Contents

analisi: your Swiss Army Knife of molecular dynamics analysis

Link to the pdf version of this document

Binary packages available on linux and macOS conda-forge:

```
conda config --add channels conda-forge    #if not already done
conda config --set channel_priority strict #if not already done
conda install analisi
```

Name	Downloads	Version	Platforms
recipe analisi	downloads 162k	conda-forge v0.5.0	

or by compiling from source following this section

Fastest possible example usage:

command line example

- mean square displacement

```
./analisi -i tests/lammps.bin -Q > output_file
```

- spherical harmonics correlation functions

```
./analisi -i tests/lammps.bin -Y 4 -F 0.7 3.5 > output_file
```

- g(r), with time lags too

```
./analisi -i tests/lammps.bin -g 200 -F 0.7 3.5 > output_file
```

- green kubo autocorrelation integrals

```
./analisi -l tests/gk_integral.txt -H -a 'c_flux[1]' 'c_vcm[1][1]' > output_file
```

and many others...

python example

```
#read trajectory
import numpy as np
pos = np.load( 'tests/data/positions.npy')
vel = np.load( 'tests/data/velocities.npy')
box = np.load( 'tests/data/cells.npy')
#atomic types: load an array with length equal to the number of atoms
#in this example I set all the atoms to type 0 but the last 16,
#wich are 8 of type 1 and 8 of type 2
types = np.zeros(pos.shape[1], dtype = np.int32)
types[-16:-8]=1
types[-8:]=2
print('position array shape is {}'.format(pos.shape))
print('first cell matrix is {}'.format(box[0]))

#create trajectory object
import pyanalisi
```

```
analisi_traj = pyanalisi.Trajectory(
      pos, #shape (ntimesteps, natoms, 3)
      vel, #same shape as pos
      types, #shape (natoms)
      box, #shape (ntimesteps, 3,3)
           # or (ntimesteps, 6)
           # or (ntimesteps, 9)
      pyanalisi.BoxFormat.CellVectors,
                  # input box format is
                  # matrix of cell vectors
      False, # don't wrap atoms around the center of
             # the cell
      False # don't save rotation matrix that is used
            # internally if the cell is triclinic
  )

#do the calculation that you want
#see MSD section
msd=pyanalisi.MeanSquareDisplacement(analisi_traj,10,4000,4,True,False,False)
msd.reset(3000)
msd.calculate(0)

#the calculation object can be used as an array
result=np.array(msd,copy=False)


#other calculations
#...
#...
```

Heat transport coefficient calculation: correlation functions and gk integral for a multicomponent fluid example

```
import numpy as np
#read first line, that is an header, and the column formatted data file
with open(filepath_tests + '/data/gk_integral.dat', 'r') as flog:
    headers = flog.readline().split()
log = np.loadtxt(filepath_tests + '/data/gk_integral.dat', skiprows=1)

import pyanalisi
#wrapper for the pyanalisi interface
traj = pyanalisi.ReadLog(log, headers)
#see Green Kubo section
gk = pyanalisi.GreenKubo(analisi_log,'',
       1,['c_flux[1]','c_vcm[1][1]'],
       False, 2000, 2,False,0,4,False,1,100)
gk.reset(analisi_log.getNtimesteps()-2000)
gk.calculate(0)

#the calculation object can be used as a python array
```

```
result = np.array(gk,copy=False)
```

note

If you use this program and you like it, spread the word around and give it credits! Implementing stuff that is already implemented can be a waste of time. And why don't you try to implement something that you like inside it? You will get for free MPI parallelization and variance calculation, that are already implemented in a very generic and abstracted way.

Description

This is a framework for computing averages on molecular dynamics trajectories and block averages. An mpi parallelization over the blocks is implemented in a very abstract and general way, so it is easy to implement a new calculation that can be parallelized with no effort using MPI. The code has two interfaces: a command line interface that is parallelized with MPI and threads, and a python interface that is parallelized on threads only. With such many parallelization levels more accurate averages can be performed, as described in details in the next sections.

Every MPI processes uses the same amount of memory by loading the part of the trajectory that it is used to do the calculation in every block. For this reason it is suggested to use one MPI process per machine, and parallelize by using threads only inside the single machine.

Features:

- python interface (reads numpy array)
- command line interface (reads binary lammps-like files or time series in column format)
- multithreaded (defaults to number of threads specifies by the shell variable OMP_NUM_THREADS)
- command line interface has MPI too (for super-heavy multi-machine calculations)
- command line calculates variance of every quantity and every function (in python you can do it by yourselves with numpy.var)
- jupyter notebook example of the python interface

Calculations:

- g of r (and time too!)
- vibrational spectrum with FFTW3
- histogram of number of neighbours
- mean square displacement
- green kubo integral of currents
- multicomponent green kubo time domain formula (MPI here can be useful. . .)
- spherical harmonics number density time correlation analysis (MPI here can be useful. . .)
- atomic position histogram
- (averaged) Steinhardt local order parameters
- SANN first neighbors algorithm
- . . .

Building from source

If you clone the repository you have to initialize the submodules with

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

this will initialize the pybind11 repository for the (optional) python interface Dependencies:

- C++17 compatible compiler (GCC 7+, clang 6+, intel 19.0.1+, MSVC 19.29 source)
- cmake 3.8+
- linux or macOS for full functionalities, that include the command line tool and reading lammps binary files and full test suite
- windows can use only the python library, and only python arrays can be used as trajectory data. Some functionalities are not tested. Reading of lammps trajectory is not supported (this would require rewriting some of the code in the core library, it is doable)
- FFTW3 (included in the package)
- Eigen3 (included in the package)
- Boost (included in the package)
- Mpi (optional)
- libxdrfile (for gromacs file conversion – included in the package)
- python (optional, 3.6+)

Documentation build:

- r-rsvg
- pandoc
- rsvg-convert
- texlive
- readme2tex (pip)

MPI build (why not?)

```
mkdir build
cd build
cmake ../ -DUSE_MPI=ON
make
```

non-MPI build (shame on you!)

```
mkdir build
cd build
cmake ../
make
```

installation and test suite

After compiling the code you will find the executable `analisi` and the shared library `pyanalisi.*.so` (with a part of the name that depends on your particular python installation)

in the build folder. To install the python library, simply copy the pyanalisi folder it in your site-library folder, and then copy inside it the `pyanalisi.*.so library`. This is done by the script `install/install_python.sh` after exporting the variables `SP_DIR`, setted to your python's package directory, `BUILD_DIR`, setted to the build directory, and `SOURCE_DIR`, setted to the main directory of the repository. The python's packages folders can be found with the command `python -m site`

```
export SP_DIR="/path/to/python/package/folder"
export SOURCE_DIR="/path/to/repository/dir"
export BUILD_DIR="/path/to/build/directory"
install/install_python.sh
```

You can set `SP_DIR=__DETECT__` to try to autodetect the python package directory. Be careful to choose the correct python executable, the same that you used to compile the library.

To test that the library was compiled correctly and that there are no regressions, you can run the (small) test suite with the command

```
make test
```

Then you can run the

```
test_cli.sh
```

script inside the `tests` folder, that tests the command line interface. You have to export the same variables used by `install/install_python.sh`. Then after the python library is installed, you can test it with

```
pytest -sv
```

in the tests folder. If all tests passed, your installation probably is working correctly. Testing is strongly suggested, you should *always* do it, since often you can find unstable environments or buggy compilers on you super-machine.

additional cmake options

- `-DBUILD_TESTS=OFF` skips the building of the C++ tests
- `-DSYSTEM_FFTW3=ON` use system's FFTW3 library
- `-DSYSTEM_EIGEN3=ON` use installed system's eigen3 library
- `-DSYSTEM_BOOST=ON` use detected system's boost library
- `-DSYSTEM_XDRFILE=ON` use detected system's xdrfile library. If not found simply disable the support for gromacs file conversion
- `-DPYTHON_EXECUTABLE=/path/to/python` use this python installation to build the python interface
- `-DPYTHON_INTERFACE=OFF` don't build any python interface (the building process will not depend on python)
- `-DBUILD_MMAP=OFF` don't use unix's mmap . This prevents the compilation of the c++ test suite and the command line tool. Only the python lib can be built.

Documentation

This document is better rendered in the pdf version. Link to the pdf version. Note that is generated by the script `build_pdf.sh`, and the original file is `README_.md`.

Command line interface

The command line utility is able to read only binary trajectory files in the LAMMPS format, specified with the command line option -i input_file, or a time series in a column formatted text file with a header, specified with the command line option -l input_file. The trajectory file can be generated in many ways: - by LAMMPS :-) - by using the command line utility with the command line options -i input_file -binary-convert output_file, where input_file is the name of a plain text trajectory in the format:

```
[natoms]
[xlo] [xhi]
[ylo] [yhi]
[zlo] [zhi]
[id_1] [type_1] [x_1] [y_1] [z_1] [vx_1] [vy_1] [vz_1]
 .      .        .     .     .     .      .      .
 .      .        .     .     .     .      .      .
 .      .        .     .     .     .      .      .
[id_natoms] [type_natoms] [x_natoms] [y_natoms] [z_natoms] [vx_natoms] [vy_natoms] [vz_natoms]
.
.
.
```

That is: for every step you have to provide the number of atoms, low and high coordinates of the orthorombic cell, and then for every atom its id, type id, positions and velocities. - by using the command line utility with the command line options -i input_file -binary-convert-gromacs output_file typefile and a gromacs trajectory (you have to provide the xdr library in the building process) - by using the python interface: see section Buffer protocol interface

Command line interface common arguments

Where meaningful, the following arguments are shared by all calculation performed by the command line utility - -i input_file specify input trajectory binary file. See lammps documentation for the documentation of how to produce a binary dump from lammps. The order of the dumped atomic quantities **must be** `id type xu yu zu vx vy vz` - -l input_file specify input time series in text column format with headers - -N thread_number specify the number of threads to use where parallelization is implemented with threads - -B number_of_block when the code calculates variances of the quantity, it splits the trajectory in many blocks. With this option you can specify the number of blocks. If MPI parallelization is used, since different blocks are calculated in parallel, you may want to spacify a number of blocks that is a multiple of the number of MPI processes. In this way, at the final iteration over the blocks, all processes will be busy. - -s timestep_number_skip usually neighbour configurations in the trajectory are strongly correlated. With this option you specify how far must be two consecutive configuration used to calculate

averages. - -S timestep_number_length specify how long must be the function that the code calculates, measured in number of timesteps. (for example the length of the MSD plot)

Python interface

Creating a trajectory object

You can create a trajectory python object to be used in this library in two ways: - start from python arrays generated by other code - use a LAMMPS binary file that you have on the filesystem. This is the same file that is used by the command line interface.

to access the positions, velocities and cell data you can use in both the python buffer protocol interface and the lammps binary one the functions `get_positions_copy()`, `get_velocities_copy()` and `get_box_copy()`. Note that each call of these functions allocates more memory, as needed.

Internal format used to store the cell information

Internally the format used for storing the cell information is:

`[x_low, y_low, z_low, lx/2, ly/2, lz/2, xy, xz, yz]`

and is accessible by using the python function `get_box_copy()` common to the trajectory objects. Note that internally the first cell vector is always in the same direction of the x axis, the second one is on the xy plane while the third has an arbitrary direction. This is equivalent to requiring that the cell matrix is triangular. If necessary, this is achieved by doing a QR decomposition of the input cell matrix and then rotating everything with the Q matrix.

using Python arrays: the buffer protocol interface

You must have 4 arrays. In general the interface supports any object that supports python's buffer protocol. This include, among others, numpy arrays. Suppose you have a trajectory of `t` timesteps and `n` atoms. You need the following arrays: - position array, of shape `(t,n,3)` - velocity array, of shape `(t,n,3)` - cell array, of shape `(t,3,3)` or if a lammps formated cell is provided `(t,6)` for orthorombic and `(t,9)` for triclinic. The lammps format simply list the low and high coordinates for each dimension and the tilts factors as described below - types array, of shape `(n)` (integer array)

In general no particular units of measure are required, and the output will reflect the units that you used in the input. The calculations that the program does are reported exactly in the following sections. From those formulas you can deduce the units of the output given the units that you used in the input.

Then you must decide if you want that the coordinates are rewrapped around the center of the cell or not. This is not equivalent to wrapping the coordinates inside the cell for the triclinic case, but generates a compact list of positions suitable for an efficient calculation of the minimum image distance.

The lammps format for the cell is `[x_lo, x_hi, y_lo, y_hi, z_lo, z_hi, xy, xz, yz]`, as described in the lammps documentation.

If the plain cell vectors are provided in the cell matrix and this matrix is not diagonal, a QR decomposition is performed to get a triangular cell matrix and the achieve the desidered internal format. In this case all velocities and positions vector are rotated with the rotation matrix Q, that can be obtained with the function `get_rotation_matrix()`.

The syntax for creating the object is

```
import pyanalisi
analisi_traj = pyanalisi.Trajectory(positions_array,
                                    velocity_array,
                                    types_array,
                                    box_array,
                                    input_box_format_id,
                                    wrap_atomic_coordinates,
                                    save_Q_rotation_matrix
)
```

where `input_box_format_id` is one of `pyanalisi.BoxFormat.CellVectors`, `pyanalisi.BoxFormat.LammpsOrtho`, `pyanalisi.BoxFormat.LammpsTriclinic` and describe the format of the array `box_array`. If `wrap_atomic_coordinates` is `True` the code will wrap all the atomic coordinates around the center of the cell (that does not mean inside the cell in the triclinc case). This makes the code for computing the minimum image distance more efficient if the atoms were far away out of the cell, but invalidates all the calculations were the atoms are not supposed to be wrapped back inside the cell.

You can write a LAMMPS binary trajectory (that can be used by the command line interface with mpi, for example) by calling

```
analisi_traj.write_lammps_binary('output_path', start_timestep, end_timestep)
```

where `start_timestep` is the first timestep index to dump (indexes start from 0) and `end_timestep` is the first timestep that will not be written. If `end_timestep == -1`, it will dump everything till the end of the trajectory. This is a very convenient way of moving heavy computations on a cluster where MPI can be used, or more in general to convert a generic trajectory format in the format used by the command line tool. For example

```
#read trajectory. It can come from everywhere
 import numpy as np
 pos = np.load( 'tests/data/positions.npy') #shape (N_timesteps, N_atoms, 3)
 vel = np.load( 'tests/data/velocities.npy') #shape (N_timesteps, N_atoms, 3)
 box = np.load( 'tests/data/cells.npy') #shape (N_timesteps, 3, 3)
 types = np.load( 'tests/data/types.npy') #shape (N_atoms), dtype=np.int32

 #create trajectory object and dump to file
 import pyanalisi
 analisi_traj = pyanalisi.Trajectory(pos, vel, types, box,
                                     True, # matrix format for the box array
                                     False, # don't wrap the coordinates
                                     False # not interested in Q matrix
                                     )
 analisi_traj.write_lammps_binary("output_filename.bin"
                              , 0, # starting timestep
```

```
-1   # last timestep:
)    #  -1 dumps full trajectory
```

LAMMPS binary trajectory interface

This interface is a little more complicated, since it was designed for computing block averages of very big files. It can read the same files that the command line program reads. The object is created with

```
analisi_traj = pyanalisi.Traj('path_of_binary_file')
```

Then you have to call some more functions, BEFORE calling the compute routines :

```
analisi_traj.setWrapPbc(True) #optional: if you want to wrap positions inside the cell
analisi_traj.setAccessWindowSize(size_in_number_of_steps_of_the_reading_block)
analisi_traj.setAccessStart(first_timestep_to_read)
```

The first line is to set the pbc wrapping (don't use this if you have to compute the MSD!). Since the access to the trajectory is done in blocks, the second line sets the size of the reading block. The bigger the block the bigger the memory allocated to store the positions and the velocities. The last call sets the index of the first timestep that is going to be read, and then read it. In this moment the selected chunk of the trajectory is loaded into your memory. At this point you are able to call the needed compute routine, making sure that it is not going to read past the last timestep of the block. Later you can call again `setAccessStart` to load a different part of the trajectory and compute the quantity again. Only the data of the current block is stored in the memory, and if evenutally there is some overlap with the previous block, the data is copied from memory and the overlapped part is not read again from the filesystem.

The `Traj` class has a normal buffer protocol interface, so you can use it as a numpy array. By defaults the array interfaced with python is the position's one. If you want to read the velocities, you can do

```
analisi_traj.toggle_pos_vel() # to change from positions to velocities and vice-versa
#do whatever you want
if not analisi_traj.serving_pos():
   np.sum(analisi_traj) # sum all velocities
else:
   np.sum(analisi_traj) # sum all positions
```

To access the atomic lammps ids and the atomic lamms types, you can use the following two functions:

```
analisi_traj.get_lammps_id()
analisi_traj.get_lammps_type()
```

don't call them too often since every time a new numpy array of ints is created

Common functions

Some functions are common to both the LAMMPS and the numpy interfaces:

- `get_positions_copy()` that returns a copy of the loaded positions

- `get_velocities_copy()` that returns a copy of the loaded velocities
- `get_box_copy()` that returns a copy of the cell data stored in the intenal format
- `write_lammps_binary(fname, start, stop)` that writes a dump of the selected part of the trajectory in the lammps format. This is useful also in the lammps interface for extracting a part of the full trajectory.

Creating a time series object

Before analyzing a time series, for example to calculate the integral of the autocorrelation function, it is necessary to create a time series object. This object will hold the data that a different function can analyze. It is created with the following:

```
time_series = pyanalisi.ReadLog(data_array, headers_array)
```

where `data_array` and `header_array` are python objects that support the buffer protocol interface, for example numpy arrays, or a plain python list. The `data_array` is a two dimensional array where the first index is the timestep index and the second index is the column index. The `header_array` object must describe the columns that are stored in the bidimensional floating point array `data_array` using Python Strings.

MSD

Given a trajectory ${}^{\mathbf{i}}\mathbf{x_t}$ where $i \in \{1, \ldots, N_{atoms}\}$ is the atomic index and t is the timestep index, defining the center of mass position of the atomic species j at the timestep t as

$$ {}^{j}cm_t = \frac{1}{N_j} \sum_{i|type(i)=j} \mathbf{x}_t $$

where N_j is the number of atoms of the specie j, the code computes the following

$$ MSD_t^{typej} = \frac{1}{N_{typej}} \sum_{i|type(i)=typej} \frac{1}{N_{ave}} \sum_{l=1}^{N_{ave}} \left| {}^{i}\mathbf{x}_{t+l} - {}^{i}\mathbf{x}_l \right|^2 $$

$$ MSDcm_t^{typej} = \frac{1}{N_{ave}} \sum_{l=1}^{N_{ave}} \left| {}^{typej}cm_{t+l} - {}^{typej}cm_l \right|^2 $$

If the option `--mean-square-displacement-self` is provided in the command line or in the python interface the documented argument is set to `True`, the atomic mean square displacement for each atomic species is calculated in the reference system of the center of mass of that particular atomic specie. That is, in this case the following is computed:

$$ MSD_t^{typej} = \frac{1}{N_{typej}} \sum_{i|type(i)=typej} \frac{1}{N_{ave}} \sum_{l=1}^{N_{ave}} \left| ({}^{i}\mathbf{x}_{t+l} - {}^{typej}cm_{t+l}) - ({}^{i}\mathbf{x}_l - {}^{typej}cm_l) \right|^2 $$

$$ MSDcm_t^{typej} = \frac{1}{N_{ave}} \sum_{l=1}^{N_{ave}} \left| {}^{typej}cm_{t+l} - {}^{typej}cm_l \right|^2 $$

In the output you have many columns, one for each of the N_{types} atomic species, first the block of the atomic MSD and then eventually the block of the center of mass MSD if asked to compute.

The center of mass MSD is computed only if the command line option `-Q` is provided or the documented argument is set to `True` in the python interface constructor. The output is the following:

$$\{MSD_t^1, \ldots, MSD_t^{N_{types}}, MSDcm_t^1, \ldots, MSDcm_t^{N_{types}}\}$$

In the command line output after each column printed as described in the line above you will find the variance calculated with a block average over the specified number of blocks.

The options that you can use for this calculation, in summary, are:

./analisi -i lammp_binary [-N number of threads] [-S stop timestep] [-s skip every] [-B number of blocks] -q | -Q [–mean-square-displacement-self]

where any option of `-q`, `-Q`, `--mean-square-displacement-self` activates the calculation of the MSD.

To use this calculation in the python interface, you have to first create a trajectory object (that can be both from a lammps binary or from python arrays objects - described in the section creating a trajectory object), then you have to create the following instance if the numnpy array trajectory is used:

```
msd_calculation = pyanalisi.MeanSquareDisplacement(
    trajectory_instance, # Trajectory instance
    skip, # in calculating averages skip this amount of timestep,
          # as in -s option
    tmax, # size of the MSD(t) plot in number of timesteps
    nthreads, # number of parallel threads to use in the computation
    center_of_mass, # if True calculates center of mass displacement too
    use_cm_reference, # use the reference system of the center of mass
                      # of all atoms of the same type
    debug_flag # if True produces some dump files...
   )
```

if the lammps binary trajectory instance is used, you must use the `pyanalisi.MeanSquareDisplacement_lammps`, with exactly the same arguments. The calculation will be exactly the same. After initializing the object, the calculation can be initialized with

```
msd_calculation.reset(block_size)
```

that sets the number of steps that will be used to calculate the average over the trajectory. If `tmax` is greater than `block_size`, then `tmax` will be setted to `block_size`. The calculation is started with:

```
msd_calculation.calculate(first_timestep)
```

where `first_timestep` is the index of the first timestep that will be used to calculate the averages. After the calculation over this block is finished, the result can be collected in a numpy array with:

```
result = np.array(msd_calculation, copy=False)
```

In general the object msd_calculation supports the buffer protocol interface, so to get the result you can use anything that can interface with the buffer.

Green-Kubo

Given M vector time series of length N ${}^{m}\mathbf{J}_t$, $m \in \{1 \dots M\}$, $t \in \{1 \dots N\}$, implements an expression equivalent to the following formula:

$$
\begin{aligned}
{}^{ij}C_l &= \frac{1}{N_{ave}} \sum_{m=1}^{N_{ave}} \frac{1}{3} \sum_{c=1}^{3} {}^{i}J_m^c \cdot {}^{j}J_{m+l}^c \\
{}^{ij}L_t &= \sum_{l=0}^{t} {}^{*} \, {}^{ij}C_l \\
{}^{ij}\bar{L}_t &= \frac{1}{t} \sum_{l=0}^{t} {}^{*} \, {}^{ij}C_l \cdot l \\
GK_t &= \frac{1}{{}^{00}[(L_t)^{-1}]} \\
G\bar{K}_t &= \frac{1}{{}^{00}[(L_t - \bar{L}_t)^{-1}]}
\end{aligned}
$$

but with the trapezoidal rule in place of the sums marked with *. Note that L_t is a matrix. To get the correct units of measure, you have still to multiply all the quantities but the C_ts by the integration timestep. N_{ave} is the number of timesteps on which the code runs the average. Every quantity is written in the output in the following order:

$$\{{}^{00}C_t, {}^{00}L_t, {}^{00}\bar{L}_t, {}^{01}C_t, {}^{01}L_t, {}^{01}\bar{L}_t, \dots, {}^{MM}C_t, {}^{MM}L_t, {}^{MM}\bar{L}_t, GK_t, G\bar{K}_t\}$$

If the command line tool is used, the variance of the block average is automatically computed, and after each column you find its variance. Moreover you find in the output a useful description of the columns with their indexes.

g(r,t)

Given a central atom of type j at timestep l, calculates the histogram of the minimum image's radial distance of atoms of type i at timestep $l + t$. The histogram can be of the same atom (you now, it spreads arond while time is passing) or of all atoms different from the original one. Everything is averaged over l and atoms of the same type. In particular, defining the index of a pair of atoms i and j and a timelag t at a timestep l as

$${}^{ij}h_t = \lfloor (|{}^{j}\mathbf{x}_{l+t} - {}^{i}\mathbf{x}_l|_{\text{min.image}} - rmin)/dr \rfloor,$$

the histogram $g(r,t)$ between specie I and J is defined as

$${}^{IJ}g_t^r = \sum_{i|type(i)=I} \sum_{j|type(j)=J} \delta({}^{ij}h_t, r)$$

where $\delta(\cdot,\cdot)$ is the Kronecker delta. In practice the program, for each atoms, it adds 1 to the corresponding bin of the histogram.

For each t, I, J, with $J \geq I$, the position of the histograms in the memory is $N_{types}(N_{types} + 1)/2 - (J+1)(J+2)/2 + I$ (0 is the first). Given this order of the histograms, the layout in the memory is the following:

$$\{{}^{0}g_t^0, \dots, {}^{0}g_t^{r_{max}}, \dots, {}^{N_{types}(N_{types}+1)/2}g_t^0, \dots, {}^{N_{types}(N_{types}+1)/2}g_t^{r_{max}}, \dots\}$$

The layout of the command line output is a gnuplot-friendly one, where the output is organized in blocks, one for each t, separated by two blank lines, and every line corresponds to a histogram bin for every combination of I, J:

$$
\begin{array}{lll}
\{ & {}^{0}g_0^0, \dots & ,^{N_{types}(N_{types}+1)/2}g_0^0 \quad \} \\
\{ & \vdots,\vdots & ,\vdots \quad \} \\
\{ & {}^{0}g_0^{r_{max}}, \dots & ,^{N_{types}(N_{types}+1)/2}g_0^{r_{max}} \quad \} \\
\vdots & & \\
\{ & {}^{0}g_{t_{max}}^0, \dots & ,^{N_{types}(N_{types}+1)/2}g_{t_{max}}^0 \quad \} \\
\{ & \vdots,\vdots & ,\vdots \quad \} \\
\{ & {}^{0}g_{t_{max}}^{r_{max}}, \dots & ,^{N_{types}(N_{types}+1)/2}g_{t_{max}}^{r_{max}} \quad \}
\end{array}
$$

where we collapsed the indexes I, J into a single number, the position order of the histogram. As usual in the command line output every column is an average over all the blocks and it is followed by the variance of the mean.

Spherical harmonics correlations

Calculation procedure:

The implemented formula for the real spherical harmonics is the following:

$$
Y_{\ell m} = \begin{cases}
(-1)^m \sqrt{2} \sqrt{\dfrac{2\ell+1}{4\pi} \dfrac{(\ell-|m|)!}{(\ell+|m|)!}}\ P_\ell^{|m|}(\cos\theta)\ \sin(|m|\varphi) & \text{if } m < 0 \\
\sqrt{\dfrac{2\ell+1}{4\pi}}\ P_\ell^m(\cos\theta) & \text{if } m = 0 \\
(-1)^m \sqrt{2} \sqrt{\dfrac{2\ell+1}{4\pi} \dfrac{(\ell-m)!}{(\ell+m)!}}\ P_\ell^m(\cos\theta)\ \cos(m\varphi) & \text{if } m > 0\,.
\end{cases}
$$

Where $\cos\theta$, $\sin\varphi$, $\cos\varphi$ are calculated using cartesian components:

$$
\begin{aligned}
\cos\theta &= \frac{z}{\sqrt{x^2+y^2+z^2}} \\
\cos\varphi &= \frac{x}{\sqrt{x^2+y^2}} \\
\sin\varphi &= \frac{y}{\sqrt{x^2+y^2}}
\end{aligned}
$$

and $\sin|m|\varphi$, $\cos|m|\varphi$ are evaluated using Chebyshev polynomials with a recursive definition:

$$
\begin{cases}
\cos m\varphi &= 2\cos(m-1)\varphi\cos\varphi - \cos(m-2)\varphi \\
\sin m\varphi &= 2\cos\varphi\sin(m-1)\varphi - \sin(m-2)\varphi
\end{cases}
$$

and P_ℓ^m are the associated Legendre polynomials, calculated with the following set of recursive definition:

$$P_{\ell+1}^{\ell+1}(x) = -(2\ell+1)\sqrt{1-x^2}P_\ell^\ell(x) P_{\ell+1}^\ell(x) = x(2\ell+1)P_\ell^\ell(x)$$

that allows us to calculate every (ℓ,ℓ) and every $(\ell,\ell+1)$ element of the (ℓ,m) values. Then we have the recursion to go up in ℓ, for any value of it:

$$P_\ell^m = \frac{x(2\ell-1)P_{\ell-1}^m - (\ell+m-1)P_{\ell-2}^m}{\ell-m}$$

The program, given a number ℓ_{max} and a triplet (x,y,z), is able to calculate every value of $Y_{\ell m}(x,y,z) \forall \ell \in [0,\ell_{max}]$ and for all allowed values of m with a single recursion. Let

$$y_{\ell m}^{Ij}(t) = \int_{V_I} \mathrm{d}\theta\mathrm{d}\varphi\mathrm{d}r\rho^j(r,\theta,\varphi,t)Y_{\ell m}(\theta,\varphi)$$

for some timestep t in some volume V_I taken as a the difference of two concentric spheres centered on the atom I of radius r_{inner} and r_{outer}, and where ρ_j is the atomic density of the species j. Since the densities are taken as sums of dirac delta functions, it is sufficient to evaluate the spherical harmonics functions at the position of the atoms. Then we calculate the following:

$$c_\ell^{Jj}(t) = \langle \frac{1}{N_J} \sum_{I \text{ is of type } J} \sum_{m=-\ell}^{\ell} y_{\ell m}^{Ij}(0) y_{\ell m}^{Ij}(t) \rangle$$

where $\langle\cdot\rangle$ is an average operator, and we do an additional average over all the N_J central atoms of the type J. The $\langle\cdot\rangle$ average is implemented as an average over the starting timestep.

Vibrational Spectrum

Calculation procedure:

The implemented equation is:

$$D_\alpha^k(\omega) = \frac{1}{3N_\alpha} \sum_n^{N_\alpha} \int_{-\infty}^{\infty} \langle v_n^k(0) v_n^k(t) \rangle e^{i\omega t} dt$$

where α is the type of atom and k is one of the spatial direction x,y,z. The diffusivity can be computed as half of the zero value of D.

The columns of the output are ordered like the following:

$$\begin{array}{lll} \{0 & ,D_0^x(\omega_0), D_0^y(\omega_0), D_0^z(\omega_0), & \ldots, D_M^x(\omega_0), D_M^y(\omega_0), D_M^z(\omega_0) \quad \} \\ \{\vdots & ,\ldots, & \ldots,\vdots \quad \} \\ \{N & ,D_N^x(\omega_N), D_N^y(\omega_N), D_N^z(\omega_N), & \ldots, D_M^x(\omega_N), D_M^y(\omega_N), D_M^z(\omega_N) \quad \} \end{array}$$

where M is the number of atomic species and N the number of timesteps. Note that in the command line versione each column but the first is followed by its variance

SANN algorithm

The algorithm is described in J. Chem. Phys. 136, 234107 (2012); van Meel, Filion, Valeriani,Frenkel.

It is implemented internally in the Steinhardt descriptor python ojects and the algorithm is exposed in the python interface as the following:

```
nns=pa.Neighbours(tr,[(40,4.0**2,0.0),(35,3.5**2,0.0)])
```

where `tr` is a Trajectory object, and you must provide the list of the maximum number of atoms and maximum cutoff square for each atomic species. The last number of the 3-tuple is not used at the moment. You can calculate the neighbour list for each atom for a particular timestep:

```
nns.calculate_neigh(0,True)
```

where the first argument is the timestep index, and if the second argument is true the list is sorted by distance from the central atom. Then you can ask for the list of atomic indexes with the following function call:

```
nns.get_neigh_idx(1,0)
```

where the first argument is the atomic index and the second argument is the list index, one for each type. You can ask also for the list of the atomic distances (in order: modulus and vector) with the call:

```
nns.get_neigh(1,0)
```

In the same way you can ask for the SANN first neighbor list:

```
nns.get_sann(1,0)
```

or, for the atomic indexes only:

```
nns.get_sann_idx(1,0)
```

Steinhardt descriptors

To distinguish the various phases of the system this is a useful tool. It is defined as

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

$$q_{lm}(i) = \frac{1}{N_b(i)}\sum_{j=1}^{N_b(i)} Y_{lm}(\mathbf{r_{ij}})$$

where Y_{lm} are the spherical harmonics. An issue with the Steinhardt descriptors, particularly relevant for some system, is that it is not able to distinguish in a nice way the local BCC and HCP structures when computing the $(q_6(i), q_4(i))$ histogram. This issue is solved by the averaged

order parameter, a slightly different version that is defined as:

$$\bar{q}_l(i) = \sqrt{\frac{4\pi}{2l+1}\sum_{m=-l}^{l}|\bar{q}_{lm}|^2}$$
$$\bar{q}_{lm}(i) = \frac{1}{N_b(i)}\sum_{k=0}^{N_b(i)} q_{lm}(k)$$

where the last sum is performed on all the first neighbors plus the particle itself. To find the nearest neighbors we used the SANN[**?**] algorithm.

The python interface allows you to evaluate the descriptors for all the atoms or compute directly histograms of the descriptors value. Classes in the python interface are called `SteinhardtOrderParameterHistogram*`. The usage is documented in the docstring.

command line version

The options that you can use for this calculation are simply:

./analisi -i lammp_binary [-N number of threads] [-V] [-B number of blocks]

the code will generate a file with a number of line equal to the number of frequencies, and with `ntypes_atoms*2+1` columns where:

- the first column rappresent the index of the frequencies
- then there are the block average and variance of the spectrum for each atomic type

The code is trasparent ot units of measuares of the quantities. If a user wants the diffusivity in the correct units (e.g. metal) must porcede in the following way:

- the first column can be multiplied by `1/(nstep*dt)` to obtain the frequencies in multiples of Hz
- the other columns can be multiplied by `dt/nstep`;

where `nstep` is the total number of step of the block used to compute VDOS, `dt` is the time difference of two consecutive molecular dynamis steps.

python interface

There are two optimal methods of computing the VDOS with the python interface depending on which trajectory are you using. The two function can be found in the `common.py` file

- with the lammps `Traj`: `analyze_vdos_lammps(traj,nstep=None,start=0,resetAccess=True,print=print)`
  - `traj` is the lammps trajectory file;
  - `nstep` is the number of step to use in the computation of the VDOS; if `None` all steps are included;
  - `start` is the starting step to use as starting point of the trajectory;
  - `resetAccess` if true resets `traj.setAccessWindowSize(traj.getNtimesteps())` and `traj.setAccessStart(0)`
- with numpy `Trajectory` interface: `analyze_vdos_numpy(pos,vel,types,box,nstep=ltot,start=start)`
  .

- `pos` : positions matrix (N_timesteps, N_atoms, 3)
- `vel` : velocities matrix (N_timesteps, N_atoms, 3)
- `types` : types vector (N_atoms)
- `box` : box matrix

- `matrix_format` :bool matrix format for the box array
- `wrap`: bool wrap coordinate
- `nstep` : int nstep to use for the computation of vdos if None it uses all
- `start` : int initial step The first 6 input are the same of the numpy `Trajectory` interface

Both functions returns a numpy array with dimensions: `(ntypes, freq, Cartesian_coordinate)`. The units are the same of the `command line version`, thus the matrix must be multiplied by `dt/nstep`.

Abstract multithreaded calculation C++ class implementation

The `CalcolaMultiThread` class tries to abstract away common features of most of the calculations over the trajectory, allowing to easily write parallel calculations that take advantage of the modern multicore architectures of every computer.

Abstract block averages calculation for transparent use of MPI

All the logic about MPI and block averages is self contained in `mediablocchi.h`, a small file of only ~200 lines, that take care of doing block averages of both the MPI and the non MPI case. Calculations that needs (or take advantage of) block averages and want to use this tool must implement few functions, as found in all the classes, and must use the Traiettoria class to read the trajectory. In this way the user can concentrate on implementing the actual calculation without thinking of the boilerplate code needed to compute block averages and MPI-parallelize it. The calculation class with the data that must be averaged over the blocks must be a class derived from `OperazioniSuLista`

Credits

Written by Riccardo Bertossa during his lifetime at SISSA