Manual of Molecular Dynamics Simulation Package

Contents

In the front

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 homepage via GitHub: 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*' in Command Prompt (Windows)]) to edit and run the python file. Python program can be run directly in Sublime [*Ctrl* + *B*].

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.

The package is distributed in the hope that it will be helpful, but WITHOUT ANY WARRANTY. Although the codes were benchmarked when developed, the users are responsible for the correctness.

Read Snapshots

Syntax:
from dump import readdump
classname(inputfile, ndim, filetype, moltypes).functioname()

- classname = readdump
- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- filetype = ‘lammps’ (default); ‘lammpscenter’, ‘gsd’; ‘gsd_dcd’
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpsmolecule and lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default (‘’)
- 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 reduced coordinates ***xs*** will be rescaled to the absolute values ***x*** in this module. Therefore, it is highly recommended to use the ***x***-type coordinates. 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** (i.e. **x y**) as particle coordinates at two dimension, while for three dimension, it takes three parameters after **type** (i.e. **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 placed 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.

For orthogonal cells, box lengths are derived from box boundaries from input. A h-matrix (diagonal) is obtained for future analysis to remove periodic boundary conditions. The case of triclinic cells (i.e. under shear) is also included. For the ***xs***-type coordinates, they will be rescaled to ***x*** under some rules (http://lammps.sandia.gov/doc/Section_howto.html#howto-12). The real box lengths are calculated while box bounds represent the low and high boundaries of the coordinates. Please be careful that box boundaries and box lengths are not directly correlated in triclinic cells. In LAMMPS, dump file only contains the box bounds without real box lengths. A h-matrix is also calculated using a lower triangular matrix to remove periodic boundary conditions in future analysis. Therefore, in future analysis, h-matrix is used in both triclinic and orthogonal cells to make the codes general.

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 in orthogonal samples with periodic boundary conditions, particle coordinates are warp to the inside of the box by default, which could be changed by hand when necessary*. In non-periodic boundary conditions, there should be no particles at the outside of the cell.

One way to deal with molecular trajectory is to use the center-of-mass of each molecule. This is the what the function 'lammpscenter 'does. In this mode, we should know what atom type the center of each molecule is. Then it will select out the information of these atoms to represent the molecules. The argument *moltypes* is also important, it should only contain the center atoms. For example, if the system has 5 types of atoms in which 1-3 is one type of molecules and 4-5 is the other, and type 3 and 5 are the centers. Then *moltypes* should be {3: 1, 5: 2}. The keys ([3, 5]) of the dict (*moltypes*) are used to select specific atoms to present the corresponding molecules. The values ([1, 2]) is used to record the type of molecules. In this case, there are two types of molecules which are made by different number of atoms. In this mode, no molecule ID is needed.

The dump files from Hoomd-blue MD engine are also included for calculations. There are two new types: gsd and dcd. Gsd file has all the information with periodic boundary conditions, while dcd only has the particle positions. Normally only gsd file is needed [use the keyword 'gsd']. But if one wants to analyze the dynamical properties, the dcd file should be dumped accompanying the gsd file to get the unwarp coordinates. In this case, using the keyword 'gsd_dcd'. More specifically, all the information about the trajectory will be obtained from the gsd file except the particle positions which will be obtained from the dcd file. Therefore, the dcd and gsd files shall be dumped with the same period or concurrently. Another important point in this case is the file name of gsd and dcd. They should share the same name with the only difference of the last three string, ie. 'gsd 'or 'dcd'. For example, if the file name of gsd is 'dumpfile.gsd 'then the dcd file name must be 'dumpfile.dcd'. To read the hoomd-blue outputs, two new modules should be installed first: i) gsd; ii) mdtraj. These modules are available by conda. Currently, the dump files from Hoomd-blue only support orthogonal box.

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. Temporary snapshot can be generated for analysis of each snapshot.

Variable names:

```
self.TimeStep       = []      #simulation timestep @ each snapshot
self.ParticleNumber = []      #particle's Number @ each snapshot
self.ParticleType   = []      #particle's type @ each snapshot
self.Positions      = []      #a list containing all snapshots, each element is a
snapshot
self.SnapshotNumber = 0       #snapshot number
self.Boxlength      = []      #box length @ each snapshot
self.Boxbounds      = []      #box boundaries @ each snapshot
self.Realbounds     = []      #real box bounds of a triclinic box
self.hmatrix        = []      #h-matrix of the cells
```

ParticleNeighbors

Syntax:
from ParticleNeighbors import functionname
functionname(arguments)

- dumpfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system, default is 3
- filetype = 'lammps' (default); 'lammpscenter', 'gsd'; 'gsd_dcd'
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpsmolecule and lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default ('')
- ***ppp*** is the periodic boundary conditions, set 1 for yes and 0 for no. default [1,1,1]
- ***N*** = the desired nearest neighbor number, default is 12
- r_cut = the global cutoff distance to select the nearest neighbors
- see output file from the listed functions.

Example:
Nnearests (inputfile, fnfile = 'neighborlist.dat')

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 provides a function Voropp() to read the neighbor list and facearea list either from Voronoi tessellation or other resources, and two functions to determine the nearest neighbors. One is N neighbors that are getting the N nearest neighbors such as N = 12. The other is by setting up a cutoff distance r_cut. The particles with the cutoff distance from the center are considered as its nearest neighbors. Usually r_cut can be obtained from the pair correlation functions. In these two methods, only neighbor list can be generated, but a fake facearea list file can be written if needed by setting fffile = 'sth'. This is convenient for the calculation of bond orientational order using this package. The output neighbor/facearea list file is directly compatible with the package.

Class/Function lists in the module (indentation indicates relationship):
Voropp(f, ParticleNumber)

Nnearests(dumpfile, ndim = 3, filetype = 'lammps', moltypes = '', N = 12, ppp = [1,1,1], fnfile = ' neighborlist.dat ')

cutoffneighbors(dumpfile, r_cut, ndim = 3, filetype = 'lammps', moltypes = '', ppp = [1,1,1], fnfile = ' neighborlist.dat ')

Pair Correlation Functions

Syntax:
from paircorrelationfunctions import gr
gr(inputfile, ndim, filetype, moltypes).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- filetype = 'lammps' (default); 'gsd'; 'gsd_dcd'; 'lammpscenter'
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default ('')
- functionname = *getresults, Unary, Binary, Ternary, Quarternary, Quinary, Senary*
- args = list of arguments to run the function (*outputfile, ppp, rdelta*)
  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;

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

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) in orthogonal and triclinic cells at various dimensional systems by giving *ndim*. g(r) is defined as:

$$g(r) = \frac{1}{N\rho}\sum_{i=1}^{N}\sum_{j\neq i}^{N}\left\langle \delta\left(\acute{r} + \acute{r_j} - \acute{r_i}\right)\right\rangle$$

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):

```
gr (self, inputfile, ndim, filetype = 'lammps', moltypes = ''):
getresults(self, ppp, rdelta = 0.01, outputfile = '')
Unary(self, ppp, rdelta = 0.01, outputfile = '')
Binary(self, ppp, rdelta = 0.01, outputfile = '')
Ternary(self, ppp, rdelta = 0.01, outputfile = '')
Quarternary(self, ppp, rdelta = 0.01, outputfile = '')
Quinary(self, ppp, rdelta = 0.01, outputfile = '')
Senary(self, ppp, rdelta = 0.01, outputfile = '')

    return (results, names)
```

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, filetype, moltypes).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- filetype = 'lammps' (default); 'lammpscenter', 'gsd'; 'gsd_dcd'
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default ('')
- functionname = *getresults, Unary, Binary, Ternary, Quarternary, Quinary, Senary*
- args = list of arguments to run the function (*outputfile*)

  outputfile is the filename of outcomes without file path;

from structurefactors import functioname1

functioname1(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')

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) in orthogonal and triclinic cells 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):

```
sq (self, inputfile, ndim, filetype = 'lammps', moltypes = ''):
getresults(self, outputfile = '')
    return results, names

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 functionnames
functionnames(inputfile, ndim, radii, ppp, filetype, moltypes, results_path, outpufile, SnapshotNumber, ParticleNumber)

- functionnames = cal_voro, voronowalls, indiceshis
- inputfile = snapshots from MD simulations (trajectories in one file) for cal_voro() and voronowalls(); should be voronoi index file for indiceshis()
- ndim = dimensionality of the system
- filetype = ‘lammps’ (default); ‘lammpscenter’, ‘gsd’; ‘gsd_dcd’
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default (‘’)
- 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 for cal_voro()
- results_path is the file path of outputfile. The default value is '../../analysis/sq/
- outpufile = file name of statistics of voronoi index in indiceshis()
- SnapshotNumber and ParticleNumber can be understand literally.

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 excluding voroindex:

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.

The voro++ package considers non-periodic boundary conditions by default, so there may be some negative numbers in the neighbor list for non-periodic boundary conditions (please refer to the voro++ manual to know this well). A function voronowalls() is designed to remove negative numbers in the neighbor list and other files correspondingly. Please choose cal_voro() for all periodic boundary conditions and voronowalls() for the opposite. Note that the former is much faster than the latter.

The function indiceshis() is designed to statistics the frequency of voronoi index from the output of voronoi analysis. Only the top 50 voronoi index will be output along with their fractions.

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

```
cal_voro(inputfile, ndim = 3, filetype = 'lammps', ppp = '-p', radii = {1: 1.0, 2: 1.0}, moltypes = '', results_path = './')

voronowalls(inputfile, ndim, radii, ppp, filetype = 'lammps', moltypes = '', results_path = './')

indicehis(inputfile, outputfile = '')
    return results, names
```

Dynamics test by log output

Syntax:

from logdynamics import total (total_PBC)

total(inputfile, ndim, filetype = 'lammps', moltypes = '', qmax = 0, a = 1.0, dt = 0.002, outputfile = '')

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- filetype = ‘lammps’ (default); ‘gsd’; ‘gsd_dcd’; ‘lammpscenter’
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default (‘’)
- ppp = [1,1,1] for periodic boundary conditions is used for total_PBC

This is module is designed to test the dynamical properties of a supercooled liquid, for example structural relaxation time. It use the ‘log’ style to output the atomic trajectory. This can be realized in LAMMPS by the following commands:

```
variable  outlog  equal logfreq2(10,18,10) #(90-10)/18, (900-100)/18, (9000-1000)/18
dump_modify 1 every v_outlog first yes
```

In this calculation, only the first configuration is used as the reference. Therefore, the results are not from ensemble for average. This is very useful to study a supercooled liquid if you do not know the structural relaxation time. By getting the structural relaxation time, you can run simulations to dump configurations linearly to do moving average. Or you can do large number of independent simulations to do average.

Dynamical Properties

Syntax:
from dynamics import dynamics
from dynamics_PBC import dynamics
dynamics(inputfile, ndim, filetype, moltypes).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- filetype = 'lammps' (default); 'gsd'; 'gsd_dcd'; 'lammpscenter'
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default ('')
- functionname = *total, partial, slowS4, fastS4*
- args = list of arguments to run the function (*outputfile*, *qmax, a, dt, 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;
  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)
  - ppp = [1,1,1] for periodic boundary conditions is used for all functions with PBC

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 $F_s(q,t)$:

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

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

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

3. Overlap function $Q(t)$:
   $Q(t) = N^{-1}$
   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):

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

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

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

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

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

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 $S_4(q;t)$:

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

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

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

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 $\chi_4(t)$. $S_{4,a}(q;t)$ and $S_{4,b}(q;t)$ 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: $S_4(q;\tau_p) = S_4(q=0;\tau_p)/\left[1+\left(q\xi_4\right)^2\right]$.
(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.

Important: It is mandatory to use xu-type coordinates to calculate dynamics.

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

dynamics (inputfile, ndim, filetype = 'lammps', moltypes = ''):

total(self, qmax, a = 1.0, dt = 0.002, outputfile = '')

    return results, names

partial(self, qmax, a = 1.0, dt = 0.002, outputfile = '')

    return partialresults, names

slowS4(self, X4time, dt = 0.002, a = 1.0, outputfile = '')

    return results, names

fastS4(self, a = 1.0, dt = 0.002, X4timeset = 0, outputfile = '')

    return results, names

the module dynamics_PBC is the same as dynamics but with additional variable ppp

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, filetype, moltype).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- Neighborfile = neighbor list of the corresponding inputfile
- ndim = dimensionality of the system
- filetype = 'lammps' (default); 'gsd'; 'gsd_dcd'; 'lammpscenter'
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default ('')
- functionname = *total, partial, slowS4, fastS4*
- args = list of arguments to run the function (*outputfile*, *qmax, a, dt, 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;
  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:

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

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.
dynamics (inputfile, Neighborfile, ndim, filetype = 'lammps', moltypes = ''):
total(self, qmax, a = 1.0, dt = 0.002, outputfile = '')
    return results, names

partial(self, qmax, a = 1.0, dt = 0.002, outputfile = '')
    return partialresults, names

```
slowS4(self, X4time, dt = 0.002, a = 1.0, outputfile = '')
    return results, names

fastS4(self, a = 1.0, dt = 0.002, X4timeset = 0, outputfile = '')
    return results, names
```

Bond Orientational Order at 3D

Syntax:

from BondOOrder import BOO3D

BOO3D (inputfile, neighborfile, faceareafile, filetype, moltypes).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, default ‘’. Only required if AreaR = 1
- filetype = ‘lammps’ (default); ‘gsd’; ‘gsd_dcd’; ‘lammpscenter’
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default (‘’)
- 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*)
  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

Example:

boo = BOO3D(dumpfile = ’‘, Neighborfile = ’‘, faceareafile = ’‘)
boo.qlQl(l = 6, ppp = [1,1,1], AreaR = 0, outputql = 'sq.dat', outputQl = 'bQ.dat')

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 in both orthogonal and triclinic cells. Steinhardt et al. defined the BOO of the l-fold symmetry as a $2l+1$ vector:

1. $$q_{lm}(i) = \frac{1}{N_i}\sum_0^{N_i} Y_{lm}\left(\theta(r_{ij}), \phi(r_{ij})\right)$$

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, $\hat{W}_l$ 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. $$\hat{w}_l = w_l\left(\sum_{m=-l}^{l}|q_{lm}|^2\right)^{\frac{-3}{2}}$$

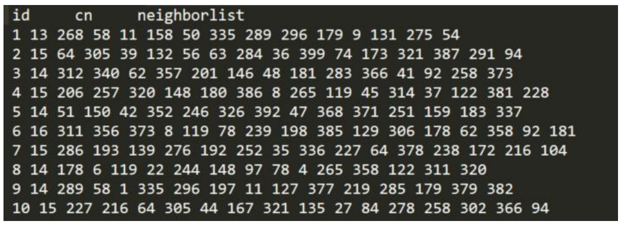
5. $$Q_{lm}(i) = \frac{1}{N_i+1}\left(q_{lm}(i) + \sum_{j=0}^{N_i} q_{lm}(j)\right)$$

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

In these calculations, one important input is the neighbor list of each particle. Currently, the module 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) [data of different snapshots is continuous without any gap or header]:



```
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. Facearea file is only needed if one wants to reweight BOO by using Voronoi polyhedron face area. 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 =)

Another two neighbor list determination methods are included in this package. They consider: 1) specific number of nearest neighbors (such as N = 12), 2) specific distance cutoff to determine the nearest neighbors (such as N = 12). The outputfiles are compatible with this module. [see module ParticleNeighbors]

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 this function, faceareafile will only be read when AreaR = 1. Otherwise it is not required.

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*(). (smallql, largeQl) will be returned.

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 and names (i.e. header of the output file) will be returned.

The functions smallwcap() and largeWcap() calculates equation (3) and (4) for (w_l, $\hat{w}_l$), (W_l, $\hat{W}_l$) 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. (smallw, smallwcap) will be returned.

When the area weighted BOO is considered, $q_{lm}(i)$ changes to $q_{lm}(i) = \sum_0^{N_i} \frac{A_j}{A} Y_{lm}\left(\theta(r_{ij}), \phi(r_{ij})\right)$ 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.

A module 'cal_multiple 'is provided so that multiple order parameters can be calculated together which saves a lot of computational time. Set the corresponding variables to be True then the order parameters will be calculated.

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 = '', filetype = 'lammps ', moltypes = ''):

```
qlQl(l, ppp = [1,1,1], AreaR = 0, outputql = '', outputQl = '')
        return (smallql, largeQl)

sijsmallql(l, ppp = [1,1,1], AreaR = 0, c = 0.7, outputql = '', outputsij = '')
        return resultssij[1:] #individual value of sij

sijlargeQl(l, ppp = [1,1,1], AreaR = 0, c = 0.7, outputQl = '', outputsij = '')
        return resultssij[1:] #individual value of sij

GllargeQ(l, ppp = [1,1,1], rdelta = 0.01, AreaR = 0, outputgl = '')
        return results, names

Glsmallq(l, ppp = [1,1,1], rdelta = 0.01, AreaR = 0, outputgl = '')
        return results, names

smallwcap(l, ppp = [1,1,1], AreaR = 0, outputw = '', outputwcap = '')
        return (smallw, smallwcap)

largeWcap(l, ppp = [1,1,1], AreaR = 0, outputW = '', outputWcap = '')
        return (largew, largewcap)

timecorr(l, ppp = [1,1,1], AreaR = 0, dt = 0.002, outputfile = '')
        return results

cal_multiple(l, ppp = [1,1,1], AreaR = 0, c = 0.7, outpath = './', cqlQl = 0, csijsmallql
= 0, csijlargeQl = 0, csmallwcap = 0, clargeWcap = 0) [c represents calculate]
```

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, filetype, moltypes).functioname(args)

- inputfile = snapshots from MD simulations (trajectories in one file)
- Neighborfile = Neighbor list file as above
- filetype = 'lammps' (default); 'gsd'; 'gsd_dcd'; 'lammpscenter'
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default ('')
- functionname = *tavephi, spatialcorr, timecorr*
- args = list of arguments to run the function (*outputphi, outputavephi, outputfile, avet, l, ppp, dt*)

  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

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 in orthogonal and triclinic cells, 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. $\psi_l^j = t\int_0^t dt\left|\varphi_l^j\right|$ (Time Average)

3. $g_l(r) = \frac{L^2}{2\pi r\Delta r N(N-1)}$ (Spatial Correlation)

4. $C_l(t) = \langle\sum_n \varphi_l^n(t)\varphi_l^n(0)\ \rangle/\left\langle\sum_n\left|\varphi_l^n(0)\right|^2\right\rangle$ (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 (self, dumpfile, Neighborfile, filetype = 'lammps', moltypes = ''):
lthorder(self, l = 6, ppp = [1, 1])
    return results

tavephi(self, outputphi, outputavephi, avet, l = 6, ppp = [1, 1], dt = 0.002)

spatialcorr(self, l = 6, ppp = [1, 1], rdelta = 0.01, outputfile = '')
    return results, names

timecorr(self, l = 6, ppp = [1, 1], dt = 0.002, outputfile = '')
    return results, names
```

References:

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

Non-affine Strain

Syntax:
from Strain import Vonmises
Vonmises (inputfile, Neighborfile, ndim, strainrate, outputfile, ppp, dt, filetype, moltypes)

- inputfile = snapshots from MD simulations (trajectories in one file)
- Neighborfile = Neighbor list file as above
- ndim = dimensionality of the system
- strainrate = strain rate of deformation, in align with standard time unit
- outputfile is the filename of outcomes without file path, please refer to individual functions to see details
- ppp is the periodic boundary conditions. Set 1 for yes and 0 for no. default=[1,1,1]
- dt is the time step of simulation. Default = 0.002.
- filetype = ‘lammps’ (default); ‘lammpscenter’, ‘gsd’; ‘gsd_dcd’
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default (‘’)

Example:
Vonmises(inputfile = , Neighborfile = , ndim = 3, strainrate = 0.01, outputfile = , ppp = [1,1,1], dt =0.002)

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 particle local shear strain η_i^{Mises} for each particle in orthogonal and triclinic cells to quantify plastic deformation at the particle level. The calculation uses the first snapshot as reference. First a local transformation matrix is defined as
$J_i = \left(\sum_{j\in N_i^0} d_{ji}^{0T} d_{ji}^0\right)^{-1} \left(\sum_{j\in N_i^0} d_{ji}^{0T} d_{ji}\right)$, then the local Lagrangian strain matrix is computed as $\eta_i = \frac{1}{2}(J_i J_i^T - I)$, where I is an identity matrix. The particle level local shear invariant is $\eta_i^{Mises} = \sqrt{\frac{1}{2} Tr\left(\eta_i - \eta_{i,m} I\right)^2}$ where $\eta_{i,m} = \frac{1}{3} Tr \eta_i$. This is also equivalent to $\eta_i^{Mises} = \sqrt{\eta_{yz}^2 + \eta_{xz}^2 + \eta_{xy}^2 + \frac{\left(\eta_{yy}-\eta_{zz}\right)^2 + \left(\eta_{xx}-\eta_{zz}\right)^2 + \left(\eta_{yy}-\eta_{xx}\right)^2}{6}}$.

Class/Function lists in the module (indentation indicates relationship):
Vonmises(inputfile, Neighborfile, ndim, strainrate, ppp = [1,1,1], dt =0.002, filetype = 'lammps', moltypes = '', outputfile = '')

Pair Entropy

Syntax:
from pairentropy import classnames
classnames(inputfile, ndim, filetype. moltypes).functionnames(*args)

- classnames = S2, S2AVE
- functionnames = timeave, spatialcorr, getS2
- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- filetype = ‘lammps’ (default); ‘lammpscenter’, ‘gsd’; ‘gsd_dcd’
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default (‘’)
- outputfile is the filename of outcomes without file path, please refer to individual functions to see details
- ppp is the periodic boundary conditions. Set 1 for yes and 0 for no.
- avetime = time scale used to average S2 or particle g(r). In time Unit.
- rdelta = bin size used in calculating g(r). default = 0.01
- dt is the time step of simulation. Default = 0.002
- outputcorr = output file name of spatial correlation of S2
- outputs2 = output file name of S2

Example:
S2(inputfile, 3).timeave(outputfile = 'S2x.dat', ppp = [1,1,1], avetime = 0.4, rdelta = 0.1, dt = 0.002, results_path='./S2/')

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 particle-level pair entropy s2 and their spatial correlations in orthogonal and triclinic cells at various dimensions. Particle s2 and their correlations are defined as:

1. $s_2^\alpha = \frac{-1}{2}\sum_v \rho_v \int dr' \left[g_{v\alpha}(r') \ln g_{v\alpha}(r') - (g_{v\alpha}(r') - 1)\right]$
2. $g_{s2}(r) = \sum_{ij} \delta(r_{ij} - r)\, s_2^j s_2^i$

The module calculates s2 up to Quinary systems. Usually, there are two methods to calculate s2: 1) calculate particle g(r) at single snapshot and integral to get s2, and then average s2 over desired time scales; 2) average particle g(r) over desired time scales first and then integral to calculate particle s2. Two classes, S2 and S2AVE, are designed for the former and latter, respectively.

In case 1) (class S2), the function timeave() can be used to output particle s2 by setting particle type number automatically. If the argument avetime is 0, particle s2 of each snapshot without averaging will be output; while averaged s2 over desired time scales will be output by giving avetime (in time units). The output results will be returned. The function spatialcorr() will call the function timeave() first and then calculate the spatial correlation of particle s2. If you want to calculate spatial correlations, you can only import the function spatialcorr() to get particle s2 and their correlations at the same time. The results will be returned.

In case 2) (class S2AVE), the function getS2() will output the results, but only particle s2 will be returned. The particle type number is also set automatically. The function spatialcorr() is similar to the case 1).

In both cases, functions from Unary() to Quinary() are designed to calculate particle s2, but the results are only returned without output. Therefore, it is unnecessary to use these functions externally.

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

```
S2(self, inputfile, ndim, filetype = 'lammps', moltypes = ''):
timeave(self, ppp, avetime, rdelta = 0.01, dt = 0.002, outputfile = '')
    return results, names

spatialcorr(self, outputs2, ppp, avetime, rdelta = 0.01, dt = 0.002, outputcorr = '')
    return results, names

S2AVE (self, inputfile, ndim, filetype = 'lammps', moltypes = ''):
getS2(self, ppp, avetime, rdelta = 0.01, dt = 0.002, outputfile = '')
    return S2results, names

spatialcorr(self, outputs2, ppp, avetime, rdelta = 0.01, dt = 0.002, outputcorr = '')
    return results, names
```

References:

Tanaka et al. Nature Materials 9, 324 (2010) (case 1)
Yang et al. Physical Review Letters 116, 238003 (2016). (case 2)

Demixing check

Syntax:
from demixcheck import neighbortypes
neighbortypes(inputfile, ndim, neighborfile, filetype, outputfile, results_path)

- inputfile = snapshots from MD simulations (trajectories in one file)
- ndim = dimensionality of the system
- neighborfile = neighbor list generated by voro++ analysis or in that format
- filetype = 'lammps' (default); 'gsd'; 'gsd_dcd'; 'lammpscenter
- outputfile is the filename of outcomes without file path, default = ''
- results_path is the file path of outputfile. The default value is '../../analysis/S2/'

Example:
neighbortypes(inputfile, ndim, neighborfile, filetype = 'lammps', outputfile = '', 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 and returns the fraction of different neighbor environments, like AA, AB, BB.

Local tetrahedral order

Syntax:

from tetrahedrality import local_tetrahedral_order
local_tetrahedral_order (inputfile, filetype, moltypes, ppp, outputfile)

- inputfile = snapshots from MD simulations (trajectories in one file)
- filetype = 'lammps' (default); 'lammpscenter', 'gsd'; 'gsd_dcd'
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default ('')
- ***ppp*** is the periodic boundary conditions, set 1 for yes and 0 for no. default [1,1,1]
- outputfile is the filename of outcomes with path, default = ''. If not given, only return the results but not write out.

Example:
local_tetrahedral_order(inputfile)

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 and returns the local tetrahedral order parameter of each atom in each snapshot in a numpy array. The shape is (particle number, snapshot number). The atom indices are included as the first column. Local tetrahedral order is defined as:

$$q_{tetra} = 1 - \frac{3}{8}\sum_{j=1}^{3}\sum_{k=j+1}^{4}\left(cos\varphi_{jk} + \frac{1}{3}\right)^2.$$

In this calculation, only 4 nearest neighbors are taken into consideration. The algorithm of selecting nearest distances is from numpy.argpartition. In this method, only the nearest neighbors are selected but not in a sorted order. j, k run over these neighbors. Currently this calculation is only available for 3D.

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

```
local_tetrahedral_order(dumpfile, filetype = 'lammps', moltypes = '', ppp = [1,1,1],
outputfile = '')
    return results, names
```

Spatial Correlation

Syntax:

from spatialcorrelation import SC_order

SC_order(inputfile, orderings, ndim = 3, filetype = 'lammps', moltypes = '', rdelta = 0.01, ppp = [1,1,1], outputfile = '')

- inputfile = snapshots from MD simulations (trajectories in one file)
- orderings = input array of particle-level property. Its shape must be **[num_of_atom, num_of_snapshot]**
- ndim = dimensionality, default 3
- filetype = 'lammps' (default); 'lammpscenter', 'gsd'; 'gsd_dcd'
- moltypes = dict of atomtype:moleculetype. *It is only needed for the lammpscenter mode. But the types of moltypes are quite different in these two cases.* Default ('')
- rdelta = bin size, default 0.01
- ***ppp*** is the periodic boundary conditions, set 1 for yes and 0 for no. default [1,1,1]
- outputfile is the filename of outcomes with path, default = ''. If not given, only return the results but not write out.

Example:

SC_order(inputfile, orderings)

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 and returns the averaged spatial correlation of an atomic-scale property in a numpy array. The spatial correlation is defined as:

$$\langle \delta X(0)\delta X(r)\rangle = \frac{\langle X(0)X(r)\rangle - \langle X\rangle^2}{\langle X^2\rangle - \langle X\rangle^2}.$$

X can be any pre-calculated atomic-scale quantity. This result can be fitted to exp(-r/A) where A is the spatial correlation length of X. By setting the orderings as binary, the correlation length of one specific structure can be calculated.

The shape of orderings must be [num_of_atom, num_of_snapshot], which is the standard output style of this package. It can be created by

orderings = np.loadtxt(filename_orderings, skiprows = 1)[:, 1:]

For a single configuration, change the numpy array by '[:, np.newaxis]'.

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

SC_order(inputfile, orderings, ndim = 3, filetype = 'lammps', moltypes = '', rdelta = 0.01, ppp = [1,1,1], outputfile = '')

    return final, names

Rewrite dump file

Syntax:
from RewriteDump import lammps
lammps(step, atomnum, boxlengths)

- step = timestep of the current snapshot
- atomnum = atom number of the current snapshot
- boxbounds = numpy array of box bounds (ndim, 2)

Example:

lammps(step, atomnum, boxbounds)

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 is used to write the header of lammps dump file. This is quite useful when you want to pick out some atoms from the trajectory to analyze. When dimension is 2, the box length at z axis should be given 0.

A string for the header will be returned.

Class/Function lists in the module (indentation indicates relationship):
lammps(step, atomnum, boxbounds, addson = '')

Read log file

Syntax:
from log import func
lammpslog(filename, output=false)

- filename
- output = file name of output

Example:

lammpslog(‘log.dat’)

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 is designed to extract the thermodynamic quantities of a simulation from the log file. This is especially useful for lammps log file because there are headers and tails in this file. For lammpslog, the thermodynamic data will be returned as a pandas dataframe. If output is true, the thermodynamic data will be written to a .csv data file. For hoomdlog, since the log file is already well formatted, it only reads it by using pandas read_csv method.

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

```
lammpslog(filename, output = False)
    return data [dataframe]

hoomdlog(filename)
    return data [dataframe]
```

Fitting data by python

Syntax:
from fitting import fits
fits(fit_func, xdata, ydata, rangea = 0, rangeb = 0, p0 = [], bounds = [])

- fit_func = the function used for fitting
- xdata, ydata = input data for fitting
- rangea, rangeb = designed interval for fitting results. Default = 0, same as input
- p0 = list of initial values of fitting parameters
- bounds = tuples of bounds list for the fitting parameters, ([lower], [upper])

Example:

fits(fit_func, xdata, ydata)

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 is designed to extract the fitting results for data by using python. It uses the designed fitting function to fit the given input data. Then fitting parameters and results are returned to plot. Both fitting parameters and their errors are returned.

Note that the fitting data for plot (xfit, yfit) are generated linearly. When do log scale, new (xfit, yfit) needs to be calculated.

Class/Function lists in the module (indentation indicates relationship):
fits(fit_func, xdata, ydata, rangea = 0, rangeb = 0, p0 = [], bounds = [])
    return (popt, perr, xfit, yfit) [fitting parameters, fit results]

Read angular vector file

Syntax:
from dumpAngular import readangular, PatchVector
PatchVector(args)
readangular(args).read_onefile()

- filename = angular vector file or dump file with coordinates of particle and patches
- ndim = dimensionality, default = 3
- num_patch = patch number of each particle, default = 12
- outputvec = output file of the vectors based on the above mentioned coordinates
- outputdump = output file of the coordinates of particles and their first patches

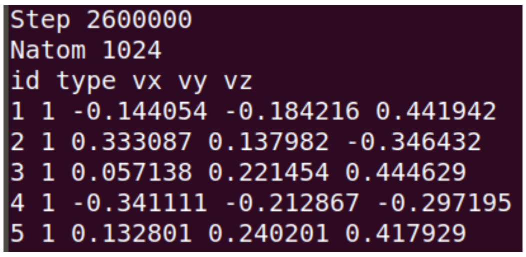
Example:

d = readangular(filename); d.read_onefile()

PatchVector(filename)

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 is designed to create and read the angular vectors (orientations) file to calculate rotational dynamics. The format of the file should be as follows:



```
Step 2600000
Natom 1024
id type vx vy vz
1 1 -0.144054 -0.184216 0.441942
2 1 0.333087 0.137982 -0.346432
3 1 0.057138 0.221454 0.444629
4 1 -0.341111 -0.212867 -0.297195
5 1 0.132801 0.240201 0.417929
```

Given the coordinates of particles and patches in the LAMMPS dump format, the vectors as above can be generated by the function ‘PatchVector’. This should be the first step to calculate rotational dynamics by using the patchy particles.

All the configurations should be included in one file. For each configuration, the first line is the time step, which can be used to get the time interval. The second is the number of particle, which is used to determine individual configuration during reading. The third is the header of each configuration, which is skipped during reading but it defines the format of input. Following the header, angular velocity of each particle is provided. The module will determine how many configurations are there by providing the above information. After reading, information about the configuration is provided in the manner of python list with elements as numpy array, just like the ‘dump ’module. The returned variables are:

TimeStep (list); ParticleNumber (list); ParticleType (list); velocity (list); SnapshotNumber

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

readangular(filename, ndim = 3)

    read_onefile()

PatchVector(filename, num_patch = 12, ndim = 3, filetype = 'lammps', outputvec = '', outputdump = '')

Rotational motion

Syntax:
from rotation import functionname
functioname(args)

- filename = angular velocity file of particles
- ndim = dimensionality, default = 3
- dt = timestep, default = 0.002
- outputfile = filename with path for output, default = ‘’; give a name for output data otherwise will not but only return
- neighborfile = Neighbor list file for calculating rotational ordering

Example:

Rorder(filename)

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 is designed to calculate the rotational dynamics and structural ordering based on particle orientation. All orientations are firstly normalized by its norm to get the unit vector $u_i(t)$where t is time. The formula to calculate the dynamics is $C_R(t) = (1/N \langle \sum_i u_i\,(t)u_i(0)\rangle)$.

There are two functions to calculate the rotational dynamics, one is using configurations dumped at the constant time interval, which enables ensemble average over different initial configurations. The other is using configurations at arbitrary dump intervals, which only use the first configuration as the reference. This is useful for log style output to test the dynamical properties at the beginning. It is also quite useful if one wants to calculate the dynamics of a specific group of particles. In that case, a new temporary dump file of the set of particles can be used as the input file.

There is also a function that characterizes the local rotational ordering of each particle, which can be used to define local crystalline order if its rotation is align with its neighbors. The formula is
$\Psi(t) = (1/N_i) \sum_{\langle i,j\rangle} \left|u_i(t)u_j(t)\right|$,
where j is the nearest neighbors of i. N_i is the the coordination number of particle i.

The function ‘RorderIJ ’is used to determine the degree of alignment of a center particle with respect to its neighbors, which is based on the value $\left|u_i(t)u_j(t)\right|$. This is similar to the calculation of sij in three-dimensional bond orientational order parameters. When this value is larger than UIJ (defalut 0.9), they are considered to be parallel or anti-parallel. The number of the aligned neighbors will be output to the file ‘outputfile’. The degree of this alignment of the center with each neighbor is written to the file ‘outputfileij’, if a name is given to ‘outputfileij’. Its format is the same as the output from voronoi neighbor list.

Note that the functions calculating the local orientational ordering ‘Rorder ’and ‘RorderIJ ’are only applicable to systems that have only one orientational direction, like spin liquid. It is not suitable for multiple patchy particles [see particle alignment analysis module].

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

CRtotal(filename, ndim = 3, dt = 0.002, outputfile = '')
    return results, names

logCRtotal(filename, ndim = 3, dt = 0.002, outputfile = '')
    return results, names

Rorder(filename, ndim = 3, neighborfile = '', outputfile = '')
    return results

RorderIJ(filename, ndim = 3, UIJ = 0.9, neighborfile = '', outputfile = '', outputfileij = '')

Ordering Time Correlation

Syntax:
from timecorrelation import Orderlife, logOrderlife
Orderlife(dumpfile, orderings, ndim = 3, dt = 0.002, outputfile = '')

- dumpfile = particle configurations to provide TimeStep
- orderings = numpy array of particle ordering quantity, its shape must be [number_of_atom, number_of_snapshot]. It should be align with the dumpfile for logOrderlife
- ndim = dimensionality, default: 3
- dt = time step of MD simulation, default: 0.002
- outputfile = name of output file, given to write out outputs

Example:

Orderlife(dumpfile, orderings)

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 is designed to calculate the time correlation of local structural ordering. The formula is

$$S(t) = \frac{\langle O(0)O(t)\rangle - \langle O\rangle^2}{\langle O^2\rangle - \langle O\rangle^2}.$$

This function can be used to evaluate the time decay of local ordering, especially for some type of structure. For example, whether the particle is icosahedron or not. By setting the orderings as binary, the lieftime of one specific structure can be calculated.

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

```
Orderlife(dumpfile, orderings, ndim = 3, dt = 0.002, outputfile = '')
    return results, names

logOrderlife(dumpfile, orderings, ndim = 3, dt = 0.002, outputfile = '')
    return results, names
```

Patchy alignment analysis

Syntax:
from patchrotation import Rorder
Rorder(filename, num_patch, ndim, ppp, neighborfile, outputfile, outputfileij)

- filename = particle configurations with both centers and patches
- num_patch = the number of patches for each particle, default: 12
- ndim = dimensionality, default: 3
- ppp = periodic boundary conditions, default: [1, 1, 1]
- neighborfile = neighbor list of each center
- outputfile = average value of particle alignment over the nearest neighbors
- outputfileij = individual particle alignment of a center with each of its neighbors

Example:

Rorder(dumpfile, 12, 3, [1,1,1], neighborfile, outputfile, outputfileij

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 is designed to calculate the patchy particle alignment of the center with each of its nearest neighbors. **This is useful for multiple patches case.** For multiple patches, two patches of each pair should be selected as the closest ones based on the angle between particle-particle and particle-patch. After getting the correct patch on each particle, it can be calculated as

$$\Psi(t) = (1/N_i)\sum_{\langle i,j \rangle} u_i(t)u_j(t)$$

where j is the nearest neighbors of i. N_i is the the coordination number of particle i. This quantity actually tells the angle of nearest particles. A perfect structure based on patch will form if its value is -1. In the module, a function 'cal_vector 'is designed to extract all the positions of centers and patches. It also functioned to remove the periodic boundary conditions of patches with respect to the center.

Class/Function lists in the module (indentation indicates relationship):
Rorder(filename, num_patch = 12, ndim = 3, ppp = [1,1,1], neighborfile = '', outputfile = '', outputfileij = ''):
    return results, names

Coarse-graining

Syntax:

```
from CoarseGraining import CG
CG(ordering, neighborfile, outputfile)
```

- ordering = input particle-level ordering as a numpy array. It must have the shape [num_of_atom, num_of_snapshot]
- neighborfile = neighbor list file over which to coarse grain
- outputfile = output of coarse-graining, will be return if not None

Example:

CG(ordering, neighborfile, outputfile = ’‘)

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 is designed to do coarse-graining over the neighbor list for each particle at each time. The property of a center will the averaged result from its neighbors and itself.

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

```
CG(ordering, neighborfile, outputfile):
    return orderingCG
```

Conditional property calculation

Syntax:
from SelectiveProperty import dynamics, logdynamics, partialgr, partialSq
funciton(**agrv)

- selection = input particle-level condition as a numpy array of **bool** type. It must have the shape [num_of_atom, num_of_snapshot]
- outputfile = output of calculations, will be return if not None
- Other parameters can be referred from the above modules

Example:

dynamics(inputfile, selection)

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 is designed to calculate the properties of specified atoms, such as the dynamics, dynamics in log style output or just use the initial configuration as the reference, pair correlation function, structure factor. The selected atoms (condition) are given by the numpy array 'selection 'in the bool type, which must have the shape [num_of_atom, num_of_snapshot]. For a single configuration, selection should be changed by '[:, np.newaxis]'. In all functions, it is the best that the configuration number of inputfile is the same as that of selection. But for logdynamics function, only the selection of the initial configuration is fine but should have the requested shape. The function dynamics do average over different initial configurations.

This module is very useful if one wants to check the special feature of some atoms. By versatile selection, it can give important information by intention.

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

```
dynamics(inputfile, selection, ndim = 3, filetype = 'lammps', moltypes = '', qmax = 0, a = 0.3, dt = 0.002, ppp = [1,1,1], outputfile = ''):
    return results, names

logdynamics(inputfile, selection, ndim = 3, filetype = 'lammps', moltypes = '', qmax = 0, a = 0.3, dt = 0.002, ppp = [1,1,1], outputfile = ''):
    return results, names

partialgr(inputfile, selection, ndim = 3, filetype = 'lammps', moltypes = '', rdelta = 0.01, ppp = [1,1,1], outputfile = ''):
    return results, names
```

```
partialSq(inputfile, selection, ndim = 3, filetype = 'lammps', moltypes = '', ppp =
[1,1,1], Numofq = 500, outputfile = '')
        return results, names
```