Squid Documentation

Release 0.0.1

Henry Herbol

Oct 26, 2018

CONTENTS

Contents:

CHAPTER

ONE

SQUID

Squid is an open-source molecular simulation codebase developed by the Clancy Lab at Cornell University. The codebase includes simplified Molecular Dynamics (MD) and Density Functional Theory (DFT) simulation submission, as well as other utilities such as file I/O and post-processing.

1.1 Installing

Currently installation involves cloning this repository.

```
[user@local]~% cd ~; git clone https://github.com/ClancyLab/squid.git
```

NOTE! If you are going to also be contributing and you want to have the ssh link instead, first get access by contacting Henry Herbol, and then clone as follows:

```
[user@local]~% cd ~; git clone git@github.com:clancylab/squid.git
```

Aftewards, open up *install.py* and adjust settings accordingly. Then, simply run:

```
[user@local]~% python install.py
```

and you are good to go with using Squid.

1.2 Contributing

If you would like to be a collaborator, first contact Henry Herbol (me) either through github or email and request permissions.

Note, you MUST use a branch for code development and only merge to master when ready for deployment. To make a new branch, use:

```
[user@local]~% git branch <new_branch>
[user@local]~% git checkout <new_branch>
[user@local]~% git push origin <new_branch>
```

To switch between branches, use:

```
[user@local]~% git checkout <new_branch>
```

Once in your new branch, work as you normally would. You can push to your branch whenever you need. When ready to merge, use:

```
[user@local]~% git checkout master
[user@local]~% git pull origin master
[user@local]~% git merge <new_branch>
[user@local]~% git push origin master
```

And finally, when done merging, delete the branch and make a new one:

```
[user@local]~% git checkout master
[user@local]~% git branch -d <branch_name>
[user@local]~% git push origin --delete <branch_name>
[user@local]~% git branch <new_branch>
[user@local]~% git checkout <new_branch>
[user@local]~% git push origin <new_branch>
```

For further information, checkout github's branch tutorial.

1.3 Documentation

Documentation is necessary, and the following steps MUST be followed during contribution of new code:

Setup

1. Download Sphinx. This can be done simply if you have pip installed via *pip install -U Sphinx*
2. Wherever you have *squid* installed, you want another folder called *squid-docs* (NOT as a subfolder of squid).

```
[user@local]~% cd ~; mkdir squid-docs; cd squid-docs; git clone -b gh-pages␣
↪git@github.com:clancylab/squid.git html
```

3. Forever more just ignore that directory (don't delete it though)

Adding Documentation

Documentation is done using ReStructuredText format docstrings, the Sphinx python package, and indices with autodoc extensions. To add more documentation, first add the file to be included in *docs/source/conf.py* under *os.path.abspath('example/dir/to/script.py')*. Secondly, ensure that you have proper docstrings in the python file, and finally run *make full* to re-generate the documentation and commit it to your local branch, as well as the git *gh-pages* branch.

For anymore information on documentation, the tutorial follwed can be found here.

CHAPTER

TWO

CODEBASE

2.1 aneb

The Auto ANEB module simplifies the submission of Auto Nudged Elastic Band simulations.

NOTE! This module is still in a very rough beta. It has been hacked together from the NEB module and is being tested. Do not use this expecting a miracle.

The following code has been tested out to some moderate success for now:

```
new_opt_params = {'step_size': 1.0,
                  'step_size_adjustment': 0.5,
                  'max_step': 0.04,
                  'maxiter': 100,
                  'linesearch': None,
                  'accelerate': False,
                  'N_reset_hess': 10,
                  'max_steps_remembered': 5,
                  'fit_rigid': True,
                  'g_rms': units.convert("eV/Ang", "Ha/Ang", 0.001),
                  'g_max': units.convert("eV/Ang", "Ha/Ang", 0.03)}

new_auto_opt_params = {'step_size': 1.0,
                       'step_size_adjustment': 0.5,
                       'max_step': 0.04,
                       'maxiter': 20,
                       'linesearch': 'backtrack',
                       'accelerate': True,
                       'reset_step_size': 20,
                       'fit_rigid': True,
                       'g_rms': units.convert("eV/Ang", "Ha/Ang", 10.0),
                       'g_max': units.convert("eV/Ang", "Ha/Ang", 0.03)}


nebs = aneb.ANEB("debug_auto", frames, "!HF-3c", fit_rigid=True,
                 opt='LBFGS',
                 new_opt_params=new_opt_params,
                 new_auto_opt_params=new_auto_opt_params,
                 ci_N=3,
                 ANEB_Nsim=5,
                 ANEB_Nmax=15)
```

- *g09_start_job()*
- *g09_results()*

- *orca_start_job()*
- *orca_results()*
- *ANEB*

---

class aneb.**ANEB**(*name*, *states*, *theory*, *extra_section=''*, *initial_guess=None*, *spring_atoms=None*, *procs=1*, *queue=None*, *mem=2000*, *priority=None*, *disp=0*, *k=0.00367453*, *charge=0*, *fit_rigid=True*, *DFT='orca'*, *opt='LBFGS'*, *start_job=None*, *get_results=None*, *new_opt_params={}*, *new_auto_opt_params={}*, *callback=None*, *ci_ANEB=False*, *ci_N=5*, *ANEB_Nsim=5*, *ANEB_Nmax=15*, *add_by_energy=False*)

A method for determining the minimum energy pathway of a reaction using DFT. Note, this method was written for atomic orbital DFT codes; however, is potentially generalizable to other programs.

Parameters

name: ***str*** The name of the ANEB simulation to be run.

states: ***list, list, structures.Atom*** A list of frames, each frame being a list of atom structures. These frames represent your reaction coordinate.

theory: ***str*** The route line for your DFT simulation.

extra_section: ***str, optional*** Additional parameters for your DFT simulation.

initial_guess: ***list, str, optional*** TODO - List of strings specifying a previously run ANEB simulation, allowing restart capabilities.

spring_atoms: ***list, int, optional*** Specify which atoms will be represented by virutal springs in the ANEB calculations. Default includes all.

procs: ***int, optional*** The number of processors for your simulation.

queue: ***str, optional*** Which queue you wish your simulation to run on (queueing system dependent). When None, ANEB is run locally.

mem: ***float, optional*** Specify memory constraints (specific to your X_start_job method).

priority: ***int, optional*** Whether to submit a DFT simulation with some given priority or not.

disp: ***int, optional*** Specify for additional stdout information.

charge: ***int*** Charge of the system.

k: ***float, optional*** The spring constant for your ANEB simulation.

fit_rigid: ***bool, optional*** Whether you want to use procrustes to minimize motion between adjacent frames (thus minimizing error due to excessive virtal spring forces).

DFT: ***str, optional*** Specify if you wish to use the default X_start_job and X_results functions where X is either g09 or orca.

opt: ***str, optional*** Select which optimization method you wish to use from the following: LBFGS.

start_job: ***func, optional*** A function specifying how to submit your ANEB single point calculations. Needed if DFT is neither orca nor g09.

get_results: ***func, optional*** A function specifying how to read your ANEB single point calculations. Needed if DFT is neither orca nor g09.

new_opt_params: ***dict, optional*** Pass any additional parameters to the optimization algorithm. Note, these parameters are for the final calculation after frames have been added in.

new_auto_opt_params: ***dict, optional*** Pass any additional parameters to the optimization algorithm. Note, these parameters are for the iterative calculations, as frames are being added to the band.

callback: ***func, optional*** A function to be run after each each to calculate().

ci_ANEB: ***bool, optional*** Whether to use the climbing image variation of ANEB.

ci_N: ***int, optional*** How many iterations to wait in climbing image ANEB before selecting which image to be used.

ANEB_Nsim: ***int, optional*** The number of frames for an auto ANEB calculation. If an even number is chosen, the expansion happens around floor(ANEB_Nsim/2).

ANEB_Nmax: ***int, optional*** The maximum number of frames to build up to in the auto ANEB.

add_by_energy: ***bool, optional*** If the user wants to add frames by the largest dE instead of dR (motion per frame), then set this flag to True.

Returns

This *ANEB* object.

References

- Henkelman, G.; Jonsson, H. The Journal of Chemical Physics 2000, 113, 9978-9985.
- Jonsson, H.; Mills, G.; Jacobson, K. W. In Classical and Quantum Dynamics in Condensed Phase Simulations;
- Berne, B. J., Ciccotti, G., Coker, D. F., Eds.; World Scientific, 1998; Chapter 16, pp 385-404.
- Armijo, L. Pacific Journal of Mathematics 1966, 16.
- Sheppard, D.; Terrell, R.; Henkelman, G. The Journal of Chemical Physics 2008, 128.
- Henkelman, G.; Uberuaga, B. P.; Jonsson, H. Journal of Chemical Physics 2000, 113.
- Atomic Simulation Environment - https://wiki.fysik.dtu.dk/ase/
- Kolsbjerg, E. L.; Groves, M. N.; Hammer, B. The Journal of Chemical Physics 2016, 145.

align_coordinates(*r*, *B=None*, *H=None*, *return_matrix=False*)

Get a rotation matrix A that will remove rigid rotation from the new coordinates r. Further, if another vector needs rotating by the same matrix A, it should be passed in B and will be rotated. If a matrix also needs rotating, it can be passed as H and also be rotated.

Parameters

r: ***list, float*** 1D array of atomic coordinates to be rotated by procrustes matrix A.

B: ***list, list, float, optional*** A list of vectors that may also be rotated by the same matrix as *r*.

H: ***list, list, float, optional***

A matrix that should also be rotated via: H = R * H * R.T

return_matrix: ***bool, optional*** Whether to also return the rotation matrix used or not.

Returns

rotations: ***dict*** A dictionary holding 'A', the rotation matrix, 'r', the rotated new coordinates, 'B', a list of all other vectors that were rotated, and 'H', a rotated matrix.

aneb.**g09_results**(*ANEB*, *step_to_use*, *i*, *state*)

A method for reading in the output of Gaussian09 single point calculations for ANEB calculations. This will both (a) assign forces to the atoms stored in state and (b) return the energy and atoms.

Parameters

ANEB: ***ANEB*** An ANEB container holding the main ANEB simulation

step_to_use: ***int*** Which iteration in the ANEB sequence the output to be read in is on.

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

Returns

new_energy: ***float*** The energy of the system in Hartree (Ha).

new_atoms: ***list,*** ***structures.Atom*** A list of atoms with the forces attached in units of Hartree per Angstrom (Ha/Ang).

aneb.**g09_start_job**(*ANEB*, *i*, *state*, *charge*, *procs*, *queue*, *initial_guess*, *extra_section*, *mem*, *priority*)
A method for submitting a single point calculation using Gaussian09 for ANEB calculations.

Parameters

ANEB: ***ANEB*** An ANEB container holding the main ANEB simulation

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

charge: ***int*** Charge of the system.

procs: ***int*** The number of processors to use during calculations.

queue: ***str*** Which queue to submit the simulation to (this is queueing system dependent).

initial_guess: ***str*** The name of a previous simulation for which we can read in a hessian.

extra_section: ***str*** Extra settings for this DFT method.

mem: ***int*** How many Mega Words (MW) you wish to have as dynamic memory.

priority: ***int*** Whether to submit the job with a given priority (NBS). Not setup for this function yet.

Returns

g09_job: ***jobs.Job*** A job container holding the g09 simulation.

aneb.**orca_results**(*ANEB*, *step_to_use*, *i*, *state*)
A method for reading in the output of Orca single point calculations for ANEB calculations. This will both (a) assign forces to the atoms stored in state and (b) return the energy and atoms.

Parameters

ANEB: ***ANEB*** An ANEB container holding the main ANEB simulation

step_to_use: ***int*** Which iteration in the ANEB sequence the output to be read in is on.

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

Returns

new_energy: ***float*** The energy of the system in Hartree (Ha).

new_atoms: ***list,*** ***structures.Atom*** A list of atoms with the forces attached in units of Hartree per Angstrom (Ha/Ang).

aneb.**orca_start_job**(*ANEB*, *i*, *state*, *charge*, *procs*, *queue*, *initial_guess*, *extra_section*, *mem*, *priority*)
A method for submitting a single point calculation using Orca for ANEB calculations.

Parameters

ANEB: ***ANEB*** An ANEB container holding the main ANEB simulation

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

charge: ***int*** Charge of the system.

procs: ***int*** The number of processors to use during calculations.

queue: ***str*** Which queue to submit the simulation to (this is queueing system dependent).

initial_guess: ***str*** The name of a previous simulation for which we can read in a hessian.

extra_section: ***str*** Extra settings for this DFT method.

mem: ***int*** How many MegaBytes (MB) of memory you have available per core.

priority: ***int*** Whether to submit to NBS with a given priority

Returns

orca_job: ***jobs.Job*** A job container holding the orca simulation.

2.2 constants

Constants useful for calculations

ENERGY: ***dict*** Various units of energy in terms of Joules. Includes: 'Ha', 'eV', 'J', 'kcal', 'kcal/mol', 'kJ/mol', 'kT_300', 'Ry'

PRESSURE: ***dict*** Various units of pressure in terms of atmospheres. Includes: 'atm', 'bar', 'Pa', 'GPa'

DISTANCE: ***dict*** Various units of idstance in terms of Angstroms. Includes: 'Bohr', 'Ang', 'Angstrom'

K_b: ***float*** Boltzmann's Constant in Joules

h: ***float*** Plank's Constant in J*s

hbar: ***float*** Reduced Plank's Constant in J*s

amu: ***float*** Atomic mass unit in Kg

c: ***float*** Speed of light in m/s

Na: ***float*** Avogadro's Number in mol^-1

pi = PI = Pi: ***float*** The constant pi (3.141592...).

PERIODIC_TABLE: ***list, dict*** A list of dictionaries holding information pertaining to each element. Note, PERIODIC_TABLE[0] is a list of entries for each element's dictionary. This dictionary has not been fully verified, but gives appropriate values for rough calculations as needed (although weight should be correct for all elements). Contains the following:

atomic_num: ***int*** The element's atomic number (equivalent to the index in *PERIODIC_TABLE*).

weight: ***float*** The weight of the element in Atomic Mass Units.

name: ***str*** The name of the element (such as Hydrogen).

sym: ***str*** The symbolic name of the element (such as 'H' for Hydrogen).

mp: ***float*** The melting point of the element in Celsius.

bp: ***float*** The boiling point of the element in Celsius.

density: ***float*** The density of the element in g/cm^3.

group: ***int*** Which column of the periodic table the element resides in.

econfig: ***list, str*** The electronic configuration of the element (such as ['[Ne]', '3s2', '3p5'] for Chlorine).

ionization: ***float*** The ionization energy in eV for the element.

vdw_r: ***float*** The van der waals radius of the element.

COLOUR = COLOR: ***dict*** A list of escape sequences for terminal output colouring on linux. Contains the following:

"BLUE", "GREEN", "YELLOW", "RED", "BOLD", "UNDERLINE", and "ENDC".

2.3 debyer

Python hooks for the debyer code. Link: https://debyer.readthedocs.io/en/latest/

- *get_pdf()*

---

debyer.**get_pdf**(*frames*, *start=0.0*, *stop=5.0*, *step=0.1*, *cutoff=10.0*, *rho=1.0*, *quanta=0.001*, *output=None*, *persist=False*)

Obtain the pair distribution function of a list of atoms using the Debyer code.

Parameters

frames: ***str or list, structures.Atom*** An xyz file name (with or without the .xyz extension) or an input frame to calculate the pdf for.

start: ***float, optional*** The starting radial distance in Angstroms for the calculated pattern.

stop: ***float, optional*** The ending radial distance in Angstroms for the calculated pattern.

step: ***float, optional*** Step in Angstroms for the calculated pattern.

cutoff: ***float, optional*** Cutoff distance in Angstroms for Interatomic Distance (ID) calculations.

rho: ***float, optional*** Numeric density of the system.

quanta: ***float, optional*** Interatomic Distance (ID) discritization quanta.

output: ***str, optional*** Output file name with NO extension given

persist: ***bool, optional*** Whether to persist made .g and .xyz files (True), or remove them (False)

Returns

pdf: ***list, tuple, float*** A list of tuples holding the pdf data (distance in Angstroms and Intensity).

References

- https://debyer.readthedocs.io/en/latest/

2.4 doe_lhs

The doe_lhs module contains code for generating random samples via the latin hypercube sampling method. The below code was developed by scilab and Abraham Lee.

This code was originally published by the following individuals for use with Scilab:

- Copyright (C) 2012 - 2013 - Michael Baudin
- Copyright (C) 2012 - Maria Christopoulou
- Copyright (C) 2010 - 2011 - INRIA - Michael Baudin
- Copyright (C) 2009 - Yann Collette
- Copyright (C) 2009 - CEA - Jean-Marc Martinez

website: forge.scilab.org/index.php/p/scidoe/sourcetree/master/macros

Much thanks goes to these individuals. It has been converted to Python by Abraham Lee.

---

doe_lhs.**lhs**(*n*, *samples=None*, *criterion=None*, *iterations=None*)

Generate a latin-hypercube design

Parameters

n: ***int*** The number of factors to generate samples for

samples: ***int, optional*** The number of samples to generate for each factor (Default: n)

criterion: ***str, optional*** Allowable values are "center" or "c", "maximin" or "m", "centermaximin" or "cm", and "correlation" or "corr". If no value given, the design is simply randomized.

iterations: ***int, optional*** The number of iterations in the maximin and correlations algorithms (Default: 5).

Returns

H: ***2d-array*** An n-by-samples design matrix that has been normalized so factor values are uniformly spaced between zero and one.

Example

A 3-factor design (defaults to 3 samples):

```
>>> lhs(3)
array([[ 0.40069325,  0.08118402,  0.69763298],
       [ 0.19524568,  0.41383587,  0.29947106],
       [ 0.85341601,  0.75460699,  0.360024  ]])
```

A 4-factor design with 6 samples:

```
>>> lhs(4, samples=6)
array([[ 0.27226812,  0.02811327,  0.62792445,  0.91988196],
       [ 0.76945538,  0.43501682,  0.01107457,  0.09583358],
       [ 0.45702981,  0.76073773,  0.90245401,  0.18773015],
       [ 0.99342115,  0.85814198,  0.16996665,  0.65069309],
       [ 0.63092013,  0.22148567,  0.33616859,  0.36332478],
       [ 0.05276917,  0.5819198 ,  0.67194243,  0.78703262]])
```

A 2-factor design with 5 centered samples:

```
>>> lhs(2, samples=5, criterion='center')
array([[ 0.3,  0.5],
       [ 0.7,  0.9],
       [ 0.1,  0.3],
       [ 0.9,  0.1],
       [ 0.5,  0.7]])
```

A 3-factor design with 4 samples where the minimum distance between all samples has been maximized:

```
>>> lhs(3, samples=4, criterion='maximin')
array([[ 0.02642564,  0.55576963,  0.50261649],
       [ 0.51606589,  0.88933259,  0.34040838],
       [ 0.98431735,  0.0380364 ,  0.01621717],
       [ 0.40414671,  0.33339132,  0.84845707]])
```

A 4-factor design with 5 samples where the samples are as uncorrelated as possible (within 10 iterations):

```
>>> lhs(4, samples=5, criterion='correlate', iterations=10)
```

2.5 files

The files module contains various functions aiding in file input and output.

- *read_cml()*
- *write_cml()*
- *read_xyz()*
- *write_xyz()*
- *read_lammpstrj()*
- write_lammpstrj()
- *read_lammps_data()*
- *write_lammps_data()*
- *last_modified()*
- *which()*

---

files.**last_modified**(*name*)

Determine when a file was last modified in seconds.

Parameters

name: ***str*** Name of the file.

Returns

time: ***datetime.datetime*** The last time this file was modified in the standard python datetime format.

files.**read_cml**(*name, new_method=False, extra_opls_parameters={}, parameter_files=[('OPLS', '/fs/europa/g_pc/Forcefields/OPLS/oplsaa.prm')], parameter_file='/fs/europa/g_pc/Forcefields/OPLS/oplsaa.prm', extra_parameters={}, test_charges=False, allow_errors=True, pair_style='lj/cut', default_angles=None, return_molecules=False, test_consistency=False*)

Read in a file written in the Chemical Markup Language (CML) format. It should be mentioned that when reading a file with multiple molecules, if you do not specify return_molecules=True, then everything will be combined into one list.

Note - When using the new_method, the following keywords are ignored: test_charges, allow_errors, pair_style, default_angles, and test_consistency.

Further, a parameter object as well as the molecule objects are returned.

Parameters

name: ***str*** File name.

new_method: ***bool, optional*** A boolean on whether to use the new method of parameter handling, or the old one. Note, the old one will be deprecated and no longer maintained, and over time removed, so we recommend getting used to this method instead.

parameter_files: ***list, tuple, str, str, optional*** A list of tuples, holding two strings: the force field type (either OPLS or SMRFF right now), and the path to the parameter file. If no path is specified, we will try to grab the one assigned in sysconst.

parameter_file: ***str, optional*** Path to the forcefield parameter file.

extra_parameters: ***dict, optional*** Additional OPLS parameters to apply to the forcefield.

test_charges: ***bool, optional*** Bypass inconsistencies in molecular charge (False) or throw errors when inconsistencies exist (True).

allow_errors: ***bool, optional*** Permit constructions of ill-conditioned molecules, such as empty bonds (True), or throw errors (False).

pair_style: ***str, optional*** Pair style to be used in the forcefield.

default_angles: ***dict, optional*** A default forcefield angle type to be set if angle types are set to None.

return_molecules: ***bool, optional*** Whether to have the return be formated as a *structures.Molecule* object or not.

test_consistency: ***bool, optional*** Whether to validate the input cml file against OPLS.

Returns

atoms: ***list,*** ***structures.Atom*** A list of atoms read in from the CML file.

bonds: ***list,*** ***structures.Bond*** A list of bonds read in from the CML file.

angles: ***list,*** ***structures.Angle*** A list of angles read in from the CML file.

dihedrals: ***list,*** ***structures.Dihedral*** A list of dihedrals read in from the CML file.

or

molecules: ***list,*** ***structures.Molecule*** A list of molecules in read in from the CML file.

or

parameters: **ff_params.Parameters** A parameter object holding all relevent data.

molecules: ***list,*** ***structures.Molecule*** A list of molecules in read in from the CML file.

files.**read_lammps_data**(*name*, *read_atoms=True*, *read_bonds=True*, *read_angles=True*, *read_dihedrals=True*)

Helper function for read_lammpstrj to read in larger files.

Parameters

name: ***str*** Name of the .lammpstrj file to be read in. NOTE, this file MUST have the extension .lammpstrj.

read_atoms: ***bool, optional*** Whether to read in the atomic information (True), or not (False).

read_bonds: ***bool, optional*** Whether to read in the bond information (True), or not (False).

read_angles: ***bool, optional*** Whether to read in the angle information (True), or not (False).

read_dihedrals: ***bool, optional*** Whether to read in the dihedral information (True), or not (False).

Returns

atoms: ***list,*** ***structures.Atom*** A list of atoms read in from the data file.

bonds: ***list,*** ***structures.Bond*** A list of bonds read in from the data file.

angles: ***list,*** ***structures.Angle*** A list of angles read in from the data file.

dihedrals: ***list,*** ***structures.Dihedral*** A list of dihedrals read in from the data file.

files.**read_lammpstrj**(*name*, *read_atoms=True*, *read_timesteps=True*, *read_num_atoms=True*, *read_box_bounds=True*, *verbose=True*, *last_frame=False*, *big_file=True*)

Imports the atom style dump file from lammps.

Parameters

name: ***str*** Name of the .lammpstrj file to be read in. NOTE, this file MUST have the extension .lammpstrj.

read_atoms: ***bool, optional*** Whether to read in the atomic information (True), or not (False).

read_timesteps: ***bool, optional*** Whether to read in the timesteps (True), or not (False).

read_num_atoms: ***bool, optional*** Whether to read in the number of atoms (True), or not (False).

read_box_bounds: ***bool, optional*** Whether to read in the system box boundaries (True), or not (False).

verbose: ***bool, optional*** Whether to output more to stdout (True), or not (False).

last_frame: ***bool, optional*** Whether to output only the last iteration of the simulation (True), or all of it (False).

big_file: ***bool, optional*** Whether to read through the file line-by-line to allow for reading of large files (True), or not (False). If True, this read operation will be slower.

Returns

data: ***results.sim_out*** Return a sim_out object containing simulation output information.

files.**read_mdl**(*name*)

Read in a file written in the molden file format.

Parameters

name: ***str*** File name with or without .mdl file extension.

Returns

frames: ***list, list,*** ***structures.Atom*** A list of atoms read in from the xyz file. If there is only one frame, then only a *list* of *structures.Atom* is returned.

files.**read_xyz**(*name, cols=['element', 'x', 'y', 'z'], cast_elem_to_sym=True, fast=True*)
Read in a file written in the XYZ file format. This is an improved version, accounting for xyz files of varying atom numbers.

Parameters

name: ***str*** File name with or without .xyz file extension.

cols: ***list, str, optional*** The specific columns in this xyz file. Note - we may not support all possibilities, and order matters!

cast_elem_to_sym: ***bool, optional*** Whether to cast the element into the symbol (ex. 2 becomes He).

fast: ***bool, optional***

If specified, you are promising that this xyz file has the columns element x y z

Further, if speed truly matters and you do not want to foce cast element into symbols, we recommend setting cast_elem_to_sym=False.

Returns

frames: ***list, list,*** ***structures.Atom*** A list of atoms read in from the xyz file. If there is only one frame, then only a *list* of *structures.Atom* is returned.

files.**read_xyz_gen**(*name, cols=['element', 'x', 'y', 'z'], cast_elem_to_sym=True, fast=False*)
This will yield a frame from an xyz file.

Parameters

name: ***str*** File name with or without .xyz file extension.

cols: ***list, str, optional*** The specific columns in this xyz file. Note - we may not support all possibilities, and order matters!

cast_elem_to_sym: ***bool, optional*** Whether to cast the element into the symbol (ex. 2 becomes He).

fast: ***bool, optional***

If specified, you are promising that this xyz file has the columns element x y z

Further, if speed truly matters and you do not want to foce cast element into symbols, we recommend setting cast_elem_to_sym=False.

Returns

yield: ***list,*** ***structures.Atom*** A frame from an xyz file.

files.**which**(*program*)
A function to return the full path of a system executable.

Parameters

program: ***str*** The name of the system executable to find.

Returns

path: ***str or None*** The path to the system executable. If none exists, then None.

References

- http://stackoverflow.com/a/377028

files.**write_cml**(*atoms_or_molecule_or_system*, *bonds=[]*, *name=None*)
Write data in (list, *structures.Atom*), or *structures.Molecule*, or *structures.System* to a file written in the Chemical Markup Language (CML) format. If a list of *structures.Molecule* is passed,

then a CML file is written in which each molecule is its own section. Note, this cannot be read into Avogadro, and it is recommended that if you plan to use Avogadro to combine these into one `structures.System`.

Parameters

atoms_or_molecule_or_system: ***structures.Atom* or *structures.Molecule* or *structures.System***
Atomic data to be written to a CML file.

bonds: ***list, structures.Bond, optional*** A list of bonds within the system. This is useful when the input is a list of `structures.Atom`.

name: ***str, optional*** The name of the output file (either ending or not in .cml).

Returns

None

`files.`**`write_lammps_data`**(*system*, *name=None*, *params=None*, *pair_coeffs_included=False*, *hybrid_angle=False*, *hybrid_pair=False*)

Writes a lammps data file from the given system.

Set pair_coeffs_included to True to write pair_coeffs in data file. Set hybrid_angle to True to detect different treatment of angles among different atom types. Set hybrid_pair to True to detect different treatment of pairing interactions among different atom types.

Parameters

system: ***structures.System*** Atomic system to be written to a lammps data file.

name: ***str, optional*** What to reassign the system name to.

new_method: ***bool, optional*** A boolean on whether to use the new method of parameter handling, or the old one. Note, the old one will be deprecated and no longer maintained, and over time removed, so we recommend getting used to this method instead.

pair_coeffs_included: ***bool, optional*** Whether to write pair coefficients into the data file (True), or not (False).

hybrid_angle: ***bool, optional*** Whether to detect different treatments of angles amongst different atom types (True), or not (False).

hybrid_pair: ***bool, optional*** Whether to detect different treatments of pairing interactions amongst different atom types(True), or not (False).

Returns

None

`files.`**`write_mdl`**(*frames*, *name*)

`files.`**`write_xyz`**(*frames_or_system*, *name_or_file=None*, *ID='Atoms'*)

Write frames of atomic conformations to a file written in the XYZ file format.

Parameters

frames_or_system: ***list, structures.Atom or structures.System*** Atoms to be written to an xyz file.

name_or_file: ***str or fptr, optional*** Either a filename (with or without the .xyz extension) or an open file buffer to write the xyz file.

ID: ***str, optional*** What is to be written on the xyz comment line.

Returns

None

2.6 frc_opls

The OPLS Forcefield module contains functionality for parsing the OPLS forcefield and typing appropriately. Note, you must first import files before ever importing frc_opls.

- *read_opls_parameters()*
- *set_forcefield_parameters()*
- *check_net_charge()*
- *check_consistency()*

---

frc_opls.**check_consistency**(*atoms*, *bonds*, *angles*, *dihedrals*, *name=''*, *allow_errors=False*)

Check to see if all possible force field parameters have been assigned. Raises exception if missing an bond or angle. Missing dihedrals allowed by default. Can turn off raising exceptions.

Parameters

atoms: *list,* ***structures.Atom*** List of atom objects to check if parameters were set accordingly.

bonds: *list,* ***structures.Bond****, optional* List of bond objects to check if parameters were set accordingly.

angles: *list,* ***structures.Angle****, optional* List of angle objects to check if parameters were set accordingly.

dihedrals: *list,* ***structures.Dihedral****, optional* List of dihedral objects to check if parameters were set accordingly.

name: *str, optional* Name of the molecule/system.

allow_errors: *bool, optional* Whether to allow incomplete parameterizations, or to throw errors.

Returns

None

frc_opls.**check_net_charge**(*atoms*, *name=''*, *q_tol=0.01*)

Check what the net charge is of the given list of atoms. An error is raised if the net_charge is greater than *q_tol*.

Parameters

atoms: *list, structures.Atom*

name: *str, optional*

Returns

None

frc_opls.**read_opls_parameters**(*parameter_file='/fs/europa/g_pc/Forcefields/OPLS/oplsaa.prm'*, *pair_style='lj/cut'*)

Reads an opls parameter file written in the Tinker file format.

Parameters

parameter_file: *str, optional* Relative or absolute path to an opls parameter file, written in the Tinker file format.

pair_style: *str, optional* The pair style to be assigned.

Returns

atom_types: *list,* ***structures.Struct*** A list of the forcefield types for atoms, stored as *structures.Struct*.

bond_types: *list,* ***structures.Struct*** A list of the forcefield types for bonds, stored as *structures.Struct*.

angle_types: *list,* ***structures.Struct*** A list of the forcefield types for angles, stored as *structures.Struct*.

dihedral_types: *list,* ***structures.Struct*** A list of the forcefield types for dihedrals, stored as *structures.Struct*.

frc_opls.**set_forcefield_parameters**(*atoms, bonds=[], angles=[], dihedrals=[], parameter_file=[('OPLS', '/fs/europa/g_pc/Forcefields/OPLS/oplsaa.prm')], name='unnamed', extra_parameters={}, test_consistency=True, test_charges=True, allow_errors=False, pair_style='lj/cut', allow_no_ffp=False*)

Reads an opls parameter file written in the Tinker file format.

Parameters

atoms: *list,* ***structures.Atom*** List of atom objects to be parameterized.

bonds: *list,* ***structures.Bond****, optional* List of bond objects to be parameterized.

angles: *list,* ***structures.Angle****, optional* List of angle objects to be parameterized.

dihedrals: *list,* ***structures.Dihedral****, optional* List of dihedral objects to be parameterized.

parameter_file: *list, tuple, str, optional* The name and path of the force field to be used. Note, we currently only accept "OPLS".

name: *str, optional* Name of the molecule/system you are parameterizing.

extra_parameters: *dict, optional* Additional parameters not found in the forcefield.

test_consistency: *bool, optional* Whether to verify all parameters have been set.

test_charges: *bool, optional* Whether to verify the system is at a neutral state.

allow_errors: *bool, optional* Whether to allow incomplete parameterizations, or to throw errors.

pair_style: *str, optional* The pair style to be used.

allow_no_ffp: *bool, optional* Whether to allow for situations in which some atoms are not included in this force field.

Returns

atoms: *list,* ***structures.Atom*** A list of all atoms with set parameters.

bonds: *list,* ***structures.Bond*** A list of all bonds with set parameters.

angles: *list,* ***structures.Angle*** A list of all angles with set parameters.

dihedrals: *list,* ***structures.Dihedral*** A list of all dihedrals with set parameters.

2.7 g09

The g09 module contains python functions for interfacing with the Gaussian09 DFT software package. NOTE! Due to implementation restrictions this code will only work on the ICSE cluster. Primarily the g09 job submission command

is specific to the ICSE system.

- *read()*
- *job()*
- *cubegen_analysis()*

---

g09.**cubegen_analysis**(*old_job*, *orbital=None*, *path='gaussian/'*, *chk_conv=True*, *skip_potential=False*)

Post process a g09 job using cubegen and vmd to display molecular orbitals and the potential surface.

Parameters

old_job: ***str*** Gaussian file name. Only use the name, such as 'water' instead of 'water.chk'. Note, do not pass a path as it is assumed you are in the parent directory of the job to analyze. If not, use the path variable.

orbital: ***str, optional*** The orbital to analyze (0, 1, 2, 3, . . .). By default HOMO and LUMO will be analyzed, thus this only is useful if you wish to see other orbitals.

path: ***str, optional*** Path to where the gaussian job was run. By default, this is a gaussian subfolder.

chk_conv: ***bool, optional*** Check if the simulation converged before proceeding. If you only have a .chk file and you are certain it converged, this can be set to False.

skip_potential: ***bool, optional*** If you only care about MO's, skip the expensive potential surface calculation.

Returns

None

g09.**job**(*run_name, route, atoms=[], extra_section='', queue='short', procs=1, verbosity='N', charge_and_multiplicity='0,1', title='run by gaussian.py', blurb=None, eRec=True, force=False, previous=None, neb=[False, None, None, None], err=False, mem=25*)

Wrapper to submitting an Gaussian09 simulation.

Parameters

run_name: ***str*** Name of the simulation to be run.

route: ***route*** The DFT route line, containing the function, basis set, etc.

atoms: ***list,*** **structures.Atom** ***,optional*** A list of atoms for the simulation.

extra_section: ***str, optional*** Additional DFT simulation parameters.

queue: ***str, optional*** What queue to run the simulation on (queueing system dependent).

procs: ***int, optional*** How many processors to run the simulation on.

verbosity: ***str, optional*** Verbosity flag for Gaussian09 output.

charge_and_multiplicity: ***str, optional*** Charge and multiplicity of the system.

title: ***str, optional*** Comment line for Gaussian09 input file.

blurb: ***str, deprecated*** Do not use

eRec: ***bool, deprecated*** Do not use

force: ***bool, optional*** Whether to overwrite a simulation with the same name.

previous: ***str, optional*** Name of a previous simulation for which to try reading in information using the MORead method.

neb: *list, bool, deprecated* Do not use

err: *bool, deprecated* Do not use

mem: *float, optional* Amount of memory per processor that is available (in MB).

Returns

job: *subprocess.Popen or* `jobs.Job` If running locally, return the process handle, else return the job container.

g09.**parse_route**(*route*)

Function that parses the route into the following situations: func/basis key(a,b,c,...) key(a) key=a key = a key=(a,b,c,...)

Parameters

route: *str* The route of a g09 simulation

Returns

parsed_list: *list, str* Split list of g09 route line.

extra: *str* What was not parsed is returned.

g09.**read**(*input_file*)

General read in of all possible data from an Gaussian09 output file (.log).

Parameters

input_file: *str* Gaussian .log file to be parsed.

Returns

data: `results.DFT_out` Generic DFT output object containing all parsed results.

2.8 geometry

The geometry module contains various functions aiding in euclidian manipulation of atomic coordinates.

- *align_centroid()*
- *align_frames()*
- *angle_size()*
- *array_to_atom_list()*
- *atom_list_to_array()*
- *center_frames()*
- *dihedral_angle()*
- *dist()*
- *dist_squared()*
- *get_bonds()*
- *get_angles_and_dihedrals()*
- *interpolate()*
- *motion_per_frame()*

---

geometry.**align_centroid**(*atoms*, *recenter=True*, *skip_H=True*)

Generate a Minimum Volume Enclosing Ellipsoid (MVEE) around atomic species to align the atoms along the x-axis.

Parameters

atoms: ***list,*** ***structures.Atom*** A list of Atom objects.

recenter: ***bool, optional*** Whether to recenter the new coordinates around the origin or not. Note, this is done via the center of geometry, NOT the center of mass.

skip_H: ***bool, optional*** Whether to skip hydrogen during recentering (that is, do not take them into accound when calculating the center of geometry).

Returns

molec.atoms: ***list,*** ***structures.Atom*** Rotated atomic coordinates.

A: ***list, list, float*** Rotated positive definite symmetric matrix of the ellipsoid's center form. This contains the ellipsoid's orientation and eccentricity.

geometry.**align_frames**(*prev_frames*)

Given a set of frames depicting some pathway, this function attempts to order atomic coordinates similarly throughout each frame.

NOTE! THIS IS A VERY SIMPLE METHOD BASED ON INTERATOMIC DISTANCES! A better procedure would be that of geometry.reorder_atoms_in_frames()

Parameters

prev_frames: ***list, list,*** ***structures.Atom*** List of lists of atoms.

Returns

frames: ***list, list,*** ***structures.Atom*** List of lists of atoms.

geometry.**angle_size**(*a*, *center*, *b*)

Determine the angle between three atoms. In this case, determin the angle a-center-b.

Parameters

a: ***structures.Atom*** First atom in the angle.

center: ***structures.Atom*** Center atom of the angle.

b: ***structures.Atom*** Last atom in the angle.

Returns

theta: ***float*** Return the angle in degrees.

geometry.**array_to_atom_list**(*A*, *elems*)

Given a list of atomic coordinates as lists of floats, and a list of elements, generate a list of atom objects.

Parameters

A: ***list, list, float*** A list of the atomic coordinates

elems: ***list, str*** A list of the elements associated with each index.

Returns

frame: ***list,*** ***structures.Atom*** A list of atom objects.

geometry.**atom_list_to_array**(*A*)

Given a list of atoms, return a list of coordinates.

Parameters

A: ***list,*** ***structures.Atom*** A list of atom objects.

Returns

coords: ***list, list, float*** A list of the atomic coordinates

elems: ***list, str*** A list of the elements associated with each index.

geometry.**center_frames**(*frames, ids, X_TOL=0.1, XY_TOL=0.1, Z_TOL=0.1, THETA_STEP=0.005, TRANSLATE=[0, 0, 0]*)

LEGACY CODE: Quickly and poorly implemented code. Only use if geometry.procrustes/geometry.orthogonal_procrustes is unable to accomplish what you need.

Recenter a list of lists of atomic coordinates to overlay based on input criteria. This is a simpler method than procrustes, but will rarely minimize the frobenius norm.

Parameters

frames: ***list, list,*** ***structures.Atom*** List of lists of atoms.

ids: ***list, int***

A list of indices for the following:

ids[0] - This is an atom that will be positioned at the origin after translating the frame

ids[1] - This is an atom that will lie on the positive x-axis after two rotations of the frame

ids[2] - This is an atom that will lie on the xy plane in the positive y direction after rotation of the frame

X_TOL: ***float, optional*** Tolerance for alignment of ids[1] along the x-axis.

XY_TOL: ***float, optional*** Tolerance for alignment of ids[2] along the positive y-axis.

Z_TOL: ***float, optional*** Tolerance for alignment of ids[2] along the xy plane.

THETA_STEP: ***float, optional*** Steps at which to adjust rotation when finding optimal rotations. Smaller implies better fit to centering criteria, but slower calculations.

TRANSLATE: ***list, float*** The desired translation from the origin.

Returns

None

See also

For more information, see *orthogonal_procrustes()* and *procrustes()*.

geometry.**check_all_bonds**(*frames*, *system*, *bond_tolerance=5.0*)

Add molecule index to the atom if it has not already been assigned. Then recursively pass bonded atoms to the function

Parameters

atom_list: ***list*, *structures.Molecule*** A list of atoms

i_list_index: ***int*** The index of the atom currently being assigned. Refers to atom_list index.

Returns

None

geometry.**dihedral_angle**(*a*, *b*, *c*, *d*)

Use the Praxeolitic formula to determine the dihedral angle between 4 atoms.

Parameters

a: ***structures.Atom*** First atom in the dihedral.

b: ***structures.Atom*** Second atom in the dihedral.

c: ***structures.Atom*** Third atom in the dihedral.

d: ***structures.Atom*** Fourth atom in the dihedral.

Returns

theta: ***float*** Return the dihedral angle in radians.

References

- http://stackoverflow.com/a/34245697

geometry.**dist**(*a*, *b*, *system=None*)

Get the distance between two atomic species.

Parameters

atom1: ***:class:'structures.Atom'*** One of the two atoms to find the distance between.

atom2: ***:class:'structures.Atom'*** Second of the two atoms to find the distance between.

system: ***:class:'structures.System', optional*** The system that the point is contained. Used for periodic distance calculations

Returns

d: ***float*** Distance between the two atoms.

geometry.**dist_squared**(*atom1*, *atom2*, *system=None*)

Get the squared distance between two atomic species. Slightly faster than geometry.dist(a, b) as we do not take the square root.

Parameters

atom1: *:class:'structures.Atom'* One of the two atoms to find the distance between.

atom2: *:class:'structures.Atom'* Second of the two atoms to find the distance between.

system: *:class:'structures.System', optional* The system that the point is contained. Used for periodic distance calculations

Returns

sqr_dist: *float* Squared distance between the two atoms.

geometry.**get_angles_and_dihedrals**(*atoms*)
Given a list of atom structures with bonded information, calculate angles and dihedrals.

Parameters

atoms: *list,* ***structures.Atom*** List of atoms for which angles and dihedrals are to be calculated.

Returns

angles: *list,* ***structures.Angle*** Calculated angles.

dihedrals: *list,* ***structures.Dihedral*** Calculated dihedrals.

geometry.**get_bonds**(*atoms*)
Given a list of atomic positions, determine all bonds based on proximity.

Parameters

atoms: *list,* ***structures.Atom*** List of atoms for which bonds are to be calculated.

Returns

bonds: *list,* ***structures.Bond*** Return the calculated bonds.

geometry.**interpolate**(*frame_1*, *frame_2*, *N*)
Linearly interpolate N frames between two given frames.

Parameters

frame_1: *list,* ***structures.Atom*** List of atoms.

frame_2: *list,* ***structures.Atom*** List of atoms.

N: *int* Number of new frames you want to generate during interpolation.

Returns

frames: *list, list, float* List of interpolated frames (non-inclusive of frame_1 nor frame_2).

geometry.**motion_per_frame**(*frames*)
Determine the root mean squared difference between atomic positions of adjacent frames.

Parameters

frames: *list, list,* ***structures.Atom*** List of lists of atoms.

Returns

motion: *list, float* List of motion between consecutive frames (frame_i vs frame_(i - 1)). As len(motion) = len(frames), this means that motion[0] = 0.

geometry.**mvee**(*points*, *tol=0.001*)
Generate a Minimum Volume Enclosing Ellipsoid (MVEE) around atomic species. The ellipsoid is calculated for the "center form": (x-c).T * A * (x-c) = 1

For useful values, you can get the radii as follows:

```
U, Q, V = np.linalg.svd(A)
r_i = 1/sqrt(Q[i])
vol = (4/3.) * pi * sqrt(1 / np.product(Q))
```

Further, note that V is the rotation matrix giving the orientation of the ellipsoid.

NOTE! You must have a minimum of 4 atoms for this to work.

Parameters

points: ***list,*** ***structures.Atom*** A list of Atom objects.

tol: ***float, optional*** Tolerance for ellipsoid generation.

Returns

A: ***list, list, float*** Positive definite symmetric matrix of the ellipsoid's center form. This contains the ellipsoid's orientation and eccentricity.

c: ***list, float*** Center of the ellipsoid.

References

- https://www.mathworks.com/matlabcentral/fileexchange/9542-minimum-volume-enclosing-ellipsoid?requestedDomain=www.mathworks.com
- http://stackoverflow.com/questions/14016898/port-matlab-bounding-ellipsoid-code-to-python/14025140#14025140

`geometry.`**`orthogonal_procrustes`**(*A*, *ref_matrix*, *reflection=False*)

Using the orthogonal procrustes method, we find the unitary matrix R with det(R) > 0 such that ||A*R - ref_matrix||^2 is minimized. This varies from that within scipy by the addition of the reflection term, allowing and disallowing inversion. NOTE - This means that the rotation matrix is used for right side multiplication!

Parameters

A: ***list,*** ***structures.Atom*** A list of atoms for which R will minimize the frobenius norm ||A*R - ref_matrix||^2.

ref_matrix: ***list,*** ***structures.Atom*** A list of atoms for which *A* is being rotated towards.

reflection: ***bool, optional*** Whether inversion is allowed (True) or not (False).

Returns

R: ***list, list, float*** Right multiplication rotation matrix to best overlay A onto the reference matrix.

scale: ***float*** Scalar between the matrices.

Derivation

Goal: minimize ||A*R - ref||^2, switch to trace

trace((A*R-ref).T*(A*R-ref)), now we distribute

trace(R'*A'*A*R) + trace(ref.T*ref) - trace((A*R).T*ref) - trace(ref.T*(A*R)), trace doesn't care about order, so re-order

trace(R*R.T*A.T*A) + trace(ref.T*ref) - trace(R.T*A.T*ref) - trace(ref.T*A*R), simplify

trace(A.T*A) + trace(ref.T*ref) - 2*trace(ref.T*A*R)

Thus, to minimize we want to maximize trace(ref.T * A * R)

u*w*v.T = (ref.T*A).T

ref.T * A = w * u.T * v

trace(ref.T * A * R) = trace (w * u.T * v * R)

differences minimized when trace(ref.T * A * R) is maximized, thus when trace(u.T * v * R) is maximized

This occurs when u.T * v * R = I (as u, v and R are all unitary matrices so max is 1)

R is a rotation matrix so R.T = R^-1

u.T * v * I = R^-1 = R.T

R = u * v.T

Thus, R = u.dot(vt)

References

- https://github.com/scipy/scipy/blob/v0.16.0/scipy/linalg/ _procrustes.py#L14
- http://compgroups.net/comp.soft-sys.matlab/procrustes-analysis -without-reflection/896635

`geometry.`**`procrustes`**(*frames*, *count_atoms=None*, *append_in_loop=True*, *reflection=False*)

Propogate rotation along a list of lists of atoms to smooth out transitions between consecutive frames. This is done by rigid rotation and translation (no scaling and no inversions). Rotation starts at frames[0].

Parameters

frames: ***list, list,*** ***`structures.Atom`*** List of lists of atoms.

count_atoms: ***list, int, optional*** A list of indices for which translation and rotations will be calculated from.

append_in_loop: ***bool, optional*** If rotation matrices for every atom (True) is desired vs rotation matrices for every frame (False). Every rotation matrix for atoms within the same frame is the same. Thus, when this is True, multiplicates will appear.

reflection: ***bool, optional*** Whether inversion is allowed (True) or not (False).

Returns

full_rotation: ***list, list, float*** List of every rotation matrix applied. NOTE - These matrices are applied via right side multiplication.

See also

For more information, see *`orthogonal_procrustes()`*.

`geometry.`**`rand_rotation`**(*limit_angle=None*, *lower_bound=0.1*, *MAXITER=1000000*)

Generate a random rotation matrix.

Parameters

limit_angle: ***float, optional*** Whether to confine your random rotation (in radians).

lower_bound: ***float, optional*** A lower bound for limit_angle, at which the identity is simply returned. This is necessary as the procedure to generate the limit_angle method is incredibly slow at small angles.

MAXITER: ***int, optional*** A maximum iteration for when we try to calculate a rotation matrix with some limit_angle specified.

Returns

frames: ***list, list, float*** A random rotation matrix.

References

- http://tog.acm.org/resources/GraphicsGems/, Ed III

geometry.**reduce_list**(*givenList*, *idfun=None*)
Remove duplicates of a list, whilst maintaining order.

Parameters

givenList: ***list*** List of anything for which __eq__ has been defined.

Returns

cleaned_list: ***list*** List with duplicates removed.

References

- https://www.peterbe.com/plog/uniqifiers-benchmark

geometry.**reorder_atoms_in_frames**(*frames*)
A function to ensure that consecutive frames of an xyz file are in the same order. This is done by minimizing the frobenious norm between consecutive frames with the application of a perterbation matrix P.

minimize ||PR_{i+1} - R_{i}||^2.

This problem boils down to maximizing Tr[P R_{i + 1} R_{i}^T]. We solve this with Munkres algorithm using the scipy.optimize.linear_sum_assignment function.

NOTE! This only works when we have the problem in which atom order only is mixed up. If we also have rotations, then the problem actually becomes:

minimize ||PAR_{i+1} - R_{i}||^2

Which is, unfortunately, harder as we now have two unknown matrices (P and A)!

Requires scipy 0.17.0 or above.

Parameters

frames: ***list,*** ***structures.Atom*** Input frames to be sorted.

Returns

Rframes: ***list,*** ***structures.Atom*** A reordered list.

geometry.**rms**(*x*)
Return the Root-Mean-Squared value of an array.

Parameters

array: ***list, float*** An array of floats to find the RMS of.

Returns

rms: ***float*** The Root-Mean-Squared value.

geometry.**rotate_frames**(*frame, theta_0=0, theta_n=360, dt=1, axis=[0, 0, 1], cog=True, origin=(0, 0, 0), last=False*)
Given a list of atoms, generate a sequential list of rotated atomic instances.

Parameters

frame: ***list,*** ***structures.Atom*** A list of Atoms.

theta_0: ***float, optional*** Starting rotation.

theta_n: ***float, optional*** Ending rotation.

dt: ***float, optional*** Change in rotation.

axis: ***list, float, optional*** Which axis to rotate around.

cog: ***bool, optional*** Whether to rotate around the center of geometry (True) or not (False).

origin: ***tuple, float*** The origin for which we will rotate around.

last: ***bool, optional*** Whether to only return the final rotation (True) or not (False).

Returns

frames: ***list, list, structures.Atom or list, structures.Atom*** Returned rotations of everything (if last is True), or just the final rotation (if last is False).

geometry.**rotate_xyz**(*alpha*, *beta*, *gamma*, *units='deg'*)

POTENTIALLY DEPRECATED CODE! WILL FAIL ON USE!

Construct general rotation matrix using yaw, pitch, and roll (alpha, beta, gamma). Performs extrinsic rotation whose Euler angles are alpha, beta, and gamma about axes z, y, and x.

Parameters

alpha: ***float*** The 'yaw' angle.

beta: ***float*** The 'pitch' angle.

gamma: ***float*** The 'roll' angle.

units: ***str, optional*** The units of the given angles.

Returns

rotatation_matrix: ***list, list, float*** The rotation matrix.

geometry.**rotation_matrix**(*axis*, *theta*, *units='deg'*)

Obtain a left multiplication rotation matrix, given the axis and angle you wish to rotate by. By default it assumes units of degrees. If theta is in radians, set units to rad.

Parameters

axis: ***list, float*** The axis in which to rotate around.

theta: ***float*** The angle of rotation.

units: ***str, optional*** The units of theta (deg or rad).

Returns

rotatation_matrix: ***list, list, float*** The left multiplication rotation matrix.

References

- http://stackoverflow.com/questions/6802577/python-rotation-of-3d-vector/25709323#25709323

geometry.**smooth_xyz**(*name*, *R_MAX=0.5*, *F_MAX=25*, *N_FRAMES=None*, *PROCRUSTES=True*, *outName=None*, *verbose=False*)

Smooth out an xyz file by linearly interpolating frames to minimize the maximum motion between adjacent frames. Further, this can use procrustes to best overlap adjacent frames.

Parameters

name: ***list, list, structures.Atom*** A list of lists of atoms.

R_MAX: ***float, optional*** The maximum motion allowed between consecutive frames.

F_MAX: ***int, optional*** The maximum number of frames allowed before failing the smooth function.

N_FRAMES: ***int, optional*** If this is specified, forgo the R_MAX and F_MAX and just interpolate out into N_FRAMES. Note, if more than N_FRAMES exists, this also cuts back into exactly N_FRAMES.

PROCRUSTES: ***bool, optional*** Whether procrustes is to be used during smoothing (True), or not (False).

outName: ***str, optional*** An output file name for the smoothed frames (without the .xyz extension).

verbose: ***bool, optional*** Whether additional stdout is desired (True), or not (False).

Returns

frames: ***list, list,*** ***structures.Atom*** Returns a list of smoothed frames

geometry.**translate_vector_1A**(*pos*, *multiple*, *system*)
Translate x,y,z coordinates across boundary vector 1/A, as many times as 'multiple'

Parameters

pos: ***list, float*** XYZ point.

multiple: ***float*** Scalar value to translate by. multiple = 1 means translate one full vector.

system: ***structures.System*** The system that the point is contained. The 1/A vector is pulled from the system.

Returns

[x_a, y_a, z_a]: ***list, float*** New XYZ point.

geometry.**translate_vector_2B**(*pos*, *multiple*, *system*)
Translate x,y,z coordinates across boundary vector 2/B, as many times as 'multiple'

Parameters

pos: ***list, float*** XYZ point.

multiple: ***float*** Scalar value to translate by. multiple = 1 means translate one full vector.

system: ***structures.System*** The system that the point is contained. The 1/A vector is pulled from the system.

Returns

[x_b, y_b, z_b]: ***list, float*** New XYZ point.

geometry.**translate_vector_3C**(*pos*, *multiple*, *system*)
Translate x,y,z coordinates across boundary vector 3/C, as many times as 'multiple'

Parameters

pos: ***list, float*** XYZ point.

multiple: ***float*** Scalar value to translate by. multiple = 1 means translate one full vector.

system: ***structures.System*** The system that the point is contained. The 1/A vector is pulled from the system.

Returns

[x_c, y_c, z_c]: ***list, float*** New XYZ point.

geometry.**unwrap_molecules**(*frames*, *system*)
Unwraps the atoms in a periodic system so that no bonds are across a periodic box. Requires either (1) the atoms in the frames to have bond information, or (2) the atoms in system to have bond information. In case (2), the atoms in system serve as a template for every frame and must contain every atom.

Parameters

frames: ***list, list, :class:'structures.Atom'*** List of lists of atoms.

system: ***:class:'structures.System'*** The system that the point is contained. Used for periodic distance calculations

Returns

frames: ***list, list, :class:'structures.Atom'*** Updated list of lists of atoms.

`geometry.`**`unwrap_xyz`**(*frames*, *system*, *motion_tolerance=3.0*)

Unwraps the atoms in a periodic system so that atoms are never reflected across periodic boundary conditions. Does this by using the previous time step as the reference and undoing any periodic reflections in the lammpstrj

Parameters

frames: ***list, list, :class:'structures.Atom'*** List of lists of atoms.

system: ***:class:'structures.System'*** The system that the point is contained. Used for periodic distance calculations

Returns

frames: ***list, list, :class:'structures.Atom'*** Updated list of lists of atoms.

2.9 jdftx

The JDFTx module. This works as a python wrapper of the JDFTx plane-wave DFT code.

- *job()*
- *read()*

`jdftx.`**`job`**(*run_name*, *atoms*, *ecut*, *ecutrho=None*, *atom_units='Ang'*, *route=None*, *pseudopotentials=None*, *periodic_distance=15*, *dumps='dump End Ecomponents ElecDensity'*, *queue=None*, *walltime='00:30:00'*, *procs=1*, *threads=None*, *redundancy=False*, *previous=None*, *mem=2000*, *priority=None*, *xhost=None*)

Wrapper to submitting a JDFTx simulation.

Parameters

run_name: ***str*** Name of the simulation to be run.

atoms: ***list, `structures.Atom`, or str*** A list of atoms for the simulation. If a string is passed, it is assumed to be an xyz file (relative or full path). If None is passed, then it is assumed that previous was specified.

ecut: ***float*** The planewave cutoff energy in Hartree.

ecutrho: ***float, optional*** The charge density cutoff in Hartree. By default this is 4 * ecut.

atom_units: ***str, optional*** What units your atoms are in. JDFTx expects bohr; however, typically most work in Angstroms. Whatever units are converted to bohr here.

route: ***str, optional*** Any additional script to add to the JDFTx simulation.

pseudopotentials: ***list, str, optional*** The pseudopotentials to use in this simulation. If nothing is passed, a default set of ultra-soft pseudo potentials will be chosen.

periodic_distance: ***float, optional*** The periodic box distance in Bohr.

dumps: ***str, optional*** The outputs for this simulation.

queue: ***str, optional*** What queue to run the simulation on (queueing system dependent).

procs: ***int, optional*** How many processors to run the simulation on.

threads: ***int, optional*** How many threads to run the simulation on. By default this is procs.

redundancy: ***bool, optional*** With redundancy on, if the job is submitted and unique_name is on, then if another job of the same name is running, a pointer to that job will instead be returned.

previous: ***str, optional*** Name of a previous simulation for which to try reading in information using the MORead method.

mem: ***float, optional*** Amount of memory per processor that is available (in MB).

priority: ***int, optional*** Priority of the simulation (queueing system dependent). Priority ranges (in NBS) from a low of 1 (start running whenever) to a high of 255 (start running ASAP).

xhost: ***list, str or str, optional*** Which processor to run the simulation on(queueing system dependent).

Returns

job: ***jobs.Job*** Teturn the job container.

jdftx.**read**(*input_file*, *atom_units='Ang'*)

General read in of all possible data from a JDFTx output file.

Parameters

input_file: ***str*** JDFTx output file to be parsed.

atom_units: ***str, optional*** What units you want coordinates to be converted to.

Returns

data: ***results.DFT_out*** Generic DFT output object containing all parsed results.

2.10 jobs

The Job module contains the Job class that wraps simulations for queue submission. Further, it contains functionality to aid in simulation submission to queueing systems.

- *Job*
- *get_all_jobs()*
- *get_running_jobs()*
- *get_pending_jobs()*
- *submit_job()*
- *pysub()*

---

class jobs.**Job**(*name*, *process_handle=None*, *job_id=None*)

Job class to wrap simulations for queue submission.

Parameters

name: ***str*** Name of the simulation on the queue.

process_handle: ***process_handle, optional*** The process handle, returned by subprocess.Popen.

Returns

This *Job* object.

is_finished()
Check if simulation has finished or not.

Returns

is_on_queue: ***bool*** Whether the simulation is still running (True), or not (False).

wait(*tsleep=60*, *verbose=False*)
Hang until simulation has finished.

Returns

None

jobs.**get_all_jobs**(*queueing_system='nbs'*, *detail=0*)
Get a list of all jobs currently on your queue. The *detail* variable can be used to specify how much information you want returned.

Parameters

queueing_system: ***str, optional*** Which queueing system you are using (NBS or PBS).

detail: ***int, optional*** The amount of information you want returned.

Returns

all_jobs: ***list*** Depending on *detail*, you get the following:

- ***details* =0:** ***list, str*** List of all jobs on the queue.
- ***details* =1:** ***list, tuple, str***

  List of all jobs on the queue as: (job name, time run, job status)
- ***details* =2:** ***list, tuple, str***

  List of all jobs on the queue as:

  (job name, time run, job status, queue, number of processors)

jobs.**get_pending_jobs**(*queueing_system='nbs'*, *detail=0*)
Get a list of all jobs currently pending on your queue. The *detail* variable can be used to specify how much information you want returned.

Parameters

queueing_system: ***str, optional*** Which queueing system you are using (NBS or PBS).

detail: ***int, optional*** The amount of information you want returned.

Returns

all_jobs: ***list*** Depending on *detail*, you get the following:

- ***details* =0:** ***list, str*** List of all pending jobs on the queue.
- ***details* =1:** ***list, tuple, str***

  List of all pending jobs on the queue as: (job name, time run, job status)
- ***details* =2:** ***list, tuple, str***

  List of all pending jobs on the queue as:

  (job name, time run, job status, queue, number of processors)

jobs.**get_running_jobs**(*queueing_system='nbs'*, *detail=1*)

Get a list of all jobs currently running on your queue. The *detail* variable can be used to specify how much information you want returned.

Parameters

queueing_system: ***str, optional*** Which queueing system you are using (NBS or PBS).

detail: ***int, optional*** The amount of information you want returned.

Returns

all_jobs: ***list*** Depending on *detail*, you get the following:

- ***details* =0:** ***list, str*** List of all running jobs on the queue.
- ***details* =1:** ***list, tuple, str***

  List of all running jobs on the queue as: (job name, time run, job status)
- ***details* =2:** ***list, tuple, str***

  List of all running jobs on the queue as:

  (job name, time run, job status, queue, number of processors)

jobs.**pysub**(*job_name*, *nprocs=1*, *ntasks=1*, *nodes=1*, *adjust_nodes=True*, *omp=None*, *queue='long'*, *walltime='00:30:00'*, *xhost=None*, *path='/fs/home/hch54/squid/docs'*, *priority=None*, *args=None*, *remove_sub_script=True*, *unique_name=False*, *redundancy=False*, *py3=False*, *use_mpi=False*, *queueing_system='nbs'*)

Submission of python scripts to run on your queue.

Parameters

job_name: ***str*** Name of the python script (with or without the .py extension).

nprocs: ***int, optional*** Number of processors to run your script on.

ntasks: ***int, optional*** (For SLURM) The number of tasks this job will run, each task uses procs number of cores.

nodes: ***int, optional*** (For SLURM) The number of nodes this job requires. If requesting ntasks * procs < 24 * nodes, a warning is printed, as on MARCC each node has only 24 cores.

adjust_nodes: ***bool, optional*** Whether to automatically calculate how many nodes is necessary when the user underspecifies nodes.

use_mpi: ***bool, optional*** Whether to run python via mpirun or not.

omp: ***int, None*** The number OMP_NUM_THREADS should be manually assigned to.

queue: ***str, optional*** Which queue you want your script to run on (specific to your queueing system).

priority: ***int, optional*** Priority of the simulation (queueing system dependent). Priority ranges (in NBS) from a low of 1 (start running whenever) to a high of 255 (start running ASAP).

xhost: ***list, str or str*** Which processors you want your script to run on (specific to your queueing system).

path: ***str, optional*** What directory your python script resides in. Note, this does NOT have a trailing /.

args: ***list, str, optional*** A list of arguments to pass to the python script on the queue.

remove_sub_script: ***bool, optional*** Whether to remove the script used to submit the job (True), or leave it (False).

unique_name: ***bool, optional*** Whether the simulation should have a unique name. By default, no.

redundancy: ***bool, optional*** With redundancy on, if the job is submitted and unique_name is on, then if another job of the same name is running, a pointer to that job will instead be returned.

py3: ***bool, optional*** Whether to run with python3 or python2 (2 is default). NOTE! This will ONLY work if you have defined python3_path in your sysconst file.

queueing_system: ***str, optional*** Which queueing system you are using (NBS or PBS).

Returns

None

jobs.**submit_job**(*name*, *job_to_submit*, *procs=1*, *ntasks=1*, *nodes=1*, *adjust_nodes=True*, *queue='long'*, *mem=1000*, *priority=None*, *walltime='00:30:00'*, *xhosts=None*, *additional_env_vars=''*, *sandbox=None*, *use_NBS_sandbox=False*, *sub_flag=''*, *email=None*, *preface=None*, *redundancy=False*, *unique_name=True*, *queueing_system='nbs'*)

Code to submit a simulation to the specified queue and queueing system.

Parameters

name: ***str*** Name of the job to be submitted to the queue. In the case of NBS, it must be without the suffix. Ex. 'job' works but 'job.nbs' fails.

job_to_submit: ***str*** String holding code you wish to submit.

procs: ***int, optional*** Number of processors requested.

ntasks: ***int, optional*** (For SLURM) The number of tasks this job will run, each task uses procs number of cores.

nodes: ***int, optional*** (For SLURM) The number of nodes this job requires. If requesting ntasks * procs < 24 * nodes, a warning is printed, as on MARCC each node has only 24 cores.

adjust_nodes: ***bool, optional*** Whether to automatically calculate how many nodes is necessary when the user underspecifies nodes.

queue: ***str, optional*** Queue you are submitting to (queueing system dependent).

mem: ***float, optional*** Amount of memory you're requesting.

priority: ***int, optional*** Priority of the simulation (queueing system dependent). Priority ranges (in NBS) from a low of 1 (start running whenever) to a high of 255 (start running ASAP).

xhosts: ***list, str or str, optional*** Which processors you want to run the job on.

additional_env_vars: ***str, optional*** Additional environment variables to be appended to the job.

sandbox: ***list, list, str, optional*** A list of two lists. The first holds a list of files to be sent to the sandbox and the latter a list of files to be returned from the sandbox.

use_NBS_sandbox: ***bool, optional*** Whether to use the NBS sandboxing headers (True), or manually copy files (False).

sub_flag: ***str, optional*** Additional flags to be used during job submission.

redundancy: ***bool, optional*** With redundancy on, if the job is submitted and unique_name is on, then if another job of the same name is running, a pointer to that job will instead be returned.

unique_name: ***bool, optional*** Whether the simulation should have a unique name. By default, no.

queueing_system: ***str, optional*** Which queueing system you are using (NBS, PBS, or SLURM).

Returns

None

2.11 joust

Job Organizer for User Simulation Tasks (JOUST).

- *Joust*

---

class joust.**Joust**(*name*, *global_system=None*, *queue=None*, *procs=1*, *mem=1000*, *priority=100*, *xhosts=None*)

The Job Organizer for User Simulation Tasks (JOUST) is the Squid workflow manager, used for automating the process of simulating consecutive jobs.

It works by allowing the user to add_task() and del_task() from a list of tasks to run. When ready, start() can be called to begin simulating.

Note, you can run tasks in parallel simply by having them in a list. That is, instead of:

task_order = ["t1", "t2", "t3", "t4"]

you can run "t1" and "t2" in parallel by doing the following:

task_order = [["t1", "t2"], "t3", "t4"]

Similarly, every task can be run in parallel as follows:

task_order = [["t1", "t2", "t3", "t4"]]

add_task(*task_name*, *task*, *append_to_run_list=True*)

Append a task to be run. This is an order dependent process and thus, tasks added first will run first. Note, not all tasks added need be run. Any task that may be called from another, but only when a specific conditional is met, should also be added here. In those cases, use:

```
append_to_run_list = False
```

Parameters

task_name: ***str*** Name of the task to be run.

task: Task object.

append_to_run_list: ***bool, optional*** Whether this will add the task to the run list, or not.

Returns

None

del_task(*task_name*)

Remove a task from the task list. This will delete the task, as well as any instances in which it would have been called by JOUST. NOTE! This does NOT remove the task from other tasks however.

Parameters

task_name: ***str*** Name of the task to be removed.

start()

Start the workflow manager.

Returns

None

2.12 lammps_job

The lammps_job module contains .

---

lammps_job.**OptJob**(*run_name, input_script, system, domain, spanMolecule, resolution=0.1, queue=None, procs=1, email='', pair_coeffs_included=True, hybrid_pair=False, orientations=10, split=1, floor=[0.0, 0.0, 0.0]*)

Runs a series of lammps jobs in a space defined by domain and resolution, at each position performing a rand_rotation of the spanMolecule. Runs the simulation at that position orienations many times. Each simulation line will be split over split many jobs.

Parameters

run_name: ***str*** Name of the simulation to be run.

input_script: ***str*** Input script for LAMMPs simulation.

system: ***structures.System*** System object for our simulation.

domain: ***list, float*** A list of 3 elements referring to the XYZ domain in which the molecule spanMolecule will raster across. Ex. [0.0, 1.0, 1.0] will span over a 1x1 angstrom box in the positive y and z directions from spanMolecule's original position.

spanMolecule: ***str*** The path to a cml file representing a molecule to be rastered across the surface.

resolution: ***float, optional*** The change in position to be taken during rastering.

queue: ***str, optional*** What queue to run the simulation on (queueing system dependent).

procs: ***int, optional*** How many processors to run the simulation on.

email: ***str, optional*** An email address for sending job information to.

pair_coeffs_included: ***bool, optional*** Whether we have included the pair coefficients to be written to our lammps data file.

hybrid_pair: ***bool, optional*** Whether to detect different treatments of pairing interactions amongst different atom types(True), or not (False).

orientations: ***int, optional*** The number of random orientations to be run at every position.

split: ***int, optional*** The number of optimizations to append to a single lammps simulation. This is primarily used when batching jobs for the queue.

floor: ***list, float, optional*** A position to set spanMolecule to prior to rastering.

Returns

None

lammps_job.**PotEngSurfaceJob**(*run_name, input_script, system, domain, spanMolecule, resolution=0.1, queue='long', procs=1, email='', pair_coeffs_included=True, hybrid_pair=False, split=0, floor=[0.0, 0.0, 0.0]*)

TO BE ADDED AGAIN

lammps_job.**job**(*run_name*, *input_script*, *system=None*, *queue='long'*, *procs=1*, *ntasks=1*, *nodes=1*, *adjust_nodes=True*, *email=None*, *write_data_file=True*, *pair_coeffs_included=True*, *hybrid_pair=False*, *hybrid_angle=False*, *TIP4P=False*, *no_echo=False*, *redundancy=False*, *params=None*, *lmp_path='/fs/europa/g_pc/lmp_serial'*)

Wrapper to submitting a LAMMPs simulation.

Parameters

run_name: ***str*** Name of the simulation to be run.

input_script: ***str*** Input script for LAMMPs simulation.

system: ***structures.System*** System object for our simulation.

queue: ***str, optional*** What queue to run the simulation on (queueing system dependent).

procs: ***int, optional*** How many processors to run the simulation on. Note, the actual number of cores mpirun will use is procs * ntasks.

ntasks: ***int, optional*** (For SLURM) The number of tasks this job will run, each task uses procs number of cores. Note, the actual number of cores mpirun will use is procs * ntasks.

nodes: ***int, optional*** (For SLURM) The number of nodes this job requires. If requesting ntasks * procs < 24 * nodes, a warning is printed, as on MARCC each node has only 24 cores.

adjust_nodes: ***bool, optional*** Whether to automatically calculate how many nodes is necessary when the user underspecifies nodes.

email: ***str, optional*** An email address for sending job information to.

pair_coeffs_included: ***bool, optional*** Whether we have included the pair coefficients to be written to our lammps data file.

hybrid_pair: ***bool, optional*** Whether to detect different treatments of pairing interactions amongst different atom types(True), or not (False).

hybrid_angle: ***bool, optional*** Whether to detect different treatments of angles amongst different atom types (True), or not (False).

TIP4P: ***bool, optional*** Whether to identify TIP4P settings within the lammps data file and update the input file (True), or not (False).

no_echo: ***bool, optional*** Whether to pipe the terminal output to a file instead of printing.

redundancy: ***bool, optional*** With redundancy on, if the job is submitted and unique_name is on, then if another job of the same name is running, a pointer to that job will instead be returned.

lmp_path: ***str, optional*** The path to the lammps executable. Note, by default this is the one defined during installation, saved in sysconst.lmp_path.

Returns

job: ***jobs.Job*** If running locally, return the process handle, else return the job container.

class lammps_job.**lmp_task**(*task_name*, *system=None*, *queue=None*, *procs=1*, *mem=1000*, *priority=None*, *xhosts=None*, *callback=None*, *no_echo=True*, *persist_system=False*)

The LAMMPs task object for JOUST. This allows for the automation of some workflows.

Parameters

task_name: ***str*** The name of this task.

system: ***structures.System*** A system object to be used for this simulation.

queue: ***str, optional*** Queue you are submitting to (queueing system dependent).

procs: ***int, optional*** Number of processors requested.

mem: ***float, optional*** Amount of memory you're requesting.

priority: ***int, optional*** Priority of the simulation (queueing system dependent). Priority ranges (in NBS) from a low of 1 (start running whenever) to a high of 255 (start running ASAP).

xhosts: ***list, str or str, optional*** Which processors you want to run the job on.

callback: ***func, optional*** A function to be run at the end of the task (only if conditional is not met).

Returns

task: ***task*** This task object.

read_results()

Parse the output of the simulation that was just run.

** Returns**

None

run()

Start the LAMMPs simulation specified by this task.

Returns

sim_handle: ***jobs.Job*** A Job container for the simulation that was submitted.

set_parameters(*input_script*, *email=None*, *pair_coeffs_included=True*, *hybrid_pair=False*, *hybrid_angle=False*, *trj_file=''*, *xyz_file=''*, *read_atoms=True*, *read_timesteps=True*, *read_num_atoms=True*, *read_box_bounds=True*)

Set parameters for the LAMMPs task.

Parameters

input_script: ***str*** Input script for LAMMPs simulation.

email: ***str, optional*** An email address for sending job information to.

pair_coeffs_included: ***bool, optional*** Whether we have included the pair coefficients to be written to our lammps data file.

hybrid_pair: ***bool, optional*** Whether to detect different treatments of pairing interactions amongs different atom types(True), or not (False).

hybrid_angle: ***bool, optional*** Whether to detect different treatments of angles amongst different atom types (True), or not (False).

trj_file: ***str, optional*** Pass the path to a lammps trajectory file. Relative paths are assumed to be in a subfolder "lammps/RUN_NAME/RUN_NAME.lammpstrj".

read_atoms: ***bool, optional*** Whether to read in the atom information.

read_timesteps: ***bool, optional*** Whether to read in the timesteps (True), or not (False).

read_num_atoms: ***bool, optional*** Whether to read in the number of atoms (True), or not (False).

read_box_bounds: ***bool, optional*** Whether to read in the system box boundaries (True), or not (False).

Returns

None

lammps_job.**read**(*run_name*, *trj_file=''*, *xyz_file=''*, *read_atoms=True*, *read_timesteps=True*, *read_num_atoms=True*, *read_box_bounds=True*)

General read in of thermo information from a lammps log file, as well as (optionally) a lammps trajectory file (.lammpstrj).

NOTE! This is important. If you plan to use this function, you MUST have "Step" in your LAMMPs Thermo output.

Parameters

run_name: ***str*** Lammps .log file to be parsed. Note, this is WITHOUT the extension (ex. test_lmp instead of test_lmp.log).

trj_file: ***str, optional*** Pass the path to a lammps trajectory file. Relative paths are assumed to be in a subfolder"lammps/RUN_NAME/RUN_NAME.lammpstrj".

xyz_file: ***str, optional*** Pass the path to a lammps xyz output. Relative paths are assumed to be in a subfolder "lammps/RUN_NAME/RUN_NAME.xyz".

read_atoms: ***bool, optional*** Whether to read in the atom information.

read_timesteps: ***bool, optional*** Whether to read in the timesteps (True), or not (False).

read_num_atoms: ***bool, optional*** Whether to read in the number of atoms (True), or not (False).

read_box_bounds: ***bool, optional*** Whether to read in the system box boundaries (True), or not (False).

Returns

lg: Lammps log file, parsed.

data_trj: Trajectory file, if it exists.

data_xyz: XYZ file, if it exists.

NOTE! THE OUTPUT SHOULD BE:

data: ***results.sim_out*** Generic LAMMPs output object containing all parsed results.

lammps_job.**read_TIP4P_types**(*data_file*)

Used to find the TIP4P water atoms, bond and angle types in the lammps data file. Returns an integer for each of the types. This method looks for particular sequences, which may not be unique under certain circumstances so it should be used with caution.

Parameters

data_file: ***str*** Lammps data file name.

Returns

otype: ***int*** The lammps atom type for TIP4P oxygen.

htype: ***int*** The lammps atom type for TIP4P hydrogen.

btype: ***int*** The lammps atom type for TIP4P bond.

atype: ***int*** The lammps atom type for TIP4P angle.

lammps_job.**read_dump**(*fptr, ext='.dump', coordinates=['x', 'y', 'z'], extras=[]*)
Function to read in a generic dump file. Currently it (1) requires element, x, y, z in the dump. You can also use xu, yu, and zu if the unwraped flag is set to True.

Due to individual preference, the extension was separated. Thus, if you dump to .xyz, have ext=".xyz", etc.

Parameters

fptr: ***str*** Name of the dump file with NO extension (ex. 'run' instead of 'run.dump'). This can also be a relative path. If no relative path is given, and the file cannot be found, it will default check in lammps/fptr/fptr+ext.

ext: ***str, optional*** The extension for the dump file. Note, this is default ".dump" but can be anything (ensure you have the ".").

coordinates: ***list, str, optional*** A list of strings describing how the coordinates are specified (x vs xs vs xu vs xsu)

extras: ***list, str, optional*** An additional list of things you want to read in from the dump file.

Returns

frames: ***list, list*** ***structures.Atom*** A list of lists, each holding atom structures.

lammps_job.**read_thermo**(*run_name*, **properties*)
Read in thermo output from a lammps log file.

Parameters

run_name: ***str*** Lammps .log file to be parsed. Note, this is WITHOUT the extension (ex. test_lmp instead of test_lmp.log).

properties: ***str*** A sequence of lammps thermo keywords to be parsed, only used when all is not required.

Returns

lj: *lammps_log*

lammps_job.**thermo_2_text**(*run_name*, **properties*)
This will convert a lammps .log file to a parsed .txt file, isolating the thermo output.

Parameters

run_name: ***str*** Lammps .log file to be parsed. Note, this is WITHOUT the extension (ex. test_lmp instead of test_lmp.log).

properties: ***str*** A sequence of lammps thermo keywords to minimize output .txt file.

Example

```
>>> thermo_2_text("test_run", "Time", "KE")
```

Returns

None

2.13 lammps_log

The lammps_log code is code from the pizza.py toolkit (www.cs.sandia.gov/~sjplimp/pizza.html) developed by Steve Plimpton (sjplimp@sandia.gov) with some additional warning interspersed through the thermo output.

- *lammps_log*

class lammps_log.**lammps_log**(**list*)

Class object to assist in parsing lammps outputs.

Parameters

list: ***str*** Path to the lammps log file that is to be parsed. Note, several files can be included in this string as long as they are separated by spaces.

read_all: ***int, optional*** If this is set to 0, don't read in the whole file upon initialization. This lets you use the *next()* functionality.

Contains

nvec: ***int*** Number of vectors.

nlen: ***int*** Length of each vector.

names: ***list, str*** List of vector names.

ptr: ***dict*** Dictionary corresponding the thermo keys to which column of the output they reside in. ptr[thermo_key] = which column this data is in

data: ***list, list, float*** Raw data from file, organized into 2d array.

style: ***int*** What style the LAMMPs log file is in. 1 = multi, 2 = one, 3 = gran

firststr: ***str*** String that begins a thermo section in log file.

increment: ***int*** 1 if log file being read incrementally

eof: ***int*** ptr into incremental file for where to start next read

Returns

This *lammps_log* object.

get(**keys*)

Read specific values from thermo output.

Parameters

keys: ***str*** Which thermo outputs you want by ID. Not, this is as many requests as you want. ex. l.get("Time", "KE", ...)

Returns

vecs: ***list*** Desired outputs.

next()

Read the next line of thermo information from the file. Note, this is used when two arguments are passed during initialization.

Returns

timestep: ***int*** The timestep of the parsed thermo output.

write(*filename*, **keys*)

Write parsed vectors to a file.

Parameters

filename: ***str*** The name of the file you want to dump all your outputs to.

keys: ***str, optional*** Which specific vectors you want output to the file. ex. >>> l.write("file.txt","Time", "KE", ...)

Returns

None

2.14 linux_helper

The Linux Helper module contains functionality to aid linux users to automate some tasks.

- *clean_up_folder()*

---

linux_helper.**clean_up_folder**(*path*, *files_to_remove=[]*, *remove_empty_folders=False*, *verbose=False*)

Automate the removal of files from a linux system. Given a parent directory, this will recursively remove specified files.

Parameters

path: ***str*** Absolute path to the parent directory to be cleaned.

files_to_remove: ***list, str, optional*** List of files to be removed. Wildcards can be used, thus [“*.txt”,”*.log”] would delete every file with the .txt and .log extension.

remove_empty_folders: ***bool, optional*** Whether to remove empty folders (True), or not (False).

verbose: ***bool, optional*** Whether to output commands used (True), or not (False).

Returns

None

2.15 neb

The NEB module simplifies the submission of Nudged Elastic Band simulations.

- *g09_start_job()*
- *g09_results()*
- *orca_start_job()*
- *orca_results()*
- *NEB*

---

class neb.**NEB**(*name*, *states*, *theory*, *extra_section=”*, *initial_guess=None*, *spring_atoms=None*, *procs=1*, *queue=None*, *mem=2000*, *priority=None*, *disp=0*, *k=0.00367453*, *charge=0*, *multiplicity=1*, *fit_rigid=True*, *DFT=’orca’*, *opt=’LBFGS’*, *start_job=None*, *get_results=None*, *new_opt_params={}*, *callback=None*, *ci_neb=False*, *ci_N=5*, *no_energy=False*)

A method for determining the minimum energy pathway of a reaction using DFT. Note, this method was written for atomic orbital DFT codes; however, is potentially generalizable to other programs.

Parameters

name: ***str*** The name of the NEB simulation to be run.

states: ***list, list, structures.Atom*** A list of frames, each frame being a list of atom structures. These frames represent your reaction coordinate.

theory: ***str*** The route line for your DFT simulation.

extra_section: ***str, optional*** Additional parameters for your DFT simulation.

initial_guess: ***list, str, optional*** TODO - List of strings specifying a previously run NEB simulation, allowing restart capabilities.

spring_atoms: ***list, int, optional*** Specify which atoms will be represented by virutal springs in the NEB calculations. Default includes all.

procs: ***int, optional*** The number of processors for your simulation.

queue: ***str, optional*** Which queue you wish your simulation to run on (queueing system dependent). When None, NEB is run locally.

mem: ***float, optional*** Specify memory constraints (specific to your X_start_job method).

priority: ***int, optional*** Whether to submit a DFT simulation with some given priority or not.

disp: ***int, optional*** Specify for additional stdout information.

charge: ***int*** Charge of the system.

multiplicity: ***int*** Multiplicity of the system.

k: ***float, optional*** The spring constant for your NEB simulation.

fit_rigid: ***bool, optional*** Whether you want to use procrustes to minimize motion between adjacent frames (thus minimizing error due to excessive virtal spring forces).

DFT: ***str, optional*** Specify if you wish to use the default X_start_job and X_results functions where X is either g09 or orca.

opt: ***str, optional*** Select which optimization method you wish to use from the following: BFGS, LBFGS, SD, FIRE, QM, CG, scipy_X. Note, if using scipy_X, change X to be a valid scipy minimize method.

start_job: ***func, optional*** A function specifying how to submit your NEB single point calculations. Needed if DFT is neither orca nor g09.

get_results: ***func, optional*** A function specifying how to read your NEB single point calculations. Needed if DFT is neither orca nor g09. Note, this function returns two things: list of energies, list of atoms. Further, the forces are contained within each atom object. It also requires that the forces on the state object be updated within said function (for more info see example codes). Finally, if using no_energy=True, then return None (or an empty list) for the energies.

new_opt_params: ***dict, optional*** Pass any additional parameters to the optimization algorithm.

callback: ***func, optional*** A function to be run after each each to calculate().

ci_neb: ***bool, optional*** Whether to use the climbing image variation of NEB.

ci_N: ***int, optional*** How many iterations to wait in climbing image NEB before selecting which image to be used.

no_energy: ***bool, optional*** A flag to turn on an experimental method, in which our selection of the tangent is based on only the force, and not the energy. Note, the code still expects the get_results function to return two things, so just have it return (None, atoms + forces).

Returns

This *NEB* object.

References

- Henkelman, G.; Jonsson, H. The Journal of Chemical Physics 2000, 113, 9978-9985.

- Jonsson, H.; Mills, G.; Jacobson, K. W. In Classical and Quantum Dynamics in Condensed Phase Simulations;
- Berne, B. J., Ciccotti, G., Coker, D. F., Eds.; World Scientific, 1998; Chapter 16, pp 385-404.
- Armijo, L. Pacific Journal of Mathematics 1966, 16.
- Sheppard, D.; Terrell, R.; Henkelman, G. The Journal of Chemical Physics 2008, 128.
- Henkelman, G.; Uberuaga, B. P.; Jonsson, H. Journal of Chemical Physics 2000, 113.
- Atomic Simulation Environment - https://wiki.fysik.dtu.dk/ase/

align_coordinates(*r*, *B=None*, *H=None*, *return_matrix=False*)

Get a rotation matrix A that will remove rigid rotation from the new coordinates r. Further, if another vector needs rotating by the same matrix A, it should be passed in B and will be rotated. If a matrix also needs rotating, it can be passed as H and also be rotated.

Parameters

r: ***list, float*** 1D array of atomic coordinates to be rotated by procrustes matrix A.

B: ***list, list, float, optional*** A list of vectors that may also be rotated by the same matrix as *r*.

H: ***list, list, float, optional***

A matrix that should also be rotated via: H = R * H * R.T

return_matrix: ***bool, optional*** Whether to also return the rotation matrix used or not.

Returns

rotations: ***dict*** A dictionary holding 'A', the rotation matrix, 'r', the rotated new coordinates, 'B', a list of all other vectors that were rotated, and 'H', a rotated matrix.

neb.**g09_results**(*NEB*, *step_to_use*, *i*, *state*)

A method for reading in the output of Gaussian09 single point calculations for NEB calculations. This will both (a) assign forces to the atoms stored in state and (b) return the energy and atoms.

Parameters

NEB: ***NEB*** An NEB container holding the main NEB simulation

step_to_use: ***int*** Which iteration in the NEB sequence the output to be read in is on.

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

Returns

new_energy: ***float*** The energy of the system in Hartree (Ha).

new_atoms: ***list,*** ***structures.Atom*** A list of atoms with the forces attached in units of Hartree per Angstrom (Ha/Ang).

neb.**g09_start_job**(*NEB*, *i*, *state*, *charge*, *multiplicity*, *procs*, *queue*, *initial_guess*, *extra_section*, *mem*, *priority*)

A method for submitting a single point calculation using Gaussian09 for NEB calculations.

Parameters

NEB: ***NEB*** An NEB container holding the main NEB simulation

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: *list,* ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

charge: *int* Charge of the system.

multiplicity: *int* Multiplicity of the system.

procs: *int* The number of processors to use during calculations.

queue: *str* Which queue to submit the simulation to (this is queueing system dependent).

initial_guess: *str* The name of a previous simulation for which we can read in a hessian.

extra_section: *str* Extra settings for this DFT method.

mem: *int* How many Mega Words (MW) you wish to have as dynamic memory.

priority: *int* Whether to submit the job with a given priority (NBS). Not setup for this function yet.

Returns

g09_job: ***jobs.Job*** A job container holding the g09 simulation.

neb.**orca_results**(*NEB*, *step_to_use*, *i*, *state*)

A method for reading in the output of Orca single point calculations for NEB calculations. This will both (a) assign forces to the atoms stored in state and (b) return the energy and atoms.

Parameters

NEB: ***NEB*** An NEB container holding the main NEB simulation

step_to_use: *int* Which iteration in the NEB sequence the output to be read in is on.

i: *int* The index corresponding to which image on the frame is to be simulated.

state: *list,* ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

Returns

new_energy: *float* The energy of the system in Hartree (Ha).

new_atoms: *list,* ***structures.Atom*** A list of atoms with the forces attached in units of Hartree per Angstrom (Ha/Ang).

neb.**orca_start_job**(*NEB*, *i*, *state*, *charge*, *multiplicity*, *procs*, *queue*, *initial_guess*, *extra_section*, *mem*, *priority*)

A method for submitting a single point calculation using Orca for NEB calculations.

Parameters

NEB: ***NEB*** An NEB container holding the main NEB simulation

i: *int* The index corresponding to which image on the frame is to be simulated.

state: *list,* ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

charge: *int* Charge of the system.

multiplicity: *int* Multiplicity of the system.

procs: *int* The number of processors to use during calculations.

queue: *str* Which queue to submit the simulation to (this is queueing system dependent).

initial_guess: *str* The name of a previous simulation for which we can read in a hessian.

extra_section: *str* Extra settings for this DFT method.

mem: ***int*** How many MegaBytes (MB) of memory you have available per core.

priority: ***int*** Whether to submit to NBS with a given priority

Returns

orca_job: *jobs.Job* A job container holding the orca simulation.

2.16 optimizers

The files module contains various functions aiding in file input and output.

- *steepest_descent()*
- *bfgs()*
- *lbfgs()*
- *quick_min()*
- *fire()*
- *conjugate_gradient()*

---

steepest_descent.**steepest_descent**(*params*, *gradient*, *NEB_obj=None*, *new_opt_params={}*, *extra_args_gradient=None*, *extra_args_target=None*)

A steepest descent optimizer, overloaded for NEB use.

Parameters

params: ***list, float*** A list of parameters to be optimized.

gradient: ***func*** A function that, given params, returns the gradient.

NEB_obj: *neb.NEB* An NEB object to use.

new_opt_params: ***dict*** A dictionary holding any changes to the optimization algorithm's parameters. This includes the following -

step_size: ***float*** Step size to take.

step_size_adjustment: ***float*** A factor to adjust step_size when a bad step is made.

max_step: ***float*** A maximum allowable step length. If 0, any step is ok.

target_function: ***func*** A function that will help decide if backtracking is needed or not. This function will be used to verify BFGS is minimizing. If nothing is passed, but NEB_obj is not None, the NEB_obj.get_error function will be called.

armijo_line_search_factor: ***float*** A factor for the armijo line search.

linesearch: ***str*** Whether to use the *armijo* or *backtrack* linesearch method. If None is passed, a static step_size is used.

reset_when_in_trouble: ***bool*** Whether to reset the Hessian to Identity when bad steps have been taken.

reset_step_size: ***int*** How many iterations of 'good' steps to take before resetting step_size to its initial value.

accelerate: ***bool*** Whether to accelerate via increasing step_size by 1/step_size_adjustment when no bad steps are taken after *reset_step_size* iterations.

maxiter: ***int*** Maximum number of iterations for the optimizer to run. If None, then the code runs indefinitely.

g_rms: ***float*** The RMS value for which to optimize the gradient to.

g_max: ***float*** The maximum gradient value to be allowed.

fit_rigid: ***bool*** Remove erroneous rotation and translations during NEB.

dimensions: ***int*** The number of dimensions for the optimizer to run in. By default this is 3 (for NEB atomic coordinates.)

callback: ***func, optional*** A function to be run after each optimization loop.

Returns

params: ***list, float*** A list of the optimized parameters.

code: ***int*** An integer describing how the algorithm converged. This can be identified in the constants file.

iters: ***int*** The number of iterations the optimizer ran for.

bfgs.**bfgs**(*params*, *gradient*, *NEB_obj=None*, *new_opt_params={}*)

A Broyden-Fletcher-Goldfarb-Shanno optimizer, overloaded for NEB use.

Parameters

params: ***list, float*** A list of parameters to be optimized.

gradient: ***func*** A function that, given params, returns the gradient.

NEB_obj: ***neb.NEB*** An NEB object to use.

new_opt_params: ***dict*** A dictionary holding any changes to the optimization algorithm's parameters. This includes the following -

step_size: ***float*** Step size to take.

step_size_adjustment: ***float*** A factor to adjust step_size when a bad step is made.

max_step: ***float*** A maximum allowable step length. If 0, any step is ok.

target_function: ***func*** A function that will help decide if backtracking is needed or not. This function will be used to verify BFGS is minimizing. If nothing is passed, but NEB_obj is not None, the NEB_obj.get_error function will be called.

armijo_line_search_factor: ***float*** A factor for the armijo line search.

linesearch: ***str*** Whether to use the *armijo* or *backtrack* linesearch method. If None is passed, a static step_size is used.

reset_when_in_trouble: ***bool*** Whether to reset the Hessian to Identity when bad steps have been taken.

reset_step_size: ***int*** How many iterations of 'good' steps to take before resetting step_size to its initial value.

N_reset_hess: ***int*** A hard reset to the hessian to be applied every N iterations.

start_hess: ***int, float, or matrix*** A starting matrix to use instead of the identity. If an integer or float is passed, then the starting hessian is a scaled identity matrix.

use_numopt_start: ***bool*** Whether to use the starting hessian guess laid out by Nocedal and Wright in the Numerical Operations textbook, page 178. H0 = (<y|s>) / (<y|y>) * I. If chosen, start_hess is set to the identity matrix.

accelerate: ***bool*** Whether to accelerate via increasing step_size by 1/step_size_adjustment when no bad steps are taken after *reset_step_size* iterations.

maxiter: ***int*** Maximum number of iterations for the optimizer to run. If None, then the code runs indefinitely.

g_rms: ***float*** The RMS value for which to optimize the gradient to.

g_max: ***float*** The maximum gradient value to be allowed.

fit_rigid: ***bool*** Remove erroneous rotation and translations during NEB.

dimensions: ***int*** The number of dimensions for the optimizer to run in. By default this is 3 (for NEB atomic coordinates.)

callback: ***func, optional*** A function to be run after each optimization loop.

Returns

params: ***list, float*** A list of the optimized parameters.

code: ***int*** An integer describing how the algorithm converged. This can be identified in the constants file.

iters: ***int*** The number of iterations the optimizer ran for.

lbfgs.**lbfgs**(*params*, *gradient*, *NEB_obj=None*, *new_opt_params={}*, *extra_args_gradient=None*, *extra_args_target=None*)

A Limited Memory Broyden-Fletcher-Goldfarb-Shanno optimizer, overloaded for NEB use.

Parameters

params: ***list, float*** A list of parameters to be optimized.

gradient: ***func*** A function that, given params, returns the gradient.

NEB_obj: ***neb.NEB*** An NEB object to use.

new_opt_params: ***dict*** A dictionary holding any changes to the optimization algorithm's parameters. This includes the following -

step_size: ***float*** Step size to take.

step_size_adjustment: ***float*** A factor to adjust step_size when a bad step is made.

max_step: ***float*** A maximum allowable step length. If 0, any step is ok.

max_steps_remembered: ***int*** The maximum number of previous iterations to save.

target_function: ***func*** A function that will help decide if backtracking is needed or not. This function will be used to verify LBFGS is minimizing. If nothing is passed, but NEB_obj is not None, the NEB_obj.get_error function will be called.

armijo_line_search_factor: ***float*** A factor for the armijo line search.

linesearch: ***str*** Whether to use the *armijo* or *backtrack* linesearch method. If None is passed, a static step_size is used.

reset_when_in_trouble: ***bool*** Whether to reset the stored parameters and gradients when a bad step has been taken.

reset_step_size: ***int*** How many iterations of 'good' steps to take before resetting step_size to its initial value.

N_reset_hess: ***int*** A hard reset to the hessian to be applied every N iterations.

start_hess: ***int, float, or matrix*** A starting integer or float to scale the starting hessian.

use_numopt_start: ***bool*** Whether to use the starting hessian guess laid out by Nocedal and Wright in the Numerical Operations textbook, page 178. H0 = (<y|s>) / (<y|y>) * I. If chosen, start_hess is set to the identity matrix.

accelerate: ***bool*** Whether to accelerate via increasing step_size by 1/step_size_adjustment when no bad steps are taken after *reset_step_size* iterations.

maxiter: ***int*** Maximum number of iterations for the optimizer to run. If None, then the code runs indefinitely.

g_rms: ***float*** The RMS value for which to optimize the gradient to.

g_max: ***float*** The maximum gradient value to be allowed.

fit_rigid: ***bool*** Remove erroneous rotation and translations during NEB.

dimensions: ***int*** The number of dimensions for the optimizer to run in. By default this is 3 (for NEB atomic coordinates.)

callback: ***func, optional*** A function to be run after each optimization loop.

Returns

params: ***list, float*** A list of the optimized parameters.

code: ***int*** An integer describing how the algorithm converged. This can be identified in the constants file.

iters: ***int*** The number of iterations the optimizer ran for.

quick_min.**quick_min**(*params*, *gradient*, *NEB_obj=None*, *new_opt_params={}*)

A quick min optimizer, overloaded for NEB use. Note, this will ONLY work for use within the NEB code.

Parameters

params: ***list, float*** A list of parameters to be optimized.

gradient: ***func*** A function that, given params, returns the gradient.

NEB_obj: ***neb.NEB*** An NEB object to use.

new_opt_params: ***dict*** A dictionary holding any changes to the optimization algorithm's parameters. This includes the following -

dt: ***float*** Time step size to take.

max_step: ***float*** The maximum step size to take.

viscosity: ***float*** The viscosity within a verlet step (used if euler is False).

euler: ***bool*** Whether to make an euler step or not.

maxiter: ***int*** Maximum number of iterations for the optimizer to run. If None, then the code runs indefinitely.

g_rms: ***float*** The RMS value for which to optimize the gradient to.

g_max: ***float*** The maximum gradient value to be allowed.

fit_rigid: ***bool*** Remove erroneous rotation and translations during NEB.

verbose: ***bool*** Whether to have additional output.

callback: ***func, optional*** A function to be run after each optimization loop.

Returns

params: ***list, float*** A list of the optimized parameters.

code: ***int*** An integer describing how the algorithm converged. This can be identified in the constants file.

iters: ***int*** The number of iterations the optimizer ran for.

`fire.`**`fire`**(*params*, *gradient*, *NEB_obj=None*, *new_opt_params={}*)

A FIRE optimizer, overloaded for NEB use.

Parameters

params: ***list, float*** A list of parameters to be optimized.

gradient: ***func*** A function that, given params, returns the gradient.

NEB_obj: ***neb.NEB*** An NEB object to use.

new_opt_params: ***dict*** A dictionary holding any changes to the optimization algorithm's parameters. This includes the following -

dt: ***float*** Time step size to take.

dtmax: ***float, optional*** The maximum dt allowed.

max_step: ***float*** The maximum step size to take.

Nmin: ***int, optional*** The minimum number of steps before acceleration occurs.

finc: ***float, optional*** The factor by which dt increases.

fdec: ***float, optional*** The factor by which dt decreases.

astart: ***float, optional*** The starting acceleration.

fa: ***float, optional*** The factor by which the acceleration is scaled.

viscosity: ***float*** The viscosity within a verlet step (used if euler is False).

euler: ***bool*** Whether to make an euler step or not.

maxiter: ***int*** Maximum number of iterations for the optimizer to run. If None, then the code runs indefinitely.

g_rms: ***float*** The RMS value for which to optimize the gradient to.

g_max: ***float*** The maximum gradient value to be allowed.

fit_rigid: ***bool*** Remove erroneous rotation and translations during NEB.

callback: ***func, optional*** A function to be run after each optimization loop.

Returns

params: ***list, float*** A list of the optimized parameters.

code: ***int*** An integer describing how the algorithm converged. This can be identified in the constants file.

iters: ***int*** The number of iterations the optimizer ran for.

`conjugate_gradient.`**`conjugate_gradient`**(*params*, *gradient*, *NEB_obj=None*, *new_opt_params={}*, *extra_args_gradient=None*, *extra_args_target=None*)

A conjugate gradient optimizer, overloaded for NEB use.

Parameters

params: ***list, float*** A list of parameters to be optimized.

gradient: ***func*** A function that, given params and an abstract list of extra arguments, returns the gradient.

NEB_obj: ***neb.NEB*** An NEB object to use.

new_opt_params: ***dict*** A dictionary holding any changes to the optimization algorithm's parameters. This includes the following -

step_size: ***float*** Step size to take.

step_size_adjustment: ***float*** A factor to adjust step_size when a bad step is made.

method: ***str*** Whether to use the Fletcher-Reeves (FR) method of calculating beta, or the Polak-Ribiere (PR) method.

max_step: ***float*** A maximum allowable step length. If 0, any step is ok.

target_function: ***func*** A function that will help decide if backtracking is needed or not. This function will be used to verify BFGS is minimizing. If nothing is passed, but NEB_obj is not None, the NEB_obj.get_error function will be called.

armijo_line_search_factor: ***float*** A factor for the armijo line search.

linesearch: ***str*** Whether to use the *armijo* or *backtrack* linesearch method. If None is passed, a static step_size is used.

reset_step_size: ***int*** How many iterations of 'good' steps to take before resetting step_size to its initial value.

accelerate: ***bool*** Whether to accelerate via increasing step_size by 1/step_size_adjustment when no bad steps are taken after *reset_step_size* iterations.

maxiter: ***int*** Maximum number of iterations for the optimizer to run. If None, then the code runs indefinitely.

g_rms: ***float*** The RMS value for which to optimize the gradient to.

g_max: ***float*** The maximum gradient value to be allowed.

fit_rigid: ***bool*** Remove erroneous rotation and translations during NEB.

dimensions: ***int*** The number of dimensions for the optimizer to run in. By default this is 3 (for NEB atomic coordinates.)

callback: ***func, optional*** A function to be run after each optimization loop.

Returns

params: ***list, float*** A list of the optimized parameters.

code: ***int*** An integer describing how the algorithm converged. This can be identified in the constants file.

iters: ***int*** The number of iterations the optimizer ran for.

2.17 orca

The Orca module contains python functions for interfacing with the Orca DFT software package.

- *engrad_read()*
- *gbw_to_cube()*

- *job()*
- *mo_analysis()*
- *orca_task*
- *pot_analysis()*
- *read()*

---

orca.**engrad_read**(*input_file*, *force='Ha/Bohr'*, *pos='Bohr'*)
General read in of all possible data from an Orca engrad file (.orca.engrad).

Parameters

input_file: ***str*** Orca .orca.engrad file to be parsed.

force: ***str, optional*** Units you want force to be returned in.

pos: ***str, optional*** Units you want position to be returned in.

Returns

atoms: ***list,*** ***structures.Atom*** A list of the final atomic state, with forces appended to each atom.

energy: ***float*** The total energy of this simulation.

orca.**gbw_to_cube**(*name*, *mo*, *spin=0*, *grid=40*, *local=False*)
Pipe in flags to orca_plot to generate a cube file for the given molecular orbital. Note, this is assumed to be running from the parent directory (ie, gbw is in the orca/BASENAME/BASENAME.orca.gbw).

Parameters

name: ***str*** The base name of the gbw file. Thus, 'water' instead of 'water.orca.gbw'.

mo: ***int*** Which molecular orbital to generate the cube file for. Note, this is 0 indexed.

spin: ***int, optional*** Whether to plot the alpha or beta (0 or 1) operator.

grid: ***int, optional*** The grid resolution, default being 40.

Returns

mo_name: ***str*** The name of the output MO file.

orca.**job**(*run_name*, *route*, *atoms=[]*, *extra_section=''*, *grad=False*, *queue=None*, *walltime='00:30:00'*, *sandbox=True*, *procs=1*, *ntasks=1*, *nodes=1*, *adjust_nodes=True*, *charge=None*, *multiplicity=None*, *charge_and_multiplicity='0 1'*, *redundancy=False*, *use_NBS_sandbox=False*, *unique_name=True*, *previous=None*, *mem=2000*, *priority=None*, *xhost=None*, *orca4=False*)
Wrapper to submitting an Orca simulation.

Parameters

run_name: ***str*** Name of the simulation to be run.

route: ***str*** The DFT route line, containing the function, basis set, etc. Note, if route=None and previous != None, the route from the previous simulation will be used instead.

atoms: ***list,*** ***structures.Atom*** **,optional** A list of atoms for the simulation. If this is an empty list, but previous is used, then the last set of atomic coordinates from the previous simulation will be used.

extra_section: ***str, optional*** Additional DFT simulation parameters. If None and previous is not None, then previous extra section is used.

grad: ***bool, optional*** Whether to force RunTyp Gradient.

queue: ***str, optional*** What queue to run the simulation on (queueing system dependent).

sandbox: ***bool, optional*** Whether to run the job in a sandbox or not.

use_NBS_sandbox: ***bool, optional*** Whether to use the NBS sandboxing headers (True), or manually copy files (False).

procs: ***int, optional*** How many processors to run the simulation on. Note, the actual number requested by orca will be procs * ntasks.

ntasks: ***int, optional*** (For SLURM) The number of tasks this job will run, each task uses procs number of cores. Note, the actual number requested by orca will be procs * ntasks.

nodes: ***int, optional*** (For SLURM) The number of nodes this job requires. If requesting ntasks * procs < 24 * nodes, a warning is printed, as on MARCC each node has only 24 cores.

adjust_nodes: ***bool, optional*** Whether to automatically calculate how many nodes is necessary when the user underspecifies nodes.

charge: ***float, optional*** Charge of the system. If this is used, then charge_and_multiplicity is ignored. If multiplicity is used, but charge is not, then default charge of 0 is chosen.

multiplicity: ***int, optional*** Multiplicity of the system. If this is used, then charge_and_multiplicity is ignored. If charge is used, but multiplicity is not, then default multiplicity of 1 is chosen.

charge_and_multiplicity: ***str, optional*** Charge and multiplicity of the system. If neither charge nor multiplicity are specified, then both are grabbed from this string.

redundancy: ***bool, optional*** With redundancy on, if the job is submitted and unique_name is on, then if another job of the same name is running, a pointer to that job will instead be returned.

unique_name: ***bool, optional*** Whether to force the requirement of a unique name or not. NOTE! If you submit simulations from the same folder, ensure that this is True lest you have a redundancy problem! To overcome said issue, you can set redundancy to True as well (but only if the simulation is truly redundant).

previous: ***str, optional*** Name of a previous simulation for which to try reading in information using the MORead method.

mem: ***float, optional*** Amount of memory per processor that is available (in MB).

priority: ***int, optional*** Priority of the simulation (queueing system dependent). Priority ranges (in NBS) from a low of 1 (start running whenever) to a high of 255 (start running ASAP).

xhost: ***list, str or str, optional*** Which processor to run the simulation on(queueing system dependent).

orca4: ***bool, optional*** Whether to use orca 4 (True) or orca 3 (False).

Returns

job: ***jobs.Job*** Teturn the job container.

`orca.`**`mo_analysis`**(*name*, *orbital=None*, *HOMO=True*, *LUMO=True*, *wireframe=True*, *hide=True*, *iso=0.04*)

Post process an orca job using orca_plot and vmd to display molecular orbitals and the potential surface. NOTE! By default Orca does not take into account degenerate energy states when populating. To do so, ensure the following is in your extra_section:

'%scf FracOcc true end'.

Parameters

name: ***str*** Orca file name. Only use the name, such as 'water' instead of 'water.gbw'. Note, do not pass a path as it is assumed you are in the parent directory of the job to analyze. If not, use the path variable.

orbital: ***list, int, optional*** **or** ***int, optional*** The orbital(s) to analyze (0, 1, 2, 3, ...). By default HOMO and LUMO will be analyzed, thus this only is useful if you wish to see other orbitals.

HOMO: ***bool, optional*** If you want to see the HOMO level.

LUMO: ***bool, optional*** If you want to see the LUMO level.

wireframe: ***bool, optional*** If you want to view wireframe instead of default surface.

hide: ***bool, optional*** Whether to have the representations all off by or not when opening.

iso: ***float, optional*** Isosurface magnitude. Set to 0.04 by default, but 0.01 may be better.

Returns

None

class orca.**orca_task**(*task_name*, *system=None*, *queue=None*, *procs=1*, *mem=1000*, *priority=None*, *xhosts=None*, *callback=None*, *no_echo=True*, *persist_system=False*)

The orca task object for JOUST. This allows for the automation of some workflows.

Parameters

task_name: ***str*** The name of this task.

system: ***structures.System*** A system object to be used for this simulation.

queue: ***str, optional*** Queue you are submitting to (queueing system dependent).

procs: ***int, optional*** Number of processors requested.

mem: ***float, optional*** Amount of memory you're requesting.

priority: ***int, optional*** Priority of the simulation (queueing system dependent). Priority ranges (in NBS) from a low of 1 (start running whenever) to a high of 255 (start running ASAP).

xhosts: ***list, str or str, optional*** Which processors you want to run the job on.

callback: ***func, optional*** A function to be run at the end of the task (only if conditional is not met).

Returns

task: ***task*** This task object.

read_results()

Parse the output of the simulation that was just run.

** Returns**

None

run()

Start the Orca simulation specified by this task.

Returns

sim_handle: ***jobs.Job*** A Job container for the simulation that was submitted.

set_parameters(*route='! HF-3c'*, *extra_section=''*, *grad=False*, *charge=None*, *multiplicity=None*, *charge_and_multiplicity='0 1'*, *previous=None*)

Set parameters for the Orca task.

Parameters

route: ***route*** The DFT route line, containing the function, basis set, etc.

extra_section: ***str, optional*** Additional DFT simulation parameters.

grad: ***bool, optional*** Whether to force RunTyp Gradient.

charge: ***float, optional*** Charge of the system.

multiplicity: ***int, optional*** Multiplicity of the system.

charge_and_multiplicity: ***str, optional*** Charge and multiplicity of the system.

previous: ***str, optional*** Name of a previous simulation for which to try reading in information using the MORead method.

Returns

None

orca.**pot_analysis**(*name*, *wireframe=True*, *npoints=80*, *orca4=False*)

Post process an orca job using orca_plot and vmd to display the electrostatic potential mapped onto the electron density surface.

Parameters

name: ***str*** Orca file name. Only use the name, such as 'water' instead of 'water.gbw'. Note, do not pass a path as it is assumed you are in the parent directory of the job to analyze.

wireframe: ***bool, optional*** If you want to view wireframe instead of default surface.

npoints: ***int, optional*** The grid size for the potential surface.

orca4: ***bool, optional*** Whether to run this for orca4 outputs or not.

Returns

None

orca.**read**(*input_file*)

General read in of all possible data from an Orca output file (.out). It should be mentioned that atomic positions are 0 indexed.

Parameters

input_file: ***str*** Orca .out file to be parsed.

Returns

data: ***results.DFT_out*** Generic DFT output object containing all parsed results.

2.18 print_helper

The Linux Helper module contains functionality to aid linux users to automate some tasks.

- *color_set()*
- *colour_set()*
- *printProgressBar()*
- *strip_color()*
- *strip_colour()*
- *spaced_print()*

print_helper.**color_set**(*s*, *c*)
Colourize a string for linux terminal output.

Parameters

s: ***str*** String to be formatted.

c: ***str*** Colour or format for the string, found in constants.COLOUR.

Returns

s: ***str*** Coloured or formatted string.

print_helper.**colour_set**(*s*, *c*)
Colourize a string for linux terminal output.

Parameters

s: ***str*** String to be formatted.

c: ***str*** Colour or format for the string, found in constants.COLOUR.

Returns

s: ***str*** Coloured or formatted string.

print_helper.**printProgressBar**(*iteration*, *total*, *prefix=''*, *suffix=''*, *decimals=1*, *length=20*, *fill='+'*, *buf=None*, *pad=False*)
NOTE! THIS IS COPIED FROM STACK OVERFLOW (with minor changes), USER Greenstick Link: https://stackoverflow.com/a/34325723

Call in a loop to create terminal progress bar.

Parameters

iteration: ***int*** Current iteration.

total: ***int*** Total number of iterations.

prefix: ***str, optional*** Prefix for the loading bar.

suffix: ***str, optional*** Suffix for the loading bar.

decimals: ***int, optional*** Positive number of decimals in percent complete

length: ***int, optional*** Character length of the loading bar.

fill: ***str, optional*** Bar fill character.

pad: ***bool, optional*** Whether to pad the right side with spaces until terminal width.

print_helper.**spaced_print**(*sOut, delim=['\t', ' '], buf=4*)
Given a list of strings, or a string with new lines, this will reformat the string with spaces to split columns. Note, this only works if there are no headers to the input string/list of strings.

Parameters

sOut: ***str*** **or** ***list, str*** String/list of strings to be formatted.

delim: ***list, str*** List of delimiters in the input strings.

buf: ***int*** The number of spaces to have between columns.

Returns

spaced_s: ***str*** Appropriately spaced output string.

print_helper.**strip_color**(*s*)
Remove colour and/or string formatting due to linux escape sequences.

Parameters

s: ***str*** String to strip formatting from.

Returns

s: ***str*** Unformatted string.

print_helper.**strip_colour**(*s*)
Remove colour and/or string formatting due to linux escape sequences.

Parameters

s: ***str*** String to strip formatting from.

Returns

s: ***str*** Unformatted string.

2.19 rate_calc

The Rate Calculation module takes themochemistry information from an Orca single point calculation and calculates the exponential pre-factor of the Arrhenius equation at a specified temperature. The algorithm uses Transition State Theory (TST) as the basis for mathematically defining the reaction kinetics.

- *translation()*
- *vibration()*
- *rotation()*
- *activation_energy()*
- *get_rate()*

---

rate_calc.**activation_energy**(*molecule*)
Extracts the final energy of a reactant from an Orca output file.

Parameters

molecule: ***str*** A string for the name of the simulation containing the optimized reactant.

Returns

E_tmp: ***float*** The energy of the optimized reactant.

rate_calc.**get_rate**()
Calculate the pre-exponential factor of the Arrenhius equation for a reaction, using Transition State Theory (TST)

rate_calc.**rotation**(*molecule*)
Calculates the rotational partition function for a reactant.

Parameters

molecule: ***str*** A string for the name of the simulation containing the optimized reactant.

Returns

qrot: ***float*** The partition function for rotation of a reactant.

rate_calc.**translation**(*molecule*, *T*)
Calculates the translational partition function for a reactant.

Parameters

molecule: ***str*** A string for the name of the simulation containing the optimized reactant.

Returns

qtrans: ***float*** The partition function for translation of a reactant.

rate_calc.**vibration**(*molecule*, *T*)
Calculates the vibrational partition function for a reactant.

Parameters

molecule: ***str*** A string for the name of the simulation containing the optimized reactant.

Returns

qvib: ***float*** The partition function for vibration of a reactant.

2.20 results

The results module contains data structures to hold simulation output.

- *DFT_out*
- *sim_out*

---

class results.**DFT_out**(*name*, *dft='orca'*)
A generic class to hold dft data.

Parameters

name: ***str*** Given name for this simulation object.

dft: ***str, optional*** Identifier for which dft code this data is from.

Contains

route: ***str*** The 'route' line describing the functional, basis set, and other dft configurations.

extra_section: ***str*** The 'extra section' in the simulation.

charge_and_multiplicity: ***str*** The charge and multiplicity, in that order, of the system.

frames: ***list, list,*** ***structures.Atom*** A list lists of atoms describing each iteration in the dft simulation.

atoms: ***list,*** ***structures.Atom*** Atomic information of the last iteration in the dft simulation.

gradients: ***list, list, float*** Gradient of the potential, stored for each atom in *atoms* and *frames[-1]*.

energy: ***float*** The total energy of the last iteration.

charges_MULLIKEN: ***list, float*** Mulliken charges for each atom in *atoms* and *frames[-1]*.

charges_LOEWDIN: ***list, float*** Loewdin charges for each atom in *atoms* and *frames[-1]*.

charges_CHELPG: ***list, float*** Chelpg charges for each atom in *atoms* and *frames[-1]*.

charges: ***list, float*** Charges for each atom in *atoms* and *frames[-1]*. Typically a copy of Mulliken charges.

MBO: *list, list,* ***structures.Atom****, float* A list of lists, each list holding (1) a list of atoms in the bond and (2) the Mayer Bond Order (MBO) of said bond.

convergence: *list, str* **VERIFY** A list of convergence criteria and matching values.

converged: *bool* Whether the simulation converged (True), or not (False).

time: *float* Total time in seconds that the simulation ran for.

bandgap: *float* Bandgap of the final configuration.

bandgaps: *float* Bandgap of each configuration.

orbitals: *list, tuple, float, float* A list of tuples, each holding the information of the occupation and energy (Ha) of a molecular orbital. NOTE! This does not take into account degenerate energy states, so ensure that whatever DFT software you're using has already done so.

finished: *bool* Whether the simulation completed normally (True), or not (False).

warnings: *list, str* Warnings output by the simulation.

class results.**sim_out**(*name*, *program='lammps'*)

A generic class to hold simulation data, particularly lammps trajectory files.

Parameters

name: *str* Given name for this simulation object.

program: *str, optional* Identifier for which program this data is from.

Contains

frames: *list, list,* ***structures.Atom*** A list lists of atoms describing each iteration in the simulation.

atoms: *list,* ***structures.Atom*** Atomic information of the last iteration in the simulation.

timesteps: *list, int* Recorded timesteps within the output.

final_timestep: *int* Final timestep of the output.

atom_counts: *list, int* List of how many atoms for each timestep.

atom_count: *int* List of how many atoms in the final timestep.

box_bounds_list: *list,* ***structures.Struct*** List of box bounds for each timestep.

box_bounds: ***structures.Struct*** List of box bounds for the final timestep.

2.21 spline_neb

The spline_NEB module simplifies the submission of Nudged Elastic Band based, curve smoothing simulations.

- *g09_start_job()*
- *g09_results()*
- *orca_start_job()*
- *orca_results()*
- *spline_NEB*

spline_neb.**g09_results**(*spline_NEB*, *step_to_use*, *i*, *state*)

A method for reading in the output of Gaussian09 single point calculations for spline_NEB calculations. This will both (a) assign forces to the atoms stored in state and (b) return the energy and atoms.

Parameters

spline_NEB: ***spline_NEB*** A spline_NEB container holding the main spline_NEB simulation

step_to_use: ***int*** Which iteration in the spline_NEB sequence the output to be read in is on.

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

Returns

new_energy: ***float*** The energy of the system in Hartree (Ha).

new_atoms: ***list,*** ***structures.Atom*** A list of atoms with the forces attached in units of Hartree per Angstrom (Ha/Ang).

spline_neb.**g09_start_job**(*spline_NEB*, *i*, *state*, *charge*, *procs*, *queue*, *initial_guess*, *extra_section*, *mem*)

A method for submitting a single point calculation using Gaussian09 for spline_NEB calculations.

Parameters

spline_NEB: ***spline_NEB*** A spline_NEB container holding the main spline_NEB simulation

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

charge: ***int*** Charge of the system.

procs: ***int*** The number of processors to use during calculations.

queue: ***str*** Which queue to submit the simulation to (this is queueing system dependent).

initial_guess: ***str*** The name of a previous simulation for which we can read in a hessian.

extra_section: ***str*** Extra settings for this DFT method.

mem: ***int*** How many Mega Words (MW) you wish to have as dynamic memory.

Returns

g09_job: ***jobs.Job*** A job container holding the g09 simulation.

spline_neb.**orca_results**(*spline_NEB*, *step_to_use*, *i*, *state*)

A method for reading in the output of Orca single point calculations for spline_NEB calculations. This will both (a) assign forces to the atoms stored in state and (b) return the energy and atoms.

Parameters

spline_NEB: ***spline_NEB*** A spline_NEB container holding the main spline_NEB simulation

step_to_use: ***int*** Which iteration in the spline_NEB sequence the output to be read in is on.

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

Returns

new_energy: ***float*** The energy of the system in Hartree (Ha).

new_atoms: ***list,*** ***structures.Atom*** A list of atoms with the forces attached in units of Hartree per Angstrom (Ha/Ang).

spline_neb.**orca_start_job**(*spline_NEB*, *i*, *state*, *charge*, *procs*, *queue*, *initial_guess*, *extra_section*, *mem*, *priority*)

A method for submitting a single point calculation using Orca for spline_NEB calculations.

Parameters

spline_NEB: ***spline_NEB*** A spline_NEB container holding the main spline_NEB simulation

i: ***int*** The index corresponding to which image on the frame is to be simulated.

state: ***list,*** ***structures.Atom*** A list of atoms describing the image on the frame associated with index *i*.

charge: ***int*** Charge of the system.

procs: ***int*** The number of processors to use during calculations.

queue: ***str*** Which queue to submit the simulation to (this is queueing system dependent).

initial_guess: ***str*** The name of a previous simulation for which we can read in a hessian.

extra_section: ***str*** Extra settings for this DFT method.

mem: ***int*** How many MegaBytes (MB) of memory you have available per core.

Returns

orca_job: ***jobs.Job*** A job container holding the orca simulation.

class spline_neb.**spline_NEB**(*name*, *states*, *theory*, *extra_section=''*, *charge=0*, *initial_guess=None*, *spring_atoms=None*, *procs=1*, *queue=None*, *mem=2000*, *priority=None*, *disp=0*, *k_max=0.1837*, *gamma=0.2*, *fit_rigid=True*, *DFT='orca'*, *opt='LBFGS'*, *start_job=None*, *get_results=None*, *new_opt_params={}*, *callback=None*)

A post-processing method for taking a converged minimum energy pathway of a reaction using DFT and smoothing the curve to a more Gaussian-like shape. Note, this method was written for atomic orbital DFT codes; however, is potentially generalizable to other programs.

Parameters

name: ***str*** The name of the spline_NEB simulation to be run.

states: ***list, list,*** ***structures.Atom*** A list of frames, each frame being a list of atom structures. These frames represent your reaction coordinate.

theory: ***str*** The route line for your DFT simulation.

extra_section: ***str, optional*** Additional parameters for your DFT simulation.

charge: ***int*** Charge of the system.

initial_guess: ***list, str, optional*** TODO - List of strings specifying a previously run NEB simulation, allowing restart capabilities.

spring_atoms: ***list, int, optional*** Specify which atoms will be represented by virutal springs in the spline_NEB calculations. Default includes all.

procs: ***int, optional*** The number of processors for your simulation.

queue: ***str, optional*** Which queue you wish your simulation to run on (queueing system dependent). When None, spline_NEB is run locally.

mem: ***float, optional*** Specify memory constraints (specific to your X_start_job method).

disp: ***int, optional*** Specify for additional stdout information.

k_max: ***float, optional*** The maximum spring constant for your spline_NEB simulation.

gamma: ***float, optional*** The percent of the magnitude of k_max at either end of the reaction coordinate

fit_rigid: ***bool, optional*** Whether you want to use procrustes to minimize motion between adjacent frames (thus minimizing error due to excessive virtal spring forces).

DFT: ***str, optional*** Specify if you wish to use the default X_start_job and X_results functions where X is either g09 or orca.

opt: ***str, optional*** Select which optimization method you wish to use from the following: BFGS, LBFGS, SD, FIRE, QM, CG, scipy_X. Note, if using scipy_X, change X to be a valid scipy minimize method.

start_job: ***func, optional*** A function specifying how to submit your NEB single point calculations. Needed if DFT is neither orca nor g09.

get_results: ***func, optional*** A function specifying how to read your NEB single point calculations. Needed if DFT is neither orca nor g09.

new_opt_params: ***dict, optional*** Pass any additional parameters to the optimization algorithm.

callback: ***func, optional*** A function to be run after each each to calculate().

Returns

This *spline_NEB* object.

References

- Henkelman, G.; Jonsson, H. The Journal of Chemical Physics 2000, 113, 9978-9985.
- Jonsson, H.; Mills, G.; Jacobson, K. W. In Classical and Quantum Dynamics in Condensed Phase Simulations;
- Berne, B. J., Ciccotti, G., Coker, D. F., Eds.; World Scientific, 1998; Chapter 16, pp 385-404.
- Armijo, L. Pacific Journal of Mathematics 1966, 16.
- Sheppard, D.; Terrell, R.; Henkelman, G. The Journal of Chemical Physics 2008, 128.
- Henkelman, G.; Uberuaga, B. P.; Jonsson, H. Journal of Chemical Physics 2000, 113.
- Atomic Simulation Environment - https://wiki.fysik.dtu.dk/ase/
- Kolsbjerg, E. L.; Groves, M. N.; Hammer, B. The Journal of Chemical Physics 2016, 145.

align_coordinates(*r*, *B=None*, *H=None*, *return_matrix=False*)

Get a rotation matrix A that will remove rigid rotation from the new coordinates r. Further, if another vector needs rotating by the same matrix A, it should be passed in B and will be rotated. If a matrix also needs rotating, it can be passed as H and also be rotated.

Parameters

r: ***list, float*** 1D array of atomic coordinates to be rotated by procrustes matrix A.

B: ***list, list, float, optional*** A list of vectors that may also be rotated by the same matrix as *r*.

H: ***list, list, float, optional***

A matrix that should also be rotated via: H = R * H * R.T

return_matrix: ***bool, optional*** Whether to also return the rotation matrix used or not.

Returns

rotations: ***dict*** A dictionary holding 'A', the rotation matrix, 'r', the rotated new coordinates, 'B', a list of all other vectors that were rotated, and 'H', a rotated matrix.

2.22 structures

The Structures module contains various class objects to describe one's molecular system. Each *System* object can be comprised of several *Molecule* objects which are, in turn, comprised of *Atom*, *Bond*, *Angle*, *Dihedral*, and *Improper* objects.

- *System*
- *Molecule*
- *Atom*
- *Bond*
- *Angle*
- *Dihedral*
- *Improper*

There also exists a dynamic data structure object *Struct*.

- *Struct*

General atom manipulation functions that can adapted by the object classes

- `_remove_atom_index()`
- `_remove_atom_type()`

---

class structures.**Angle**(*a*, *b*, *c*, *type=None*, *theta=None*)

A structure to hold angle information.

Parameters

a: ***structures.Atom*** First atom in the angle.

b: ***structures.Atom*** Second atom in the angle.

c: ***structures.Atom*** Third atom in the angle.

type: ***dict, optional*** The forcefield type.

theta: ***float, optional*** The angle.

Returns

angle: ***structures.Angle*** The Angle class container.

class structures.**Atom**(*element*, *x*, *y*, *z*, *index=None*, *type=None*, *molecule_index=1*, *bonded=[]*, *type_index=None*)

A structure to hold atom information.

Parameters

element: ***str*** The atomic element.

x: ***float*** The x coordinate of the atom.

y: ***float*** The y coordinate of the atom.

z: ***float*** The z coordinate of the atom.

index: ***int, optional*** The atomic index within a molecule.

type: ***dict, optional*** The forcefield type.

molecule_index: ***int, optional*** Which molecule the atom is contained in.

bonded: ***list, structures.Atom, optional*** A list of atoms to which this atom is bonded.

type_index: ***int, optional*** The index of the atomic type within the given forcefield.

Returns

atom: ***structures.Atom*** The Atom class container.

flatten()
Obtain simplified position output.

Returns

pos: ***list, float*** A list holding the x, y, and z position of this atom.

set_position(*pos*)
Manually set the atomic positions by passing a tuple/list.

Parameters

pos: ***list, float or tuple, float*** A vector of 3 floats specifying the new x, y, and z coordinate.

Returns

None

translate(*v*)
Translate the atom by a vector.

Parameters

v: ***list, float*** A vector of 3 floats specifying the x, y, and z offsets to be applied.

Returns

None

class structures.**Bond**(*a*, *b*, *type=None*, *r=None*)
A structure to hold bond information.

Parameters

a: ***structures.Atom*** First atom in the bond.

b: ***structures.Atom*** Second atom in the bond.

type: ***dict, optional*** The forcefield type.

r: ***float, optional*** The bond length.

Returns

bond: ***structures.Bond*** The Bond class container.

class structures.**Dihedral**(*a*, *b*, *c*, *d*, *type=None*, *theta=None*)
A structure to hold dihedral information.

Parameters

a: ***structures.Atom*** First atom in the dihedral.

b: ***structures.Atom*** Second atom in the dihedral.

c: ***structures.Atom*** Third atom in the dihedral.

d: ***structures.Atom*** Fourth atom in the dihedral.

type: ***dict, optional*** The forcefield type.

theta: ***float, optional*** The dihedral angle.

Returns

dihedral: ***structures.Dihedral*** The Dihedral class container.

class structures.**Improper**(*a*, *b*, *c*, *d*, *type=None*, *theta=None*)
A structure to hold improper information.

Parameters

a: ***structures.Atom*** First atom in the improper.

b: ***structures.Atom*** Second atom in the improper.

c: ***structures.Atom*** Third atom in the improper.

d: ***structures.Atom*** Fourth atom in the improper.

type: ***dict, optional*** The forcefield type.

theta: ***float, optional*** The improper angle.

Returns

improper: ***structures.Improper*** The Improper class container.

class structures.**Molecule**(*atoms_or_filename*, *bonds=None*, *angles=None*, *dihedrals=None*, *parameter_file='/fs/europa/g_pc/Forcefields/OPLS/oplsaa.prm'*, *extra_parameters={}*, *test_charges=False*, *allow_errors=False*, *default_angles=None*, *test_consistency=False*, *charge=None*)
A molecule object to store atoms and any/all associated interatomic connections.

Parameters

atoms_or_filename: ***list,*** ***structures.Atom*** ***or str*** Either (a) a list of atoms or (b) a string pointing to a cml file containing the atoms.

bonds: ***list,*** ***structures.Bond*****,*** ***optional*** A list of all bonds within the system.

angles: ***list,*** ***structures.Angle*****,*** ***optional*** A list of all angles within the system.

dihedrals: ***list,*** ***structures.Dihedral*****,*** ***optional*** A list of all dihedrals within the system.

parameter_file: ***str, optional*** A path to your forcefield file. Currently only supports OPLS-AA.

extra_parameters: ***dict, optional*** Additional OPLS parameters to apply to the forcefield.

test_charges: ***bool, optional*** Bypass inconsistencies in molecular charge (False) or throw errors when inconsistencies exist (True).

allow_errors: ***bool, optional*** Permit constructions of ill-conditioned molecules, such as empty bonds (True), or throw errors (False).

default_angles: ***dict, optional*** A default forcefield angle type to be set if angle types are set to None.

test_consistency: ***bool, optional*** Whether to validate the input cml file against OPLS.

charge: ***float, optional*** The total charge of this molecule.

Returns

molecule: ***structures.Molecule*** The Molecule class container.

flatten()
Flatten out all atoms into a 1D array.

Returns

atoms: ***list, float*** A 1D array of atomic positions.

get_center_of_geometry(*skip_H=True*)
Calculate the center of geometry of the molecule.

Parameters

skip_H: ***bool, optional*** Whether to include Hydrogens in the calculation (False), or not (True).

Returns

cog: ***tuple, float*** A tuple of the x, y, and z coordinate of the center of geometry.

get_center_of_mass()
Calculate the center of mass of the molecule.

Returns

com: ***list, float*** A list of the x, y, and z coordinate of the center of mass.

merge(*other*)
This function merges another molecule into this one, offsetting indices as needed.

perturbate(*dx=0.1*, *dr=5*, *center_of_geometry=True*, *rotate=True*)
Randomly perturbate atomic coordinates, and apply a slight rotation.

Parameters

dx: ***float, optional*** By how much you are willing to perturbate via translation.

dr: ***float, optional*** By how much you are willing to perturbate via rotation in degrees.

center_of_geometry: ***bool, optional*** Whether to do the random rotation by the center of geometry (True) or mass (False). Note, if types are not set, it will fail in the case of center of mass.

rotate: ***bool, optional*** Whether to randomly rotate the molecule or not.

Returns

None

rand_rotate(*in_place=True*, *limit_angle=None*, *center_of_geometry=False*)
Randomly rotate a molecule.

Parameters

in_place: ***bool, optional*** Whether to rotate randomly (False), or around the molecule's center of mass (True).

limit_angle: ***float, optional*** Whether to confine your random rotation (in radians).

center_of_geometry: ***bool, optional*** Whether to rotate around the center of geometry (True) or mass (False).

Returns

None

remove_atom_index(*indices=[]*, *verbose=False*)
Removes selected indices from system. Does so by compiling new lists for atoms, bonds, angles, and dihedrals. Will be faster than Remove in cases where you are only keeping a few atoms.

Parameters

type_indices: ***list, int, optional*** A list of OPLS types.

Returns

None

remove_atom_type(*type_indices=[]*, *verbose=False*)

Removes selected OPLS types from system. Does so by compiling new lists for atoms, bonds, angles, and dihedrals. Will be faster than Remove in cases where you are only keeping a few atoms.

Parameters

type_indices: ***list, int, optional*** A list of OPLS types.

Returns

None

rotate(*m*)

Rotate the molecule by the given matrix *m*.

Parameters

m: ***list, list, float*** A 3x3 matrix describing the rotation to be applied to this molecule.

Returns

None

set_center(*xyz=[0.0, 0.0, 0.0]*)

Recenter the molecule to the origin.

Parameters

xyz: ***list, float, optional*** A list of x, y, and z offsets to be applied post centering.

Returns

None

set_positions(*positions*, *new_atom_list=False*)

Manually specify atomic positions of your molecule.

Parameters

positions: ***list, float*** A list, either 2D or 1D, of the atomic positions. Note, this should be in the same order that the atoms are stored in.

new_atom_list: ***bool, optional*** Whether to generate an entirely new atom list (True) or re-write atom positions of those atoms already stored (False). Note, if a new list is written, connections (bonds, angles, . . .) are not changed.

Returns

None

set_types(*P*)

This will, using a parameter object, assign a pointer of which type corresponds with each atom, bond, angle, and dihedral.

Parameters

P: **squid.ff_params.Parameters** A general parameter object

translate(*v*)
Apply a translation to this molecule.

Parameters

v: ***list, float*** A list of 3 elements: the x, y, and z translations to be applied.

Returns

None

class structures.**Struct**(***kwargs*)
A generalized Structure object for python

class structures.**System**(*name=None*, *box_size=(10.0*, *10.0*, *10.0)*, *box_angles=(90.0*, *90.0*, *90.0)*, *periodic=False*)
A system object to store molecules for one's simulations.

Parameters

name: ***str, optional*** System Name.

box_size: ***tuple, float, optional*** System x, y, and z lengths.

box_angles: ***tuple, float, optional*** System xy, yz, and xz angles.

periodic: ***bool, optional*** Whether to have periodic boundaries on or off.

Returns

system: ***structures.System*** The System class container.

Contains(*molecule*)
Check if this system contains a molecule, based on the atoms, bonds, angles and dihedrals.

Parameters

molecule: ***structures.Molecule*** A molecule to be checked if it resides within this system.

Returns

is_contained: ***bool*** A boolean specifying if the molecule passed to this function is contained within this System object. This implies that all atoms, bonds, angles, and dihedrals within the molecule are present in a molecule within the system.

Remove(*target*)
If target is a molecule, removes all atoms, bonds angles and dihedrals of the passed molecule from the system. Raises a ValueError if not all aspects of molecule are found in the system.

If target is an Atom, the atom is removed from the system, and any bonds, angles, and dihedrals which contain the atom are also removed from the system. Raises a ValueError if the Atom is not found in the system.

Parameters

target: ***structures.Atom* or *structures.Molecule*** A target atom or molecule to be removed from this system. Target is a valid *structures.Molecule* instance or a valid *structures.Atom* instance.

Returns

None

add(*molecule*, *x=0.0*, *y=0.0*, *z=0.0*, *scale_x=1.0*, *scale_y=1.0*, *scale_z=1.0*)
A function to add a molecule to this system.

Parameters

molecule: ***structures.Molecule*** A Molecule structure.

x: ***float, optional*** x offset to atomic positions of the input molecule.

y: ***float, optional*** y offset to atomic positions of the input molecule.

z: ***float, optional*** z offset to atomic positions of the input molecule.

scale_x: ***float, optional*** scalar to offset x coordinates of the input molecule.

scale_y: ***float, optional*** scalar to offset y coordinates of the input molecule.

scale_z: ***float, optional*** scalar to offset z coordinates of the input molecule.

Returns

None

assign_molecule_index(*atom_list*, *i_list_index*, *molecule_count*, *elements=[]*)

Add molecule index to the atom if it has not already been assigned. Then recursively pass bonded atoms to the function

Parameters

atom_list: ***list*, *structures.Molecule*** A list of atoms

i_list_index: ***int*** The index of the atom currently being assigned. Refers to atom_list index.

molecule_count: ***int*** The index of the molecule to be assigned.

elements: ***list, list, int, optional*** A list of OPLS types. Will be skipped during this part of the molecule assignment sequence.

Returns

None

assign_molecules(*elements=[]*)

Assigns a unique molecule index for each molecule and sets each atom.molecule_index to the appropriate index. The algorithm runs by recursively searching through every bonded atom and giving the same molecule index as the origin atom. Normally, this means that molecules that are not bonded to each other will have a unique molecule index. Specific elements can be assign a predetermined molecule index by passing a list of element lists.

For example element_groups=[[6,8], [9]] will give all carbon and oxygen atoms a molecule index of 1 and fluorine atoms will have a molecule index of 2.

It is possible to get different molecule indices within the same bonded compound by giving a specific "bridging" element a predetermined molecule index. This prevents the molecular index from spreading to the entire bonded structure.

Parameters

elements: ***list, list, int, optional*** A list of OPLS types.

Returns

None

get_elements(*params=None*)

This simplifies using dump_modify by getting a list of the elements in this system, sorted by their weight. Note, duplicates will exist if different atom types exist within this system!

Returns

elements: ***list, str*** A list of the elements, sorted appropriately for something like dump_modify.

packmol(*molecules*, *molecule_ratio=(1,)*, *new_method=False*, *density=1.0*, *seed=1*, *persist=True*, *number=None*, *additional=''*, *custom=None*, *extra_block_at_end=''*, *extra_block_at_beginning=''*, *tolerance=2.0*)

Given a list of molecules, pack this system appropriately. Note, we now will pack around what is already within the system! This is done by first generating a packmol block for the system at hand, followed by a block for the solvent.

A custom script is also allowed; however, if this path is chosen, then ensure all file paths for packmol exist. We change directories within this function to a sys_packmol folder, where all files are expected to reside.

Parameters

molecules: ***list,*** **structures.Molecule** Molecules to be added to this system.

molecule_ratio: ***tuple, float, optional*** The ration that each molecule in *molecules* will be added to the system.

density: ***float, optional*** The density of the system in g/mL

seed: ***float, optional*** Seed for random generator.

persist: ***bool, optional*** Whether to maintain the generated sys_packmol directory or not.

number: ***int or list, int, optional*** Overide density and specify the exact number of molecules to pack. When using a list of molecules, you must specify each in order within a list.

custom: ***str, optional*** A custom packmol script to run for the given input molecules. Note, you should ensure all necessary files are within the sys_packmol folder if using this option.

additional: ***str, optional*** Whether to add additional constraints to the standard packmol setup.

extra_block_at_beginning: ***str, optional*** An additional block to put prior to the standard block.

extra_block_at_end: ***str, optional*** An additional block to put after the standard block.

tolerance: ***float, optional*** The tolerance around which we allow atomic overlap/proximity.

Returns

None

References

- Packmol - http://www.ime.unicamp.br/~martinez/packmol/home.shtml

remove_atom_index(*indices=[]*, *verbose=False*)

Removes selected indices from system. Does so by compiling new lists for atoms, bonds, angles, and dihedrals. Will be faster than Remove in cases where you are only keeping a few atoms.

Parameters

type_indices: ***list, int, optional*** A list of OPLS types.

Returns

None

remove_atom_type(*type_indices=[]*, *verbose=False*)

Removes selected OPLS types from system. Does so by compiling new lists for atoms, bonds, angles, and dihedrals. Will be faster than Remove in cases where you are only keeping a few atoms.

Parameters

type_indices: ***list, int, optional*** A list of OPLS types.

Returns

None

setTriclinicBox(*periodic*, *box_size*, *box_angles*)
A function to establish a triclinic box boundary condition for this system.

Parameters

periodic: ***bool, optional*** Whether to have periodic boundaries on or off. Initial guess.

box_size: ***tuple, float, optional*** System x, y, and z lengths.

box_angles: ***tuple, float, optional*** System xy, yz, and xz angles.

Returns

None

set_types(*params=None*)
Given the atoms, bonds, angles, and dihedrals in a system object, generate a list of the unique atom, bond, angle, dihedral types and assign that to the system object.

Parameters

params: **squid.ff_params.Parameters, optional** A parameters object holding all the possible parameters.

2.23 units

The units package holds various functions that aid in unit conversion, as well as periodic table data management.

- *convert_energy()*
- *convert_pressure()*
- *convert_dist()*
- *elem_i2s()*
- *elem_s2i()*
- *elem_weight()*
- *elem_sym_from_weight()*
- *convert()*

---

units.**convert**(*old*, *new*, *val*)
A generic converter of fractional units. This works only for one unit in the numerator and denomenator (such as Ha/Ang to eV/Bohr).

Parameters

old: ***str*** Units for which val is in.

new: ***str*** Units to convert to.

val: ***float*** Value to convert.

Returns

new_val: ***float*** Converted value in units of new.

units.**convert_dist**(*d0*, *d1*, *d_val*)
Convert distance units.

Parameters

d0: ***str*** Unit of distance that d_val is in.

d1: ***str*** Unit of distance that you wish to convert to.

d_val: ***float*** Value to be converted.

Returns

distance: ***float*** Converted d_val to units of d1.

units.**convert_energy**(*e0*, *e1*, *e_val*)
Convert energy units.

Parameters

e0: ***str*** Unit of energy that e_val is in.

e1: ***str*** Unit of energy that you wish to convert to.

e_val: ***float*** Value to be converted.

Returns

energy: ***float*** Converted e_val to units of e1.

units.**convert_pressure**(*p0*, *p1*, *p_val*)
Convert pressure units.

Parameters

p0: ***str*** Unit of pressure that p_val is in.

p1: ***str*** Unit of pressure that you wish to convert to.

p_val: ***float*** Value to be converted.

Returns

pressure: ***float*** Converted p_val to units of p1.

units.**elem_i2s**(*elem_int*)
Get the elemental symbol, given its atomic number.

Parameters

elem_int: ***int*** Atomic number of an element.

Returns

elem_sym: ***str*** Elemental symbol.

units.**elem_s2i**(*elem_sym*)
Get the atomic number, given its elemental symbol.

Parameters

elem_sym: ***str*** Elemental symbol of an element.

Returns

elem_int: ***int*** Atomic number.

units.**elem_sym_from_weight**(*weight*, *delta=0.1*)
Get the element that best matches the given weight (in AMU).

Parameters

weight: ***float*** Weight of an element in AMU.

delta: ***float, optional*** How close you permit the matching to be in AMU.

Returns

elem_sym: ***str*** The elemental symbol.

units.**elem_weight**(*elem*)
Get the weight of an element, given its symbol or atomic number.

Parameters

elem: ***str or int*** Elemental symbol or atomic number.

Returns

elem_weight: ***float*** Weight of the element in AMU.

2.24 utils

The utils module was one of the crucial modules used in the original Squid code. However, due to confusion in what functions belonged where, it eventually became cluttered. Thus, it has now been deprecated. You can find the following functions in their new modules:

The following have been placed in the *structures* module:

- Struct
- System
- Molecule
- Atom
- Bond
- Dihedral

The following have been placed in the *results* module:

- DFT_out
- sim_out

The following have been placed in the *geometry* module:

- angle_size
- dihedral_angle
- get_bonds
- get_angles_and_dihedrals
- orthogonal_procrustes
- procrustes
- interpolate
- motion_per_frame

- dist_squared
- dist
- rotation_matrix
- rotate_xyz
- rotate_frames
- rand_rotation
- mvee
- align_centroid
- center_frames
- pretty_xyz -> smooth_xyz (note the changed function name)

The following have been placed in the *frc_opls* module:

- opls_options

The following have been placed in the *print_helper* module:

- color_set
- colour_set
- strip_color
- strip_colour
- spaced_print

The following have been placed in the *linux_helper* module:

- clean_up_folder

The following have been placed in the *jobs* module:

- Job

The following have been placed in the *debyer* module:

- get_pdf

2.25 visualization

The visualization module automates some visualization procedures for post processing data.

An example of using this is as follows:

```
import files
import visualization as vis

vis.ovito_xyz_to_gif(files.read_xyz("CNH_HCN.xyz"), "/fs/home/hch54/tmp", renderer=
↪'Tachyon')
```

- *ovito_xyz_to_image()*
- *ovito_xyz_to_gif()*

visualization.**ovito_xyz_to_gif**(*frames*, *scratch*, *fname='image'*, *camera_pos=(10, 0, 0)*, *camera_dir=(-1, 0, 0)*, *size=(800, 600)*, *delay=10*, *renderer='OpenGLRenderer'*, *renderer_settings={}*, *overwrite=False*)

This function will, using the ovito python api, generate either a single image or a gif of the input frames. Note, a gif is only generated when more than one frame exists.

Parameters

frames: ***str*** **or** ***list,*** ***structures.Atom*** A list of frames you wish to generate an image for, or a path to an xyz file.

scratch: ***str*** A directory you want to have each image saved to.

fname: ***str, optional*** The prefix for the image names.

camera_pos: ***tuple, float, optional*** A tuple of x, y, and z coordinates for the camera to be positioned.

camera_dir: ***tuple, float, optional*** The direction the camera is facing.

size: ***tuple, int, optional*** Image size (width, height).

delay: ***int, optional*** In the event of a gif, how long it should play for.

renderer: ***str, optional*** What kind of renderer you wish to use: OpenGL or Tachyon.

renderer_settings: ***dict, optional*** Here you can change specific renderer settings.

overwrite: ***bool, optional*** Whether to delete any files already existing in the scratch dir.

Returns

None

visualization.**ovito_xyz_to_image**(*xyz*, *scratch*, *fname='image'*, *camera_pos=(10, 0, 0)*, *camera_dir=(-1, 0, 0)*, *size=(800, 600)*, *renderer='OpenGLRenderer'*, *renderer_settings={}*)

This function will, using the ovito python api, generate a png image of an xyz file.

Parameters

xyz: ***str*** A path to an xyz file.

scratch: ***str*** A directory you want to have each image saved to.

fname: ***str, optional*** The prefix for the image names.

camera_pos: ***tuple, float, optional*** A tuple of x, y, and z coordinates for the camera to be positioned.

camera_dir: ***tuple, float, optional*** The direction the camera is facing.

size: ***tuple, int, optional*** Image size (width, height).

delay: ***int, optional*** In the event of a gif, how long it should play for.

renderer: ***str, optional*** What kind of renderer you wish to use: OpenGL or Tachyon.

renderer_settings: ***dict, optional*** Here you can change specific renderer settings.

Returns

None

2.26 vmd

The vmd package automates various vmd post-processing tasks.

- *plot_MO_from_cube()*

---

vmd.**plot_MO_from_cube**(*fptrs*, *wireframe=True*, *hide=True*, *iso=0.04*)
A function to generate a VMD visualization of a molecular orbital from a cube file.

Parameters

fptrs: ***list, str, or str*** Strings giving the path to the cube file.

wireframe: ***bool, optional*** If you want to view wireframe (True) or not (False) for the orbitals.

hide: ***bool, optional*** Whether to hide the representations (True) on startup, or not (False).

iso: ***float, optional*** Isosurface magnitude. Set to 0.04 by default, but 0.01 may be better.

Returns

None

vmd.**plot_electrostatic_from_cube**(*fptr_rho*, *fptr_pot*, *wireframe=True*)
A function to generate a VMD visualization of a electrostatic potential mapped onto an electron density isosurface.

Parameters

fptr_rho: ***str*** Path to the electron density cube file.

fptr_pot: ***str*** Path to the electrostatic potential cube file.

wireframe: ***bool, optional*** If you want to view wireframe (True) or not (False) for the orbitals.

Returns

None

CHAPTER

THREE

EXAMPLES

Here we supply some example uses of the Squid codebase. These examples can all be found and run in the git repository.

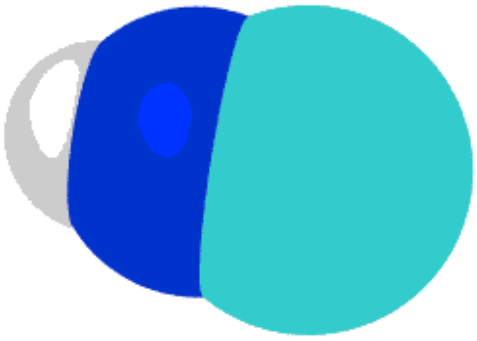
3.1 Geometry - Smoothing out a Reaction Coordinate

The below code shows how, given a folder of steps in a reaction coordinate, we can smooth out the full reaction.

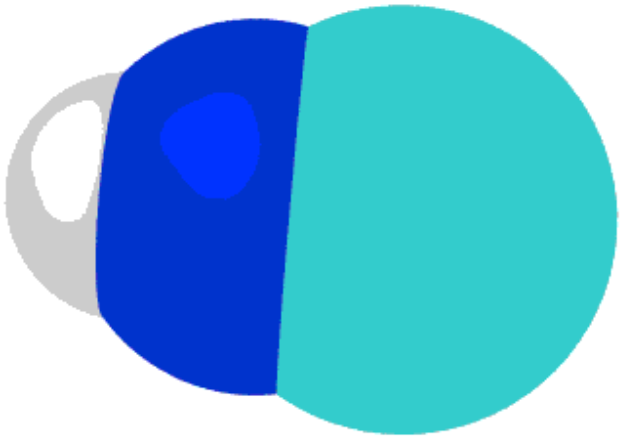
```
# System imports
import os
import copy
# Squid imports
from squid import files
from squid import geometry

# First we want to read in the manually made iterations
fptrs = [int(f.split(".xyz")[0]) for f in os.listdir("reaction_coordinate")]
fptrs.sort()

# Now, we loop through all files in numerical order and append to our reaction␣
↪coordinate
rxn = []
for f in fptrs:
        rxn.append(files.read_xyz("reaction_coordinate/%d.xyz" % f))

# Save an example of this rough reaction we made
files.write_xyz(rxn, "reaction_coordinate_rough")

# Now, we smooth it out.  There are many ways of doing so.  We'll only show the main␣
↪two methods here
# Here we just make a copy of the frames for the second method
held_rough_reaction = copy.deepcopy(rxn)

# Method 1 - Procrustes to minimize rotations and translations between consecutive␣
↪frames
geometry.procrustes(rxn)
files.write_xyz(rxn, "reaction_coordinate_procrustes")

# Method 2 - Procrustes plus linear interpolation
# Note, R_MAX is the maximum average change in atomic positions between adjacent␣
↪frames (in angstroms)
#       F_MAX is the maximum number of frames we want in the final reaction coordinate
```

```
rxn = copy.deepcopy(held_rough_reaction) # Grab the previously rough reaction
geometry.smooth_xyz(rxn, R_MAX=0.1, F_MAX=50, PROCRUSTES=True, outName="reaction_
↪coordinate_smooth", write_xyz=True)
```

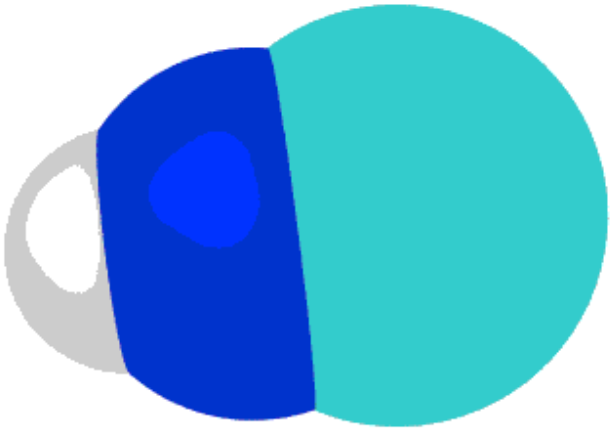
The rough reaction coordinate initially appears smooth, but as we reach the final frame we see that there is clearly a change in reference frame. This is due to the fact that when optimizing in DFT we usually get some residual rotations due to switching between internal coordinates and cartesian coordinates.



However, when using the procrustes method we remove the rigid rotation associated with this change of coordinate system.



Finally, with the added linear interpolations we end up with a smooth reaction coordinate.



3.2 DFT - Geometry Optimization of Acetic Acid

The below code shows how to use Orca to optimize the geometry of an acetic acid dimer.

```
from squid import orca
from squid import files

# Read in the xyz file
frames = files.read_xyz("acetic_acid_dimer.xyz")
# Run a simulation locally using the Hartree Fock method (with 3 corrections)
orca.job("aa_dimer_local","! HF-3c Opt",atoms=frames,queue=None)
```

3.3 DFT - Molecular Orbitals Post Processing

The below code shows how to use g09 and vmd to generate and display molecular orbitals of a DFT simulation. Note, this uses g09's cubegen and formchk code.

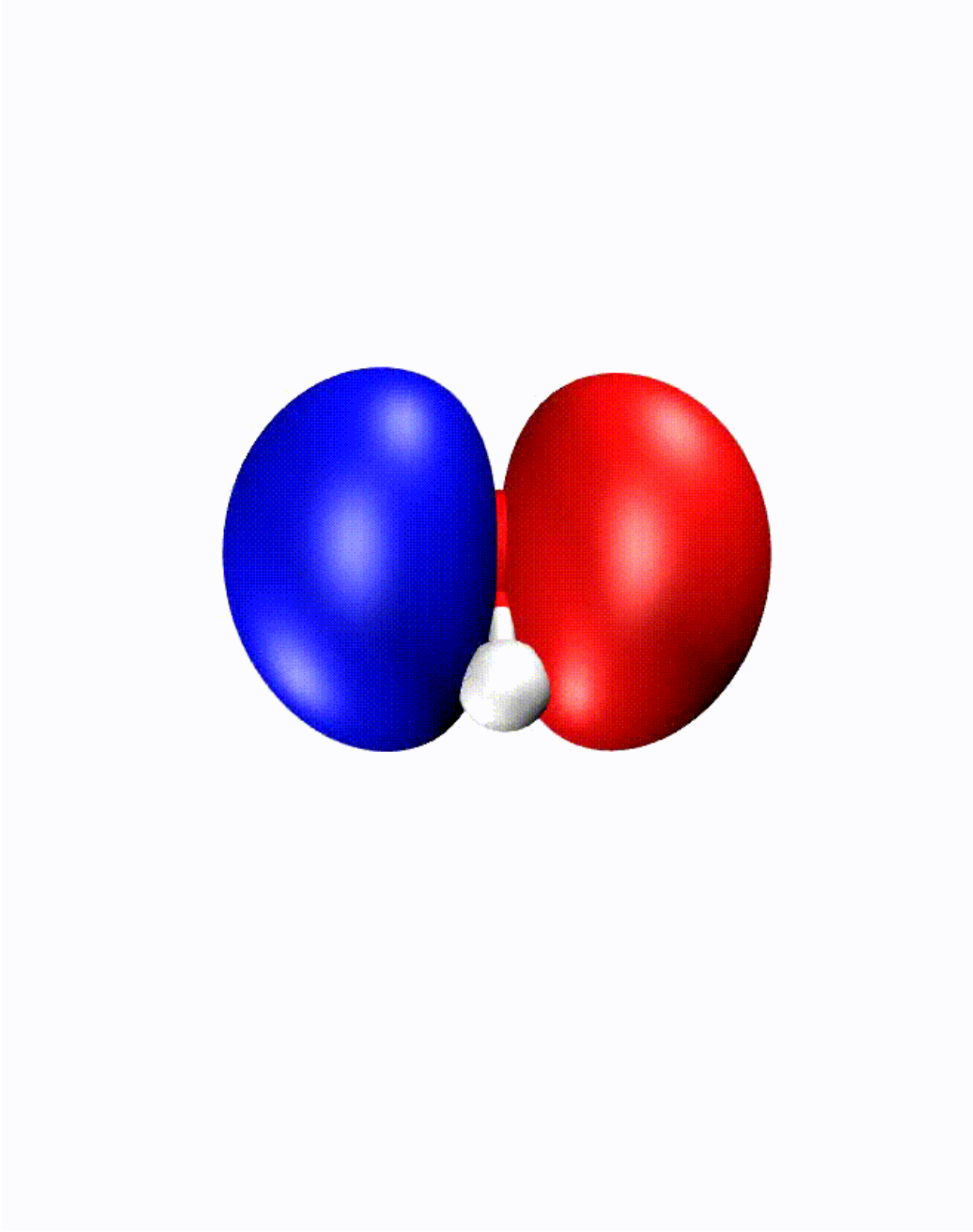
```
from squid import g09
from squid import files
```

```
# Run water simulation
def opt_water():
        frames = files.read_xyz('water.xyz')
        return g09.job('water',
                                    'HSEH1PBE/cc-pVTZ OPT=() SCRF(Solvent=Toluene)',
                                    atoms=frames,
                                    queue=None,
                                    force=True)


job = opt_water()
job.wait()
g09.cubegen_analysis("water", orbital=3)
```

This will optimize the geometry of a water molecule and then automatically generate a VMD session with various representations. In the console output it'll show the following in blue:

```
Representations are as follows:

        1 - CPK of atoms
        2 - LUMO Positive
        3 - HOMO Positive
        4 - LUMO Negative
        5 - HOMO Negative
        6 - Potential Surface
        7 - MO 3
```

Choosing only displays 1, 3, and 5 we can see the HOMO level of water as follows (positive being blue and negative being red):



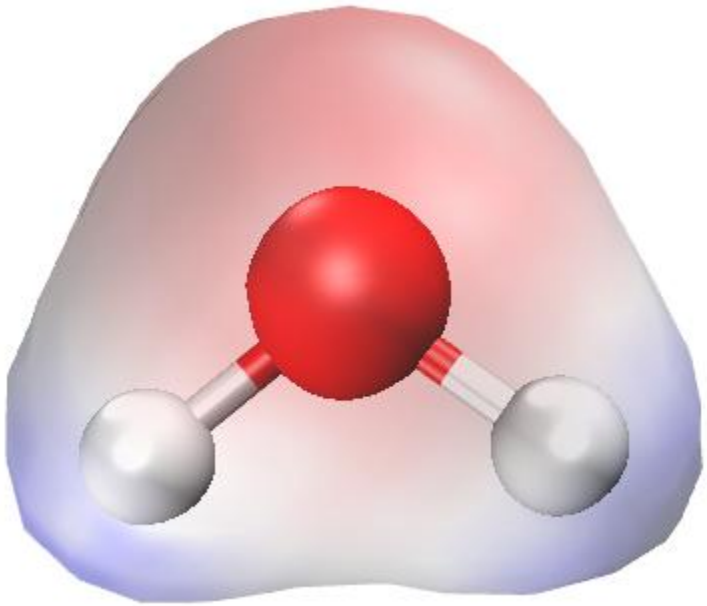
Recent updates now allow this for orca as well. NOTE! By default Orca does not take into account degenerate energy states when populating. To do so, ensure the following is in your extra_section before trying the visualization:

```
%scf FracOcc true end
```

```
from squid import orca
from squid import structures

ROUTE_OPT = '! B97-D3 def2-TZVP OPT'
EXTRA_SECTION = ''

frames = [structures.Atom("O", -0.730404, 2.443498, 0.004930),
          structures.Atom("H", 0.227213, 2.402054, -0.008942),
          structures.Atom("H", -1.008399, 1.573518, -0.286267)]

j = orca.job("water", ROUTE_OPT,
             atoms=frames,
             extra_section=EXTRA_SECTION,
             queue=None, procs=1, mem=1000)

j.wait()

orca.mo_analysis("water",
                 orbital=[0, 1, 2, 3],
                 HOMO=True,
                 LUMO=True,
                 wireframe=True)
```

3.4 DFT - Electrostatic Potential Mapped on Electron Density Post Processing

```
from squid import orca
from squid import structures

ROUTE_OPT = '! B97-D3 def2-TZVP OPT'
EXTRA_SECTION = ''

frames = [structures.Atom("O", -0.730404, 2.443498, 0.004930),
          structures.Atom("H", 0.227213, 2.402054, -0.008942),
          structures.Atom("H", -1.008399, 1.573518, -0.286267)]

j = orca.job("water", ROUTE_OPT,
             atoms=frames,
             extra_section=EXTRA_SECTION,
             queue=None, procs=1, mem=1000)

j.wait()

orca.pot_analysis("water", wireframe=True, npoints=80)
```



3.5 DFT - Nudged Elastic Band of CNH Isomerization

The below code shows how to use the Nudged Elastic Band method (NEB) to optimize for the minimum energy pathway. Note, this is a rough example, and in reality one would make sure to optimize both endpoints of *frames* at the same level of theory and then to proceed with the NEB simulation.

```
from squid import neb
from squid import files

frames = files.read_xyz("CNH_HCN.xyz")
new_opt_params = {'step_size': 0.1,
                                  'step_size_adjustment': 0.5,
                                  'max_step': 0.2,
                                  'linesearch': 'backtrack',
                                  'accelerate': True,
                                  'reset_step_size': 5}
optimizer = neb.NEB("neb_test",
                                        frames,
                                        "! HF-3c",
                                        opt="LBFGS",
                                        new_opt_params=new_opt_params)
optimizer.optimize()
```

Example output is as follows:

```
--------------------------------------------------------------------------------
↪----
Run_Name = neb_test
DFT Package = orca
Spring Constant for NEB: 0.1837 Ha/Ang = 4.99928 eV/Ang

Running neb with optimization method LBFGS
        step_size = 0.1
```

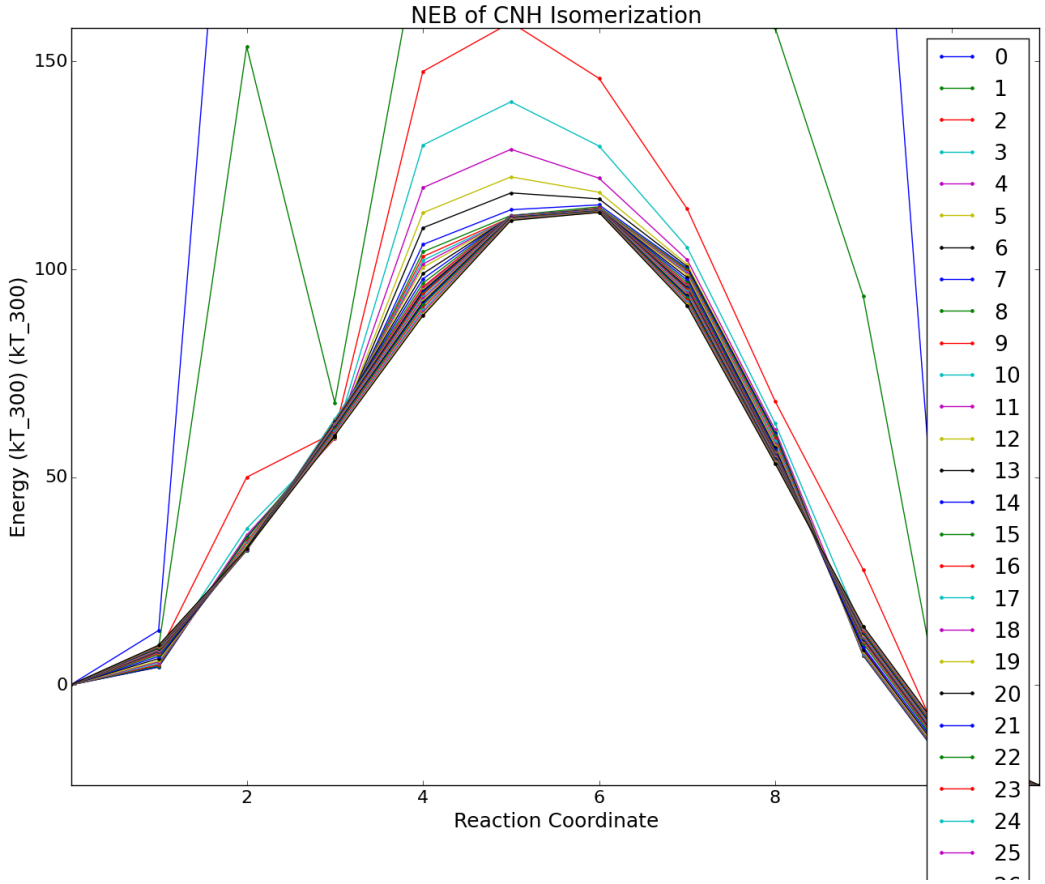
```
        step_size_adjustment = 0.5
        Linesearch method used is backtrack
        Will reset stored parameters and gradients when stepped bad.
        Will reset step_size after 5 good steps.
        Will accelerate step_size after 5 good steps.
        Will use procrustes to remove rigid rotations and translations
Convergence Criteria:
        g_rms = 0.001 (Ha/Ang) = 0.0272144 (eV/Ang)
        g_max = 0.001 (Ha/Ang) = 0.0272144 (eV/Ang)
        maxiter = 1000
-----------------------------------------------
Step    RMS_F (eV/Ang)  MAX_F (eV/Ang)  MAX_E (kT_300)  MAX Translational Force (eV/
↪Ang)        Energies (kT_300)
----
0       53.2607         95.8912         733.9           0.0000
↪           -92.232 +  13.1 273.4 269.5 695.5 733.9 693.2 610.5 384.1 262.4 -17.0 -24.
↪2
1       13.2722         31.0298         192.3           0.0000
↪           -92.232 +   9.0 153.6  67.9 184.2 192.3 177.8 167.8 158.1  93.5 -19.4 -24.
↪2
2       5.6462          9.9857          159.4           0.0000
↪           -92.232 +   7.6  50.0  60.8 147.6 159.4 145.9 114.6  68.2  27.7 -19.4 -24.
↪2
3       3.4829          6.7076          140.3           0.0000
↪           -92.232 +   5.7  37.6  59.4 129.9 140.3 129.7 105.3  62.9  12.6 -20.0 -24.
↪2
4       2.4373          5.1388          128.9           0.0000
↪           -92.232 +   4.8  36.1  59.2 119.6 128.9 121.9 102.3  61.3   8.9 -20.6 -24.
↪2
5       1.7959          4.0514          122.2           0.0000
↪           -92.232 +   4.4  35.7  59.4 113.6 122.2 118.6 101.2  60.8   7.6 -20.9 -24.
↪2
6       1.3715          3.2678          118.4           0.0000
↪           -92.232 +   4.3  35.5  59.7 110.0 118.4 117.0 100.7  60.6   7.2 -21.1 -24.
↪2
7       0.8475          2.0949          115.5           0.0000
↪           -92.232 +   4.4  35.5  60.5 106.0 114.3 115.5 100.3  60.4   7.0 -21.1 -24.
↪2
8       0.6027          1.3783          115.0           0.0000
↪           -92.232 +   4.6  35.2  61.1 104.2 113.0 115.0  99.9  60.0   7.1 -21.0 -24.
↪2
9       0.4495          0.9335          114.8           0.0000
↪           -92.232 +   4.9  34.5  61.7 103.1 112.5 114.8  99.6  59.2   7.3 -20.8 -24.
↪2
10      0.3571          0.799           114.7           0.0000
↪           -92.232 +   5.2  34.0  62.1 102.1 112.3 114.7  99.3  58.6   7.5 -20.5 -24.
↪2
11      0.2806          0.6794          114.7           0.0000
↪           -92.232 +   5.5  33.7  62.4 101.3 112.4 114.7  99.0  58.0   7.8 -20.3 -24.
↪2
12      0.2343          0.5628          114.6           0.0000
↪           -92.232 +   5.8  33.5  62.7 100.5 112.4 114.6  98.7  57.6   8.0 -20.0 -24.
↪2
13      0.1914          0.3947          114.6           0.0000
↪           -92.232 +   6.3  33.0  63.2  98.9 112.7 114.6  98.0  57.0   8.4 -19.5 -24.
↪2
14      0.1686          0.3585          114.6           0.0000
↪           -92.232 +   6.8  32.7  63.5  97.7 112.8 114.6  97.4  56.5   9.1 -19.1 -24.
↪2
```

```
15      0.1405          0.2894          114.6           0.0000
↪           -92.232 +   7.3  32.5  63.6  96.6 112.9 114.6  96.8  56.1   9.9 -18.6 -24.
↪2
16      0.1261          0.2586          114.6           0.0000
↪           -92.232 +   7.7  32.4  63.7  95.8 112.9 114.6  96.3  55.8  10.5 -18.2 -24.
↪2
17      0.1447          0.4157          114.6           0.0000
↪           -92.232 +   7.9  32.4  64.0  95.0 113.0 114.6  95.8  55.5  10.8 -17.7 -24.
↪2
18      0.1113          0.2416          114.6           0.0000
↪           -92.232 +   7.8  32.4  63.6  95.4 112.9 114.6  96.1  55.7  10.6 -18.0 -24.
↪2
19      0.1099          0.2155          114.5           0.0000
↪           -92.232 +   7.9  32.4  63.5  95.1 112.9 114.5  95.9  55.5  10.8 -17.8 -24.
↪2
20      0.1017          0.1878          114.5           0.0000
↪           -92.232 +   8.0  32.4  63.3  94.8 112.9 114.5  95.6  55.4  11.0 -17.6 -24.
↪2
21      0.0941          0.1725          114.4           0.0000
↪           -92.232 +   8.1  32.4  63.1  94.5 112.8 114.4  95.4  55.3  11.1 -17.5 -24.
↪2
22      0.0882          0.1662          114.4           0.0000
↪           -92.232 +   8.2  32.5  62.9  94.2 112.8 114.4  95.2  55.2  11.3 -17.3 -24.
↪2
23      0.0833          0.1592          114.3           0.0000
↪           -92.232 +   8.3  32.5  62.8  93.9 112.8 114.3  95.0  55.1  11.4 -17.2 -24.
↪2
24      0.0753          0.1452          114.3           0.0000
↪           -92.232 +   8.4  32.5  62.6  93.4 112.7 114.3  94.7  54.9  11.7 -16.9 -24.
↪2
25      0.0693          0.1317          114.2           0.0000
↪           -92.232 +   8.6  32.5  62.4  92.9 112.6 114.2  94.3  54.7  12.0 -16.6 -24.
↪2
26      0.0641          0.1202          114.1           0.0000
↪           -92.232 +   8.7  32.5  62.2  92.5 112.6 114.1  94.0  54.5  12.2 -16.4 -24.
↪2
27      0.0593          0.1102          114.1           0.0000
↪           -92.232 +   8.8  32.5  62.0  92.1 112.5 114.1  93.7  54.4  12.4 -16.2 -24.
↪2
28      0.055           0.1016          114.0           0.0000
↪           -92.232 +   8.9  32.6  61.8  91.7 112.4 114.0  93.4  54.2  12.6 -16.0 -24.
↪2
29      0.051           0.094           114.0           0.0000
↪           -92.232 +   9.0  32.6  61.6  91.3 112.4 114.0  93.1  54.1  12.8 -15.8 -24.
↪2
30      0.044           0.0809          113.9           0.0000
↪           -92.232 +   9.2  32.6  61.3  90.7 112.2 113.9  92.6  53.9  13.2 -15.4 -24.
↪2
31      0.0382          0.0711          113.8           0.0000
↪           -92.232 +   9.3  32.6  60.9  90.2 112.1 113.8  92.2  53.7  13.5 -15.1 -24.
↪2
32      0.0334          0.0633          113.8           0.0000
↪           -92.232 +   9.4  32.6  60.6  89.7 112.0 113.8  91.9  53.5  13.7 -14.9 -24.
↪2
33      0.0298          0.0565          113.7           0.0000
↪           -92.232 +   9.5  32.7  60.3  89.2 111.9 113.7  91.6  53.4  13.9 -14.7 -24.
↪2
34      0.0271          0.0509          113.7           0.0000
↪           -92.232 +   9.5  32.8  60.0  88.8 111.8 113.7  91.3  53.2  14.1 -14.5 -24.
↪2
```

```
NEB converged the RMS force.
------------------------------------------------------------------------------------------
↪----
```

With the following graph made using:

```
scanDFT neb_test-^-%d 1 10 -neb neb_test-0-0,neb_test-0-11 -c ^,0,34 -t "NEB of CNH␣
↪Isomerization" -lx "Reaction Coordinate" -ly "Energy (kT_300)" -u kT_300
```



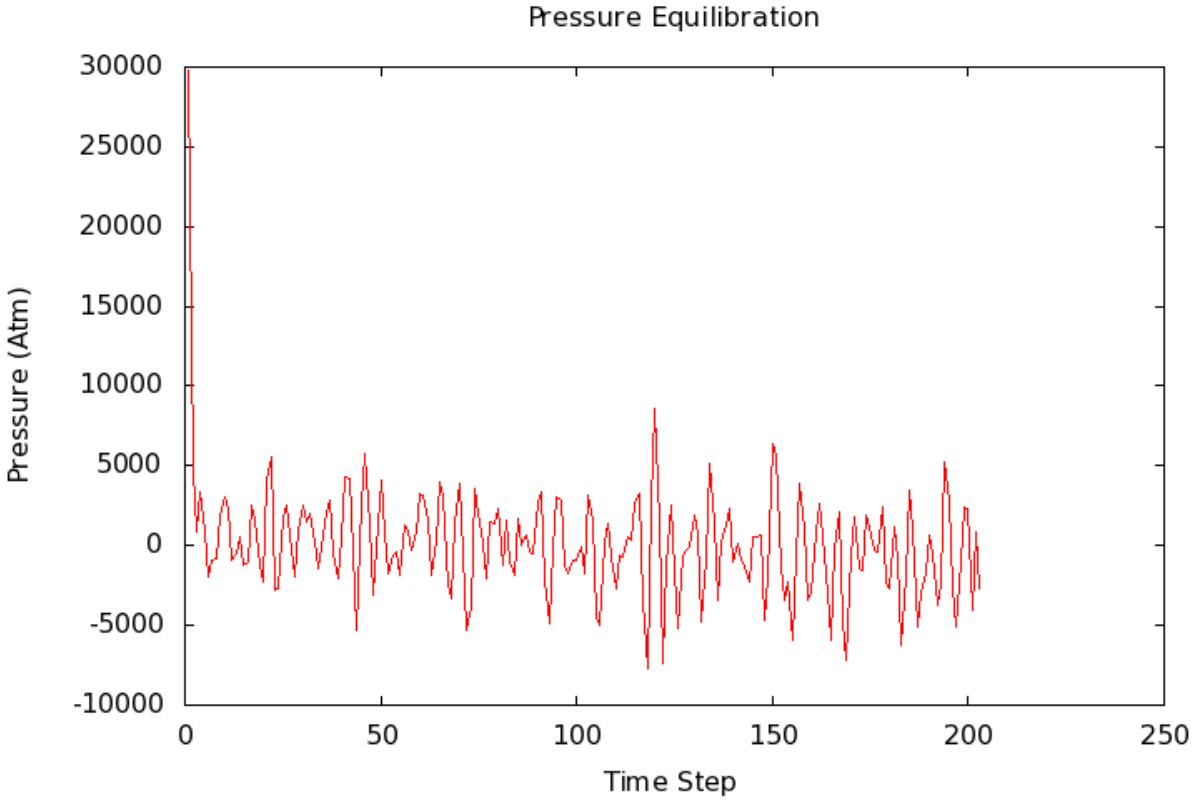
3.6 MD - Equilibration of Solvent Box

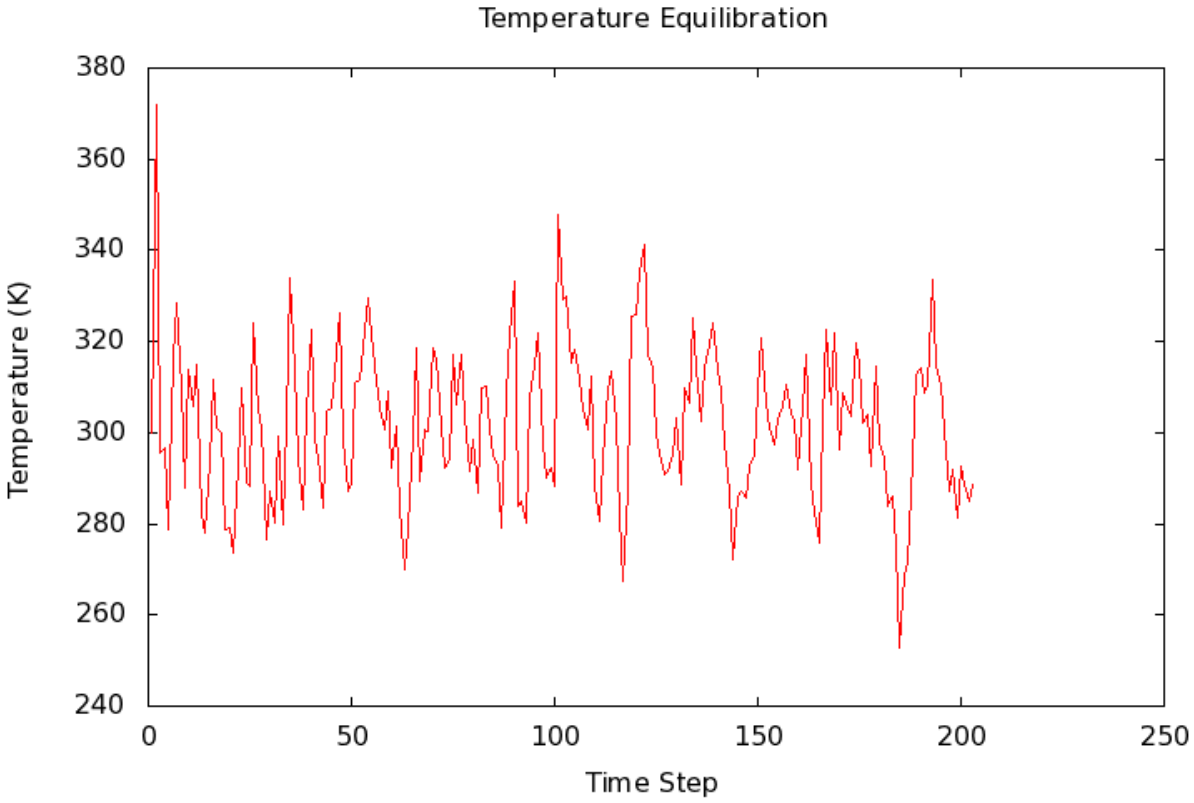
Below is a method of using squid to (1) read in a solvent molecule, (2) utilize the packmol hook to pack a box, and (3) equilibrate the system via NPT and NVT calculations.

```
from squid import units
from squid import structures
from squid import lammps_job

# Generate the system object to hold our solvent
solvent_box = structures.System(name="solv_box", box_size=(15.0, 15.0, 15.0), box_
↪angles=(90.0, 90.0, 90.0), periodic=True)
```

```
# Read in our molecule
# Note, we specified our forcefield indices in the cml file
acetone = structures.Molecule("acetone.cml")

# Using packmol, pack this box with acetic acids
solvent_box.packmol([acetone], density=0.791, seed=21321)

# Now we can run an NPT simulation using lammps
## Get a list of elements for dump_modify.  By default we organize types by heaviest␣
↪to lightest, so do so here.
atom_types = []
elems = []
for molec in solvent_box.molecules:
        for atom in molec.atoms:
                if atom.type.element_name not in atom_types:
                        atom_types.append(atom.type.element_name)
                        elems.append(atom.element)
elem_mass = [units.elem_weight(e) for e in elems]
elem_str = " ".join([x for (y,x) in sorted(zip(elem_mass,elems))][::-1])

input_script = """units real
atom_style full
pair_style lj/cut/coul/cut 10.0
bond_style harmonic
angle_style harmonic
dihedral_style opls

boundary p p p
read_data solv_box.data

dump 1 all xyz 100 solv_box.xyz
dump_modify 1 element """+elem_str+"""

thermo_style custom ke pe temp press
thermo 100

minimize 1.0e-4 1.0e-6 1000 10000

velocity all create 300.0 23123 rot yes dist gaussian
timestep 1.0

fix motion_npt all npt temp 300.0 300.0 100.0 iso 0.0 0.0 1000.0
run 10000
unfix motion_npt

fix motion_nvt all nvt temp 300.0 300.0 300.0
run 10000
unfix motion_nvt
"""

lammps_job.job("solv_box", input_script, solvent_box, queue=None, hybrid_angle=False)
```

Plotting the pressure and temperature we can verify equilibration (note, this is a rough demo so there is still a lot of noise).


Pressure Equilibration
Pressure (Atm)
30000
25000
20000
15000
10000
5000
0
-5000
-10000
0
50
100
150
200
250
Time Step


Temperature Equilibration
Temperature (K)
380
360
340
320
300
280
260
240
0
50
100
150
200
250
Time Step

3.7 Optimizers

Using the built in optimizers, you're able to extend them to mathematical problems. Take, for example, the following equation:

$y = 2x^2 + x^5 - ln(x)$

$\frac{\partial y}{\partial x} = 4x + 5x^4 - \frac{1}{4}$

Using the following, you are able to determine the value of x that would minimize y. Note, currently *quick_min()* does not work in this regard.

```
import numpy as np

from squid.optimizers.bfgs import bfgs
# from lbfgs import lbfgs
# from steepest_descent import steepest_descent
# from fire import fire


def grad(params):
        # Function is y = 2x^2 + x^5 - ln(x)
        # Derivative is y = 4x + 5x^4 - 1/x
        x = params[0]
        return np.array([float(4 * x + 5 * x**4 - 1 / x)])


def grad2(params2):
        # Function is z = (x-3)^2 + (y+2)^2 + x*y
        # Derivative is:
        #     dz/dx = 2(x-3) + y
        #     dz/dy = 2(y+2) + x
        x, y = params2
        a = 2.0 * (x - 3.0) + y
        b = 2.0 * (y + 2.0) + x
        return np.array([a, b])


params = [3.0]
params2 = [4.0, 4.0]

print bfgs(params, grad, new_opt_params={'dimensions': 1})
# print lbfgs(params, grad, new_opt_params={'dimensions': 1})
# print steepest_descent(params, grad, new_opt_params={'dimensions': 1})
# print fire(params, grad)

print bfgs(params2, grad2, new_opt_params={'dimensions': 2})
```

Due to the implementation of the optimizers, you must specify the dimensionality of your problem. A second example has been included in the above code for a two dimensional problem.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX

I

J

L

M

N

O

P

Q

R

S

T

U

V

W