MonteModes Documentation

Release latest

Mar 29, 2018

Table of contents

Montemodes is a software to calculate Monte Carlo simulations using the Metropolis algorithm. This software connects with Gaussian or Tinker to calculate the energy at each step.

CHAPTER 1

How to install

Download the code and use :file: *setup.py* to install the code using :program: *setuptools* python module. A simple setup could be

```
$ setup.py install --user
```

the code will be installed as a python module. To check that it is properly installed you can run the :program: ‘python interpret and execute

```
import montemodes
```

if the module should be loaded without warnings and errors

CHAPTER 2

Structure object

$ Structure(coordinates=None, internal=None, z_matrix=None, int_label=None, atom_types=None, atomic_elements=None, atomic_numbers=None, connectivity=None, file_name=None, charge=0, multiplicity=1, int_weights=None):

2.1 Methods

def get_coordinates(): return self._coordinates.copy()

def set_coordinates(self, coordinates): self._coordinates = coordinates self._number_of_atoms = None self._energy = None

def get_internal(): return self._internal.copy()

def set_internal(internal): self._internal = internal

def get_full_z_matrix(): return self._full_z_matrix

def get_z_matrix(): return self._z_matrix

def set_z_matrix(z_matrix): self._z_matrix = z_matrix

def get_int_label(): return self._int_label

def set_int_label(int_label): self._int_label = int_label

def get_int_dict(): self._internal_dict = {} for i, label in enumerate(self.get_int_label()[:,0]):

self._internal_dict.update({label:self.get_internal()[i, 0]})

return self._internal_dict

def get_int_weights(): return self._int_weights

def set_int_weights(int_weights): self._int_weights = int_weights

def get_atomic_elements_with_dummy(): return self._atomic_elements

def get_atom_types(): return self._atom_types

def set_atom_types(atom_types): self._atom_types = atom_types

def get_atomic_numbers(): return self._atomic_numbers

def set_atomic_numbers(atomic_numbers):

def get_atomic_elements(self): return np.array([i for i in self._atomic_elements if i != "X"], dtype=str)

def set_atomic_elements(atomic_elements):

def get_connectivity(): return self._connectivity

def set_connectivity(connectivity):

def get_number_of_atoms(): return self._number_of_atoms

def get_number_of_internal(): return self._number_of_internal

def get_energy(method=None): return self._energy

def get_modes(method=None): return self._modes

def get_atomic_masses(self): return self._atomic_masses

2.2 Properties

@property def charge(self):

return self._charge

@property def multiplicity(self):

return self._multiplicity

2.2.1 example

```
initial_coordinates = [[ 0.5784585,  0.7670811,  1.3587379],
                       [-1.7015514, -0.0389921, -0.0374715],
                       [ 0.5784290, -1.6512236, -0.0374715],
                       [ 0.5784585,  0.7670811, -1.4336809],
                       [ 0.0000000,  0.0000000,  0.0000000]]
initial_coordinates = np.array(initial_coordinates)

atomic_elements = ['O', 'O', 'O', 'O', 'P']
atomic_elements = np.array(atomic_elements)[None].T

molecule = Structure(coordinates=initial_coordinates,
                     atomic_elements=atomic_elements)

molecule.charge = 0
molecule.multiplicity = 1
```

CHAPTER 3

MonteCarlo simulation

3.1 Prepare the simulation

3.1.1 Calculation method

The calculation method define the software to use to calculate the atomic interactions. Two software are implemented: **`Gaussian09`** and **`Tinker`**. To use these calculators, Gaussian and/or Tinker should be installed in your system and the binaries (or hard links to them) should be placed in a directory included in the $PATH environment variable with the name **`g09`** and **`tinker`**, respectively. Note that setting up an alias in `.profile` or `.bashrc` for these software will not work.

Gaussian 09

```
from montecarlo.functions.method import gaussian
calculator = gaussian(methodology='pm6' [String],
                      internal=False [Boolean],
                      processors=None [Integer])
```

- methodology: Method label to use in **`Gaussian09`**. This argument should contain the basis set if it is necessary (Ex: B3LYP/6-31G).
- Internal: Use internal coordinates (z-matrix).
- Set number of processors to use in the :program:Gaussian09‘ calculation.

Tinker

```
from montecarlo.functions.method import tinker
calculator = tinker(parameter_set='mm3.prm' [String]):
```

- parameter_set: name of the force field file to use. This file should be placed in the work directory or full path should be specified

3.1.2 Conditions

Conditions object contains the parameters of the Monte Carlo simulation

```
from montecarlo.classes.results import Conditions
conditions = Conditions(temperature=None [Float],
                        number_of_cycles=100000 [Integer],
                        kb=0.0019872041, # kcal/mol
                        initial_expansion_factor=0.05 [Float],
                        acceptation_regulator=1.0 [Float],
                        number_of_values_for_average=2000 [Integer],
                        energy_method=calculator [Method object])
```

- temperature: Temperature at which the MonteCarlo simulation is calcuculated.
- number_of_cycles: Number of simulation steps
- energy_method: Calculation method object
- initial_expansion_factor: Initial factor of acceptance
- acceptation_regulator: Ratio of alteration of the factor of acceptance as a function of acceptance.
- number_of_values_for_average: Number of last simulation steps used to calculate the averaged properties.
- kb: Boltzmann constant according to the units of energy and temperature

3.2 Run the simulation

Montecarlo object contains all the information concerting to the simulation. This object is generated from a structure object tha contains the initial structure

```
from montecarlo.classes.results import MonteCarlo
simulation = MonteCarlo(structure)
```

To Run the simulation the *calculate_MonteCarlo()* is used. This function returns a MonteCarlo object that contains the results

```
from montecarlo.functions.montecarlo import calculate_MonteCarlo
simulation = calculate_MonteCarlo(simulation [Montecarlo type],
                                  conditions [Conditions type],
                                  show_text=True [Boolean],
                                  alteration_type='cartesian' [String])
  Return result [Montecarlo type]
```

- simulation: Initial Montecarlo object. If this object already contains information of a previous simulation, the simulation will continue adding the data of the new simulation.
- conditions: Conditions object.
- show_text: If True writes montecarlo information on screen during the simulation calculation. If False the calculation is carried out silently.
- alteration_type: Defines the way the structures are altered during each simulation step. The possible options are ‘cartesian’ ‘internal’ or ‘modes’.

The returned Montecarlo object can be used again in the *calculate_MonteCarlo()* function to continue the simulation.

3.3 Save results to data files

To save the MonteCarlo data into files some helper functions are available in

```
montemodes.functions.reading
```

Save the energy, acceptation of each simulation

```
write_result_to_file(result, 'test.out')
```

Save the trajectory into a file in xyz format

```
write_result_trajectory(result.trajectory, 'trajectory.xyz')
```

Save the full simulation objects into a file

```
save_to_dump(conditions, result, filename='full.obj')
```

Load the simulation objects from a file

```
load_from_dump(filename='full.obj')
```

3.4 Example

```
import montemodes.functions.reading as io_monte
import montemodes.functions.montecarlo as monte
import montemodes.functions.methods as method
import montemodes.classes.results as res


gaussian_calc = method.gaussian(methodology='pm6',
                                internal=False)

conditions = res.Conditions(temperature=500,
                            number_of_cycles=1000,
                            initial_expansion_factor=0.05,
                            acceptation_regulator=0.1,
                            number_of_values_for_average=20,
                            energy_method=gaussian_calc)

initial_structure = io_monte.reading_from_xyz_file('molecule.xyz')
initial_structure.charge = 0
initial_structure.multiplicity = 1

simulation = res.MonteCarlo(initial_structure)

result = monte.calculate_MonteCarlo(simulation,
                                    conditions,
                                    show_text=True,
                                    alteration_type='cartesian')

io_monte.write_result_to_file(result, 'montecarlo.out')
io_monte.write_result_trajectory(result.trajectory, 'trajectory.xyz')
```

CHAPTER 4

Symmetry and shape analysis

The symetry and shape analysis is calculated using the external software :program: *shape*, :program: *symop*, and :program: *symgroup*. To use the interfaces to the binaries have to be placed in a directory included in the $PATH environment variable with the name **shape**, **symop**, and **symgroup**, respectively. Note that setting up an alias in `.profile` or `bashrc` for these software will not work.

4.1 Shape

To use shape interface, shape module has to be loaded by

```
$ module load montemodes.functions.shape as shape
```

The use of shape module is divided in two parts: First a shape input object is created. This shape object defines the kind of shape calculation to perform

```
$ input_shape = shape.Shape(code=1,
                            central_atom=0,
                            custom_atom_list=None)
```

- code: corresponds to the shape code available in shape manual. It depends on the number of vertices.
- central_atom: defines the atom number that will be used as central atom. Atom number uses the same rule as **shape** where the first atom is atom number 1. If central_atom is 0 no central atom will be defined.
- custom_atom_list: defines a list of atoms of the structure that will be used in the shape calculation. If this value is None all atoms are used.

4.1.1 methods

- get_shape(structure [type Structure], input_shape [type Shape]):

  Return: Float

Get the shape measure of a structure

- get_shape_trajectory(trajectory [list of Structure objects], input_shape [type Shape]):

  Return: List of Float

Get the shape of a list of structures

- **get_info(vertices=None):** Return: Null

Get information about available shapes. (equivalent to shape +).

4.1.2 example

```
$ import montemodes.functions.reading as io_monte
$ import montemodes.functions.shape as shape

$ structure = io_monte.reading_from_xyz_file('ch4.xyz')
$ input_shape = shape.Shape(code=2,
                            central_atom=1,
                            custom_atom_list=None)

$ measure = get_shape(structure, input_shape)
$ print ('The T-4 shape measure of CH4 is {}'.format(measure))
```

4.2 Symop

To use symop interface, symop module has to be loaded by

```
$ module load montemodes.functions.symop as symop
```

Likewise shape module, symop module is divided in two parts: First a symop input object is created. This symop object defines the kind of symmetry calculation to perform

```
$ input_symop = symop.Symop(symmetry='c 3',
                            label=False,
                            connect=False,
                            central_atom=0,
                            custom_atom_list=None)
```

- symmetry: corresponds to the symmetry operation to be measured.
- label : if True adds %label keyword to symop input (check symop manual for further information).
- connect : if True adds %connect keyword to symop input (check symop manual for further information).
- central_atom: defines the atom number that will be used as central atom. Atom number uses the same rule as **`symop`** where the first atom is atom number 1. If central_atom is 0 no central atom will be defined.
- custom_atom_list: defines a list of atoms of the structure that will be used in the shape calculation. If this value is None all atoms are used.

4.2.1 methods

- get_symmetry(structure [type Structure], symop_input [type symop]):

Return: Float

Get the symmetry measure of a structure.

- get_symmetry_trajectory(trajectory [type Structure], symop_input [type symop]):

  Return: List of Float

Get the symmetry measure of a list of Structure type objects.

4.2.2 example

```
$ import montemodes.functions.reading as io_monte
$ import montemodes.functions.symop as symop

$ structure = io_monte.reading_from_xyz_file('ch4.xyz')

$ input_symop = symop.Symop(symmetry='c 3',
                            label=False,
                            connect=False,
                            central_atom=0,
                            custom_atom_list=[1,2,3,4])

$ measure = get_symmetry(structure, input_symop)
$ print ('The C3 symmetry measure of CH4 is {}'.format(measure))
```

CHAPTER 5

Distortion index

5.1 Functions

The calculation of the distortion index as defined in the article: *Baur WH. Acta Crystallogr Sect B Struct Crystallogr Cryst Chem. 1974;30(5):1195–215.* is implemented in the functions

```
import montecarlo.analysis.distorsion as distorsion
distorsion.get_distortion_indices_angles(structure [Structure], 'A', 'B', 'C')
    Return distortion_index [Float]

distorsion.get_distortion_indices_distances(structure [Structure] , 'A', 'B')
    Return distortion_index [Float]
```

where structure is a Structure type object and 'A', 'B', and 'C' are the chemical symbol of the elements to analyze.

These functions can be called from a list of Structure type objects to return a dictionary with statistic data

```
dist_OPO = distorsion.get_distortion_statistic_analysis(structures [List of␣
↪Structure],
                                                        distorsion.get_distortion_
↪indices_angles [Distorsion function],
                                                        ['A', 'B', 'C'],
                                                        show_plots=False)
        return {'average': average [Float],
                'deviation': deviation [Float]}

dist_OP = distorsion.get_distortion_statistic_analysis(structures [List of Structure],
                                                       distorsion.get_distortion_
↪indices_distances [Distorsion function],
                                                       ['A', 'B'],
                                                       show_plots=False)
        return {'average': average [Float],
                'deviation': deviation [Float]}
```

5.2 Example

```
import montemodes.functions.reading as io_monte
molecule1 = io_monte.reading_from_xyz_file('PO4_1.xyz')
molecule2 = io_monte.reading_from_xyz_file('PO4_2.xyz')
molecule3 = io_monte.reading_from_xyz_file('PO4_3.xyz')

import montemodes.analysis.distortion as distortion

di_OPO = distorsion.get_distortion_indices_angles(molecule1, 'P', 'O', 'P')
di_OP = distorsion.get_distortion_indices_distances(molecule2, 'P', 'O')

print 'results: {} {}'.format(di_OPO, di_OP)

####
structures = [molecule1, molecule2, molecule3]
stat_OPO = distorsion.get_distortion_statistic_analysis(structures,
                                                        distorsion.get_distortion_
↪indices_angles,
                                                        ['O', 'P', 'O'],
                                                        show_plots=False)

stat_OP = distorsion.get_distortion_statistic_analysis(structures,
                                                       distorsion.get_distortion_
↪indices_distances,
                                                       ['O', 'P'],
                                                       show_plots=False)

print 'averages: {} {} and deviations: {} {}'.format(stat_OP['average'], stat_OPO[
↪'average'],
                                                     stat_OP['deviation'], stat_OPO[
↪'deviation'])
```

CHAPTER 6

Symmetry classification

Classify the symmetry of a list of structures is symmetry categories defined by the user

```
import montecarlo.analysis.symmetry_analysis
get_symmetry_analysis(structures [List of Structure Objects],
                                  symmetry_to_analyze=None [List of Strings],
                                  shape_to_analyze=1 [Integer],
                                  central_atom=0 [Integer],
                                  symmetry_threshold=0.1 [Float],
                                  cutoff_shape=3.0 [Float],
                                  show_plots=True [Boolean])

    return {symmetry_label : percentage} [Dictionary]
```

- structures: List of Structure type objects to be analyzed
- symmetry_to_analyze : List of symmetry operations to classify the structures into.
- shape_to_analyze: Ideal shape of the structures
- cutoff_shape: Maximum value of shape measurement (defined in shape_to_analyze) to be accepted. Structures with a higher value will be discarded.
- symmetry_threshold: Maximum value of a symmetry measurement of a structure to consider that the structure has the measured symmetry.
- show_plots: If True, graphical data is shown. This includes histrograms showing the distribution of symmetry and shape measurements.

6.1 Exemple

```
import montemodes.functions.reading as io_monte
import montecarlo.analysis.symmetry_analysis

molecule1 = io_monte.reading_from_xyz_file('PO4_1.xyz')
```

```
molecule2 = io_monte.reading_from_xyz_file('PO4_2.xyz')
molecule3 = io_monte.reading_from_xyz_file('PO4_3.xyz')

structures = [molecule1, molecule2, molecule3]

percentage_dict = get_symmetry_analysis(structures [List of Structure Objects],
                                        symmetry_to_analyze=['c 2', 'c 3', 's 4', 'r
↪'],
                                        shape_to_analyze=2,
                                        central_atom=5,
                                        symmetry_threshold=0.15,
                                        cutoff_shape=5.0,
                                        show_plots=False)

for key in percentage_dict:
    print '{} : {} '.format(key, percentage_dict[key])
```