freud Documentation

Release 0.6.4

The Regents of the University of Michigan

Feb 07, 2018

Contents

"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 to view/clone the source; post issues or bugs to our issue tracker ; and ask questions and discuss on our forum

CHAPTER 1

Contents:

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 download the documentation in the downloads section, or you may build the documentation yourself:

Building the documentation

Documentation written in sphinx, not doxygen. Please install sphinx:

```
$: conda install sphinx
```

OR

```
$: pip install sphinx
```

To view the full documentation run the following commands in the source directory:

Linux

```
$: cd doc
$: make html
$: firefox build/html/index.html
```

Mac

```
$: cd doc
$: make html
$: open build/html/index.html
```

If you have latex and/or pdflatex, you may also build a pdf of the documentation:

Linux

```
$: cd doc
$: make latexpdf
$: xdg-open build/latex/freud.pdf
```

Mac

```
$: cd doc
$: make latexpdf
$: open build/latex/freud.pdf
```

1.2.3 Installation

Install freud via conda, glotzpkgs, or compile from source.

Conda install

To install freud with conda, make sure you have the glotzer channel and conda-private channels installed:

```
$: conda config --add channels glotzer
$: conda config --add channels file:///nfs/glotzer/software/conda-private
```

Now, install freud

```
$: conda install freud
# you may also install into a new environment
$: conda create -n my_env python=3.5 freud
```

glotzpkgs install

Please refer to the official 'glotzpkgs <http://glotzerlab.engin.umich.edu/glotzpkgs/>'_ documentation

Make sure you have a working glotzpkgs env.

```
# install from provided binary
$: gpacman -S freud
# installing your own version
$: cd /pth/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's 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)
- cython (not required, but a correct _freud.cpp file must be present to compile)
- tbb
- cmake
- icu (because of boost for now)

You may either make a build directory *inside* the freud source directory, or create a separate build directory somewhere on your system:

```
$: mkdir /pth/to/build
$: cd /pth/to/build
$: ccmake /pth/to/freud
# adjust settings as needed, esp. ENABLE_CYTHON=ON
$: make install -j6
# enjoy
```

By default, freud installs to the USER_SITE directory. Which is in *~/.local* on linux and in *~/Library* on mac. *USER_SITE* is on the python search path by default, there is no need to modify *PYTHONPATH*.

To run out of the build directory, run *make -j20* instead of *make install -j20* 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. See http://pythontesting.net/framework/nose/nose-introduction/ for an introduction to writing nose tests.

```
# Install nose if necessary
$: conda install nose
# run tests
$: cd source
$: nosetests
```

1.3 freud modules

Modules in freud may be imported individually or as a whole as part of a Python script.

Several modules are available currently. Feel free to contribute your own modules to the source.

1.3.1 Bond Module

The bond modules 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 bonding 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: θ):**
  - X, Y
  - X, Y, θ_2
  - r, θ_1, θ_2
- 3D particle coordinates -> X, Y, Z

Bonding Analysis

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

Analyze the bonds as calculated by freud's Bonding modules.

Determines 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 – lifetime of bonds :rtype: `numpy.ndarray`, shape=($N_{particles}$, varying), dtype= `numpy.uint32`

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*)

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*)

Returns lifetime of bonds

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

getTransitionMatrix (*self*)

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

Returns number of bonds

Return type unsigned int

num_frames
Get number of frames calculated

Returns number of frames

Return type unsigned int

num_particles
Get number of particles being tracked

Returns number of particles

Return type unsigned int

overall_lifetimes
return – lifetime of bonds :rtype: `numpy.ndarray`, shape=($N_{particles}$, varying), dtype= `numpy.uint32`

transition_matrix
return – transition matrix :rtype: `numpy.ndarray`

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 – particle bonds :rtype: `numpy.ndarray`

box
Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box()*

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*)

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

Returns list_map

Return type dict

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

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

Returns list_map

Return type dict

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

Coordinate System: x, y, θ_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 – particle bonds :rtype: `numpy.ndarray`

box

Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box()*

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*)

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

Returns list_map

Return type dict

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

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

Returns list_map

Return type dict

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

Coordinate System: r, θ_1, θ_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 – particle bonds :rtype: numpy.ndarray

box
Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box()*

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*)

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

Returns list_map

Return type dict

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

rev_list_map

Get the dict used to map list idx to bond idx

Returns list_map

Return type dict

```
>>> bond_idx = list_map[list_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 – particle bonds :rtype: numpy.ndarray

box

Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box()*

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*)

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

Returns list_map

Return type dict

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

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

Returns list_map

Return type dict

```
>>> bond_idx = list_map[list_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: Carl Simon Adorf <*csadorf@umich.edu*>

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.

classmethod cube(*L*)
Construct a cubic box.

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

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

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

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

to_matrix()
Returns the box matrix (3x3).

to_tuple()
Returns the box as named tuple.

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, 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>

Parameters

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

- **rcut** (*float*) – Particle distance cutoff

Note: 2D: Cluster properly handles 2D boxes. As with everything else in freud, 2D points must be passed in as 3 component vectors $(x, y, 0)$. Failing to set 0 in the third component will lead to undefined behavior.

---

box
Return the stored freud Box

Returns freud Box

Return type *freud.box.Box*

cluster_idx
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`

cluster_keys
Returns the keys contained in each cluster

Returns list of lists of each key contained in clusters

Return type list

computeClusterMembership (*self*, *keys*)
Compute the clusters with key membership

Loops overa 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*)
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

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 :return: number of particles :rtype: int

num_clusters

Returns the number of clusters

Returns number of clusters

Return type int

num_particles

Returns the number of particles

Returns number of particles

Return type int

class freud.cluster.**ClusterProperties** (*box*)

Routines for computing properties of point clusters

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

- Center of mass
- Gyration radius tensor

m_cluster_com stores the computed center of mass for each cluster (properly handling periodic boundary conditions, of course) as a numpy.ndarray, shape= $(N_{clusters}, 3)$.

m_cluster_G stores a 3×3 G tensor for each cluster. Index cluster c, element j, i with the following: m_cluster_G[c*9 + j*3 + i]. The tensor is symmetric, so the choice of i and j are irrelevant. This is passed back to python as a $N_{clusters} \times 3 \times 3$ numpy array.

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

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

box

Return the stored freud Box

Returns freud Box

Return type *freud.box.Box*

cluster_COM

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

cluster_G

Returns the cluster G tensors computed by the last call to computeProperties

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

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

cluster_sizes
Returns the cluster sizes computed by the last call to computeProperties

Returns numpy array of sizes of each cluster

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

computeProperties (*self*, *points*, *cluster_idx*)
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 compute 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) – Index of which cluster each point belongs to

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

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 numpy array of 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

Returns number of clusters

Return type int

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.

2D: CorrelationFunction properly handles 2D boxes. As with everything else in freud, 2D points must be passed in as 3 component vectors x,y,0. Failing to set 0 in the third component 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 <mspells@umich.edu>

Parameters

- **r_max** (*float*) – distance over which to calculate
- **dr** (*float*) – bin size

R

return – values of bin centers :rtype: `numpy.ndarray`, shape=(N_{bins}), dtype= `numpy.float32`

RDF

return – expected (average) product of all values at a given radial distance :rtype: `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

Returns freud Box

Return type *freud.box.Box*

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
return – counts of each histogram bin :rtype: numpy.ndarray, shape=(N_{bins}), dtype= numpy.int32

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 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.

2D: CorrelationFunction properly handles 2D boxes. As with everything else in freud, 2D points must be passed in as 3 component vectors x,y,0. Failing to set 0 in the third component 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 <mspells@umich.edu>

Parameters

- **r_max** (*float*) – distance over which to calculate
- **dr** (*float*) – bin size

R

return – values of bin centers :rtype: `numpy.ndarray`, shape=(N_{bins}), dtype= `numpy.float32`

RDF

return – expected (average) product of all values at a given radial distance :rtype: `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.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

Returns freud Box

Return type *freud.box.Box*

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
return – counts of each histogram bin :rtype: numpy.ndarray, shape=(N_{bins}), dtype= numpy.int32

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*)

Returns values of bin centers

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

getRDF (*self*)

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>

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

Returns freud Box

Return type *freud.box.Box*

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
return – Image (grid) with values of gaussian :rtype: `numpy.ndarray`, shape=(w_x, w_y, w_z), dtype= `numpy.float32`

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.

2D: RDF properly handles 2D boxes. Requires the points to be passed in [x, y, 0]. Failing to z=0 will lead to undefined behavior.

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

Parameters

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

box

Get the box used in the calculation

Returns freud Box

Return type `freud.box.Box`

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

return – Density array for each particle :rtype: `numpy.ndarray`, shape=($N_{particles}$), dtype= `numpy.float32`

getBox (*self*)

Returns freud Box

Return type `freud.box.Box`

getDensity (*self*)

Returns Density array for each particle

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

getNumNeighbors (*self*)

Returns Number of neighbors for each particle

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

num_neighbors

return – Number of neighbors for each particle :rtype: `numpy.ndarray`, shape=($N_{particles}$), dtype= `numpy.float32`

Radial Distribution Function

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

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 rmax, dr in the constructor. rmax sets the maximum distance at which to calculate the $g(r)$ while dr determines the step size for each bin.

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

Note: 2D: RDF properly handles 2D boxes. Requires the points to be passed in [x, y, 0]. Failing to z=0 will lead to undefined behavior.

Parameters

- **rmax** (*float*) – maximum distance to calculate
- **dr** (*float*) – distance between histogram bins

R

return – values of bin centers :rtype: `numpy.ndarray`, shape=(N_{bins}), dtype= `numpy.float32`

RDF

return – expected (average) product of all values at a given radial distance :rtype: `numpy.ndarray`, shape=(N_{bins}), dtype= `numpy.float64`

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

Returns freud Box

Return type *freud.box.Box*

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*)

Returns histogram of cumulative rdf values

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

getR (*self*)

Returns values of the histogram bin centers

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

getRDF (*self*)

Returns histogram of rdf values

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

n_r

return – histogram of cumulative rdf values :rtype: numpy.ndarray, shape=(N_{bins}, 3), dtype= numpy.float32

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>

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*)

Returns number of elements in the array

Return type unsigned int

num_elements

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

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>

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*)

Returns number of elements in the array

Return type unsigned int

num_elements

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

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 <mspells@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 $S\left(\vec{q}\right) = C_0 \left|\sum_{m=1}^{N} \exp i\vec{q} \cdot \vec{r_i}\right|^2$ where C_0 is a scaling constant chosen so that $S\left(0\right) = 1$, 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 $h, k, l : \left[-g, g\right]$ and L_x, L_y, L_z are the box lengths in each direction.

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

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

`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,c] 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 $S\left(q\right)$ 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 $S\left(|q|\right)$ 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 to real q values in simulation units.

getPeakDegeneracy(*cut*)

Get a dictionary of peaks indexed by q^2

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

Returns a dictionary with key q^2 and each element being a list of peaks

Return type `dict`

getPeakList(*cut*)

Get a list of peaks in the structure factor

Parameters **cut** – All $S\left(q\right)$ values greater than cut will be counted as peaks

Returns peaks, q as lists

Return type `list`

getSvsQ()

Get a list of all $S\left(|q|\right)$ values vs q^2

Returns S, qsquared

Return type `numpy.ndarray`

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

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

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

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=($N_{particles}$, 3), dtype= numpy.float32) – array of particle positions in nm
- **orientation** (numpy.ndarray, shape=($N_{particles}$, 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** (class) – 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, λ, affecting the scattering density $\rho(r)$, $S_{lambda}(k) == \lambda^3 * S(\lambda * k)$

class freud.kspace.**FTsphere**
Fourier transform for sphere

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

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_{verts}, 3), dtype= numpy.float32) – list of vertices
- **facets** (numpy.ndarray, shape=(N_{facets}, N_{verts}), dtype= numpy.float32) – list of facets
- **norms** (numpy.ndarray, shape=(N_{facets}, 3), dtype= numpy.float32) – list of norms
- **d** (numpy.ndarray, shape=(N_{facets}), dtype= numpy.float32) – list of d values
- **areas** (numpy.ndarray, shape=(N_{facets}), 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 calcular $S(i)$

Spoly3D (*k*)
Calculate Fourier transform of polyhedron

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

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** (np.complex) – 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** (np.complex) – complex valued amplitude used to generate spot intensity

set_sigma(*sigma*)
Define Gaussian

Parameters **sigma** (*float*) – width of the Guassian 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, *dot(g[i], a[j]) = 2 * pi * 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 $log(n_{bonds})$ time using `find_first_index()`.

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

Note: Typically, in python you will only manipulate a `freud.locality.NeighborList` object that you receive from a neighbor search algorithm, such as `freud.locality.LinkCell` and `freud.locality.NearestNeighbors`.

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 ourself; otherwise, return a copy of ourself.

filter(*self, filt*)

Removes bonds that satisfy a boolean criterion.

Parameters **filt** – Boolean-like array of bonds to keep (True => 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** – reference points to use for filtering

- **points** – target points to use for filtering
- **rmax** – maximum bond distance in the resulting neighbor list
- **rmin** – 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 index >=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
Returns the reference point indices from the last set of points we were evaluated with. This array is read-only to prevent breakage of find_first_index.

index_j
Returns the target point indices from the last set of points we were evaluated with. This array is read-only to prevent breakage of find_first_index.

neighbor_counts
Returns 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 we were evaluated with.

segments
Returns 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 we were evaluated with.

weights
Returns the per-bond weights from the last set of points we were 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: *freud.locality.LinkCell* supports 2D boxes; in this case, make sure to set the z coordinate of all points to 0.

Example:

```
# assume we have position as 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[0])
    # list of cell's neighboring cells
    cellNeighbors = lc.getCellNeighbors(cell)
    # iterate over neighboring cells (including our own)
    for neighborCell in cellNeighbors:
        # iterate over particles in each neighboring cell
        for neighbor in lc.itercell(neighborCell):
            pass # do something with neighbor index

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

box

return – freud Box :rtype: *freud.box.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= $(N_{refpoints}, 3)$, dtype= numpy.float32) – reference point coordinates
- **points** (numpy.ndarray, shape= $(N_{points}, 3)$, dtype= numpy.float32) – point coordinates
- **exlude_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= $(N_{refpoints}, 3)$, dtype= numpy.float32) – reference point coordinates
- **points** (numpy.ndarray, shape= $(N_{points}, 3)$, dtype= numpy.float32) – point coordinates
- **exlude_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*)

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= $(N_{neighbors})$, dtype= numpy.uint32

getNumCells (*self*)

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
Returns the neighbor list stored by this object, generated by *compute*.

num_cells
return – the number of cells in this box :rtype: unsigned int

NearestNeighbors

class freud.locality.**NearestNeighbors** (*rmax*, *n_neigh*)
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: rmax will be expanded to find requested number of neighbors. If rmax increases to the point that a cell list cannot be constructed, a warning will be raised and neighbors found will be returned

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

Parameters

- **rmax** (*float*) – Initial guess of a distance to search within to find N neighbors
- **n_neigh** (*unsigned int*) – Number of neighbors to find for each point
- **scale** (*float*) – multiplier by which to automatically increase rmax value by if 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

Example:

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

UINTMAX
return – value of C++ UINTMAX used to pad the arrays :rtype: unsigned int

box
return – freud Box :rtype: `freud.box.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`) – coordinated of reference points
- **points** (`numpy.ndarray`, shape=($N_{particles}$, 3), dtype= `numpy.float32`) – coordinates of points
- **exlude_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*)

Returns freud Box

Return type `freud.box.Box`

getNRef (*self*)

Returns the number of particles this object found neighbors of

Return type unsigned int

getNeighborList (*self*)
Return the entire neighbors 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 to fetch the neighboring points of

getNumNeighbors (*self*)

Returns the number of neighbors this object will find

Return type unsigned int

getRMax (*self*)
Return the current neighbor search distance guess :return: nearest neighbors search radius :rtype: float

getRsq (*self*, *unsigned int i*)
Return the Rsq values for 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 of the N nearest neighbors

Returns Neighbor List

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

getRsqList (*self*)
Return the entire Rsq values list

Returns Rsq 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
return – the number of particles this object found neighbors of :rtype: unsigned int

nlist
Returns the neighbor list stored by this object, generated by *compute*.

num_neighbors
return – the number of neighbors this object will find :rtype: unsigned int

r_max
Return the current neighbor search distance guess :return: nearest neighbors search radius :rtype: 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 requested number of neighbors. If rmax increases to the point that a cell list cannot be constructed, a warning will be raised and neighbors 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 :param rmax: nearest neighbors search radius :type rmax: float

wrapped_vectors

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`

1.3.9 PMFT Module

The PMFT Module allows for the calculation of the Potential of Mean Force and Torque (PMFT) *[Cit2]* 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: θ):**
  - X, Y
  - X, Y, θ_2
  - r, θ_1, θ_2
- 3D particle coordinates -> X, Y, Z

Coordinate System: x, y, θ_2

class `freud.pmft.`**`PMFTXYT`**(*x_max*, *y_max*, *n_x*, *n_y*, *n_t*)

Computes the PMFT *[Cit2]* 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, θ listed in the x, y, and t arrays.

The values of x, y, t to compute the pcf at are controlled by x_max, y_max and n_bins_x, n_bins_y, n_bins_t parameters to the constructor. 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: This calculation is defined for 2D systems only. However particle positions are still required to be (x, y, 0)

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.

Returns PCF

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

PMFT
Get the positional correlation function.

Returns PCF

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

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

Returns bin centers of t-dimension of histogram

Return type numpy.ndarray, shape= (N_θ), dtype= numpy.float32

X
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

Y
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

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.

Returns Bin Counts

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

box
Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box()*

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_{θ}

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_θ, 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_θ, 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

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_θ), 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

Returns Inverse Jacobian

Return type float

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

Returns N_θ

Return type unsigned int

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

Returns N_x

Return type unsigned int

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

Returns N_y

Return type unsigned int

r_cut
Get the r_cut value used in the cell list

Returns r_cut

Return type float

reducePCF (*self*)
Reduces the histogram in the values over N processors to a single histogram. This is called automatically by *freud.pmft.PMFTXYT.getPCF()*.

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]* 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: This calculation is defined for 2D systems only.

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.

Returns PCF

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

PMFT
Get the positional correlation function.

Returns PCF

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

X
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

Y

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`

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.

Returns Bin Counts

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

box

Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box()*

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_y), dtype= numpy.float32

getPMFT (*self*)
Get the Potential of Mean Force and Torque.

Returns PMFT

Return type numpy.ndarray, shape= (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

jacobian
Get the jacobian used in the pmft

Returns Inverse Jacobian

Return type float

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

Returns N_x

Return type unsigned int

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

Returns N_y

Return type unsigned int

r_cut
Get the r_cut value used in the cell list

Returns r_cut

Return type float

reducePCF(*self*)
Reduces the histogram in the values over N processors to a single histogram. This is called automatically by *freud.pmft.PMFTXY2D.getPCF()*.

resetPCF(*self*)
Resets the values of the pcf histograms in memory

Coordinate System: r, θ_1, θ_2

class freud.pmft.**PMFTR12**(*r_max*, *n_r*, *n_t1*, *n_t2*)
Computes the PMFT *[Cit2]* 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, θ_1, θ_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. rmax 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: This calculation is defined for 2D systems only. However particle positions are still required to be (x, y, 0)

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.

Returns PCF

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

PMFT

Get the positional correlation function.

Returns PCF

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

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

Returns bin centers of T1-dimension of histogram

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

T2

Get the array of T2-values for the PCF histogram

Returns bin centers of T2-dimension of histogram

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

Returns Bin Counts

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

box

Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box()*

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= $(N_r, N_{\theta 1}, N_{\theta 2})$, 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= $(N_r, N_{\theta 1}, N_{\theta 2})$, 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_{θ_1}

Return type unsigned int

getNBinsT2 (*self*)

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

Returns N_{θ_2}

Return type unsigned int

getPCF(*self*)
Get the positional correlation function.

Returns PCF

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

getPMFT(*self*)
Get the Potential of Mean Force and Torque.

Returns PMFT

Return type `numpy.ndarray`, shape= $(N_r, N_{\theta 1}, N_{\theta 2})$, 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 array of T2-values for the PCF histogram

Returns bin centers of T2-dimension of histogram

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

n_bins_T1
Get the number of bins in the T1-dimension of histogram

Returns N_{θ_1}

Return type unsigned int

n_bins_T2
Get the number of bins in the T2-dimension of histogram

Returns N_{θ_2}

Return type unsigned int

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

Returns N_r

Return type unsigned int

r_cut
Get the r_cut value used in the cell list

Returns r_cut

Return type float

reducePCF (*self*)
Reduces the histogram in the values over N processors to a single histogram. This is called automatically by `freud.pmft.PMFTR12.getPCF()`.

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]* 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: This calculation is defined for 3D systems only.

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.

Returns PCF

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

PMFT
Get the positional correlation function.

Returns PCF

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

X

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`

Y

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`

Z

Get the array of z-values for the PCF histogram

Returns bin centers of z-dimension of histogram

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

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.

Returns Bin Counts

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

box

Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box()*

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`

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

Returns Inverse Jacobian

Return type float

n_bins_X

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

Returns N_x

Return type unsigned int

n_bins_Y

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

Returns N_y

Return type unsigned int

n_bins_Z

Get the number of bins in the z-dimension of histogram

Returns N_z

Return type unsigned int

reducePCF (*self*)

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

resetPCF (*self*)

Resets the values of the pcf histograms in memory

1.3.10 Order Module

The order module contains functions which deal with the order of the system

Bond Order

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

Compute the bond order diagram for the system of particles.

Available Modues of Calculation: * If mode=bod (Bond Order Diagram): 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. This is the default.

- If mode=lbod (Local Bond Order Diagram): Create the 2D histogram containing the number of bonds formed, rotated into the local orientation of the central particle, through the surface of a unit sphere based on the azimuthal (θ) and polar (ϕ) angles.
- If mode=obcd (Orientation Bond Correlation Diagram): Create the 2D histogram containing the number of bonds formed, rotated by the rotation that takes the orientation of neighboring particle j to the orientation of each particle i, through the surface of a unit sphere based on the azimuthal (θ) and polar (ϕ) angles.
- If mode=oocd (Orientation Orientation Correlation Diagram): Create the 2D histogram containing the directors of neighboring particles ($\hat{z}$ rotated by their quaternion), rotated into the local orientation of the central particle, through the surface of a unit sphere based on the azimuthal (θ) and polar (ϕ) angles.

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

Parameters

- **r_max** (*float*) – distance over which to calculate
- **k** (*unsigned int*) – order parameter i. to be removed
- **n** (*unsigned int*) – number of neighbors to find
- **n_bins_t** (*unsigned int*) – number of 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
return – bond order :rtype: `numpy.ndarray`, shape= (N_{ϕ}, N_{θ}), dtype= `numpy.float32`

box
Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box*

compute (*self*, *box*, *ref_points*, *ref_orientations*, *points*, *orientations*, *str 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*)

Returns bond order

Return type `numpy.ndarray`, shape= (N_{ϕ}, N_{θ}), 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_{ϕ}

Return type unsigned int

getNBinsTheta (*self*)
Get the number of bins in the Theta-dimension of histogram

Returns N_{θ}

Return type unsigned int

getPhi (*self*)

Returns values of bin centers for Phi

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

getTheta (*self*)

Returns values of bin centers for Theta

Return type numpy.ndarray, shape= (N_θ), 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 taking Spherical Harmonics of the bond order diagram, which is the spherical analogue of Fourier Transforms.

Cubatic Order Parameter

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

Compute the Cubatic Order Parameter *[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 OP

Parameters

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

get_cubatic_order_parameter (*self*)

Returns Cubatic Order parameter

Return type float

get_cubatic_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

Hexatic Order Parameter

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

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

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

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

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

Note: 2D: This calculation is defined for 2D systems only. However particle positions are still required to be (x, y, 0)

Module author: Eric Harper <harperic@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)

Note: While k is a float, this is due to its use in calculations requiring floats. Passing in non-integer values will result in undefined behavior

box
Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box*

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 float

Note: While k is a float, this is due to its use in calculations requiring floats. Passing in non-integer values will result in undefined behavior

getNP (*self*)
Get the number of particles

Returns $N_{particles}$

Return type unsigned int

getPsi (*self*)

Returns order parameter

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

k
Get the symmetry of the order parameter

Returns k

Return type float

Note: While k is a float, this is due to its use in calculations requiring floats. Passing in non-integer values will result in undefined behavior

num_particles
Get the number of particles

Returns $N_{particles}$

Return type unsigned int

psi
return – order parameter :rtype: `numpy.ndarray`, shape= $(N_{particles})$, dtype= `numpy.complex64`

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 <mspells@umich.edu>

Parameters

- **num_neighbors** (*unsigned int*) – Maximum number of neighbors to compute descriptors for
- **lmax** – Maximum spherical harmonic l to consider
- **rmax** (*float*) – Initial guess of the maximum radius to looks for neighbors
- **negative_m** – 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

- **points_ref** (`numpy.ndarray`, shape= $(N_{particles}, 3)$, dtype= `numpy.float32`) – source points to calculate the order parameter
- **num_neighbors** – Number of neighbors to compute with
- **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= $(N_{bonds}, \text{SphWidth})$, dtype= `numpy.complex64`

l_max
Get the maximum spherical harmonic l to calculate for

Returns l

Return type unsigned int

num_neighbors
Get the number of neighbors

Returns $N_{neighbors}$

Return type unsigned int

num_particles
Get the number of particles

Returns $N_{particles}$

Return type unsigned int

r_max
Get the cutoff radius

Returns r

Return type float

sph

Get a reference to the last computed spherical harmonic array

Returns order parameter

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

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

Returns freud Box

Return type *freud.box.Box*

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

Returns order parameter

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

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

Returns $N_{particles}$

Return type unsigned int

Local Q_l

class freud.order.**LocalQl**(*box*, *rmax*, *l*, *rmin*)

LocalQl(box, rmax, l, rmin=0) Compute the local Steinhardt rotationally invariant Ql *[Cit4]* order parameter for a set of points.

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

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 Ql order parameter for a set of points:

- Variation of the Steinhardt Ql 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>

param box simulation box

param rmax Cutoff radius for the local order parameter. Values near first minima of the rdf are recommended

param l Spherical harmonic quantum number l. Must be a positive number

param rmin can look at only the second shell or some arbitrary rdf region

type box *freud.box.Box()*

type rmax float

type l unsigned int

type rmin float

Ql

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

Returns order parameter

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

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.

Returns order parameter

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

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.

Returns order parameter

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

box
Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box*

compute (*self*, *points*, *nlist=None*)
Compute the local rotationally invariant Ql 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 Ql 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 Ql 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 Ql 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 Ql for each particle. Returns NaN instead of Ql 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.

Returns order parameter

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

num_particles
Get the number of particles

Returns $N_{particles}$

Return type unsigned int

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 Ql order parameter [Cit4] for a set of points.

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

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 Ql order parameter for a set of points:

- Variation of the Steinhardt Ql 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>

param box simulation box

param rmax Cutoff radius for the local order parameter. Values near first minima of the rdf are recommended

param l Spherical harmonic quantum number l. Must be a positive number

param kn number of nearest neighbors. must be a positive integer

type box `freud.box.Box`

type rmax float

type l unsigned int

type kn unsigned int

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

Compute the local rotationally invariant Ql 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 Ql 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 Ql 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 Ql 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>

param box simulation box

param rmax Cutoff radius for the local order parameter. Values near first minima of the rdf are recommended

param l Spherical harmonic quantum number l. Must be a positive number

type box *freud.box.Box()*

type rmax float

type l unsigned int

Ql

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

Returns order parameter

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

Wl

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`

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.

Returns order parameter

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

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.

Returns order parameter

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

box

Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box*

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

Compute the local rotationally invariant Ql order parameter.

Parameters

- **points** – 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 Ql order parameter.

Parameters

- **points** – 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** – 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** – 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.

Returns order parameter

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

num_particles
Get the number of particles

Returns $N_{particles}$

Return type unsigned int

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>

param box simulation box

param rmax Cutoff radius for the local order parameter. Values near first minima of the rdf are recommended

param l Spherical harmonic quantum number l. Must be a positive number

param kn Number of nearest neighbors. Must be a positive number

type box *freud.box.Box*

type rmax float

type l unsigned int

type kn unsigned int

compute (*self*, *points*, *nlist=None*)
Compute the local rotationally invariant Ql 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 Ql 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 Ql 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 Ql 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>

param box simulation box

param rmax Cutoff radius for the local order parameter. Values near first minima of the rdf are recommended

param Qthreshold 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)

param Sthreshold 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)

param l Choose spherical harmonic Q_l. Must be positive and even.

type box *freud.box.Box()*

type rmax float

type Qthreshold float

type Sthreshold unsigned int

type l unsigned int

Ql_dot_ij
Get a reference to the number of connections per particle

Returns clusters

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

Ql_mi
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`

box
Get the box used in the calculation

Returns freud Box

Return type *freud.box.Box*

cluster_sizes
Return the sizes of all clusters

Returns largest cluster size

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

clusters
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`

compute (*self*, *points*, *nlist=None*)
Compute the local rotationally invariant Ql 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 Ql 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 Ql 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 compute sol-liq 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 compute sol-liq first

Returns largest cluster size

Return type unsigned int

num_connections
Get a reference to the number of connections per particle

Returns clusters

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

num_particles
Get the number of particles

Returns $N_{particles}$

Return type unsigned int

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>

param box simulation box

param rmax Cutoff radius for the local order parameter. Values near first minima of the rdf are recommended

param Qthreshold 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)

param Sthreshold 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)

param l Choose spherical harmonic Q_l. Must be positive and even.

param kn Number of nearest neighbors. Must be a positive number

type box `freud.box.Box`

type rmax float

type Qthreshold float

type Sthreshold unsigned int

type l unsigned int

type kn unsigned int

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

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*, *nlist=None*)
Determine clusters of particles with matching environments.

Parameters

- **points** (`numpy.ndarray`, shape= $(N_{particles}, 3)$, dtype= `numpy.float32`) – particle positions
- **threshold** (*float*) – maximum magnitude of the vector difference between two vectors, below which you call them 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** – if true, do an exhaustive search wherein you compare the environments of every single pair of particles in the simulation. If false, only compare the environments of neighboring particles.

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

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 you call them 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 you call them 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= $(N_{particles}, 3)$, 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= $(N_{particles}, 3)$, dtype= `numpy.float32`), map[int, int]]

num_clusters
Get the number of clusters

Returns $N_{clusters}$

Return type unsigned int

num_particles
Get the number of particles

Returns $N_{particles}$

Return type unsigned int

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

Returns the array of vectors

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

Pairing

Note: This module is deprecated is 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

Returns freud Box

Return type *freud.box.Box*

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= $(N_{particles}, 3)$, dtype= `numpy.float32`) – reference points to calculate the local density

- **orientations** (`numpy.ndarray`, shape= $(N_{particles})$, dtype= `numpy.float32`) – orientations to use in computation
- **compOrientations** (`numpy.ndarray`, shape= $(N_{particles})$, 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*)

Returns match

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

getPair (*self*)

Returns pair

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

match
return – match :rtype: `numpy.ndarray`, shape= $(N_{particles})$, dtype= `numpy.uint32`

pair
return – pair :rtype: `numpy.ndarray`, shape= $(N_{particles})$, dtype= `numpy.uint32`

1.4 Design

1.4.1 Vision

The freud library is designed to be:

1. Powerful
2. Flexible
3. Maintainable

Powerful

The amount of data produced by simulations is always increasing. By being powerful, freud allows users to analyze their simulation data as fast as possible so that it can be used in real-time visualization and on-line simulation analysis.

Flexible

The number of simulation packages, analysis packages, and other software packages keeps growing. Rather than attempt to understand and interact with all of these packages, freud achieves flexibility by providing a simple Python interface and making no assumptions regarding data, operating on and returning NumPy arrays to the user.

Maintainable

Code which cannot be maintained is destined for obscurity. In order to be maintainable, freud uses Git for version control; Bitbucket for code hosting and issue tracking; and the PEP8 standard for code, stressing explicitly written code which is easy to read. Additionally, freud employs unit tests to ensure that any changes or new features do not break existing functionality.

1.4.2 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.

1.4.3 Code Guidelines

Code in freud should follow PEP8, as well as the following guidelines. Anything listed here takes precedence over PEP8, but we try to deviate as little as possible from PEP8. *When in doubt, follow the guidelines!*

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).

Python and Cython naming conventions

- Variables should be named using `lower_case_with_underscores`
- Functions and methods should be named using `lowerCaseWithNoUnderscores`
- Classes should be named using `CapWords`

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++ naming conventions

To intuitively distinguish between C++ and Python code, the following conventions should be used:

- Variables are named with `lower_case_with_underscores`
- Functions and methods are named with `lowerCaseWithNoUnderscores`
- Classes are named with `CapWords`

Example:

```
class FreudCPPClass
    {
    FreudCPPClass()
        {
        }
    computeSomeValue(int variable_a, float variable_b)
        {
        // do some things in here
        }
    };
```

Make things explicit, not automatic

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 do this explicitly. This helps avoid issues in debugging and undocumented behavior:

```
#!python
# 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:

```
#!c++
// somewhere deep in a 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
                    }
            }

#!c++
// somewhere deep in a 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:

```
#!python
# this is bad
def heavyLiftingInFreud(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 callC++ForHeavyLifting(positions):
    # check that positions are fine
    c++_heavy_function(computed_array, positions, len(pos))
    return computed_array

#!c++

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

However, 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.

Semicolons in Python

Semicolons should not be used to mark the end of lines in Python

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.
- C++ code should follow Whitesmith's style. An extended set of examples follows:

```
#!c++
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.)
    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 it
↪'s mate
    }
```

- Documentation comments and items broken over multiple lines should be *aligned* with spaces

```
#!c++
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);
```

- TBB sections should use lambdas, not templates

```
#!c++
void someC++Function(float some_var,
                     float other_var)
    {
    // code before parallel section
    parallel_for(blocked_range<size_t>(0,n),
        [=] (const blocked_range<size_t>& r)
            {
            // do stuff
            });
```

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

```
#!c++
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:

```
#!c++
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 120 character limit, leave it all on one line

```
#!c++
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:

```
#!c++
// 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.

```
#!c++
// go from
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)
// to
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)
```

Documentation Comments

- 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*.
- See http://www.sphinx-doc.org/en/stable/index.html
- If you copy an existing file as a template, *DO NOT* simply 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.

1.5 References and Citations

CHAPTER 2

Indices and tables

- genindex
- modindex
- 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. Strong Orientational Coordinates and Orientational Order Parameters for Symmetric Objects. Journal of Physics A: Mathematical and Theoretical 2015, 48, 485201.

[Cit2] van Anders, G. ; Ahmed, N. K. ; Klotsa, D. ; Engel, M. ; Glotzer, S. C. Unified Theoretical Framework for Shape Entropy in Colloids", arXiv:1309.1187.

[Cit3] van Anders, G. ; Ahmed, N. K. ; Smith, R. ; Engel, M. ; Glotzer, S. C. Entropically Patchy Particles, arXiv:1304.7545.

[Cit4] Wolfgan Lechner (2008) (DOI: 10.1063/Journal of Chemical Physics 129.114707)

Index

Symbols

A

B

C

D

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X

Y

Z