pyfragment Documentation

Release 0.1

Misha Salim

Apr 25, 2017

CONTENTS

PyFragment is a collection of Python modules that facilitate the **setup**, **parallel execution**, and **analysis** of *embedded-fragment* calculations on molecular clusters, liquids, and solids. It currently interfaces with the quantum chemistry software packages Psi4 and NWChem.

The binary interaction method (**BIM**) module performs a variety of calculations:

- Total energy (or unit cell energy) evaluation at HF, MP2, or beyond
- Molecular clusters or systems with 1- to 3-dimensional periodic boundary conditions
- Nuclear gradients and stress tensor, even for nonorthogonal lattice vectors
- Nuclear hessian and vibrational analysis tools
- Parallel execution using MPI (through the mpi4py bindings)

The experimental valence bond charge-transfer (**VBCT**) module is part of a new method development effort. The intent is to extend molecular fragment calculations to systems with significant charge-resonance, where integer electron counts cannot be assigned to individual fragments.

Modules can be invoked from the command line with a freeform input file. They may also be imported to user-written Python programs to create new functionality or automate tedious tasks. Several such **driver** scripts are already included, which use the BIM and VBCT modules to:

- Integrate a **molecular dynamics** trajectory, with optional Nose-Hoover or Berendsen thermostats/barostats
- Perform a **PES scan** along a user-defined coordinate
- **Optimize** crystal structure with the BFGS algorithm

Pyfragment also includes a suite of **tools** to process the output of the above calculations and facilitate data analysis:

- Phonon dispersion/density of states calculations from solid Hessian data
- Molecular dynamics trajectory analysis tool

CHAPTER

ONE

USER GUIDE

1.1 Setup Instructions

1.1.1 Prerequisites

PyFragment requires Python version 2.7. It uses the modules numpy, scipy, h5py, and MPI4Py. These must be readily importable, i.e.:

```
import numpy
import scipy
import mpi4py
import h5py
import argparse
```

should execute without any errors in your Python interpreter.

The `PYTHONPATH` environment variable must include the path to the top level pyfrag directory. This is necessary for the subpackages to find each other.

PyFragment requires at least one quantum chemistry backend to perform the fragment calculations. Currently supported packages are Psi4 and NWChem. The executables `psi4` and `nwchem.x` must therefore be in the system `PATH`. The `backend` subpackage can easily be extended for PyFragment to interface with other quantum chemistry backends.

1.1.2 Gellmann setup tips

To replace the default system Python 2.6 with Python 2.7, the **opt-python** module must be loaded. A convenient way to set up the requisite Python environment is with virtualenv and the pip package manager. Virtualenv allows users to manage a Python installation and install packages in a loadable environment stored in the home directory. This does not require administrator priveleges and the environment can easily be loaded/unloaded as necessary. In a subdirectory of your home directory, perhaps called my-env: run the following commands

```
module load opt-python # replace default Python 2.6 with 2.7
wget https://pypi.python.org/packages/source/p/pip/pip-1.1.tar.gz --no-check-
↪certificate # download PIP
virtualenv --system-site-packages your_environment_name # setup a new environment␣
↪with PIP
source your_environment_name/bin/activate # load the environment
```

Notice that the terminal now indicates your virtual environment is loaded. You are now able to install packages to this environment using **pip**. Invoke

```
pip install mpi4py
pip install scipy
```

In future interactive sessions or any job submissions that require Python 2.7, mpi4py, or scipy, be sure that **opt-python** is loaded and the **source** line is invoked.

1.1.3 Blue Waters setup tips

Blue Waters comes equipped with high-performance builds of the necessary python modules. Your job submission scripts can set up the environment with the two lines

```
module load bwpy
module load bwpy-mpi
```

Installing Psi4 on Blue Waters

Checkout the Psi4 repository and prepare the environment for setup:

```
git clone https://github.com/psi4/psi4.git
module load bwpy
export CRAYPE_LINK_TYPE=dynamic
export CRAY_ADD_RPATH=yes
mkdir ~/libsci
```

Now, query the currently (default) loaded Cray LibSci module to find the directory containg the LibSci libaries. Make symlinks in ~/libsci to all the necessary versions, naming the links corresponding to libsci.* as liblapack.*. This is necessary for CMake to recognize the math libraries. Finally, export the MATH_ROOT environment variable:

```
export MATH_ROOT=~/libsci
```

You will need a newer version of cmake than is provided on Blue Waters. A convenient way to quickly get the binary is with Miniconda (conda install cmake). This binary will work fine for the build process.

```
cd psi4
cmake -H. -Bobjdir
cd objdir
nohup make >& make.log &
```

The compilation is rather lengthy; using nohup will allow you to launch the build and then log off without interrupting the process.

1.1.4 Running PyFragment

The directory containing PyFragment should be included in the `PYTHONPATH` environment variable.

To run as an executable on 16 cores, invoke

```
mpirun -n 16 python /directory/to/pyfrag <input-file>
```

from the command line. This causes Python to run the __main__.py module located in the pyfrag directory. Alternatively, the program can be invoked using

```
mpirun -n 16 python -m pyfrag <input-file>
```

1.2 Quick Start Guide

Follow the setup instructions and start a Python shell. Try the command

```
from pyfrag import bim
```

If any errors occur, it is likely that a prerequisite Python package is missing or the directory containing `pyfrag` is missing from the `PYTHONPATH` environment variable. Be sure that `nwchem.x` and `psi4` are callable from the shell.

1.2.1 A minimal input file

The following is an input file to run a BIM calculation of the water trimer Hartree-Fock/sto-3g energy:

```
geometry {
 O    -0.167787    1.645761    0.108747
 H    0.613411    1.10262    0.113724
 H    -0.093821    2.20972    -0.643619
 O    1.517569    -0.667424    -0.080674
 H    1.989645    -1.098799    0.612047
 H    0.668397    -1.091798    -0.139744
 O    -1.350388    -0.964879    -0.092208
 H    -1.908991    -1.211298    0.626207
 H    -1.263787    -0.018107    -0.055536
}
correlation = False
basis = sto-3g
task = bim_e
```

Save the file to `minimal.inp` and invoke:

```
mpirun -n 3 python -m pyfrag minimal.inp -v
```

If all is working correctly, your output should end with the lines

```
Computing Fragment sums
Task bim_e done in 00m:07s

  E(monomer)    -224.91589306
    E(dimer)       0.01182801
  E(coulomb)       0.00000000
-----------------------------
    E(total)    -224.90406505
  E(total)/N     -74.96802168
```

1.2.2 Setting up a molecular dynamics run

The following input file is more involved and overrides various defaults in the code. It loads a liquid water geometry from an external .xyz file and requests a new NPT molecular dynamics trajectory.

```
scrdir = /scratch
backend = psi4
mem_mb = 3800

basis = aug-cc-pvdz
correlation = MP2
embedding = True
r_qm = 8.0 # cutoff in angstroms
r_bq = 10.0
r_lr = 200.0

task = bim_md

geometry = wat_init.xyz # pbc specified in file as "a b c alpha beta gamma 0"
fragmentation = auto


# md_restart_file = md_example.hdf5
pressure_bar = 0
num_steps = 8000
save_intval = 2
dt_fs = 0.5
temperature = 250 # kelvin
T_bath = nose-hoover
P_bath = berendsen
nose_tau_fs = 30.0 # time constant for computing fictitious masses
```

All of the trajectory information will be logged to `md_example.hdf5` which is a convenient storage format for later trajectory analyses with Python. It also serves as the restart file, which may be requested by the `md_restart_file` parameter in a later job to resume MD integration from the last time step.

1.3 Input to PyFragment

1.3.1 Modular usage

The modules of PyFragment can be imported into other Python programs or interactive sessions. Then, the relevant calculation input can be set programatically by interfacing with the **Globals** modules. The following code snippet shows an example of the syntax:

```
from pyfrag.Globals import params, geom
from pyfrag.bim import bim
# ... other code here ...
params.options['basis'] = 'cc-pvtz'
params.options['fragmentation'] = 'auto'
params.options['r_qm'] = 10.3
params.options['task'] = 'bim_grad'
geomtxt = '''He 0 0 0
             He 1 0 0
             He 2 0 0'''
geom.load_geometry(geomtxt) # build the geometry object
geom.perform_fragmentation() # auto-fragment
result = bim.kernel()
grad = result['gradient']
# ... more code here ...
```

All imports from PyFragment should be in the form of

```
from pyfrag.Globals import logger, params
from pyfrag.backend import nw
```

Warning: NEVER import shared data directly from modules, as in:

```
from pyfrag.Globals.params import options
```

This will produce local objects that do not change in the scope of other modules when updated. This will result in very difficult bugs to track. By importing the modules themselves and referencing their attributes, data is correctly shared between the program modules.

1.3.2 Standalone execution

If PyFragment is invoked from the command line, input must come in the form of an input file argument. The input format is somewhat flexible:

- case-insensitive
- ignores whitespace
- ignores comments starting with '#' character

The parser recognizes two types of entries in the input file.

1. **One line** entries use an = (equals sign)

   ```
   geometry = geom1.xyz
   ```

2. **Multi-line** entries are enclosed in curly braces

   ```
   geometry {
   2.0 0.0 0.0 90.0 90.0 0.0 0
   H 0 0 0
   F 1 0 0
   }
   ```

Input File Structure

Here is a sample input file with comments explaining the meaning of the parameters. The order of input does not matter and parameters irrelvant to the calculation can be omitted.

```
# To run this program, use:
# mpirun -n <nproc> python pyfrag <inputfilename> <-v>
#
# This is a sample input file
# comments begin with '#' character
# one-line entries are parsed as:
#     <keyword> = <value>
# multi-line entries are enclosed in braces and parsed as lists:
#     <keyword> = <list of newline-separated values>

# MAIN PARAMETERS
# ---------------
scrdir = /home/misha/scratch    # optional scratch directory (default /tmp)
```

```
backend = pyscf                  # Quantum chemistry backend: NW or psi4
mem_mb = 3800                    # memory-per-process for QC backend

basis = aug-cc-pvdz
hftype = rohf                      # uhf or rohf
correlation = off                  # can omit this line or specify off/no/false for HF␣
↪theory
embedding = True                   # Use embedding field
r_qm = 8.0 # cutoff in angstroms
r_bq = 8.0
r_lr = 200.0

task = bim_e # bim_e bim_grad bim_hess
             # vbct_e
             # bim_opt
             # bim_md

# GEOMETRY / ANGSTROMS
# --------------------
# list of atoms with formal charges (repeated + or -)
# fragment charges will be sum of formal charges
# example for (H2O)(H3O+) cluster:
geometry {
O 4 0 0
H 4 1 0
H 4 0 1
O 0 0 0
H+ 1 0 0
H 0 1 0
H 0 0 1
}

# if PBC, include a line containing "a b c alpha beta gamma axis0"
# if not periodic in b or c dimension, set lattice constant to 0.0
# example for linear, 1D-periodic HF chain:
#geometry {
#2.0 0.0 0.0 90.0 90.0 0.0 0
#H 0 0 0
#F 1 0 0
#}
# geometry = geom1.xyz  # or .xyz file path (lattice constants on line 2)

# FRAGMENTATION (3 options)
# ------------
fragmentation = auto          # use bond cutoffs in Globals.geom
#fragmentation = full_system # no fragmentation (reference calculation)
#fragmentation {              # newline-separated atom indices
#0 2 3 4
#1
#5 6 7 8 9 10
#11
#}

# VBCT-SPECIFIC OPTIONS
# ---------------------
vbct_scheme = chglocal #chglocal or mono_ip

# OPTIMIZER options
# -----------------
atom_gmax = 0.0015 # max nuclear gradient tolerance
lat_gmax = 0.0003 # max lattice gradient
opt_maxiter = 50

# MD-only options
# ---------------
pressure = 5000 # bar
temperature = 300 # temperature
num_steps = 500
md_restart_file = restart.hdf5
save_intval = 2
dt_fs = 1.0 # femtosecond
T_bath = nose   # None, Nose-Hoover, or Berendsen
P_bath = berend # None or Berendsen

# HESSIAN: force constant matrices
# -------------------------------
interaction_cells = 2 2 2
```

CHAPTER

TWO

SUBPACKAGE DOCUMENTATION

2.1 The `bim` module: BIM energies, gradients, and hessians

2.1.1 `bim`: main BIM routine

Binary Interaction Method – module for embedded-fragment calculations on weakly-interacting molecular clusters (energy, gradient, hessian)

`bim.bim.`**`create_bim_fragment`**`(`*specifier*, *espcharges*`)`

Create and dispatch a backend fragment calculation.

Args:

specifier: tuple specifying the monomer and cell indices for the requested calculation.

espcharges: embedding field charges

Returns: results: results dict from fragment calculation

`bim.bim.`**`get_task`**`()`

map from params.options['task'] –> bim summation type

Returns: task: one of 'energy', 'gradient', 'hessian'

sum_fxn: the corresponding BIM summation function

`bim.bim.`**`kernel`**`(`*comm=None*`)`

Get fragments, do monomer SCF, and dispatch list of fragment calcs.

Controlled by setting values in params.options and geom.geometry

Args: comm (optional): pass a subcommunicator (generated with comm.split) for nested parallelism

Returns: results: dictionary of fragment sums

2.1.2 `monomerscf`: self-consistent fragment ESP charges

`bim.monomerscf.`**`monomerSCF`**`(`*comm=None*`)`

Cycle embedded monomer calculations until ESP charges converge.

BIM version: include all monomers, take charges from input geometry, bq_lists from Globals.neighbor, and embedding option from input file. PBC is implicitly handled No need to do anything if embedding option is off.

Args:

comm: specify a sub-communicator for parallel execution. Default None: use the top-level communicator in Globals.MPI

Returns: espcharges: a list of esp-fit atom-centered charges

2.1.3 `sums`: summation of fragment energies

BIM summation functions to compute full-system properties.

Each sum function takes the same arguments: specifiers: a list of specifier tuples (as defined in bim.kernel)

calcs: a dict (with keys matching specifiers.keys()) of dicts containing the results from each fragment calculation.

and each returns a dictionary of calculation results.

`bim.sums.`**`energy_sum`**(*specifiers*, *calcs*)
Determine total energy

`bim.sums.`**`gradient_sum`**(*specifiers*, *calcs*)
compute gradient of total energy

`bim.sums.`**`hessian_sum`**(*specifiers*, *calcs*)
compute interaction force constants (hessian)

2.2 The `drivers` modules: Molecular dynamics, optimization, and beyond

2.2.1 `bim_md`: Molecular Dynamics

Molecular dynamics (NVE, NVT, NPT) integration with BIM Forces

class `drivers.bim_md.`**`Integrator`**(*forcefield*)
Contains data and methods for trajectory initialization, Velocity Verlet integration (with Nose-Hoover thermostat, Berendensen thermostat, and Berendsen barostat), and HDF5 I/O for trajectory storage

`apply_berend_baro`()
Apply Berendsen cell scaling for pressure control. berend_tau must be > 10 fs, as explained for thermostat.

`apply_berend_thermo`()
Apply Berendsen velocity scaling for temperature control. berend_tau should be at least 10 fs (fast equilibration) at tau>100 fs, the fluctuations should be consistent with NVT

`apply_nose_chain`()
Half-update NH degrees of freedom together with system velocity.

`clean_up`(*istep*)
trim and flush hdf5 trajectory data

`create_trajectory_file`()
Create new trajectory in hdf5 file; set handles to data

`detect_and_fix_pbc_crossings`()
translate fragments with COM outside unit cell inside

`get_MD_options`()
Set relevant parameters from input file

`init_velocity`()
Set initial vel according to Maxwell-Boltzmann

integrate()
Velocity Verlet with thermo/barostatting

kinetic_com_tensor()
Compute kinetic energy tensor based on fragment centers of mass; needed for stress tensor calculation

nose_init()
Initialize NH coordinates & masses.

restart_trajectory_file()
Load trajectory from hdf5 file and append

summary_log(*istep*, *wall_seconds*)
Print MD summary statistics to stdout

update_kinetic_and_temperature()
Update kinetic energy & temperature

write_trajectory(*istep*)
Write to hdf5 trajectory file

drivers.bim_md.**bim_force**()
BIM gradients

drivers.bim_md.**hooke_force**(*k=0.2*, *r_eq=2.0*)
toy potential: harmonic oscillator

drivers.bim_md.**kernel**()
MD Main

2.2.2 `bim_opt`: Geometry Optimization

Geometry optimization (fixed unit cell)

drivers.bim_opt.**from_flat**(*x*)
reconstruct geom from flat ndarray

drivers.bim_opt.**gopt_callback**(*xk*)
Report geometry by appending to gopt.xyz

drivers.bim_opt.**kernel**()
Invoke scipy optimizer

drivers.bim_opt.**objective_gopt**(*x*)
energy opt wrt geometry (using E and gradient)

This objective function packs/unpacks the geometry from a 1D ndarray, updates geom.geometry, and invokes bim.kernel to update the energy/gradient.

drivers.bim_opt.**to_flat**()
return 1D ndarray input for objective fxn

2.3 The `tools` modules: post-processing and analysis

2.3.1 `phonon`: Vibrational Analysis

Tool for computing phonon dispersion and vibrational normal modes, given a BIM Hessian

`tools.phonon.`**`make_brillouin_zone3D`**(*kmesh_density=None*)
Brute-force generate uniform grid inside 1st Brillouin zone.

Brute force but clean approach. Truncates a uniform rectangular grid; therefore, nonorthogonal cells may require a high density of points to sample adequately near the zone boundaries.

`tools.phonon.`**`mass_weight`**(*hess*)
Return mass-weighted hessian

`tools.phonon.`**`parseargs`**()
Parse command line arguments using argparse

`tools.phonon.`**`phonon`**(*hess_mw*, *kmesh*, *smooth_mat=None*)
Compute phonon dispersion and normal modes on a given k-mesh

`tools.phonon.`**`phonon_freqs`**(*hess_mw*, *kmesh*, *smooth_mat=None*)
Compute phonon dispersion without returning normal mode eigenvectors

`tools.phonon.`**`read_force_consts_np`**(*fname*)
Read force constants from numpy binary file

`tools.phonon.`**`read_force_consts_txt`**(*fname*)
Read force constants from text file

`tools.phonon.`**`smoothing_matrix`**(*hess_mw*)
Generate matrix for projecting out translational normal modes

2.4 The `vbct` module: charge resonance in cluster cations

2.4.1 `energy`: VBCT energy calculation

Compute GS doublet energy and charge distribution in VBCT scheme

1. No PBC or truncation of summations: only cluster ions are calculated
2. only charge +1 is supported: single hole hopping between fragments

`vbct.energy.`**`build_secular_equations`**(*diag_results*, *offdiag_results*)
Build Hamiltonian and overlap matrix

`vbct.energy.`**`calc_chg_distro`**(*prob0*, *diag_results*)
Generate linear combination of esp charge distros

Args prob0: ground state probability distribution diag_results: diagonal element calculation results list,

contains esp_charges for each charge-local state

Returs charge_distro: approximate charge distribution computed as linear combination of VBCT basis charge distros

`vbct.energy.`**`calc_diagonal`**(*idx*, *comm=None*)
Dispatch diagonal element calculation

Args idx: integer index ranging from 0-len(geom.fragments). Which diagonal element to calculate comm: MPI communicator or subcommunicator

Returns res: dict, results from diagonal element calculation.

`vbct.energy.`**`fraglabel`**(*frag*, *chg*)
Generate chemical formula string for a fragment

`vbct.energy.`**`kernel`**`()`
SP energy: form and diagonalize VBCT matrix

`vbct.energy.`**`print_vbct_init`**`(`*states*`)`
Print header for calculation

2.4.2 `vbct_calc`: algorithms for effective Hamiltonian generation

`vbct.vbct_calc.`**`coupl_chglocal`**`(`*A*, *B*`)`
Charge-local dimer method for coupling

Only works with NW backend and singly ionized molecular cluster cations.

Args A, B: indices for off-diagonal H element calculation

`vbct.vbct_calc.`**`diag_chglocal`**`(`*charges*, *espfield*, *movecs*, *comm=None*`)`
Charge-local dimer method for diagonal element calculation.

Only works with NW backend and singly ionized molecular cluster cations.

Args charges: list of net charges on each fragment espfield: list of esp-fit atomic charges for entire system movecs: list of MO coeff files for each fragment comm: MPI communicator or subcommunicator

`vbct.vbct_calc.`**`make_embed_list`**`(`*qm_fragment*, *all_monomers*`)`
Generate embedding field

Args qm_fragment: index of monomer or dimer (no PBC)

Returns bq_list: list of fragments in bq field of qm_fragment

2.4.3 `monomerscf`: self-consistent VBCT state ESP charges

`vbct.monomerscf.`**`fullsys_best_guess`**`(`*comm=None*`)`
Get conventional E of full system by finding best initial guess.

Iterates over fragments to produce Nfrag charge-local initial guesses. Takes best UHF energy from each of the corresponding initial densities, then does correlated calculation.

`vbct.monomerscf.`**`monomerSCF`**`(`*monomers*, *net_charges*, *embedding=None*, *comm=None*`)`
Cycle embedded monomer calculations until the ESP charges converge.

VBCT version: specify N monomers and their net charges explicitly. No support for cutoffs/periodicity: every monomer is embedded in the field of all other N-1 monomers. Able to override default embedding option. Monomer MO vectors are saved; thus one cycle runs even if embedding option is turned off.

Args monomers: a list of monomer indices net_charges: net charge of each monomer embedding: Override True/False specified in input.

Default None: use the value specified in input.

comm: specify a sub-communicator for parallel execution. If string ‘serial’ is specified, bypass MPI communication. Default None: use the top-level communicator in Globals.MPI

Returns espcharges: a list of esp-fit atom-centered charges movecs: a list of MO vectors for each monomer

2.5 The `backend` modules: interface to quantum chemistry software

2.5.1 `backend`: Generic QC functionality

Wrapper functions for quantum chemistry backends

`backend.backend.`**`build_atoms`**(*frags*, *bq_list*, *bq_charges*)

Make the input geometry/embedding for a QM calculation.

Args frags: a list of 4-tuples (i,a,b,c) where i is the fragment index, abc indicate the lattice cell. bq_list: a list of (4-tuples) indicating the molecules to be placed in the embedding field. bq_charges: a list of point charges for each atom in the geometry.

Returns atoms: a list of Atoms for easy printing to the QM "geometry" input. bq_field: a list of numpy arrays(length-4) in format (x,y,z,q)

`backend.backend.`**`run`**(*calc*, *frags*, *charge*, *bq_list*, *bq_charges*, *noscf=False*, *guess=None*, *save=False*)

QM backend dispatcher: invoke a calculation.

Currently, for ESP calculation the NWChem package is always dispatched for its higher performance. Otherwise, the backend is determined by the params backend option.

Args calc: one of esp, energy, energy_hf, gradient, hessian frags: list of 4-tuples (i,a,b,c) (See build_atoms documentation) charge: the net charge of the fragments in QM calculation bq_list: the embedding fragments, list of 4-tuples bq_charges: the embedding charges for each atom in the geometry noscf (default False): if True, only build Fock matrix from initial

guess and diagonalize once.

guess (default None): if supplied, provide prior MO for initial density.

save (default False): if True, return a handle to the MO resulting from calculation. For NW & Gaussian, this is a file path & MO file will be copied to a shared directory. For Psi4 and PySCF, the MO vectors are directly returned as a 2D numpy array.

Returns results: a dictionary packaging all of the results together. The contents of 'dictionary' depend on the type of calculation invoked and other arguments.

2.5.2 `nw`: NWChem wrapper

NWChem Backend

`backend.nw.`**`calculate`**(*inp*, *calc*, *save*)

Run nwchem on input, return raw output

Args inp: NWChem input object (input file path) calc: calculation type save: save calculation results

Returns output_lines: nwchem stdout lines

`backend.nw.`**`inp`**(*calc*, *atoms*, *bqs*, *charge*, *noscf=False*, *guess=None*, *save=False*)

Write NWchem input file to temp file. Return filename.

`backend.nw.`**`invecs`**(*guess*)

Create initial guess string for NWchem scf input

Args guess: string or list of strings for fragment guess

`backend.nw.`**`parse`**(*data*, *calc*, *inp*, *atoms*, *bqs*, *save*)

Parse raw NWchem output.

2.5.3 `psi4`: Psi4 wrapper

Backend for Psi4 – using subprocess and file input/output

`backend.psi4.`**`calculate`**(*inp*, *calc*, *save*)
    run psi4 on input, return text output lines from psi4

`backend.psi4.`**`inp`**(*calc*, *atoms*, *bqs*, *charge*, *noscf=False*, *guess=None*, *save=False*)
    Generate psi4 input file

`backend.psi4.`**`parse`**(*data*, *calc*, *inp*, *atoms*, *bqs*, *save*)
    Parse psi4 output text, return results dict

2.6 The `test` modules: unit testing with Python

2.6.1 `test`: serial test cases

Unit Tests using Python unittest framework

Run all tests from command line using:

```
>>> python -m unittest pyfrag/test
```

Run one test from command line by specifying a specific test case:

```
>>> python -m unittest pyfrag/test.TestTrimerRHF
```

Add new test cases by creating new classes that extend unittest.TestCase:

class `test.test.`**`TestTrimerMP2`**(*methodName='runTest'*)
    Test MP2 gradients on water trimer with both NW/Psi4 backends

class `test.test.`**`TestTrimerRHF`**(*methodName='runTest'*)
    Test RHF energy and gradient on water trimer

2.7 The `Globals` Modules

The **Globals** modules contains data and functionality shared between all types of fragment calculations.

2.7.1 geom

This module contains the shared `geometry` and `fragments` lists.

It defines the Atom class and supporting data structures to load and print geometry information. It also contains the logic for performing *fragmentation*, that is, assigning which atoms belong to which fragments. The `geometry` and `fragments` lists are shared across all modules.

class `Globals.geom.`**`Atom`**(*atomstr*, *units='angstrom'*)
    Convenience class for loading and storing geometry data

`Globals.geom.`**`com`**(*frag*)
    Get center of mass of a fragment.

    Args frag: the list of atom indices

Returns com: numpy array pointing at COM

`Globals.geom.`**`load_geometry`**`(`*data*, *units='angstrom'*`)`

Loads geometry from input text, lists, or filename.

Tries to be flexible with the form of input 'data' argument. Uses regex to extract atomic coordinates from text.

Args data: string, list of strings, list of lists, or filename containing the xyz coordinate data units (default Angstrom): "bohr" or "angstrom"

Returns None: the geometry is saved as a module-level variable

`Globals.geom.`**`nuclear_repulsion_energy`**`()`

Return nuclear repulsion energy / a.u.

`Globals.geom.`**`set_frag_auto`**`()`

Auto-generate list of fragments based on bond-length frag_cutoffs.

Use this if you don't wish to manually assign atoms to fragments.

Args: None (module-level geometry is used)

Returns: None (module-level fragments list is set)

`Globals.geom.`**`set_frag_full_system`**`()`

No fragmentation: all atoms in system belong to one fragment.

Use this to perform one big reference QM calculation

Args: None (module-level geometry is used)

Returns: None (module-level fragments list is set)

`Globals.geom.`**`set_frag_manual`**`()`

Read list of fragments from input file.

Fragmentation is manually specified in the input file.

Args: None (module-level geometry is used)

Returns: None (module-level fragments list is set)

2.7.2 lattice

This module contains the globally-shared lat_vecs which defines the Bravais lattice vectors of the system in a 3x3 ndarray. It comes with supporting functionality for 3D PBC calculations:

- updating lattice vectors from lattice parameters (a,b,c,alpha,beta,gamma)
- the inverse of lat_vecs for transformation to fractional(scaled) coords
- computing cell volume computing gradient wrt lattice parameters, using virial tensor and applied stress
- rescaling cell by translating fragment centers of mass, while preserving fragment internal coordinates

`Globals.lattice.`**`lat_angle_differential`**`()`

Compute partial derivatives of lat_vecs wrt angle parameters.

This is just a quick finite difference calculation.

Args: None

Returns: Three 3x3 numpy arrays. They contain the derivatives of the lattice vector components with respect to alpha, beta, and gamma, respectively. Units of each matrix element are Angstroms/degree.

`Globals.lattice.`**`lattice_gradient`**`(`*virial*, *p0_bar*`)`

Compute the energy gradient in lattice parameters (a.u. / bohr)

Args: virial: 3x3 numpy array, from gradient calculation p0_bar: external pressure in bar

Returns: lat_grad: 6-dimensional gradient vector in au/bohr, au/degrees

`Globals.lattice.`**`rescale`**`(`*scaling*`)`

Rescale the lattice vectors and shift fragment centers of mass to preserve scaled COM coordinates while maintaining internal fragment coordinates.

Args:

scaling: either a 3x3 lattice vector transformation matrix (ndarray) or a list of lattice parameters (a,b,c,alpha,beta,gamma,axis) in Angstrom.

Returns: None.

`Globals.lattice.`**`update_lat_params`**`()`

Update lattice parameters NamedTuple from “lat_vecs”

The inverse of update_lat_vecs; recomputes the lattice parameters based on the current vectors. Lattice parameter units are angstrom/degree.

`Globals.lattice.`**`update_lat_vecs`**`()`

Update lattice vectors matrix (3x3, angstrom) from “lattice”.

Invoke this function whenever the lattice variable is changed to recompute the lattice vectors. No input arguments or return values.

`Globals.lattice.`**`volume`**`()`

Return volume in Angstrom**3

3D case: return the determinant of lat_vecs matrix For 2D,1D,0D systems: return cell area, length, unity, respectively.

2.7.3 params

This module contains the globally-shared options dict and a method to generate it by parsing an input file. Other modules set module-level attributes like “quiet” and “verbose”.

`Globals.params.`**`convert_params`**`()`

Sanitize parsed options.

Make string–>(list, float, boolean) conversions, wherever possible

`Globals.params.`**`parse`**`(`*inFile*`)`

Crude input file parser.

Populates the “options” dictionary from an input file.

Args: inFile: input file handle for reading

Returns: None

`Globals.params.`**`set_defaults`**`()`

Set default parameters if not specified

`Globals.params.`**`tryFloat`**`(`*s*`)`

Try to cast to float; no big deal

2.7.4 MPI

Wrap mpi4py, so that the code works as expected whether running in serial or parallel

`Globals.MPI.`**`allgather`**(*comm*, *data*)
: default allgather fxn is dumb... data = MPI_allgather(comm, mydata)

`Globals.MPI.`**`create_split_comms`**(*N*)
: Evenly divide nproc into N subcommunicators.

  Returns subcommunicator and color.

`Globals.MPI.`**`gather`**(*comm*, *data*, *master=0*)
: default gather fxn is dumb... data = MPI_gather(comm, data, 0)

`Globals.MPI.`**`scatter`**(*comm*, *data*, *master=0*)
: default scatter fxn is dumb; can only handle one item per rank. This will split a list evenly among the ranks by creating a list of lists. mydata = MPI_scatter(comm, data, 0)

2.7.5 neighbor

Generate pair lists for QM calculations and BQ embedding

`Globals.neighbor.`**`BFS_lattice_traversal`**(*pair_accumulate_fxn*, ***args*)
: Generic floodfill algorithm to visit cells & build neighbor lists.

  This function performs a Breadth-First Search (BFS) starting from the central unit cell (0,0,0) and moving outwards in all periodic dimensions. It requires an accumulation function as its first argument, which is used to count up all the relevant interactions between origin and a given cell (a,b,c). Once no more interactions are counted, the outward fill ends.

  Args

  pair_accumulate_fxn: function of the form f(cell, **args) which builds neighbor lists between the given cell and origin cell. It must return the number of pairs within range.

  **args: additional arguments to pair_accumulate_fxn

  Returns None

`Globals.neighbor.`**`build_lists`**()
: (re)compute neighbor lists via BFS floodfill search, based on cutoffs defined in params.options

`Globals.neighbor.`**`pair_dist`**(*pair_tup*)
: Compute dimer separation given a pair tuple

`Globals.neighbor.`**`pairlist_accumulator`**(*cell*, *com*)
: count up dimers within R_QM and R_BQ cutoffs

  This is called by a generic breadth-first search method which traverses all unit cells outward from cell 0 (3D flood-fill).

2.7.6 coulomb

Evaluate classical point charge interactions between fragments

`Globals.coulomb.`**`accumulate_pair`**(*idx1*, *idx2*, *cell*, *scale*, *charges*)
: Coluomb E, grad, virial for a specific dimer.

  The coulomb interactions are accumulated into module-level variables energy_coulomb, gradient_coulomb, and virial_coulomb.

Args: idx1, idx2: indices of monomers cell: lattice vectors for molecule at idx2 scale: scale factor for energy contribution charges: list of charges for the whole system

Returns None

`Globals.coulomb.`**`coulomb_accumulator`**`(`*cell*, *charges*`)`

Accumulate all the dimer interactions between cell 0 and a given cell.

This is called by a generic breadth-first search method which traverses all unit cells outward from cell 0 (3D flood-fill). The pair interactions are computed in parallel and must be accumulated later.

Args: cell: a,b,c lattice indices of cell interacting with cell 0 charges: list of esp charges on each fragment

Returns: num_pairs: the number of interactions counted with the given cell

`Globals.coulomb.`**`evaluate_coulomb`**`(`*espfield*`)`

Evaluate all coulomb interactions via BFS over cells

2.7.7 logger

Convenience functions for pretty printing/file IO

2.7.8 utility

`Globals.utility.`**`make_scratch_dirs`**`(`*top_dir=None*`)`

Establish current directory, scratch directories, and temporary shared directories

`Globals.utility.`**`mw_execute`**`(`*specifiers*, *run_calc*, *comm=None*, **args*`)`

Run a series of generic calculations in master-worker mode.

Args:

specifiers: a list of tuples which uniquely determine a fragment calculation.

run_calc: the function which accepts a specifier tuple, sets up a QM calculation, invokes the backend, and returns the relevant calculation results.

comm: optional subcommunicator, default None: use Globals.MPI communicator

Returns: calcs: the dictionary of results indexed by specifiers

`Globals.utility.`**`parse_input`**`(`*input_file*`)`

Open input_file, parse it, load geometry, and MPI-broadcast the data

`Globals.utility.`**`pretty_time`**`(`*seconds*`)`

Convert elapsed seconds to a nice string representation

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX