analisi: your Swiss Army Knife of molecular dynamics analysis

Fastest possible example usage:

command line

- 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
```

many others…

python

```
#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')
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, vel, types, box,True, False)

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

```
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
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
traj = pyanalisi.ReadLog(log, headers)
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)
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

Features:

- python interface (reads numpy array)
- command line interface (reads binary lammps-like files)
- multithreaded
- command line interface has MPI too (for super-heavy calculations)
- command line calculates variance of every quantity and every function (in python you can do it by yourselves with numpy.var)

Calculations:

- g of r (and time too!)

- vibrational spectrum (this is nothing special)
- 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
- and more …
- …

Building from source

Dependencies:

- C++17 capable compiler
- cmake
- linux (mmap) (maybe can be removed if only python interface is needed)
- FFTW3 (included in the package)
- Eigen3 (included in the package)
- Boost (included in the package)
- Mpi (optional)
- libxdrfile (for gromacs file conversion – optional)
- python (optional)

If you want to use system's fftw3 library, you have to provide to cmake the option:

```
-DSYSTEM_FFTW3=ON
```

If you don't want any python interface you have to provide to cmake the option:

```
-DPYTHON_INTERFACE=OFF
```

MPI build (why not?)

```
mkdir build
cd build
cmake ../ -DCMAKE_CXX_COMPILER=mpicxx -DCMAKE_C_COMPILER=mpicc -DUSE_MPI=ON
make
```

non-MPI build (shame on you!)

```
mkdir build
```

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

Documentation

This document is better rendered in the pdf version. Link to the pdf version.

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_natom
.
.
.
```

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 [trr_file] -binary-convert-gromacs [output_file] [typefile]` and a gromacs trajectory (you have to provide the xdr library) - by using the python interface: see section Buffer protocol interface

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 that some other routine that you have gived to you - use a LAMMPS binary file that you have on the filesystem. This is the same file that is used by the command line interface.

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)` only diagonal matrices (orthorombic cells) are supported at the moment, or if a lammps formated cell is provided `(t,6)` - types array, of shape `(n)`

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 inside the cell or not. At the moment only orthorombic cells are supported in all calculations but those that need only unwrapped coordinates, like MSD.

The lammps format for the cell is `[x_lo, x_hi, y_lo, y_hi, z_lo, z_hi]`: you have to provide only the coordinates boundaries.

The syntax for creating the object is

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

where `use_matrix_or_lammps_cell_format` is `True` if usual matrix format for the cell is given and `False` if a lammps formatted cell is provided and `wrap_atomic_coordinates` is `True` if you want to wrap all the atomic coordinates inside the cell.

You can write a LAMMPS bynary trajectory (that can be used by the command line interface) 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 writed. If `end_timestep == -1`, it will dump 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, False)
 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. 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)
```

then call the relevant 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 compute the quantity in a different region of the trajectory. Only the data of the current block is stored in the memory.

Creating a time series object

TODO

MSD

Given a trajectory $\mathbf{{}^i x_t}$ where $i \in \{1, \dots, N_{atoms}\}$ 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

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

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:

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

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.

GreenKubo

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:

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

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:

$$\{{}^{0\,0}C_t, {}^{0\,0}L_t, {}^{0\,0}\bar{L}_t, {}^{0\,1}C_t, {}^{0\,1}L_t, {}^{0\,1}\bar{L}_t, \dots, {}^{M\,M}C_t, {}^{M\,M}L_t, {}^{M\,M}\bar{L}_t, GK_t, \bar{GK}_t\}$$

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

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

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

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

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:

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

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:

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

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:

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

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.

Credits

Written by Riccardo Bertossa during his lifetime at SISSA