Manual

Contents

Introduction

My name is Yuan-Chao Hu who is a PhD student currently in Institute of Physics, Chinese Academy of Sciences. You can reach me by email: ychu0213@gmail.com or visit my web: https://yuanchaohu.github.io/.

This package is designed for who are interested in analyzing the snapshots from molecular dynamics simulations, i.e. by LAMMPS. It is flexible for other computer simulations as long as you change the method of reading coordinates to suitable formats in 'dump.py'. The modules in the package are written in Python3 by importing some high-efficiency modules like Numpy and Pandas. I strongly recommend the user to install Anaconda3 or/and Sublime Text 3 (with properly installed Python and individual packages [by '*pip install package*']) to edit and run the python file. Python program can be run directly in Sublime.

To use the package efficiently, one intelligent way is to write a python script by importing desired modules and functions. In this way, all results can be obtained in sequence with suitable settings.

Read Snapshots

Syntax:

from dump import readdump

classname(inputfile, ndim).functioname()

- classname = readdump
- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- functionname = *read_onefile*

Example:

d = readdump('./dumpfile', 3)

d.read_onefile()

Description:

This module reads the snapshots (or trajectories) from MD simulations. Only x, xs, xu coordinates are acceptable at current stage. The code is suitable for various dimensions. The coordinate queue should be '**id type x y z (…)**'. For example, the code will take two quantities after type (ie. x y) as particle coordinates at two dimension, while for three dimension, it takes three parameters after type (ie. x y z) as particle coordinates. The cases at higher dimensions are similar. If you have other quantities after the coordination, the code will just neglect them. In other words, the code can read the coordinates if there are other parameters in the input file as long as they are sit after the coordinates. If you have 'id type x y z a b c' in the file, x y z will be stored as coordinates while a, b, c will be neglected. After executing the code in Example, all information including TimeStep, Particle Number, Box Lengths, Box Boundaries, Particle Types, Particle Positions is accessible. This module is the basis of the following analysis.

Since reading coordinates are essential for data analysis, I strongly recommend the reader to read and understand this module. If you use other types of input, you can change the format to use the following analysis modules. *For the xs and x types, particle coordinates are warp to the inside of the box by default, which could be changed by hand when necessary*.

Important Notes: All snapshots should be in one file at this stage. This module has been imported in the following analysis modules, so it is not necessary to import it again.

Pair Correlation Functions

Syntax:

from paircorrelationfunctions import gr

gr(inputfile, ndim).functionname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- functionname = *getresults, Unary, Binary, Ternary, Quarternary, Quinary, Senary*
- args = list of arguments to run the function (*outputfile, ppp, rdelta, results_path*)
  outputfile is the filename of outcomes without file path;
  ppp is periodic boundary conditions along different directions, set 1 for yes and 0 for no at one direction. For example, set [1,1,1] for 3D and [1,1] for 2D;
  rdelta is the bin size calculating g(r), the default value is 0.01;
  results_path is the file path of outputfile. The default value is '../../analysis/gr/'

Example:

gr('./dumpfile', 3).getresults(outputfile = 'gr.dat', ppp =[1,1,1], results_path = './gr/')

Please refer to the specific Class/Function lists below when using the functions. You can copy the function below and reset the parameters.

Description:

This module calculates the overall and partial pair correlation functions g(r) for various dimensional systems by giving *ndim*. g(r) is defined as:

$$\mathrm{g}(r) = \frac{1}{N\rho}\sum_{i=1}^{N}\sum_{j\neq i}\langle\delta\left(\vec{r}+\vec{r}_j-\vec{r}_i\right)\rangle$$

where N is particle number, ρ is number density. The code is written referring to (Allen Book).

If you know the particle type number and want to get the returned numpy array of the results, please use functions from *Unary*() to *Senary*() according to your system. In these functions, the results will not only be written to an output file, but also will be returned as a numpy array for further analysis in Python. However, if you just want to get the analysis results in file, it is more convenient to choose the function *getresults*() without worrying about the particle type number. Because the code itself will set the type number based on the input file. However, no numpy arrays will be returned. This module is not limited to cubic systems, but is also suitable for rectangular boxes. In the latter case, g(r) ranges to half of the box length minimum ($L_{min}/2$).

Notes: At the current stage, *Senary*() only calculates the overall g(r).

Class/Function lists in the module (indentation indicates relationship):

Class gr (inputfile, ndim):

getresults (outputfile, ppp, rdelta = 0.01, results_path='../../analysis/gr/');

Unary (outputfile, ppp, rdelta = 0.01, results_path='../../analysis/gr/');

Binary (outputfile, ppp, rdelta = 0.01, results_path='../../analysis/gr/');

Ternary (outputfile, ppp, rdelta = 0.01, results_path='../../analysis/gr/');

Quarternary (outputfile, ppp, rdelta = 0.01, results_path='../../analysis/gr/');

Quinary (outputfile, ppp, rdelta = 0.01, results_path='../../analysis/gr/');

Senary (outputfile, ppp, rdelta = 0.01, results_path='../../analysis/gr/');

References:

Hu et al. Nature Communications, 6: 8310 (2015)

Hu et al. The Journal of Chemical Physics, **145** (10), 104503 (2016)

Hu et al. The Journal of Chemical Physics, **146** (2), 024507 (2017)

Hu et al. Physical Review E, **96** (2), 022613 (2017)

Structure Factors

Syntax:

from structurefactors import sq

sq(inputfile, ndim).functionname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- functionname = *getresults, Unary, Binary, Ternary, Quarternary, Quinary, Senary*
- args = list of arguments to run the function (*outputfile, results_path*)
  outputfile is the filename of outcomes without file path;
  results_path is the file path of outputfile. The default value is '../../analysis/sq/'

from structurefactors import functionname1

functionname1(arg1, arg2)

- functionname1 = *wavevector3d, wavevector2d, choosewavevector(arg1, arg2)*
- arg1 = *Numofq*
- arg2 = ndim (see above)
  Numofq is the considered number of wavenumber. Default is 500

Example:

sq('./dumpfile', 3).getresults(outputfile = 'Sq.dat', results_path = './sq/')

wavevector3d(Numofq = 100)

choosewavevector(Numofq = 100, ndim = 3)

Please refer to the specific Class/Function lists below when using the functions. You can copy the function below and reset the parameters.

Description:

This module calculates the overall and partial structure factors S(q) at different dimensions directly. S(q) is defined as:

$$S(q) = N^{-1}\left\langle \sum_k \sum_j e^{-i\vec{q}\cdot(\vec{r}_k-\vec{r}_j)} \right\rangle$$

where N is particle number. In this code, if the box length L is smaller than 40.0, S(q) will be computed to L; if L is smaller than 80.0, S(q) will be computed to $L/2$; if L is larger than 80.0, S(q) will be computed to $L/4$. This aims to save the computer time and can be changed in the source code. The module now is only applicable for cubic systems.

If you know the particle type number and want to get the returned numpy array of the results, please use functions from *Unary*() to *Senary*() according to your system. In these functions, the results will not only be written to an output file, but also will be returned as a numpy array for further analysis in Python. However, if you just want to get the analysis results in file, it is more convenient to choose the function *getresults*() without worrying about the particle type number. Because the code itself will set the type number based on the input file. However, no numpy arrays will be returned.

The module also provides wavenumber design method in functions *wavevector3d, wavevector2d* and *choosewavevector*. The last one is designed to choose ether of the first two by given *ndim*. A numpy array will be returned as [d, a, b, c] where $d = a^2 + b^2 + c^2$ for 3D or [d, a, b] where $d = a^2 + b^2$ for 2D. These functions are useful for further analysis related to 'structure factors' like the four-point dynamic structure factor. If you want to study structure factors at higher dimensions, you can just design the wavevector and add it in the *choosewavevector* function.

Class/Function lists in the module (indentation indicates relationship):

Class sq (inputfile, ndim):

    getresults(outputfile, results_path='../../analysis/sq/');
    Unary(outputfile, results_path='../../analysis/sq/');
    Binary(outputfile, results_path='../../analysis/sq/');
    Ternary(outputfile, results_path='../../analysis/sq/');
    Quarternary(outputfile, results_path='../../analysis/sq/');
    Quinary(outputfile, results_path='../../analysis/sq/');
    Senary(outputfile, results_path='../../analysis/sq/');

wavevector3d(Numofq = 500)

wavevector2d(Numofq = 500)

choosewavevector(Numofq, ndim)

References:

Hu et al. The Journal of Chemical Physics, **146** (2), 024507 (2017)

Hu et al. Physical Review E, **96** (2), 022613 (2017)

Voronoi Tessellation

Syntax:

from Voronoi import cal_voro

cal_voro(inputfile, ndim, radii, ppp, results_path)

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- radii = a dict contains the particles' radii, like {1 : 1.28, 2 : 1.60}
- ppp = set of periodic boundary conditions. '-px', '-py', '-pz' can be used to set along each direction, like ppp = '-px -py' for XY. If three directions are periodic, set ppp = '-p', which is also the default
- results_path is the file path of outputfile. The default value is '../../analysis/sq/

Example:

cal_voro(inputfile = './dump/CuZr.neighbors.lammpstrj', ndim = 3, radii = {1 : 1.0, 2 : 1.0}, ppp = '-p', results_path = './voro/')

Please refer to the specific Class/Function lists below when using the functions. You can copy the function below and reset the parameters.

Description:

This module performs voronoi tessellation by employing the voro++ package. The particles' radii are considered by giving different values in radii. If you want to neglect this consideration, you can just set the radii to be the same over different species. During calculations, the command line used will be printed. Four output files will be generated with headers for each snapshot:

xx.facearea.dat: ID, Number of Neighbor, Voronoi polyhedron face area list
xx.neighbor.dat: ID, Number of Neighbor, Neighbor list
xx.overall.dat: ID, Number of Neighbor, Voronoi Volume, Voronoi face area
xx.voroindex.dat: ID, Voronoi Index (from 0 to 7 faces)
xx is the name of inputfile without format

These outputs are in align with the format needed in the module *ParticleNeighbors*. This module provides important method to define instantaneous neighbors of a particle, which is better than using some distance cutoffs.

Class/Function lists in the module (indentation indicates relationship):

cal_voro(inputfile, ndim, radii, ppp = '-p', results_path = '../../analysis/voro/')

Dynamical Properties

Syntax:

from dynamics import dynamics

dynamics(inputfile, ndim).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- functionname = *total, partial, slowS4, fastS4*
- args = list of arguments to run the function (*outputfile*, *qmax, a, dt, results_path, X4time, X4timeset*)

  outputfile is the filename of outcomes without file path;

  qmax is the q values (usually the first peaks of structure factors) for calculating self-intermediate scattering functions; for the function *total*(), *qmax* is a value, but for the function *partial*(), *qmax* is a list containing the q values of different particle types in sequence;

  a is the cutoff value in the Overlap function Q(t), default is 1.0;

  dt is the timestep in MD simulations, default is 0.002;

  results_path is the file path of outputfile. The default value is '../../analysis/dynamics/';

  X4time is time scale (usually the peak time of the dynamic sysceptibility X4) for calculating four-point dynamic structure factor S4(q) in the function *slowS4*(). (Time Unit);

  X4timeset is similar to *X4time* above but for the function *fastS4*(). If set *X4timeset >0, fastS4*() will use the given value; but if set *X4timeset = 0*, *fastS4*() will use the internal calculated peak time scale of X4 of fast particles. (Time Unit)

Example:

dynamics('./dumpfile', 3).total(outputfile = 'total.dat', qmax = 2.5, results_path = './dynamics/')

Please refer to the specific Class/Function lists below when using the functions. You can copy the function below and reset the parameters.

Description:

This module calculates the dynamical properties at different dimensions, such as:

(1). self-intermediate scattering function $F_s(q,t)$:

$$F_s(q,t) = \frac{1}{N}\left\langle \sum_{j=1}^{N} \exp\left[i\vec{\mathrm{q}} \cdot \left(\vec{\mathrm{r}}_j(t) - \vec{\mathrm{r}}_j(0)\right)\right]\right\rangle$$

(2). $F_s(q,t)$ susceptibility $\chi_4(t)$ ($F_s(q,t)$ is the non-averaged values):

$$\chi_4(t) = N^{-1}[\langle F_s(q,t)^2\rangle - \langle F_s(q,t)\rangle^2]$$

(3). Overlap function $Q(t)$:

$$Q(t) = N^{-1}\langle \sum_{j=1}^{N} \omega\left(|\vec{\mathrm{r}}_j(0) - \vec{\mathrm{r}}_j(t)|\right)\rangle$$

slow particles: where $\omega(r) = 1$ if $r \leq a$ and zero otherwise
fast particles: where $\omega(r) = 1$ if $r \geq a$ and zero otherwise

(4). Dynamic susceptibility $\chi_4(t)$ ($Q(t)$ is the non-averaged values):

$$\chi_4(t) = N^{-1}[\langle Q(t)^2\rangle - \langle Q(t)\rangle^2]$$

(5). mean-square displacements $\langle \Delta r^2(t)\rangle$:

$$\langle \Delta r^2(t)\rangle = \frac{1}{N}\left\langle \sum_{j=1}^{N} \left[\vec{\mathrm{r}}_j(t) - \vec{\mathrm{r}}_j(0)\right]^2 \right\rangle$$

(6). Non-Gaussion parameter $\alpha_2(t)$:

$$\alpha_2(t) = \frac{3\langle \Delta r^4(t)\rangle}{5\langle \Delta r^2(t)\rangle^2} - 1 \ \ (3\mathrm{D}); \ \ \alpha_2(t) = \frac{\langle \Delta r^4(t)\rangle}{2\langle \Delta r^2(t)\rangle^2} - 1 \ \ (2\mathrm{D})$$

To calculate $F_s(q,t)$, a wavenumber is required, which is usually the first peak of corresponding structure factors so you should first calculate the structure factors. If you are only interested in the overall dynamics without considering particle type, choose the function *total*() to calculate the above dynamic properties. The results in a numpy array will be returned. If you are interested in the dynamics of different particle types, choose the function *partial*() to calculate the above dynamic properties. A list containing results of different particle types (in numpy array) in sequence will be returned. The results of different types will be written to separate files indicated by 'Type*i*' in the filename, where *i* is the type number.

(7). Four-point dynamic structure factor $S_4(q;t)$:

$$S_{4,\mathrm{a}}(q;t) = N^{-1}\langle W(\vec{\mathrm{q}},t)W(-\vec{\mathrm{q}},t)\rangle$$

$$S_{4,\mathrm{b}}(q;t) = N^{-1}[\langle W(\vec{\mathrm{q}},t)W(-\vec{\mathrm{q}},t)\rangle - \langle W(\vec{\mathrm{q}},t)\rangle\langle W(-\vec{\mathrm{q}},t)\rangle]$$

$$W(\vec{\mathrm{q}};t) = \sum_{j=1}^{N} \exp\left[i\vec{\mathrm{q}} \cdot \vec{\mathrm{r}}_j(0)\right]\omega\left(\left|\vec{\mathrm{r}}_j(t) - \vec{\mathrm{r}}_j(0)\right|\right)$$

slow particles: where $\omega(r) = 1$ if $r \leq a$ and zero otherwise

fast particles: where $\omega(r) = 1$ if $r \geq a$ and zero otherwise

To calculate $S_4(q;t)$, a time scale to calculate particle mobility is required, which is usually defined as the peak time scale of $\chi_4(t)$. $S_{4,\mathrm{a}}(q;t)$ and $S_{4,\mathrm{b}}(q;t)$ are somehow equivalent, and I recommend to use the former one due to simplicity. In the result file, both of them will be written with explicit header. In this code, S4 for slow and fast particles are calculable with function *slowS4*() and *fastS4*(), respectively. The difference lies in calculating the mobility field as defined above. In the function *fastS4*(), the $Q(t)$ and $\chi_4(t)$ of the fast particles are calculated first. The results will be written in a file with a name 'Dynamics.' in given by the code. The $S_4(q;t)$ results of fast particles will be written to a separate file instead. So please do not worry about the only given output filename.

In this code, if the box length L is smaller than 40.0, S(q) will be computed to L; if L is smaller than 80.0, S(q) will be computed to $L/2$; if L is larger than 80.0, S(q) will be computed to $L/4$. This aims to save computer time and compare with the static structure factors and can be changed in the source code. After calculating S4, the four-point dynamic correlation length ξ_4 can be achieved by fitting the low wavenumber region to the function:

$$S_4(q;\tau_p) = S_4(q=0;\tau_p)/[1+(q\xi_4)^2].$$

(see Hu et al. The Journal of Chemical Physics, **146** (2), 024507 (2017))

Notes: In the 2D case, the module only calculates the absolute dynamics without considering the Mermin-Wagner fluctuations.

Class/Function lists in the module (indentation indicates relationship):

Class dynamics (inputfile, ndim):

total(outputfile, qmax, a = 1.0, dt = 0.002, results_path = '../../analysis/dynamics');

partial(outputfile, qmax, a = 1.0, dt = 0.002, results_path = '../../analysis/dynamics');

slowS4(outputfile, X4time, dt = 0.002, a= 1.0, results_path= '../../analysis/dynamics');

fastS4(outputfile, a=1.0, dt=0.002, X4timeset=0, results_path= '../../analysis/dynamics');

References:

Hu et al. Nature Communications, 6: 8310 (2015)

Hu et al. The Journal of Chemical Physics, **145** (10), 104503 (2016)

Hu et al. The Journal of Chemical Physics, **146** (2), 024507 (2017)

Hu et al. Physical Review E, **96** (2), 022613 (2017)

Cage Relative Dynamics

Syntax:

from cagedynamics import dynamics

dynamics(inputfile, Neighborfile, ndim).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- Neighborfile = neighbor list of the corresponding inputfile
- ndim = dimensionality of the system
- functionname = *total, partial, slowS4, fastS4*
- args = list of arguments to run the function (*outputfile*, *qmax, a, dt, results_path, X4time, X4timeset*)

  outputfile is the filename of outcomes without file path;

  qmax is the *q* values (usually the first peaks of structure factors) for calculating self-intermediate scattering functions; for the function *total*(), *qmax* is a value, but for the function *partial*(), *qmax* is a list containing the *q* values of different particle types in sequence;

  a is the cutoff value in the Overlap function Q(t), default is 1.0;

  dt is the timestep in MD simulations, default is 0.002;

  results_path is the file path of outputfile. The default value is '../../analysis/dynamics/';

  X4time is time scale (usually the peak time of the dynamic sysceptibility X4) for calculating four-point dynamic structure factor S4(q) in the function *slowS4*(). (Time Unit);

  X4timeset is similar to *X4time* above but for the function *fastS4*(). If set *X4timeset >0, fastS4*() will use the given value; but if set *X4timeset = 0*, *fastS4*() will use the internal calculated peak time scale of X4 of fast particles. (Time Unit)

The syntax of this module is similar to the Dynamical Properties, please refer to the above to see details. The only difference lies in considering the relative displacements to the neighbors instead of the absolute ones as:

$$\Delta \boldsymbol{r} = \left[\vec{r}_j(t) - \vec{r}_j(0)\right] - \frac{1}{N_i}\sum_i^{N_i}\left[\vec{r}_i(t) - \vec{r}_i(0)\right]$$

Since we consider cage relative dynamics in this module, the neighbor list is needed as an input (see Voronoi Tessellation). This is important in two dimensional systems where long wavelength fluctuations are prominent, which is also known as Mermin-Wagner effects.

Bond Orientational Order at 3D

Syntax:

from BondOOrder import BOO3D

BOO3D (inputfile, neighborfile, faceareafile).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- neighborfile = neighbor list generated by voro++ analysis or in that format
- faceareafile = facearea list generated by voro++ analysis or in that format
- functionname = *qlQl, sijsmallql, sijlargeQl, GllargeQ, Glsmallq, smallwcap, largeWcap, timecorr*
- args = list of arguments to run the function (*l, ppp, AreaR, outputql, outputQl, outputsij, rdelta, outputgl, outputw, outputW, outputwcap, outputWcap, dt, results_path*)

  output* is the filename of outcomes without file path, please check the specific name according to the function

  l is the degree of spherical harmonics

  ppp is the periodic boundary conditions, set 1 for yes and 0 for no. default [1,1,1]

  AreaR is used to declare whether traditional (AreaR = 0) BOO or voronoi polyhedron face area weighted BOO (AreaR = 1). You can modify the source code accordingly if other weighting methods are wanted. Default 0.

  c is the cutoff in s(i, j) demonstrating whether a bond is crystalline. Default 0.7.

  rdelta is the bin size in calculating bond order spatial correlation Gl, default 0.01.

  dt is the time step of simulations for calculating time correlation of BOO

  results_path is the file path of outputfile. The default value is '../../analysis/BOO/'

Example:

boo = BOO3D(dumpfile = '', Neighborfile = '', faceareafile = '')

boo.qlQl(l = 6, ppp = [1,1,1], AreaR = 0, outputql = 'sq.dat', outputQl = 'bQ.dat', results_path = '')

Please refer to the specific Class/Function lists below when using the functions. You can copy the function below and reset the parameters.

Description:

This module calculates the bond orientational order parameters and their spatial and time correlations for three dimensional systems. Steinhardt et al. defined the BOO of the l-old symmetry as a $2l+1$ vector:

(1). $q_{lm}(i) = \frac{1}{N_i}\sum_0^{N_i} Y_{lm}(\theta(\mathbf{r}_{ij}), \phi(\mathbf{r}_{ij}))$

where the Y_{lm} are the spherical harmonics and N_i the number of bonds of particles i. In the calculations, one uses the rotational invariants defined as (2), (3) and (4). In (3), the first part is the Wigner 3-j symbol. The coarse-grained BOO over the neighbors is defined in (5). The coarse-grained Q_l, W_l, $\widehat{W_l}$ can then be defined in the same way as *w* quantities. One standard method in literature to detect crystal nuclei is calculating the normalized scalar product of the (non-coarse-grained) q_{lm} ($l = 6$). The bond between the center particle and a neighbor is considered crystalline if $s(i,j) > c$ (usually 0.7). A particle is crystalline if the number of crystalline bond is larger than a threshold (usually half of the coordination number). The spatial correlation of bond order can be defined as $G_l(r)$. In (6) and (7), the variable can be replaced by q_{lm} or Q_{lm}. This module calculates *l* from 2 to 10.

(2). $q_l = \sqrt{\frac{4\pi}{2l+1}\sum_{m=-l}^{l}|q_{lm}|^2}$

(3). $w_l = \sum_{m_1+m_2+m_3=0}\begin{pmatrix} l & l & l \\ m_1 & m_2 & m_3 \end{pmatrix} q_{lm_1}q_{lm_2}q_{lm_3}$

(4). $\widehat{w_l} = w_l(\sum_{m=-l}^{l}|q_{lm}|^2)^{-\frac{3}{2}}$

(5). $Q_{lm}(i) = \frac{1}{N_i+1}(q_{lm}(i) + \sum_{j=0}^{N_i} q_{lm}(j))$

(6). $s(i,j) = \frac{\sum_{m=-l}^{l} q_{lm}(i)q_{lm}^*(j)}{\sqrt{\sum_{m=-l}^{l}|q_{lm}(i)|^2}\sqrt{\sum_{m=-l}^{l}|q_{lm}(j)|^2}}$

(7). $G_l(r) = \frac{4\pi}{2l+1}\frac{\sum_{ij}\sum_{m=-l}^{l} Q_{lm}(i)Q_{lm}^*(j)\delta(r_{ij}-r)}{\sum_{ij}\delta(r_{ij}-r)}$

(8). $C_l^*(t) = \frac{4\pi}{2l+1}\langle\sum_i^N\sum_{m=-l}^{l} Q_{lm}^i(t)Q_{lm}^i(0)^*\rangle / \langle\sum_i^N\sum_{m=-l}^{l}\left|Q_{lm}^i(0)\right|^2\rangle$

In these calculations, one important input is the neighbor list of each particle. Currently, the module only accepts data generated by the given module Voronoi Tessellation or in that format. The format of the files must be (particle index, neighbor number, neighbor list):

```
id     cn     neighborlist
1 13 268 58 11 158 50 335 289 296 179 9 131 275 54
2 15 64 305 39 132 56 63 284 36 399 74 173 321 387 291 94
3 14 312 340 62 357 201 146 48 181 283 366 41 92 258 373
4 15 206 257 320 148 180 386 8 265 119 45 314 37 122 381 228
5 14 51 150 42 352 246 326 392 47 368 371 251 159 183 337
6 16 311 356 373 8 119 78 239 198 385 129 306 178 62 358 92 181
7 15 286 193 139 276 192 252 35 336 227 64 378 238 172 216 104
8 14 178 6 119 22 244 148 97 78 4 265 358 122 311 320
9 14 289 58 1 335 296 197 11 127 377 219 285 179 379 382
10 15 227 216 64 305 44 167 321 135 27 84 278 258 302 366 94
```

If the file is neighbor list, the string 'neighborlist' should be in the header so that the list will be stored as integers or else as float numbers. In the module, a function *Voropp*() in a separate module '***ParticleNeighbors'*** is used to read the neighbor list. *Voropp*() is suitable to read data in the above format, such as the voronoi polyhedron face area list of each particle for the face area weighted BOO. It is important to reminder that after using *Voropp*(), the values have been subtracted by 1 because Python counts from 0. This is designed for the convenience in using neighbor list in other modules. If you use other data other than neighbor list, please add 1 to use the correct values. If you have used LAMMPS output voronoi analysis results by using the command line:

compute voro all voronoi/atom neighbors yes
dump name all local N dumpfile index c_voro[1] c_voro[2] c_voro[3]

in the lammps strcipt. A module '***ParseList'*** has been provided to rewrite the dump file to the required format [the same as from Voronoi Tessellation]. In the output file, '0' only indicates no number at that site. Then you can read the generated data for BOO analysis and others. It is good to remind that this code provides the model method to write a numpy array in a file by using python *write*() method. To use this module, please give the python code:

from ParseList import readall
readall(fnfile = 'neighborlistfile', fffile = 'facearealistfile', fread = 'dumpfile', ParticleNumber = , Snapshotnumber =)

To calculate the spherical harmonics at different *l*, a module '***SphericalHarmonics'*** has been given by lots of efforts from $l = 0$ to $l = 10$. The formulas are taken from Wikipedia. By importing different functions inside (from *SphHarm0*() to *SphHarm10*()), you can calculate spherical harmonics with given parameters (theta, phi). The results at a *l* will be returned as a list. This module is powerful to calculate different BOO parameters by choosing a *l* (from 2 to 10). In the ***BondOOrder*** module, corresponding spherical harmonics have been chosen by giving a *l*.

The core part of this module is the function *qlmQlm*() which gives q_{lm} and Q_{lm} values of different particles in complex numbers, actually in numpy array. The results in

different snapshots are stored in a list for each of them, separately. And then they are returned in a tuple as (q_{lm}, Q_{lm}), where q_{lm} and Q_{lm} are types of list consisting of numpy arraies.

In the function *qlQl*(), output file names should be given to get the computation results or it will just be returned in computer memory without writing to files. In the result files, different columns represent results at different snapshots. The row index is the particle index as indicated by 'id'. Function return is similar to the case in *qlmQlm*().

The functions *sijsmallql*() and *sijlargeQl*() calculate $s(i,j)$ based on q_{lm} and Q_{lm}, respectively. Although it is more frequently to use qlm. Two files are written, one is the $s(i,j)$ value of each particle with its every neighbor. Results in different snapshots are written in sequence. Thus, this file is very large akin to the neighbor list file. Since at most 50 neighbors of each particle are considered, if the value in the list is 0, it represents there is no particle. The non-zero value in a row excluding particle index is equal to its neighbor number. These values are helpful to identify which pair or bond is crystalline using different criterion. In the other file, results in different snapshots are also written in sequence. The first column is particle index, and the second is the crystal bond number of a particle according to the given cutoff *c*. The third one shows whether a bond is crystalline (1) or not (0), if the crystal bond number is larger than half of its neighbor number. In this function, the data in the first file is returned.

The functions *Glsmallq*() and *GllargeQ*() compute the spatial correlation of bond orientational order based on q_{lm} and Q_{lm}, respectively, which can then be used to get a static correlation length (see Ref.). Since some values of g(r) at small distance is 0, a warning of dividing zero is given when running, this is OK since numpy operation is used. At these distances, Gl(r)/g(r) is 'nan' in the output file. The results will be returned.

The functions smallwcap() and largeWcap() calculates equation (3) and (4) for (w_l, $\widehat{w_l}$), (W_l, $\widehat{W_l}$) by using q_{lm} and Q_{lm}, respectively. This function is a little bit slow due do large efforts of calculation including use Wigner 3-j symbol. All results will be written to separate files if given a filename and will be returned in a tuple similar to above.

When the area weighted BOO is considered, $q_{lm}(i)$ changes to $q_{lm}(i) = \sum_0^{N_i} \frac{A_j}{A} Y_{lm}(\theta(\mathbf{r}_{ij}), \phi(\mathbf{r}_{ij}))$ where A_j is the polyhedral face area between particle i and j and A is the total face area around i. In this way, a series of BOO parameters can be calculated according to the above.

Importantly, all the output files without given names bellow will not be written unless a name is provided for the data you are interested in.

Class/Function lists in the module (indentation indicates relationship):

Class BOO3D(dumpfile = , Neighborfile = , faceareafile =):

boo.qlQl(l = , ppp = [1,1,1], AreaR = 0, outputql = '', outputQl = '', results_path = '../../analysis/BOO/')

boo.sijsmallql(l = , ppp = [1,1,1], AreaR = 0, c = 0.7, outputql = '', outputsij = '', results_path = '../../analysis/BOO/')

boo.sijlargeQl(l = , ppp = [1,1,1], AreaR = 0, c = 0.7, outputQl = '', outputsij = '', results_path = '../../analysis/BOO/')

boo.GllargeQ(l = , ppp = [1,1,1], rdelta = 0.01, AreaR = 0, outputgl = 'GllargeQ.dat', results_path = '../../analysis/BOO/')

boo.Glsmallq(l = , ppp = [1,1,1], rdelta = 0.01, AreaR = 0, outputgl = 'Glsmallq.dat', results_path = '../../analysis/BOO/')

boo.smallwcap(l = , ppp = [1,1,1], AreaR = 0, outputw = '', outputwcap = '', results_path = '../../analysis/BOO/')

boo.largeWcap(l = , ppp = [1,1,1], AreaR = 0, outputW = '', outputWcap = '', results_path = '../../analysis/BOO/')

timecorr(l = , ppp = [1,1,1], AreaR = 0, dt = 0.002, outputfile = 'timecorr.dat', results_path = '../../analysis/BOO/')

References:

Steinhardt et al. Physical Review B 28, 784 (1983)

Tanaka et al. Nature Communications 3, 974 (2012)

Bond Orientational Order at 2D

Syntax:

from Order2D import BOO2D

BOO2D(inputfile, Neighborfile).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- Neighborfile = Neighbor list file as above
- functionname = *tavephi, spatialcorr, timecorr*
- args = list of arguments to run the function (*outputphi, outputavephi, outputfile, avet, l, ppp, dt, results_path*)
  output* is the filename of outcomes without file path, please refer to individual functions to see details
  avet is the time scale used to average phi. (Time Unit)
  l is the order of BOO, usually from 4 to 8 at 2D. default = 6
  ppp is the periodic boundary conditions. Set 1 for yes and 0 for no. default = [1,1]
  dt is the time step of simulation. Default = 0.002
  results_path is the file path of outputfile. The default value is '../../analysis/sq/'

Example:

BOO2D('./dumpfile', './Neighborfile').tavephi(outputphi = 'phi.dat', outputavephi = 'ave_phi.dat', avet = 1.0, l = 6, ppp = [1, 1], dt = 0.002, results_path = './order2d/')

Please refer to the specific Class/Function lists below when using the functions. You can copy the function below and reset the parameters.

Description:

This module calculates bond orientational order of different degrees at two dimension, such as the known hexatic order φ_6 ($l = 6$). The order parameter is defined as:

(1). $\varphi_l^j = \frac{1}{N_j}\sum_{m=1}^{N_j} e^{il\theta_m^j}$ $(i = \sqrt{-1})$

(2). $\psi_l^j = \mathrm{t}\int_0^t dt\left|\varphi_l^j\right|$ (Time Average)

(3). $\mathrm{g}_l(r) = \frac{L^2}{2\pi r\Delta r N(N-1)}\langle\sum_{j\neq k}\delta\left(r-\left|\vec{r}_{jk}\right|\right)\varphi_l^j\varphi_l^{k*}\rangle$ (Spatial Correlation)

(4). $\mathrm{C}_l(t) = \langle\sum_n \varphi_l^n(t)\varphi_l^n(0)^*\rangle/\langle\sum_n|\varphi_l^n(0)|^2\rangle$ (Time Correlation)

Equation (2) – (4) are different post-processing methods after calculating the order parameter. In the function *tavephi*(), individual file names should be given to do the calculation and write the results.

Class/Function lists in the module (indentation indicates relationship):

BOO2D(dumpfile, Neighborfile):

tavephi(outputphi, outputavephi, avet, l = 6, ppp = [1, 1], dt = 0.002, results_path = '../../analysis/order2d/')

spatialcorr(outputfile, l = 6, ppp = [1, 1], rdelta = 0.01, results_path = '../../analysis/order2d/')

timecorr(outputfile, l = 6, ppp = [1, 1], dt = 0.002, results_path = '../../analysis/order2d/')

References:

Hu et al. Physical Review E, **96** (2), 022613 (2017)