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 packagename*']) 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).functioname()

- classname = readdump (for 2D and 3D systems)
- inputfile = snapshots from MD simulations (trajectories in one file)
- functionname = *read_onefile*

Example:
d = readdump('./dumpfile')
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 both two dimensional and three dimensional systems, but for 2D (x, y, z) are all needed. The coordinate queue should be '**id type x y z (…)**'. 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.

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 gr3d (for 3D or gr2d for 2D)
classname(inputfile).functioname(args)

- classname = gr3d (3D systems) or gr2d (2D systems)
- inputfile = snapshots from MD simulations (trajectories in one file)
- functionname = *getresults, Unary, Binary, Ternary, Quarternary, Quinary, Senary* for 2D and 3D cases (see below for details)
- args = list of arguments to run the function (*outputfile, rdelta, ppp, results_path*)
  outputfile is the filename of outcomes without file path;
  rdelta is the bin size calculating g(r), the default value is 0.01;
  ppp is periodic boundary conditions along different directions, set 1 for yes and 0 for no at one direction. The default value is [1,1,1] for 3D and [1,1] for 2D; ***results_path*** is the file path of outputfile. The default value is '../../analysis/gr/'

Example:
gr3d('./dumpfile').getresults(outputfile = 'gr.dat', 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 three dimensional and two dimensional systems. g(r) is defined as:

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

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

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 ($L_{\min}/2$).

The cases in 3D and 2D are quite similar in the module. But only remember to change *ppp* argument because there are only two values in 2D.

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

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

Class gr3d (inputfile):

getresults (outputfile, rdelta = 0.01, ppp = [1,1,1], results_path='../../analysis/gr/');
Unary (outputfile, rdelta = 0.01, ppp = [1,1,1], results_path='../../analysis/gr/');
Binary (outputfile, rdelta = 0.01, ppp = [1,1,1], results_path='../../analysis/gr/');
Ternary (outputfile, rdelta = 0.01, ppp = [1,1,1], results_path='../../analysis/gr/');
Quarternary (outputfile, rdelta = 0.01, ppp = [1,1,1], results_path='../../analysis/gr/');
Quinary (outputfile, rdelta = 0.01, ppp = [1,1,1], results_path='../../analysis/gr/');
Senary (outputfile, rdelta = 0.01, ppp = [1,1,1], results_path='../../analysis/gr/');

Class gr2d (inputfile):

getresults (outputfile, rdelta = 0.01, ppp = [1,1], results_path='../../analysis/gr/');
Unary (outputfile, rdelta = 0.01, ppp = [1,1], results_path='../../analysis/gr/');
Binary (outputfile, rdelta = 0.01, ppp = [1,1], results_path='../../analysis/gr/');
Ternary (outputfile, rdelta = 0.01, ppp = [1,1], results_path='../../analysis/gr/');
Quarternary (outputfile, rdelta = 0.01, ppp = [1,1], results_path='../../analysis/gr/');
Quinary (outputfile, rdelta = 0.01, ppp = [1,1], results_path='../../analysis/gr/');
Senary (outputfile, rdelta = 0.01, ppp = [1,1], 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 sq3d (for 3D or sq2d for 2D)

classname(inputfile).functioname(args)

- classname = sq3d (3D systems) or sq2d (2D systems)
- inputfile = snapshots from MD simulations (trajectories in one file)
- functionname = *getresults, Unary, Binary, Ternary, Quarternary, Quinary, Senary* for 2D and 3D cases (see below for details)
- 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/gr/'

from structurefactors import functioname1

functioname1(args1)

- functionname1 = *wavevector3d, wavevector2d*
- args1 = *Numofq*
  Numofq is the considered number of wavenumber. Default is 500

Example:

sq3d('./dumpfile').getresults(outputfile = 'Sq.dat', results_path = './sq/')

wavevector3d(Numofq = 100)

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) for three dimensional and two dimensional systems directly. S(q) is defined as:

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

where N is particle number. The code is written referring to (). In this code, if the box length L is smaller than 40.0, S(q) will be computed to L; but if L is larger than 40.0, S(q) will be computed to $L/2$. 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* and *wavevector2d*. 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.

The cases in 3D and 2D are quite similar in the module.

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

Class sq3d (inputfile):

    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)

Class sq2d (inputfile):

    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/');

References:

Hu et al. The Journal of Chemical Physics, **146** (2), 024507 (2017)
Hu et al. Physical Review E, **96** (2), 022613 (2017)

Dynamical Properties

Syntax:
from dynamics import dynamics3d (for 3D or dynamics2d for 2D)
classname(inputfile).functioname(args)

- classname = dynamics3d (3D systems) or dynamics2d (2D systems)
- inputfile = snapshots from MD simulations (trajectories in one file)
- functionname = *total, partial, slowS4, fastS4* for 2D and 3D cases (see below for details)
- 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:
dynamics3d('./dumpfile').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 in 3D and 2D such as:

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

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

(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 $Q(t)$:

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

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

fast particles: where $\omega(r) = 1$ if $r \ge a$ 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 $\langle \Delta r^2(t)\rangle$:

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

(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 $S_4(q;t)$:

$$S_4(q;t) = N^{-1}[\langle W(\vec{q},t)W(-\vec{q},t)\rangle - \langle W(\vec{q},t)\rangle\langle W(-\vec{q},t)\rangle]$$

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

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

fast particles: where $\omega(r) = 1$ if $r \ge a$ 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 X4. 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; but if L is larger than 40.0, S(q) will be computed to L/2. 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))

The cases in 3D and 2D are quite similar in the module.

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 dynamics3d (inputfile):

```
    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');
```

Class dynamics2d (inputfile):

```
    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)

Bond Orientational Order at 3D

Syntax:
from BondOOrder import BOO3D
classname(inputfile, neighborfile, faceareafile).functionname(args)

- classname = BOO3D
- 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, BoowWcap*
- args = list of arguments to run the function (*l, ppp, AreaR, outputql, outputQl, outputsij, rdelta, outputgl, outputw, outputW, outputwcap, outputWcap, 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.

  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 correlations for three dimensional. S(q) is defined as:

where *N* is particle number. The code is written referring to ().

In these calculations, one important input is the neighbor list of each particle. Currently, the module only accepts data generated by the given shell script 'voronoi.sh' (using Voro++ Package). Please visit the hyperlink to review the details. 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
```

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. If you use other data other than the neighbor list, please add 1 to use the correct value. 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.sh']. Then you can read the generated data for BOO analysis and others. It is good to reminder 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 the 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 qlm and Qlm 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 (qlm, Qlm), where qlm and Qlm 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’. The return is similar to the case in qlmQlm().

The functions *sijsmallql*() and *sijlargeQl*() calculate sij based on qlm and Qlm, respectively. Although it is more frequently to use qlm. Two files are written, one is the sij 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 qlm and Qlm, 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 function BoowWcap() calculates equation (). This function is a little bit slow due do large efforts of calculation including use Wigner 3j symbol. All results will be written to separate files if given a filename and will be returned in a tuple similar to 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.BoowWcap(l = , ppp = [1,1,1], AreaR = 0, outputw = '', outputW = '', outputwcap = '', outputWcap = '', results_path = '../../analysis/BOO')

References: