np$  by  $m\_maxk$  by 3 matrix of all environments for all particles.

**Returns** the array of vectors

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ,  $N_{neighbors}$ , 3), dtype= `numpy.float32`

**isSimilar** (*self*, *refPoints1*, *refPoints2*, *threshold*, *registration=False*)

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

**Parameters**

- **refPoints1** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – vectors that make up motif 1
- **refPoints2** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – 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*) – 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.

**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[(`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`), map[int, int]]

**matchMotif** (*self*, *points*, *refPoints*, *threshold*, *registration=False*, *nlist=None*)

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

#### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – particle positions
- **refPoints** (`numpy.ndarray`, shape= ( $N_{neighbors}$ , 3), dtype= `numpy.float32`) – 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*) – 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.
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**minRMSDMotif** (*self*, *points*, *refPoints*, *registration=False*, *nlist=None*)

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

#### Parameters

- **points** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – particle positions
- **refPoints** (`numpy.ndarray`, shape= ( $N_{neighbors}$ , 3), dtype= `numpy.float32`) – vectors that make up the motif against which we are matching
- **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.
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**Returns** vector of minimal RMSD values, one value per particle.

**Return type** `numpy.ndarray`, shape= ( $N_{particles}$ ), dtype= `numpy.float32`

**minimizeRMSD** (*self*, *refPoints1*, *refPoints2*, *registration=False*)

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

#### Parameters

- **refPoints1** (`numpy.ndarray`, shape= ( $N_{particles}$ , 3), dtype= `numpy.float32`) – vectors that make up motif 1

- **refPoints2** (`numpy.ndarray`, `shape= (Nparticles, 3)`, `dtype= numpy.float32`)  
– vectors that make up motif 2
- **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

**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, ( numpy.ndarray, shape= (Nparticles, 3), dtype= numpy.float32), map[int, int]]`

**num\_clusters**

Get the number of clusters.

**num\_particles**

Get the number of particles.

**setBox** (*self*, *box*)

Reset the simulation box.

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

**tot\_environment**

Returns the entire `m_Np` by `m_maxk` by 3 matrix of all environments for all particles.

## Pairing

---

**Note:** This module is deprecated and is replaced with [Bond Module](#).

---

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

Compute pairs for the system of particles.

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

**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

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *points*, *orientations*, *compOrientations*, *nlist=None*)

Calculates the correlation function and adds to the current histogram.

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **points** (`numpy.ndarray`, `shape= (Nparticles, 3)`, `dtype= numpy.float32`) – reference points to calculate the local density
- **orientations** (`numpy.ndarray`, `shape= (Nparticles)`, `dtype= numpy.float32`) – orientations to use in computation

- **compOrientations** (`numpy.ndarray`, `shape= (Nparticles)`, `dtype= numpy.float32`) – possible orientations to check for bonds
- **nlist** (`freud.locality.NeighborList`) – `freud.locality.NeighborList` object to use to find bonds

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** `freud.box.Box`

**getMatch** (*self*)

Get the match.

**Returns** match

**Return type** `numpy.ndarray`, `shape= (Nparticles)`, `dtype= numpy.uint32`

**getPair** (*self*)

Get the pair.

**Returns** pair

**Return type** `numpy.ndarray`, `shape= (Nparticles)`, `dtype= numpy.uint32`

**match**

Match.

**pair**

Pair.

### 1.3.10 Parallel Module

The `freud.parallel` module tries to use all available threads for parallelization unless directed otherwise, with one exception. On the *flux* and *nyx* clusters, freud will only use one thread unless directed otherwise.

`parallel.setNumThreads` (*nthreads=None*)

Set the number of threads for parallel computation.

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

**Parameters** **nthreads** (*int or None*) – number of threads to use. If *None* (default), use all threads available

**class** `freud.parallel.NumThreads` (*N=None*)

Context manager for managing the number of threads to use.

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

**Parameters** **N** (*int or None*) – Number of threads to use in this context. Defaults to *None*, which will use all available threads.

### 1.3.11 PMFT Module

The PMFT Module allows for the calculation of the Potential of Mean Force and Torque (PMFT) [Cit2] [Cit3] in a number of different coordinate systems.

---

**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:  $\theta$ ):**
    - $X, Y$
    - $X, Y, \theta_2$
    - $R, \theta_1, \theta_2$
  - **3D particle coordinates:  $X, Y, Z$**
- 

**Coordinate System:**  $x, y, \theta_2$

**class** `freud.pmft.PMFTXYT` ( $x\_max, y\_max, n\_x, n\_y, n\_t$ )

Computes the PMFT [Cit2] [Cit3] for a given set of points.

A given set of reference points is given around which the PCF is computed and averaged in a sea of data points. Computing the PCF results in a PCF array listing the value of the PCF at each given  $x, y, \theta$  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.  $x\_max, y\_max$  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>

#### 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  $t$

#### PCF

Get the positional correlation function.

#### PMFT

Get the potential of mean force and torque.

#### T

Get the array of  $t$ -values for the PCF histogram.

#### X

Get the array of  $x$ -values for the PCF histogram.

#### Y

Get the array of  $y$ -values for the PCF histogram.

**accumulate** (*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *nlist*=None)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – angles of reference points to use in calculation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – angles of particles to use in calculation
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**bin\_counts**

Get the raw bin counts.

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *nlist*=None)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – angles of reference points to use in calculation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – angles of particles to use in calculation
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**getBinCounts** (*self*)

Get the raw bin counts.

**Returns** Bin Counts

**Return type** *numpy.ndarray*, shape= ( $N_{\theta}$ ,  $N_y$ ,  $N_x$ ), dtype= *numpy.uint32*

**getBox** (*self*)

Get the box used in the calculation.

**Returns** *freud.Box*

**Return type** *freud.box.Box*

**getJacobian** (*self*)

Get the Jacobian used in the PMFT.

**Returns** Inverse Jacobian

**Return type** float

**getNBinsT**(*self*)

Get the number of bins in the t-dimension of histogram.

**Returns**  $N_\theta$

**Return type** unsigned int

**getNBinsX**(*self*)

Get the number of bins in the x-dimension of histogram.

**Returns**  $N_x$

**Return type** unsigned int

**getNBinsY**(*self*)

Get the number of bins in the y-dimension of histogram.

**Returns**  $N_y$

**Return type** unsigned int

**getPCF**(*self*)

Get the positional correlation function.

**Returns** PCF

**Return type** `numpy.ndarray`, shape= ( $N_\theta, N_y, N_x$ ), dtype= `numpy.float32`

**getPMFT**(*self*)

Get the potential of mean force and torque.

**Returns** PMFT

**Return type** `numpy.ndarray`, shape= (matches PCF), dtype= `numpy.float32`

**getRCut**(*self*)

Get the `r_cut` value used in the cell list.

**Returns** `r_cut`

**Return type** float

**getT**(*self*)

Get the array of t-values for the PCF histogram.

**Returns** bin centers of t-dimension of histogram

**Return type** `numpy.ndarray`, shape= ( $N_\theta$ ), dtype= `numpy.float32`

**getX**(*self*)

Get the array of x-values for the PCF histogram.

**Returns** bin centers of x-dimension of histogram

**Return type** `numpy.ndarray`, shape= ( $N_x$ ), dtype= `numpy.float32`

**getY**(*self*)

Get the array of y-values for the PCF histogram.

**Returns** bin centers of y-dimension of histogram

**Return type** `numpy.ndarray`, shape= ( $N_y$ ), dtype= `numpy.float32`

**jacobian**

Get the Jacobian used in the PMFT.

**n\_bins\_T**

Get the number of bins in the T-dimension of histogram.

**n\_bins\_X**

Get the number of bins in the x-dimension of histogram.

**n\_bins\_Y**

Get the number of bins in the y-dimension of histogram.

**r\_cut**

Get the r\_cut value used in the cell list.

**reducePCF** (*self*)

Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.pmft.PMFT.PCF()`.

**resetPCF** (*self*)

Resets the values of the PCF histograms in memory.

**Coordinate System:** *x, y*

**class** `freud.pmft.PMFTXY2D` (*x\_max, y\_max, n\_x, n\_y*)

Computes the PMFT [Cit2] [Cit3] for a given set of points.

A given set of reference points is given around which the PCF is computed and averaged in a sea of data points. Computing the PCF results in a PCF array listing the value of the PCF at each given *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. *x\_max* and *y\_max* 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>*

**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*

**PCF**

Get the positional correlation function.

**PMFT**

Get the potential of mean force and torque.

**X**

Get the array of *x*-values for the PCF histogram.

**Y**

Get the array of *y*-values for the PCF histogram.

**accumulate** (*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *nlist*=None)

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

#### Parameters

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – orientations of reference points to use in calculation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – orientations of particles to use in calculation
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**bin\_counts**

Get the raw bin counts.

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *nlist*=None)

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

#### Parameters

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – orientations of reference points to use in calculation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – orientations of particles to use in calculation
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**getBinCounts** (*self*)

Get the raw bin counts (non-normalized).

**Returns** Bin Counts

**Return type** *numpy.ndarray*, shape= ( $N_y$ ,  $N_x$ ), dtype= *numpy.uint32*

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** *freud.box.Box*

**getJacobian** (*self*)

Get the Jacobian.

**Returns** Jacobian

**Return type** float

**getNBinsX** (*self*)

Get the number of bins in the x-dimension of histogram.

**Returns**  $N_x$

**Return type** unsigned int

**getNBinsY** (*self*)

Get the number of bins in the y-dimension of histogram.

**Returns**  $N_y$

**Return type** unsigned int

**getPCF** (*self*)

Get the positional correlation function.

**Returns** PCF

**Return type** `numpy.ndarray`, shape= ( $N_y, N_x$ ), dtype= `numpy.float32`

**getPMFT** (*self*)

Get the potential of mean force and torque.

**Returns** PMFT

**Return type** `numpy.ndarray`, shape= (matches PCF), dtype= `numpy.float32`

**getRCut** (*self*)

Get the `r_cut` value used in the cell list.

**Returns** `r_cut`

**Return type** float

**getX** (*self*)

Get the array of x-values for the PCF histogram.

**Returns** bin centers of x-dimension of histogram

**Return type** `numpy.ndarray`, shape= ( $N_x$ ), dtype= `numpy.float32`

**getY** (*self*)

Get the array of y-values for the PCF histogram.

**Returns** bin centers of y-dimension of histogram

**Return type** `numpy.ndarray`, shape= ( $N_y$ ), dtype= `numpy.float32`

**jacobian**

Get the Jacobian used in the PMFT.

**n\_bins\_X**

Get the number of bins in the x-dimension of histogram.

**n\_bins\_Y**

Get the number of bins in the y-dimension of histogram.

**r\_cut**

Get the `r_cut` value used in the cell list.

**reducePCF** (*self*)

Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.pmft.PMFT.PCF()`.

**resetPCF** (*self*)

Resets the values of the PCF histograms in memory.

**Coordinate System:**  $r, \theta_1, \theta_2$ **class** `freud.pmft.PMFTTR12` (*r\_max, n\_r, n\_t1, n\_t2*)

Computes the PMFT [Cit2] [Cit3] for a given set of points.

A given set of reference points is given around which the PCF is computed and averaged in a sea of data points. Computing the PCF results in a PCF array listing the value of the PCF at each given  $r, \theta_1, \theta_2$  listed in the *r*, *t1*, and *t2* arrays.

The values of *r*, *t1*, *t2* to compute the PCF at are controlled by *r\_max* and *nbins\_r*, *nbins\_t1*, *nbins\_t2* parameters to the constructor. *r\_max* determines the minimum/maximum  $r$  ( $\min(\theta_1) = \min(\theta_2) = 0$ ,  $\max(\theta_1) = \max(\theta_2) = 2\pi$ ) at which to compute the PCF and *nbins\_r*, *nbins\_t1*, *nbins\_t2* is the number of bins in *r*, *t1*, *t2*.

---

**Note:** 2D: `freud.pmft.PMFTTR12` 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>

**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*

**PCF**

Get the positional correlation function.

**PMFT**

Get the potential of mean force and torque.

**R**

Get the array of *r*-values for the PCF histogram.

**Returns** bin centers of *r*-dimension of histogram

**Return type** `numpy.ndarray`, shape= ( $N_r$ ), dtype= `numpy.float32`

**T1**

Get the array of *T1*-values for the PCF histogram.

**T2**

Get the array of *T2*-values for the PCF histogram.

**Returns** bin centers of *T2*-dimension of histogram

**Return type** `numpy.ndarray`, shape= ( $N_{\theta_2}$ ), dtype= `numpy.float32`

**accumulate** (*self, box, ref\_points, ref\_orientations, points, orientations, nlist=None*)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – angles of reference points to use in calculation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – angles of particles to use in calculation
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**bin\_counts**

Get the raw bin counts.

**box**

Get the box used in the calculation.

**compute** (*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *nlist=None*)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – angles of reference points to use in calculation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ ), dtype= *numpy.float32*) – angles of particles to use in calculation
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**getBinCounts** (*self*)

Get the raw bin counts.

**Returns** Bin Counts

**Return type** *numpy.ndarray*, shape= ( $N_r$ ,  $N_{\theta 2}$ ,  $N_{\theta 1}$ ), dtype= *numpy.uint32*

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** *freud.box.Box*

**getInverseJacobian** (*self*)

Get the inverse Jacobian used in the PMFT.

**Returns** Inverse Jacobian

**Return type** `numpy.ndarray`, shape=  $(N_r, N_{\theta 2}, N_{\theta 1})$ , dtype= `numpy.float32`

**getNBinsR** (*self*)

Get the number of bins in the r-dimension of histogram.

**Returns**  $N_r$

**Return type** unsigned int

**getNBinsT1** (*self*)

Get the number of bins in the T1-dimension of histogram.

**Returns**  $N_{\theta 1}$

**Return type** unsigned int

**getNBinsT2** (*self*)

Get the number of bins in the T2-dimension of histogram.

**Returns**  $N_{\theta 2}$

**Return type** unsigned int

**getPCF** (*self*)

Get the positional correlation function.

**Returns** PCF

**Return type** `numpy.ndarray`, shape=  $(N_r, N_{\theta 2}, N_{\theta 1})$ , dtype= `numpy.float32`

**getPMFT** (*self*)

Get the potential of mean force and torque.

**Returns** PMFT

**Return type** `numpy.ndarray`, shape= (matches PCF), dtype= `numpy.float32`

**getR** (*self*)

Get the array of r-values for the PCF histogram.

**Returns** bin centers of r-dimension of histogram

**Return type** `numpy.ndarray`, shape=  $(N_r)$ , dtype= `numpy.float32`

**getRCut** (*self*)

Get the `r_cut` value used in the cell list.

**Returns** `r_cut`

**Return type** `float`

**getT1** (*self*)

Get the array of T1-values for the PCF histogram.

**Returns** bin centers of T1-dimension of histogram

**Return type** `numpy.ndarray`, shape=  $(N_{\theta 1})$ , dtype= `numpy.float32`

**getT2** (*self*)

Get the array of T2-values for the PCF histogram.

**Returns** bin centers of T2-dimension of histogram

**Return type** `numpy.ndarray`, shape=  $(N_{\theta 2})$ , dtype= `numpy.float32`

**inverse\_jacobian**

Get the inverse Jacobian used in the PMFT.

**n\_bins\_T1**

Get the number of bins in the T1-dimension of histogram.

**n\_bins\_T2**

Get the number of bins in the T2-dimension of histogram.

**n\_bins\_r**

Get the number of bins in the r-dimension of histogram.

**r\_cut**

Get the r\_cut value used in the cell list.

**reducePCF** (*self*)

Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.pmft.PMFT.PCF()`.

**resetPCF** (*self*)

Resets the values of the PCF histograms in memory.

### Coordinate System: $x, y, z$

**class** `freud.pmft.PMFTXYZ` ( $x\_max, y\_max, z\_max, n\_x, n\_y, n\_z$ )

Computes the PMFT [Cit2] [Cit3] for a given set of points.

A given set of reference points is given around which the PCF is computed and averaged in a sea of data points. Computing the PCF results in a PCF array listing the value of the PCF at each given  $x, y, z$ , 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.  $x\_max, y\_max$ , and  $z\_max$  determine the minimum/maximum distance at which to compute the PCF and  $n\_x, n\_y, n\_z$  is 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>

#### 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

**PCF**

Get the positional correlation function.

**PMFT**

Get the potential of mean force and torque.

**x**

Get the array of x-values for the PCF histogram.

**Y**

Get the array of y-values for the PCF histogram.

**Z**

Get the array of z-values for the PCF histogram.

**accumulate**(*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *face\_orientations*=None, *nlist*=None)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ , 4), dtype= *numpy.float32*) – orientations of reference points to use in calculation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ , 4), dtype= *numpy.float32*) – orientations of particles to use in calculation
- **face\_orientations** (*numpy.ndarray*, shape= (( $N_{particles}$ , ),  $N_{faces}$ , 4), dtype= *numpy.float32*) – 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.

**bin\_counts**

Get the raw bin counts.

**box**

Get the box used in the calculation.

**compute**(*self*, *box*, *ref\_points*, *ref\_orientations*, *points*, *orientations*, *face\_orientations*, *nlist*=None)

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

**Parameters**

- **box** (*freud.box.Box*) – simulation box
- **ref\_points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – reference points to calculate the local density
- **ref\_orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ , 4), dtype= *numpy.float32*) – orientations of reference points to use in calculation
- **points** (*numpy.ndarray*, shape= ( $N_{particles}$ , 3), dtype= *numpy.float32*) – points to calculate the local density
- **orientations** (*numpy.ndarray*, shape= ( $N_{particles}$ , 4), dtype= *numpy.float32*) – orientations of particles to use in calculation
- **face\_orientations** (*numpy.ndarray*, shape= (( $N_{particles}$ , ),  $N_{faces}$ , 4), dtype= *numpy.float32*) – orientations of particle faces to account for particle symmetry
- **nlist** (*freud.locality.NeighborList*) – *freud.locality.NeighborList* object to use to find bonds

**getBinCounts**(*self*)

Get the raw bin counts.

**Returns** Bin Counts

**Return type** `numpy.ndarray`, shape=  $(N_z, N_y, N_x)$ , dtype= `numpy.uint32`

**getBox** (*self*)

Get the box used in the calculation.

**Returns** freud Box

**Return type** `freud.box.Box`

**getJacobian** (*self*)

Get the Jacobian.

**Returns** Jacobian

**Return type** `float`

**getNBinsX** (*self*)

Get the number of bins in the x-dimension of histogram.

**Returns**  $N_x$

**Return type** unsigned int

**getNBinsY** (*self*)

Get the number of bins in the y-dimension of histogram.

**Returns**  $N_y$

**Return type** unsigned int

**getNBinsZ** (*self*)

Get the number of bins in the z-dimension of histogram.

**Returns**  $N_z$

**Return type** unsigned int

**getPCF** (*self*)

Get the positional correlation function.

**Returns** PCF

**Return type** `numpy.ndarray`, shape=  $(N_z, N_y, N_x)$ , dtype= `numpy.float32`

**getPMFT** (*self*)

Get the potential of mean force and torque.

**Returns** PMFT

**Return type** `numpy.ndarray`, shape=  $(N_z, N_y, N_x)$ , dtype= `numpy.float32`

**getRCut** (*self*)

Get the `r_cut` value used in the cell list.

**Returns** `r_cut`

**Return type** `float`

**getX** (*self*)

Get the array of x-values for the PCF histogram.

**Returns** bin centers of x-dimension of histogram

**Return type** `numpy.ndarray`, shape=  $(N_x)$ , dtype= `numpy.float32`

**getY**(*self*)

Get the array of y-values for the PCF histogram.

**Returns** bin centers of y-dimension of histogram**Return type** `numpy.ndarray`, shape= ( $N_y$ ), dtype= `numpy.float32`**getZ**(*self*)

Get the array of z-values for the PCF histogram.

**Returns** bin centers of z-dimension of histogram**Return type** `numpy.ndarray`, shape= ( $N_z$ ), dtype= `numpy.float32`**jacobian**

Get the Jacobian used in the PMFT.

**n\_bins\_X**

Get the number of bins in the x-dimension of histogram.

**n\_bins\_Y**

Get the number of bins in the y-dimension of histogram.

**n\_bins\_Z**

Get the number of bins in the z-dimension of histogram.

**r\_cut**Get the `r_cut` value used in the cell list.**reducePCF**(*self*)Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.pmft.PMFTXYZ.PCF()`.**resetPCF**(*self*)

Resets the values of the PCF histograms in memory.

### 1.3.12 Voronoi Module

**class** `freud.voronoi.Voronoi` (*box*, *buff*=0.1)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.

**compute** (*positions*, *box*=None, *buff*=None)

Compute Voronoi diagram.

**Parameters**

- **box** (`freud.box.Box`) – simulation box
- **buff** (`float`) – buffer width

**computeNeighbors** (*positions*, *box=None*, *buff=None*, *exclude\_ii=True*)

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.

---

**computeVolumes** ()

Computes volumes (areas in 2D) of Voronoi cells.

New in version 0.8.

Must call *compute()* before this method.

Retrieve the results with *getVolumes()*.

**getBuffer** ()

Returns the buffer width.

**Returns** buffer width

**Return type** float

**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** (*numShells*)

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 `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 `compute()` method, if all the polytopes are closed. Otherwise try using a larger buffer width.

**Returns** `numpy.ndarray` containing Voronoi polytope volumes/areas.

**Return type** `numpy.ndarray`, shape= ( $N_{cells}$ ), dtype= `numpy.float32`

#### **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 `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(box)**

Reset the simulation box.

**Parameters** `box` (`freud.box.Box`) – simulation box

#### **setBufferWidth(buff)**

Reset the buffer width.

**Parameters** `buff` (`float`) – buffer width

## 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
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

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

```

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

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

## Python vs. Cython vs. C++

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

```

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

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

```

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

```

void cplusplus_heavy_function(float* computed_array,
                             float* positions,
                             int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (i != j)

```

(continues on next page)

(continued from previous page)

```
        {
            r_ij = pos_j - pos_i;
            ...
            computed_array[i] += some_val;
        }
    }
}
```

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

## 1.4.2 Source Code Conventions

The guidelines below should be followed for any new code added to freud. This guide is separated into three sections, one for guidelines common to python and C++, one for python alone, and one for C++.

### Both

#### Naming Conventions

The following conventions should apply to Python, Cython, and C++ code.

- Variable names use `lower_case_with_underscores`
- Function and method names use `lowerCaseWithNoUnderscores`
- Class names use `CapWords`

Python example:

```
class FreudClass(object):
    def __init__(self):
        pass
    def calcSomething(self, position_i, orientation_i, position_j, orientation_j):
        r_ij = position_j - position_i
        theta_ij = calcOrientationThing(orientation_i, orientation_j)
    def calcOrientationThing(self, orientation_i, orientation_j):
        ...
```

C++ example:

```
class FreudCPPClass
{
    FreudCPPClass()
    {
    }
    computeSomeValue(int variable_a, float variable_b)
    {
        // do some things in here
    }
};
```

## Indentation

- Spaces, not tabs, must be used for indentation
- *4 spaces* are required per level of indentation
- *4 spaces* are *required*, not optional, for continuation lines
- There should be no whitespace at the end of lines in the file.
- Documentation comments and items broken over multiple lines should be *aligned* with spaces

```
class SomeClass
{
    private:
        int m_some_member;          //!< Documentation for some_member
        int m_some_other_member;    //!< Documentation for some_other_member
};

template<class BlahBlah> void some_long_func(BlahBlah with_a_really_long_argument_
↪list,
                                           int b,
                                           int c);
```

## Formatting Long Lines

All code lines should be hand-wrapped so that they are no more than *79 characters* long. Simply break any excessively long line of code at any natural breaking point to continue on the next line.

```
cout << "This is a really long message, with "
     << message.length()
     << "Characters in it:"
     << message << endl;
```

Try to maintain some element of beautiful symmetry in the way the line is broken. For example, the *above* long message is preferred over the below:

```
cout << "This is a really long message, with " << message.length() << "Characters in_
↪it:"
     << message << endl;
```

There are *special rules* for function definitions and/or calls:

- If the function definition (or call) cleanly fits within the character limit, leave it all on one line

```
int some_function(int arg1, int arg2)
```

- (Option 1) If the function definition (or call) goes over the limit, you may be able to fix it by simply putting the template definition on the previous line:

```
// go from
template<class Foo, class Bar> int some_really_long_function_name(int with_really_
↪long, Foo argument, Bar lists)
// to
template<class Foo, class Bar>
int some_really_long_function_name(int with_really_long, Foo argument, Bar lists)
```

- (Option 2) If the function doesn't have a template specifier, or splitting at that point isn't enough, split out each argument onto a separate line and align them.

```
// Instead of this...
int someReallyLongFunctionName(int with_really_long_arguments, int or, int maybe,
↪float there, char are, int just, float a, int lot, char of, int them)

// ...use this.
int someReallyLongFunctionName(int with_really_long_arguments,
                               int or,
                               int maybe,
                               float there,
                               char are,
                               int just,
                               float a,
                               int lot,
                               char of,
                               int them)
```

## Python

Code in freud should follow [PEP 8](#), as well as the following guidelines. Anything listed here takes precedence over PEP 8, but try to deviate as little as possible from PEP 8. When in doubt, follow these guidelines over PEP 8.

If you are unsure if your code is PEP 8 compliant, you can use `autopep8` and `flake8` (or similar) to automatically update and check your code.

## Semicolons

Semicolons should not be used to mark the end of lines in Python.

## Documentation Comments

- Python documentation uses sphinx, not doxygen
- See the [sphinx documentation](#) for more information
- Documentation should be included at the Python-level in the Cython wrapper.
- Every class, member variable, function, function parameter, macro, etc. must be documented with *Python docstring* comments which will be converted to documentation with sphinx.
- If you copy an existing file as a template, do not leave the existing documentation comments there. They apply to the original file, not your new one!
- The best advice that can be given is to write the documentation comments *first* and the actual code *second*. This allows one to formulate their thoughts and write out in English what the code is going to be doing. After thinking through that, writing the actual code is often *much easier*, plus the documentation left for future developers to read is top-notch.
- Good documentation comments are best demonstrated with an in-code example.

## 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:

- Edit `cpp/CMakeLists.txt`
  - Add `${CMAKE_CURRENT_SOURCE_DIR}/moduleName` to `include_directories`.
  - Add `moduleName/SubModule.cc` and `moduleName/SubModule.h` to the `FREUD_SOURCES` in `set`.
- 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.
- 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

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

### 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/glutzer/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.



## CHAPTER 2

---

### Index

---

- `genindex`
- `search`



---

## Bibliography

---

- [Cit0] Bokeh Development Team (2014). Bokeh: Python library for interactive visualization URL <http://www.bokeh.pydata.org>.
- [Cit1] 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>
- [Cit2] 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>
- [Cit3] 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>
- [Cit4] 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>



### f

`freud.parallel`, [70](#)  
`freud.voronoi`, [83](#)



## Symbols

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

## A

`accumulate()` (freud.density.ComplexCF method), 23  
`accumulate()` (freud.density.FloatCF method), 22  
`accumulate()` (freud.density.RDF method), 27  
`accumulate()` (freud.order.BondOrder method), 48  
`accumulate()` (freud.pmft.PMFTR12 method), 77  
`accumulate()` (freud.pmft.PMFTXY2D method), 75  
`accumulate()` (freud.pmft.PMFTXYT method), 71  
`accumulate()` (freud.pmft.PMFTXYZ method), 81  
`add_ptype()` (freud.kspace.SingleCell3D method), 33  
`addFT()` (freud.kspace.FTfactory method), 35  
`AlignedBoxConstraint` (class in freud.kspace), 40  
`AnalyzeSFactor3D` (class in freud.kspace), 32  
`ave_norm_Ql` (freud.order.LocalQl attribute), 56  
`ave_norm_Wl` (freud.order.LocalWl attribute), 60  
`ave_Ql` (freud.order.LocalQl attribute), 56  
`ave_Wl` (freud.order.LocalWl attribute), 60

## B

`bin_counts` (freud.pmft.PMFTR12 attribute), 78  
`bin_counts` (freud.pmft.PMFTXY2D attribute), 75  
`bin_counts` (freud.pmft.PMFTXYT attribute), 72  
`bin_counts` (freud.pmft.PMFTXYZ attribute), 81  
`bond_lifetimes` (freud.bond.BondingAnalysis attribute), 6  
`bond_order` (freud.order.BondOrder attribute), 48  
`BondingAnalysis` (class in freud.bond), 6  
`BondingR12` (class in freud.bond), 10  
`BondingXY2D` (class in freud.bond), 7  
`BondingXYT` (class in freud.bond), 9  
`BondingXYZ` (class in freud.bond), 11  
`BondOrder` (class in freud.order), 48  
`bonds` (freud.bond.BondingR12 attribute), 10  
`bonds` (freud.bond.BondingXY2D attribute), 7  
`bonds` (freud.bond.BondingXYT attribute), 9  
`bonds` (freud.bond.BondingXYZ attribute), 12

`Box` (class in freud.box), 13  
`box` (freud.bond.BondingR12 attribute), 10  
`box` (freud.bond.BondingXY2D attribute), 7  
`box` (freud.bond.BondingXYT attribute), 9  
`box` (freud.bond.BondingXYZ attribute), 12  
`box` (freud.cluster.Cluster attribute), 18  
`box` (freud.cluster.ClusterProperties attribute), 20  
`box` (freud.density.ComplexCF attribute), 24  
`box` (freud.density.FloatCF attribute), 22  
`box` (freud.density.GaussianDensity attribute), 25  
`box` (freud.density.LocalDensity attribute), 26  
`box` (freud.density.RDF attribute), 28  
`box` (freud.locality.LinkCell attribute), 43  
`box` (freud.locality.NearestNeighbors attribute), 45  
`box` (freud.order.BondOrder attribute), 48  
`box` (freud.order.HexOrderParameter attribute), 52  
`box` (freud.order.LocalQl attribute), 56  
`box` (freud.order.LocalWl attribute), 60  
`box` (freud.order.Pairing2D attribute), 69  
`box` (freud.order.SolLiq attribute), 63  
`box` (freud.order.TransOrderParameter attribute), 55  
`box` (freud.pmft.PMFTR12 attribute), 78  
`box` (freud.pmft.PMFTXY2D attribute), 75  
`box` (freud.pmft.PMFTXYT attribute), 72  
`box` (freud.pmft.PMFTXYZ attribute), 81

## C

`calculate()` (freud.kspace.SingleCell3D method), 33  
`Cluster` (class in freud.cluster), 18  
`cluster()` (freud.order.MatchEnv method), 66  
`cluster_COM` (freud.cluster.ClusterProperties attribute), 20  
`cluster_G` (freud.cluster.ClusterProperties attribute), 20  
`cluster_idx` (freud.cluster.Cluster attribute), 18  
`cluster_keys` (freud.cluster.Cluster attribute), 19  
`cluster_sizes` (freud.cluster.ClusterProperties attribute), 20  
`cluster_sizes` (freud.order.SolLiq attribute), 63  
`ClusterProperties` (class in freud.cluster), 20  
`clusters` (freud.order.SolLiq attribute), 63

ComplexCF (class in freud.density), 23  
compute() (freud.bond.BondingAnalysis method), 6  
compute() (freud.bond.BondingR12 method), 10  
compute() (freud.bond.BondingXY2D method), 7  
compute() (freud.bond.BondingXYT method), 9  
compute() (freud.bond.BondingXYZ method), 12  
compute() (freud.density.ComplexCF method), 24  
compute() (freud.density.FloatCF method), 22  
compute() (freud.density.GaussianDensity method), 25  
compute() (freud.density.LocalDensity method), 26  
compute() (freud.density.RDF method), 28  
compute() (freud.interface.InterfaceMeasure method), 31  
compute() (freud.kspace.FTdelta method), 37  
compute() (freud.kspace.FTpolyhedron method), 38  
compute() (freud.kspace.SFactor3DPoints method), 32  
compute() (freud.locality.LinkCell method), 44  
compute() (freud.locality.NearestNeighbors method), 45  
compute() (freud.order.BondOrder method), 49  
compute() (freud.order.CubaticOrderParameter method), 50  
compute() (freud.order.HexOrderParameter method), 52  
compute() (freud.order.LocalDescriptors method), 53  
compute() (freud.order.LocalQI method), 56  
compute() (freud.order.LocalQINear method), 58  
compute() (freud.order.LocalW1 method), 60  
compute() (freud.order.LocalWINear method), 62  
compute() (freud.order.NematicOrderParameter method), 51  
compute() (freud.order.Pairing2D method), 69  
compute() (freud.order.SolLiq method), 63  
compute() (freud.order.SolLiqNear method), 65  
compute() (freud.order.TransOrderParameter method), 55  
compute() (freud.pmft.PMFTR12 method), 78  
compute() (freud.pmft.PMFTXY2D method), 75  
compute() (freud.pmft.PMFTXYT method), 72  
compute() (freud.pmft.PMFTXYZ method), 81  
compute() (freud.voronoi.Voronoi method), 83  
compute\_py() (freud.kspace.FTconvexPolyhedron method), 39  
computeAve() (freud.order.LocalQI method), 57  
computeAve() (freud.order.LocalQINear method), 59  
computeAve() (freud.order.LocalW1 method), 60  
computeAve() (freud.order.LocalWINear method), 62  
computeAveNorm() (freud.order.LocalQI method), 57  
computeAveNorm() (freud.order.LocalQINear method), 59  
computeAveNorm() (freud.order.LocalW1 method), 60  
computeAveNorm() (freud.order.LocalWINear method), 62  
computeCellList() (freud.locality.LinkCell method), 44  
computeClusterMembership() (freud.cluster.Cluster method), 19  
computeClusters() (freud.cluster.Cluster method), 19  
computeNeighbors() (freud.voronoi.Voronoi method), 84

computeNList() (freud.order.LocalDescriptors method), 54  
computeNorm() (freud.order.LocalQI method), 57  
computeNorm() (freud.order.LocalQINear method), 59  
computeNorm() (freud.order.LocalW1 method), 60  
computeNorm() (freud.order.LocalWINear method), 63  
computeProperties() (freud.cluster.ClusterProperties method), 20  
computeSolLiqNoNorm() (freud.order.SolLiq method), 64  
computeSolLiqNoNorm() (freud.order.SolLiqNear method), 66  
computeSolLiqVariant() (freud.order.SolLiq method), 64  
computeSolLiqVariant() (freud.order.SolLiqNear method), 66  
computeVolumes() (freud.voronoi.Voronoi method), 84  
constrainedLatticePoints() (in module freud.kspace), 40  
Constraint (class in freud.kspace), 40  
copy() (freud.locality.NeighborList method), 42  
counts (freud.density.ComplexCF attribute), 24  
counts (freud.density.FloatCF attribute), 22  
CubaticOrderParameter (class in freud.order), 50  
cube() (freud.box.Box class method), 14

## D

d\_r (freud.order.TransOrderParameter attribute), 55  
DeltaSpot (class in freud.kspace), 39  
density (freud.density.LocalDensity attribute), 26  
dimensions (freud.box.Box attribute), 14

## F

filter() (freud.locality.NeighborList method), 42  
filter\_r() (freud.locality.NeighborList method), 42  
find\_first\_index() (freud.locality.NeighborList method), 42  
FloatCF (class in freud.density), 21  
freud.parallel (module), 70  
freud.voronoi (module), 83  
from\_arrays() (freud.locality.NeighborList method), 42  
from\_box() (freud.box.Box class method), 14  
from\_matrix() (freud.box.Box class method), 14  
FTbase (class in freud.kspace), 36  
FTconvexPolyhedron (class in freud.kspace), 38  
FTdelta (class in freud.kspace), 37  
FTfactory (class in freud.kspace), 35  
FTpolyhedron (class in freud.kspace), 38  
FTsphere (class in freud.kspace), 37

## G

gaussian\_density (freud.density.GaussianDensity attribute), 25  
GaussianDensity (class in freud.density), 25  
GaussianSpot (class in freud.kspace), 40

get\_cubic\_order\_parameter() (freud.order.CubicOrderParameter method), 50  
 get\_cubic\_tensor() (freud.order.CubicOrderParameter method), 50  
 get\_density() (freud.kspace.FTbase method), 36  
 get\_director() (freud.order.NematicOrderParameter method), 51  
 get\_form\_factors() (freud.kspace.SingleCell3D method), 33  
 get\_gen\_r4\_tensor() (freud.order.CubicOrderParameter method), 50  
 get\_global\_tensor() (freud.order.CubicOrderParameter method), 50  
 get\_gridPoints() (freud.kspace.DeltaSpot method), 39  
 get\_nematic\_order\_parameter() (freud.order.NematicOrderParameter method), 51  
 get\_nematic\_tensor() (freud.order.NematicOrderParameter method), 52  
 get\_orientation() (freud.order.CubicOrderParameter method), 51  
 get\_parambyname() (freud.kspace.FTbase method), 36  
 get\_params() (freud.kspace.FTbase method), 36  
 get\_particle\_op() (freud.order.CubicOrderParameter method), 51  
 get\_particle\_tensor() (freud.order.CubicOrderParameter method), 51  
 get\_particle\_tensor() (freud.order.NematicOrderParameter method), 52  
 get\_ptypes() (freud.kspace.SingleCell3D method), 33  
 get\_radius() (freud.kspace.FTconvexPolyhedron method), 39  
 get\_radius() (freud.kspace.FTpolyhedron method), 38  
 get\_radius() (freud.kspace.FTsphere method), 37  
 get\_scale() (freud.kspace.FTbase method), 36  
 get\_scale() (freud.order.CubicOrderParameter method), 51  
 get\_t\_final() (freud.order.CubicOrderParameter method), 51  
 get\_t\_initial() (freud.order.CubicOrderParameter method), 51  
 getAveQl() (freud.order.LocalQl method), 57  
 getAveWl() (freud.order.LocalWl method), 61  
 getBinCounts() (freud.pmft.PMFTR12 method), 78  
 getBinCounts() (freud.pmft.PMFTXY2D method), 75  
 getBinCounts() (freud.pmft.PMFTXYT method), 72  
 getBinCounts() (freud.pmft.PMFTXYZ method), 81  
 getBondLifetimes() (freud.bond.BondingAnalysis method), 6  
 getBondOrder() (freud.order.BondOrder method), 49  
 getBonds() (freud.bond.BondingR12 method), 11  
 getBonds() (freud.bond.BondingXY2D method), 8  
 getBonds() (freud.bond.BondingXYT method), 9  
 getBonds() (freud.bond.BondingXYZ method), 12  
 getBox() (freud.bond.BondingR12 method), 11  
 getBox() (freud.bond.BondingXY2D method), 8  
 getBox() (freud.bond.BondingXYT method), 9  
 getBox() (freud.bond.BondingXYZ method), 12  
 getBox() (freud.cluster.Cluster method), 19  
 getBox() (freud.cluster.ClusterProperties method), 20  
 getBox() (freud.density.ComplexCF method), 24  
 getBox() (freud.density.FloatCF method), 22  
 getBox() (freud.density.GaussianDensity method), 25  
 getBox() (freud.density.LocalDensity method), 26  
 getBox() (freud.density.RDF method), 28  
 getBox() (freud.locality.LinkCell method), 44  
 getBox() (freud.locality.NearestNeighbors method), 46  
 getBox() (freud.order.BondOrder method), 49  
 getBox() (freud.order.HexOrderParameter method), 52  
 getBox() (freud.order.LocalQl method), 57  
 getBox() (freud.order.LocalWl method), 61  
 getBox() (freud.order.Pairing2D method), 70  
 getBox() (freud.order.SolLiq method), 64  
 getBox() (freud.order.TransOrderParameter method), 55  
 getBox() (freud.pmft.PMFTR12 method), 78  
 getBox() (freud.pmft.PMFTXY2D method), 75  
 getBox() (freud.pmft.PMFTXYT method), 72  
 getBox() (freud.pmft.PMFTXYZ method), 82  
 getBuffer() (freud.voronoi.Voronoi method), 84  
 getCell() (freud.locality.LinkCell method), 44  
 getCellNeighbors() (freud.locality.LinkCell method), 44  
 getClusterCOM() (freud.cluster.ClusterProperties method), 20  
 getClusterG() (freud.cluster.ClusterProperties method), 20  
 getClusterIdx() (freud.cluster.Cluster method), 19  
 getClusterKeys() (freud.cluster.Cluster method), 19  
 getClusters() (freud.order.MatchEnv method), 67  
 getClusters() (freud.order.SolLiq method), 64  
 getClusterSizes() (freud.cluster.ClusterProperties method), 21  
 getClusterSizes() (freud.order.SolLiq method), 64  
 getCoordinates() (freud.box.Box method), 14  
 getCounts() (freud.density.ComplexCF method), 24  
 getCounts() (freud.density.FloatCF method), 22  
 getDensity() (freud.density.LocalDensity method), 27  
 getDr() (freud.order.TransOrderParameter method), 55  
 getEnvironment() (freud.order.MatchEnv method), 67  
 getFT() (freud.kspace.FTbase method), 36  
 getFTlist() (freud.kspace.FTfactory method), 35  
 getFTObject() (freud.kspace.FTfactory method), 35  
 getGaussianDensity() (freud.density.GaussianDensity method), 26  
 getImage() (freud.box.Box method), 14  
 getInverseJacobian() (freud.pmft.PMFTR12 method), 78  
 getJacobian() (freud.pmft.PMFTXY2D method), 75  
 getJacobian() (freud.pmft.PMFTXYT method), 72

getJacobian() (freud.pmft.PMFTXYZ method), 82  
getK() (freud.order.HexOrderParameter method), 52  
getL() (freud.box.Box method), 14  
getLargestClusterSize() (freud.order.SolLiq method), 64  
getLatticeVector() (freud.box.Box method), 14  
getLinv() (freud.box.Box method), 14  
getListMap() (freud.bond.BondingR12 method), 11  
getListMap() (freud.bond.BondingXY2D method), 8  
getListMap() (freud.bond.BondingXYT method), 9  
getListMap() (freud.bond.BondingXYZ method), 12  
getLMax() (freud.order.LocalDescriptors method), 54  
getLx() (freud.box.Box method), 15  
getLy() (freud.box.Box method), 15  
getLz() (freud.box.Box method), 15  
getMatch() (freud.order.Pairing2D method), 70  
getNBinsPhi() (freud.order.BondOrder method), 49  
getNBinsR() (freud.pmft.PMFTR12 method), 79  
getNBinsT() (freud.pmft.PMFTXYT method), 73  
getNBinsT1() (freud.pmft.PMFTR12 method), 79  
getNBinsT2() (freud.pmft.PMFTR12 method), 79  
getNBinsTheta() (freud.order.BondOrder method), 49  
getNBinsX() (freud.pmft.PMFTXY2D method), 76  
getNBinsX() (freud.pmft.PMFTXYT method), 73  
getNBinsX() (freud.pmft.PMFTXYZ method), 82  
getNBinsY() (freud.pmft.PMFTXY2D method), 76  
getNBinsY() (freud.pmft.PMFTXYT method), 73  
getNBinsY() (freud.pmft.PMFTXYZ method), 82  
getNBinsZ() (freud.pmft.PMFTXYZ method), 82  
getNeighborList() (freud.locality.NearestNeighbors method), 46  
getNeighborList() (freud.voronoi.Voronoi method), 84  
getNeighbors() (freud.locality.NearestNeighbors method), 46  
getNeighbors() (freud.voronoi.Voronoi method), 84  
getNP() (freud.order.HexOrderParameter method), 53  
getNP() (freud.order.LocalDescriptors method), 54  
getNP() (freud.order.LocalQl method), 57  
getNP() (freud.order.LocalWl method), 61  
getNP() (freud.order.MatchEnv method), 67  
getNP() (freud.order.SolLiq method), 64  
getNP() (freud.order.TransOrderParameter method), 55  
getNr() (freud.density.RDF method), 28  
getNRef() (freud.locality.NearestNeighbors method), 46  
getNSphs() (freud.order.LocalDescriptors method), 54  
getNumberOfConnections() (freud.order.SolLiq method), 64  
getNumBonds() (freud.bond.BondingAnalysis method), 6  
getNumCells() (freud.locality.LinkCell method), 44  
getNumClusters() (freud.cluster.Cluster method), 19  
getNumClusters() (freud.cluster.ClusterProperties method), 21  
getNumClusters() (freud.order.MatchEnv method), 67  
getNumElements() (freud.index.Index2D method), 29  
getNumElements() (freud.index.Index3D method), 30  
getNumFrames() (freud.bond.BondingAnalysis method), 6  
getNumNeighbors() (freud.density.LocalDensity method), 27  
getNumNeighbors() (freud.locality.NearestNeighbors method), 46  
getNumParticles() (freud.bond.BondingAnalysis method), 6  
getNumParticles() (freud.cluster.Cluster method), 19  
getOverallLifetimes() (freud.bond.BondingAnalysis method), 7  
getPair() (freud.order.Pairing2D method), 70  
getPCF() (freud.pmft.PMFTR12 method), 79  
getPCF() (freud.pmft.PMFTXY2D method), 76  
getPCF() (freud.pmft.PMFTXYT method), 73  
getPCF() (freud.pmft.PMFTXYZ method), 82  
getPeakDegeneracy() (freud.kspace.AnalyzeSFactor3D method), 32  
getPeakList() (freud.kspace.AnalyzeSFactor3D method), 33  
getPeriodic() (freud.box.Box method), 15  
getPeriodicX() (freud.box.Box method), 15  
getPeriodicY() (freud.box.Box method), 15  
getPeriodicZ() (freud.box.Box method), 15  
getPhi() (freud.order.BondOrder method), 49  
getPMFT() (freud.pmft.PMFTR12 method), 79  
getPMFT() (freud.pmft.PMFTXY2D method), 76  
getPMFT() (freud.pmft.PMFTXYT method), 73  
getPMFT() (freud.pmft.PMFTXYZ method), 82  
getPsi() (freud.order.HexOrderParameter method), 53  
getQ() (freud.kspace.SFactor3DPoints method), 32  
getQl() (freud.order.LocalQl method), 57  
getQl() (freud.order.LocalWl method), 61  
getQlAveNorm() (freud.order.LocalQl method), 57  
getQldot\_ij() (freud.order.SolLiq method), 65  
getQlmi() (freud.order.SolLiq method), 65  
getQlNorm() (freud.order.LocalQl method), 58  
getR() (freud.density.ComplexCF method), 24  
getR() (freud.density.FloatCF method), 23  
getR() (freud.density.RDF method), 28  
getR() (freud.pmft.PMFTR12 method), 79  
getRCut() (freud.pmft.PMFTR12 method), 79  
getRCut() (freud.pmft.PMFTXY2D method), 76  
getRCut() (freud.pmft.PMFTXYT method), 73  
getRCut() (freud.pmft.PMFTXYZ method), 82  
getRDF() (freud.density.ComplexCF method), 24  
getRDF() (freud.density.FloatCF method), 23  
getRDF() (freud.density.RDF method), 28  
getRevListMap() (freud.bond.BondingR12 method), 11  
getRevListMap() (freud.bond.BondingXY2D method), 8  
getRevListMap() (freud.bond.BondingXYT method), 10  
getRevListMap() (freud.bond.BondingXYZ method), 12  
getRMax() (freud.locality.NearestNeighbors method), 46

getRMax() (freud.order.LocalDescriptors method), 54  
 getRsq() (freud.locality.NearestNeighbors method), 46  
 getRsqList() (freud.locality.NearestNeighbors method), 46  
 getS() (freud.kspace.SFactor3DPoints method), 32  
 getSComplex() (freud.kspace.SFactor3DPoints method), 32  
 getSph() (freud.order.LocalDescriptors method), 54  
 getSvsQ() (freud.kspace.AnalyzeSFactor3D method), 33  
 getT() (freud.pmft.PMFTXYT method), 73  
 getT1() (freud.pmft.PMFTR12 method), 79  
 getT2() (freud.pmft.PMFTR12 method), 79  
 getTheta() (freud.order.BondOrder method), 49  
 getTiltFactorXY() (freud.box.Box method), 15  
 getTiltFactorXZ() (freud.box.Box method), 15  
 getTiltFactorYZ() (freud.box.Box method), 16  
 getTotEnvironment() (freud.order.MatchEnv method), 67  
 getTransitionMatrix() (freud.bond.BondingAnalysis method), 7  
 getUINTMAX() (freud.locality.NearestNeighbors method), 46  
 getVolume() (freud.box.Box method), 16  
 getVolumes() (freud.voronoi.Voronoi method), 84  
 getVoronoiPolytopes() (freud.voronoi.Voronoi method), 85  
 getWl() (freud.order.LocalWl method), 61  
 getWlAveNorm() (freud.order.LocalWl method), 61  
 getWlNorm() (freud.order.LocalWl method), 61  
 getWrappedVectors() (freud.locality.NearestNeighbors method), 47  
 getX() (freud.pmft.PMFTXY2D method), 76  
 getX() (freud.pmft.PMFTXYT method), 73  
 getX() (freud.pmft.PMFTXYZ method), 82  
 getY() (freud.pmft.PMFTXY2D method), 76  
 getY() (freud.pmft.PMFTXYT method), 73  
 getY() (freud.pmft.PMFTXYZ method), 82  
 getZ() (freud.pmft.PMFTXYZ method), 83

## H

HexOrderParameter (class in freud.order), 52

## I

Index2D (class in freud.index), 29  
 Index3D (class in freud.index), 30  
 index\_i (freud.locality.NeighborList attribute), 43  
 index\_j (freud.locality.NeighborList attribute), 43  
 initialize() (freud.bond.BondingAnalysis method), 7  
 InterfaceMeasure (class in freud.interface), 31  
 inverse\_jacobian (freud.pmft.PMFTR12 attribute), 79  
 is2D() (freud.box.Box method), 16  
 isSimilar() (freud.order.MatchEnv method), 67  
 itercell() (freud.locality.LinkCell method), 44

## J

jacobian (freud.pmft.PMFTXY2D attribute), 76  
 jacobian (freud.pmft.PMFTXYT attribute), 73  
 jacobian (freud.pmft.PMFTXYZ attribute), 83

## K

k (freud.order.HexOrderParameter attribute), 53

## L

L (freud.box.Box attribute), 13  
 l\_max (freud.order.LocalDescriptors attribute), 54  
 largest\_cluster\_size (freud.order.SolLiq attribute), 65  
 LinkCell (class in freud.locality), 43  
 Linv (freud.box.Box attribute), 13  
 list\_map (freud.bond.BondingR12 attribute), 11  
 list\_map (freud.bond.BondingXY2D attribute), 8  
 list\_map (freud.bond.BondingXYT attribute), 10  
 list\_map (freud.bond.BondingXYZ attribute), 12  
 LocalDensity (class in freud.density), 26  
 LocalDescriptors (class in freud.order), 53  
 LocalQl (class in freud.order), 56  
 LocalQlNear (class in freud.order), 58  
 LocalWl (class in freud.order), 59  
 LocalWlNear (class in freud.order), 62  
 Lx (freud.box.Box attribute), 13  
 Ly (freud.box.Box attribute), 13  
 Lz (freud.box.Box attribute), 14

## M

makeCoordinates() (freud.box.Box method), 16  
 makeFraction() (freud.box.Box method), 16  
 makeSpot() (freud.kspace.DeltaSpot method), 39  
 makeSpot() (freud.kspace.GaussianSpot method), 40  
 match (freud.order.Pairing2D attribute), 70  
 MatchEnv (class in freud.order), 66  
 matchMotif() (freud.order.MatchEnv method), 68  
 meshgrid2() (in module freud.kspace), 31  
 minimizeRMSD() (freud.order.MatchEnv method), 68  
 minRMSDMotif() (freud.order.MatchEnv method), 68

## N

n\_bins\_r (freud.pmft.PMFTR12 attribute), 80  
 n\_bins\_T (freud.pmft.PMFTXYT attribute), 74  
 n\_bins\_T1 (freud.pmft.PMFTR12 attribute), 79  
 n\_bins\_T2 (freud.pmft.PMFTR12 attribute), 80  
 n\_bins\_X (freud.pmft.PMFTXY2D attribute), 76  
 n\_bins\_X (freud.pmft.PMFTXYT attribute), 74  
 n\_bins\_X (freud.pmft.PMFTXYZ attribute), 83  
 n\_bins\_Y (freud.pmft.PMFTXY2D attribute), 76  
 n\_bins\_Y (freud.pmft.PMFTXYT attribute), 74  
 n\_bins\_Y (freud.pmft.PMFTXYZ attribute), 83  
 n\_bins\_Z (freud.pmft.PMFTXYZ attribute), 83  
 n\_r (freud.density.RDF attribute), 28

`n_ref` (freud.locality.NearestNeighbors attribute), 47  
`NearestNeighbors` (class in freud.locality), 45  
`neighbor_counts` (freud.locality.NeighborList attribute), 43  
`NeighborList` (class in freud.locality), 41  
`NematicOrderParameter` (class in freud.order), 51  
`nlist` (freud.locality.LinkCell attribute), 45  
`nlist` (freud.locality.NearestNeighbors attribute), 47  
`norm_Ql` (freud.order.LocalQl attribute), 58  
`norm_Wl` (freud.order.LocalWl attribute), 61  
`num_bonds` (freud.bond.BondingAnalysis attribute), 7  
`num_cells` (freud.locality.LinkCell attribute), 45  
`num_clusters` (freud.cluster.Cluster attribute), 19  
`num_clusters` (freud.cluster.ClusterProperties attribute), 21  
`num_clusters` (freud.order.MatchEnv attribute), 69  
`num_connections` (freud.order.SolLiq attribute), 65  
`num_elements` (freud.index.Index2D attribute), 29  
`num_elements` (freud.index.Index3D attribute), 30  
`num_frames` (freud.bond.BondingAnalysis attribute), 7  
`num_neighbors` (freud.density.LocalDensity attribute), 27  
`num_neighbors` (freud.locality.NearestNeighbors attribute), 47  
`num_neighbors` (freud.order.LocalDescriptors attribute), 54  
`num_particles` (freud.bond.BondingAnalysis attribute), 7  
`num_particles` (freud.cluster.Cluster attribute), 19  
`num_particles` (freud.order.HexOrderParameter attribute), 53  
`num_particles` (freud.order.LocalDescriptors attribute), 54  
`num_particles` (freud.order.LocalQl attribute), 58  
`num_particles` (freud.order.LocalWl attribute), 61  
`num_particles` (freud.order.MatchEnv attribute), 69  
`num_particles` (freud.order.SolLiq attribute), 65  
`num_particles` (freud.order.TransOrderParameter attribute), 55  
`NumThreads` (class in freud.parallel), 70

## O

`overall_lifetimes` (freud.bond.BondingAnalysis attribute), 7

## P

`pair` (freud.order.Pairing2D attribute), 70  
`Pairing2D` (class in freud.order), 69  
`PCF` (freud.pmft.PMFTR12 attribute), 77  
`PCF` (freud.pmft.PMFTXY2D attribute), 74  
`PCF` (freud.pmft.PMFTXYT attribute), 71  
`PCF` (freud.pmft.PMFTXYZ attribute), 80  
`periodic` (freud.box.Box attribute), 16  
`PMFT` (freud.pmft.PMFTR12 attribute), 77  
`PMFT` (freud.pmft.PMFTXY2D attribute), 74  
`PMFT` (freud.pmft.PMFTXYT attribute), 71

`PMFT` (freud.pmft.PMFTXYZ attribute), 80  
`PMFTR12` (class in freud.pmft), 77  
`PMFTXY2D` (class in freud.pmft), 74  
`PMFTXYT` (class in freud.pmft), 71  
`PMFTXYZ` (class in freud.pmft), 80  
`psi` (freud.order.HexOrderParameter attribute), 53

## Q

`Ql` (freud.order.LocalQl attribute), 56  
`Ql` (freud.order.LocalWl attribute), 60  
`Ql_dot_ij` (freud.order.SolLiq attribute), 63  
`Ql_mi` (freud.order.SolLiq attribute), 63

## R

`R` (freud.density.ComplexCF attribute), 23  
`R` (freud.density.FloatCF attribute), 21  
`R` (freud.density.RDF attribute), 27  
`R` (freud.pmft.PMFTR12 attribute), 77  
`r_cut` (freud.pmft.PMFTR12 attribute), 80  
`r_cut` (freud.pmft.PMFTXY2D attribute), 76  
`r_cut` (freud.pmft.PMFTXYT attribute), 74  
`r_cut` (freud.pmft.PMFTXYZ attribute), 83  
`r_max` (freud.locality.NearestNeighbors attribute), 47  
`r_max` (freud.order.LocalDescriptors attribute), 54  
`r_sq_list` (freud.locality.NearestNeighbors attribute), 47  
`RDF` (class in freud.density), 27  
`RDF` (freud.density.ComplexCF attribute), 23  
`RDF` (freud.density.FloatCF attribute), 21  
`RDF` (freud.density.RDF attribute), 27  
`reciprocalLattice3D()` (in module freud.kspace), 41  
`reduceBondOrder()` (freud.order.BondOrder method), 49  
`reduceCorrelationFunction()` (freud.density.ComplexCF method), 25  
`reduceCorrelationFunction()` (freud.density.FloatCF method), 23  
`reducePCF()` (freud.pmft.PMFTR12 method), 80  
`reducePCF()` (freud.pmft.PMFTXY2D method), 76  
`reducePCF()` (freud.pmft.PMFTXYT method), 74  
`reducePCF()` (freud.pmft.PMFTXYZ method), 83  
`reduceRDF()` (freud.density.RDF method), 28  
`remove_ptype()` (freud.kspace.SingleCell3D method), 34  
`resetBondOrder()` (freud.order.BondOrder method), 50  
`resetCorrelationFunction()` (freud.density.ComplexCF method), 25  
`resetCorrelationFunction()` (freud.density.FloatCF method), 23  
`resetDensity()` (freud.density.GaussianDensity method), 26  
`resetPCF()` (freud.pmft.PMFTR12 method), 80  
`resetPCF()` (freud.pmft.PMFTXY2D method), 77  
`resetPCF()` (freud.pmft.PMFTXYT method), 74  
`resetPCF()` (freud.pmft.PMFTXYZ method), 83  
`resetRDF()` (freud.density.RDF method), 28  
`rev_list_map` (freud.bond.BondingR12 attribute), 11

rev\_list\_map (freud.bond.BondingXY2D attribute), 8  
 rev\_list\_map (freud.bond.BondingXYT attribute), 10  
 rev\_list\_map (freud.bond.BondingXYZ attribute), 12

## S

satisfies() (freud.kspace.AlignedBoxConstraint method), 40  
 satisfies() (freud.kspace.Constraint method), 40  
 segments (freud.locality.NeighborList attribute), 43  
 set2D() (freud.box.Box method), 16  
 set\_active() (freud.kspace.SingleCell3D method), 34  
 set\_box() (freud.kspace.SingleCell3D method), 34  
 set\_density() (freud.kspace.FTbase method), 36  
 set\_density() (freud.kspace.FTdelta method), 37  
 set\_density() (freud.kspace.FTpolyhedron method), 38  
 set\_dK() (freud.kspace.SingleCell3D method), 34  
 set\_form\_factor() (freud.kspace.SingleCell3D method), 34  
 set\_inactive() (freud.kspace.SingleCell3D method), 34  
 set\_K() (freud.kspace.FTbase method), 36  
 set\_K() (freud.kspace.FTdelta method), 37  
 set\_K() (freud.kspace.FTpolyhedron method), 38  
 set\_k() (freud.kspace.SingleCell3D method), 34  
 set\_ndiv() (freud.kspace.SingleCell3D method), 34  
 set\_param() (freud.kspace.SingleCell3D method), 34  
 set\_parambyname() (freud.kspace.FTbase method), 36  
 set\_params() (freud.kspace.FTpolyhedron method), 38  
 set\_radius() (freud.kspace.FTconvexPolyhedron method), 39  
 set\_radius() (freud.kspace.FTpolyhedron method), 38  
 set\_radius() (freud.kspace.FTsphere method), 37  
 set\_rq() (freud.kspace.FTbase method), 36  
 set\_rq() (freud.kspace.FTdelta method), 37  
 set\_rq() (freud.kspace.FTpolyhedron method), 38  
 set\_rq() (freud.kspace.SingleCell3D method), 35  
 set\_scale() (freud.kspace.FTbase method), 37  
 set\_scale() (freud.kspace.FTdelta method), 37  
 set\_scale() (freud.kspace.SingleCell3D method), 35  
 set\_sigma() (freud.kspace.GaussianSpot method), 40  
 set\_xy() (freud.kspace.DeltaSpot method), 39  
 set\_xy() (freud.kspace.GaussianSpot method), 40  
 setBox() (freud.order.LocalQI method), 58  
 setBox() (freud.order.LocalWI method), 61  
 setBox() (freud.order.MatchEnv method), 69  
 setBox() (freud.order.SolLiq method), 65  
 setBox() (freud.voronoi.Voronoi method), 85  
 setBufferWidth() (freud.voronoi.Voronoi method), 85  
 setClusteringRadius() (freud.order.SolLiq method), 65  
 setCutMode() (freud.locality.NearestNeighbors method), 47  
 setL() (freud.box.Box method), 16  
 setNumThreads() (freud.parallel method), 70  
 setPeriodic() (freud.box.Box method), 16  
 setPeriodicX() (freud.box.Box method), 17

setPeriodicY() (freud.box.Box method), 17  
 setPeriodicZ() (freud.box.Box method), 17  
 setRMax() (freud.locality.NearestNeighbors method), 47  
 SFactor3DPoints (class in freud.kspace), 31  
 SingleCell3D (class in freud.kspace), 33  
 SolLiq (class in freud.order), 63  
 SolLiqNear (class in freud.order), 65  
 sph (freud.order.LocalDescriptors attribute), 54  
 Spoly2D() (freud.kspace.FTconvexPolyhedron method), 38  
 Spoly3D() (freud.kspace.FTconvexPolyhedron method), 39  
 square() (freud.box.Box class method), 17

## T

T (freud.pmft.PMFTXYT attribute), 71  
 T1 (freud.pmft.PMFTR12 attribute), 77  
 T2 (freud.pmft.PMFTR12 attribute), 77  
 to\_matrix() (freud.box.Box method), 17  
 to\_tuple() (freud.box.Box method), 17  
 tot\_environment (freud.order.MatchEnv attribute), 69  
 transition\_matrix (freud.bond.BondingAnalysis attribute), 7  
 TransOrderParameter (class in freud.order), 55

## U

UINTMAX (freud.locality.NearestNeighbors attribute), 45  
 unwrap() (freud.box.Box method), 17  
 update\_bases() (freud.kspace.SingleCell3D method), 35  
 update\_K\_constraint() (freud.kspace.SingleCell3D method), 35  
 update\_Kpoints() (freud.kspace.SingleCell3D method), 35

## V

volume (freud.box.Box attribute), 17  
 Voronoi (class in freud.voronoi), 83

## W

weights (freud.locality.NeighborList attribute), 43  
 W1 (freud.order.LocalWI attribute), 60  
 wrap() (freud.box.Box method), 17  
 wrapped\_vectors (freud.locality.NearestNeighbors attribute), 47

## X

X (freud.pmft.PMFTXY2D attribute), 74  
 X (freud.pmft.PMFTXYT attribute), 71  
 X (freud.pmft.PMFTXYZ attribute), 80  
 xy (freud.box.Box attribute), 18  
 xz (freud.box.Box attribute), 18

## Y

Y (freud.pmft.PMFTXY2D attribute), [74](#)

Y (freud.pmft.PMFTXYT attribute), [71](#)

Y (freud.pmft.PMFTXYZ attribute), [80](#)

yz (freud.box.Box attribute), [18](#)

## Z

Z (freud.pmft.PMFTXYZ attribute), [81](#)