pyzfs

Mar 27, 2020

Contents

PyZFS is an MPI-parallelized Python code for the first-principles calculation of the spin-spin zero-field-splitting (ZFS) tensor based on wavefunctions obtained from density functional theory (DFT) calculations.

PyZFS can work with wavefunctions generated by various plane-wave pseudopotential DFT codes including **Quantum Espresso** (https://www.quantum-espresso.org/) and **Qbox** (http://qboxcode.org/). **PyZFS** also supports the standard cube file format. **PyZFS** computes the spin-spin ZFS tensor from normalized pseudo-wavefunctions without projected-augmented-wave type corrections and is designed to be scalable to large calculations. For instance, **PyZFS** has been applied to study spin-defects in semiconductors using supercells containing thousands of valence electrons.

CHAPTER 1

Installation

PyZFS uses the **mpi4py** package for parallelization. An existing MPI implementation (e.g. **MPICH** or **OpenMPI**) is required to install **mpi4py** and **PyZFS**. Many supercomputers provide modules for pre-compiled MPI implementations. To install MPI manually (taking **MPICH** as example), execute the following command on Linux

```
$ sudo apt-get install mpich libmpich-dev
```

or the following command on Mac

```
$ brew install mpich
```

PyZFS can be executed with Python 2.7 and Python 3.5+. However, to run **PyZFS** with Python 2.7 one may need to build certain legacy versions of dependencies such as **ase** (**ase** v3.17.0 is tested to work with **PyZFS** in Python 2).

It is recommended to install **PyZFS** using **pip**. First, clone the git repository into a local directory

```
$ git clone https://github.com/hema-ted/pyzfs.git
```

Then, execute **pip** in the folder containing **setup.py**

```
$ pip install .
```

PyZFS depends on the following packages, which will be installed automatically if installed through **pip**

- `numpy`
- `scipy`
- `mpi4py`
- `h5py`
- `ase`
- `lxml`

If using **pip** is not possible, one can manually install the above dependencies, and then include the directory of **PyZFS** repository to the **PYTHONPATH** by appending the following command to the **.bashrc** file

```
$ export PYTHONPATH=$PYTHONPATH:path/to/pyzfs
```

CHAPTER 2

Tutorial

After installation, **PyZFS** can be executed in two manners:

1. Construct WavefunctionLoader and ZFSCalculation loader from within Python terminal or Jupyter notebook, and call ZFSCalculation.solve to perform the calculation.

   An example Python script for computing the ZFS tensor for oxygen molecule is shown below. */path/to/o2.xml* should be replaced by the path to the *pyzfs/examples/o2_qbox_xml/o2.xml* file in the **PyZFS** folder.

   ```
   >>> from pyzfs.common.wfc.qboxloader import QboxWavefunctionLoader
   >>> from pyzfs.zfs.main import ZFSCalculation
   >>> wfcloader = QboxWavefunctionLoader(filename='/path/to/o2.xml')  # Construct␣
   ↪wavefunction loader
   >>> zfscalc = ZFSCalculation(wfcloader=wfcloader)  # Set up ZFS calculation
   >>> zfscalc.solve()  # Perform ZFS calculation
   ```

   Example Jupyter notebooks can be found at /examples/o2_qbox_xml/run.ipynb and /examples/o2_qe_hdf5/run.ipynb.

2. Directly execute **PyZFS**. This approach works more smoothly with MPI.

   For serial execution, simply type the following command in the folder that contains DFT wavefunction file(s)

   ```
   $ pyzfs [--flags]
   ```

   For parallel execution, use the following command

   ```
   $ mpiexec [-n num_of_processes] pyzfs [--flags]
   ```

   where *num_of_processes* is the number of processes. **PyZFS** distributes the calculations on a square grid of processes. If *num_of_processes* is not a square number, **PyZFS** will use the largest square number of processes smaller than *num_of_processes* for calculations.

   Note that to use the above *pyzfs* command, one needs to install **PyZFS** through **pip** (see *Installation*). If one manually added **PyZFS** directory to the **PYTHONPATH** without installing it, one needs to replace the above commands with

```
$ python -m pyzfs.run [--flags]
```

and

```
$ mpiexec [-n num_of_processes] python -m pyzfs.run [--flags]
```

Acceptable flags [–flags] are listed below, for detailed explanation see *pyzfs/run.py*.

- *path*: working directory for this calculation. Python will first change the working dir before any calculations. Default is “.”.
- *wfcfmt*: format of input wavefunction. Default is “qeh5”. Supported values are:
  - “qeh5”: Quantum Espresso HDF5 save file. path should contains “prefix.xml” and save folder.
  - “qe”: Quantum Espresso (v6.1) save file. path should be the save folder that contains “data-files.xml”, etc.
  - “qbox”: Qbox xml file.
  - “cube-wfc”: cube files of (real) wavefunctions (Kohn-Sham orbitals).
  - “cube-density”: cube files of (signed) squared wavefunction, this option is to support *pp.x* output with *plot_num = 7* and *lsign = .TRUE.*.
- *filename*: name of the Qbox sample XML file that contains input wavefunction. Only used if *wfcfmt = “qbox”*.
- *fftgrid*: FFT grid used. Supported values are “density” or “wave”. “density”: the density grid is used for FFT; “wave”: a reduced grid is used for FFT. Default is “wave”, which is computationally less expensive and is recommended for large-scale calculations.
- *memory*: Controls whether certain intermediate quantities are kept in memory or re-computed every time. Supported values are “high”, “low” and “critical”, which keeps the decreasing amount of quantities in memory. Default is “critical”, which costs least memory and is recommended for large-scale calculations.

An example execution command for Quantum Espresso HDF5 save file is

```
$ mpiexec pyzfs --wfcfmt qeh5 --prefix pwscf
```

where pwscf is the prefix used for the Quantum Espresso calculation.

An example execution command for Qbox XML save file is

```
$ mpiexec pyzfs --wfcfmt qbox --filename gs.xml
```

where gs.xml is the XML save file generated by Qbox.

See *pyzfs/examples* for examples of computing the ZFS tensor for the oxygen molecule and the nitrogen-vacancy (NV) center in diamond.

After **PyZFS** is executed, the D tensor, its eigenvalues and eigenvectors are printed by the end of the output. The widely-used scalar D and E parameters are also printed. A “zfs.xml” file is generated that includes these information, facilitating parsing the results through scripts.

PyZFS can scale to hundreds of MPI processes, and has been applied to systems with up to 3000 valence electrons. For large calculations, typical walltime for a calculation is on the order of 12-24 hours.

CHAPTER 3

Code documentation

PyZFS can be extended to support more wavefunction formats by defining subclasses of **WavefunctionLoader** abstract class. The abstract method **scan** and **load** have to be override to parse and read the wavefunction data into memory and store as a **Wavefunction** object.

PyZFS API documentation:

3.1 ZFS

class `pyzfs.zfs.main.`**`ZFSCalculation`**`(`*`**kwargs`*`)`

Bases: `object`

Zero field splitting D tensor calculation.

Generally, calculation of D tensor involves pairwise iteration over many wavefuctions (KS orbitals). Physically, wavefunction is uniquely labeled by a 2-tuple of band index (int) and spin (“up” or “down”). Internally, each wavefunction is labeled by an integer index. Several maps are defined to describe related transformations.

`wfc`

container for all KS orbitals

Type *Wavefunction*

`cell`

defines cell size, R and G vectors

Type *Cell*

`ft`

defines grid size for fourier transform

Type *FourierTransform*

`ddig`

dipole-dipole interaction tensor in G space. Shape = (6, n1, n2, n3), where first index labels cartisian directions (xx, xy, xz, yy, yz, zz), last 3 indices iterate over G space

Type ndarray

`Iglobal`
global I array of shape (norbs, norbs, 6) first two indices iterate over wavefunctions, last index labels catesian directions in xx, xy, xz, yy, yz, xz manner

Type ndarray

`I`
local I matrix, first two dimensions are distributed among processors

Type ndarray

`D`
3 by 3 matrix, total D tensor

Type ndarray

`ev, evc`
eigenvalues and eigenvectors of D tensor

Type ndarray

`Dvalue, Evalue`
scalar D and E parameters for triplet

Type float

`get_xml()`
Generate an xml to store information of this calculation.

Returns A string containing xml.

3.2 Common

class `pyzfs.common.wfc.baseloader.`**`WavefunctionLoader`**(*memory='critical'*)
A wavefunction loader that can load the wavefunction generated by given DFT codes into memory, stored as a Wavefunction object.

`load`(*iorbs*, *sdm*)
Load read space KS orbitals to memory, store in wfc.iorb_psir_map.

Parameters

- **`iorbs`** – a list of integers representing orbital indices.
- **`sdm`** – a SymmetricDistributedMatrix object indicating how the wavefunction is distributed.

Returns After load is called, the wavefunction will be loaded into self.wfc.

`scan`()
Scan current directory, construct wavefunction object

class `pyzfs.common.wfc.wavefunction.`**`Wavefunction`**(*cell*, *ft*, *nuorbs*, *ndorbs*, *iorb_sb_map*, *iorb_fname_map*, *dft=None*, *gamma=True*, *gvecs=None*)
Container class for Kohn-Sham orbitals

Physically, wavefunction is uniquely labeled by a 2-tuple of band index (int) and spin ("up" or "down"). Internally, each wavefunction is labeled by an integer index. Several maps are defined to describe related transformations.

`norbs`
total number of KS orbitals to be considered

Type int

`nuorbs/ndorbs`
number of spin up/down orbitals

Type int

`sb_iorb_map`
(spin, band index) -> orb index map

Type dict

`iorb_sb_map`
orb index -> (spin, band index) map

Type list

`iorb_psir_map`
orb index -> orb object (3D array) map

Type dict

`cell`
defines cell size, R and G vectors

Type *Cell*

`ft`
defines grid size for fourier transform

Type *FourierTransform*

Right now only consider ground state, insulating, spin-polarized case. No occupation number considerations are implemented yet.

`compute_psir_from_psig_arr`(*psig_arr*)
Compute psi(r) based on psi(G) defined on self.gvecs

`get_psir`(*iorb*)
Get psi(r) of certain index

`get_rhog`(*iorb*)
Get rho(G) of certain index

`normalize`(*psir*)
Normalize psir.

`class` `pyzfs.common.cell.`**`Cell`**(*ase_cell*)
A wrapper class for ASE Atoms that defines R and G vectors.

`class` `pyzfs.common.ft.`**`FourierTransform`**(*n1*, *n2*, *n3*)
Define forward/backward 3D FT on a given grid

Forward/backward FT are defined with following conventions: f(G) = 1/omega * int{ f(r) exp(-iGr) dr } f(r) = sigma{ f(G) exp(iGr) }

`backward`(*fg*)
Fourier backward transform a function.

Parameters **`fg`** (*`np.ndarray`*) – function in G space (3D array)

Returns function in R space (with same grid size)

forward(*fr*)
Fourier forward transform a function.

Parameters **fr** (*np.ndarray*) – function in R space (3D array)

Returns function in G space (with same grid size)

interp(*fr*, *n1*, *n2*, *n3*)
Fourier interpolate a function to a smoother grid.

Parameters

- **fr** – function to be interpolated
- **n2, n3** (*n1,*) – new grid size

Returns interpolated function (3D array of size n1 by n2 by n3)

class pyzfs.common.parallel.**DistributedMatrix**(*pgrid*, *shape*, *dtype*)
An array whose first two dimensions are distributed.

Convention: a variable indexing local block of a distributed matrix should have trailing "loc" in its name, otherwise it is considered a global index

collect()
Gather the distributed matrix to all processor.

Returns: global matrix.

gtol(*i*, *j=None*)
global -> local index map

ltog(*iloc*, *jloc=None*)
local -> global index map

class pyzfs.common.parallel.**ProcessorGrid**(*comm*, *square=False*)
2D Grid of processors used to wrap MPI communications.

class pyzfs.common.parallel.**SymmetricDistributedMatrix**(*pgrid*, *shape*, *dtype*)
A array whose first two dimensions are distributed and symmetric.

get_triu_iterator()
Get a list of 2D indices to iterate over upper triangular part of the local matrix.

Returns list of 2-tuples of ints.

symmetrize()
Compute lower triangular part of the matrix from upper triangular part.

Installation Instructions on how to install the **PyZFS** package.

Tutorial Demonstration of usage of **PyZFS** with wavefunctions from various DFT codes.

Code documentation Detailed documentation of the **PyZFS** package.

Python Module Index

Index