PWDFT.jl Documentation

Fadjar Fathurrahman

February 14, 2019

Contents

Part I
User Guide

1 Overview

PWDFT.jl is a Julia package to carry out (electronic) density functional theory calculations on materials. It uses plane wave basis sets to discretize Kohn-Sham equations. It also uses pseudopotentials to replace strong Coulombic nuclei-electron interactions [1].

I assumes that the readers already have some experience with similar electronic structure programs such as Quantum Espresso, ABINIT, VASP, and others.

PWDFT.jl uses Hartree atomic units internally.

2 Installation

You need the following to use `PWDFT.jl`:

- Julia version 0.7 or higher
- LibXC
- SPGLIB
- C compiler to compile LibXC and SPGLIB

PWDFT.jl is a Julia package, so you need to copy (or clone) under Julia's package depot so that it can be used within Julia. Presently, it is not yet a registered package. There are at least two ways of doing this.

The first way is using Julia's package manager. The following command can be issued at the Julia's console.

```
Pkg.add(PackageSpec(url="https://github.com/f-fathurrahman/PWDFT.jl"))
```

[1] You can carry out the calculation with full potential anyway, however this will requires a lot of plane waves (large cutoff energy)

The second way is to use Julia's development directory located at `$HOME/.julia/dev`. To enable this, please put the following in the your `$HOME/.julia/config/startup.jl` file:

```
push!(LOAD_PATH, expanduser("~/.julia/dev"))
```

I usually sym-links my original PWDFT.jl directory to a directory named `PWDFT` (without `.jl` part) under `$HOME/.julia/dev`.

```
ln -fs /path/to/PWDFT.jl ~/.julia/dev/PWDFT
```

To test the installation, you can run the following:

```
using PWDFT
```

at the Julia's console. If there are no error messages then the installation is OK. [2]

2.1 LibXC

PWDFT.jl uses LibXC to calculate exchange correlation energy and potential. The following commands can be used to configure, build, and install LibXC in your computer.

```
./configure --prefix=/home/user/softwares/libxc \
  --disable-fortran --enable-shared
make
make install
```

2.2 SPGLIB

PWDFT.jl uses SPGLIB to generate reduced k-point for integration over Brillouin zone. The following commands can be used to configure, build, and install SPGLIB in your computer.

```
cd spglib-master
mkdir build
cd build
cmake -D CMAKE_INSTALL_PREFIX=/home/user/softwares/spglib
make
make install
```

3 Total energy calculation

We need to prepare several things before using PWDFT.jl:

- Atomic structures: coordinates and unit cell
- Pseudopotentials

This code will initialize a hydrogen molecule (H_2) inside a box of $16 \times 16 \times 16$ bohr.

[2] This does not check the LibXC or SPGLIB installation.

```
atoms = Atoms(
  xyz_string="""
  2

  H      3.83653478       4.23341768       4.23341768
  H      4.63030059       4.23341768       4.23341768
  """,
  LatVecs=gen_lattice_sc(16.0))
```

After creating an instance of `Atoms`, we need to create an instance of Hamiltonian:

```
DIR_PWDFT = joinpath(dirname(pathof(PWDFT)), "..")
DIR_PSP = joinpath(DIR_PWDFT, "pseudopotentials", "pade_gth")
pspfiles = [joinpath(DIR_PSP, "H-q1.gth")]
ecutwfc = 30.0
Ham = Hamiltonian(atoms, pspfiles, ecutwfc)
```

Once the instance of `Hamiltonian is initialized`, we can solve the Kohn-Sham problem by SCF method (or your own solver!), for example:

```
KS_solve_SCF!(Ham)
```

During the call of this function several info will be printed to `stdout`. You can redirect this print out to file if you want to save the print out and analyze it later

```
# save the reference to original stdout
stdout_orig = stdout
# Open file for writing
fileout = open("LOG_"*molname, "w")
# Redirect stdout
redirect_stdout(fileout)
# Solve the Kohn-Sham problem
KS_solve_SCF!(Ham)
# Close IO
close(fileout)
```

`KS_solve_SCF!` has several options that you can used. For more information you can see the original code in `src/KS_solve_SCF.jl`.

Part II

Implementation Notes

In this note, I will describe several user-defined data types that are used throughout PWDFT.jl. Beginners are not expected to know them all, however they are needed to know the internal of PWDFT.jl.

When referring to a file, I always meant to the file contained in the `src/` directory.

Even though it is not strictly required in Julia, I have tried to always use type annotations.

4 Atoms

`Atoms` can be used to represent molecular or crystalline structures. This type is implemented in the file `src/Atoms.jl`.
It has the following fields:

- `Natoms::Int64`: number of atoms present in the system
- `Nspecies::Int64`: number of atomic species present in the system
- `positions::Array{Float64,2}`: An array containing coordinates of atoms in bohr units. Its shape is `(3,Natoms)`
- `atm2species::Array{Int64,1}`: An array containing mapping between atom index to species index. Its shape is `(Natoms,)`
- `atsymbs::Array{String,1}`: An array containing atomic symbols for each atoms. Its shape is `(Natoms,)`.
- `SpeciesSymbols::Array{String,1}`: An array containing unique symbols for each atomic species present in the system. Its shape is `(Nspecies,)`.
- `LatVecs::Array{Float64,2}`: A 3 by 3 matrix describing lattice vectors for unit cell of the system.
- `Zvals::Array{Float64,1}`: An array containing number of (valence) electrons of each atomic species. Its shape is `(Nspecies,)`.

An instance of `Atoms` can be initialized using any of the following ways:

- Using `Atoms` constructor, which has the following signature

```
function Atoms( ;xyz_file="", xyz_string="", xyz_string_frac="",
                in_bohr=false,
                LatVecs=10*[1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0] )
```

- Using `init_atoms_xyz` or `init_atoms_xyz` which takes either a string containing path to xyz file or the content of the xyz file itself. When using this function, one must set `LatVecs` field manually. The `Atoms` constructor mentioned before is actually a wrapper for the functions `init_atoms_xyz` and `init_atoms_xyz_string`.

A note about `Zvals`:
Both `Atoms` and `init_atoms_*` function set `Zvals` to `zeros(Nspecies)`. After passed to `Hamiltonian` constructor `Zvals` will be set according to the pseudopotentials used.
TODO: examples

5 Hamiltonian

An instance `Hamiltonian` is a central object in PWDFT.jl. It is used to store various instances of other important types such as atoms, plane wave grids, pseudopotentials, etc. It is implemented in the file `src/Hamiltonian.jl`.

To create an instance of `Hamiltonian`, we normaly need to provide at least three arguments to the `Hamiltonian` constructor:

- `atoms::Atoms`: an instance of `Atoms`
- `pspfiles::Array{String,1}`: a list of strings specifying the locations of pseudopotentials used in the calculations. Note that, the order should be the same as species ordering of `atoms`, i.e. `pspfiles[isp]` is the path of pseudopotentials of species with symbols `atoms.SpeciesSymbols[isp]`.
- `ecutwfc::Float64`: cutoff energy for wave function expansion using plane wave basis set.

A simplified version of `Hamiltonian` constructor only needs two arguments: `atoms::Atoms` and `ecutwfc::Float64`. In this case, full Coulomb potential will be used. We usually need very large cutoff energy in this case (probably in the order or 10^2 Hartree to obtain good convergence).

The structure of `Hamiltonian` is designed such that we can perform application or multiplication of Hamiltonian to wave function:

```
Hpsi = op_H(H, psi)
```

or, (by overloading the * operator [3])

```
Hpsi = H*psi
```

6 Plane wave basis

The plane wave basis is described by the type `PWGrid`. This type is defined in the file `PWGrid.jl`. It has the following fields:

- `ecutwfc::Float64`:
  cutoff for wave function expansion
- `ecutrho::Float64`:
  cutoff for electron density expansion, for norm-converving pseudopotential we have `ecutrho = 4*ecutwfc`.
- `Ns::Tuple{Int64,Int64,Int64}`:
  parameters defining real-space grid points.
- `LatVecs::Array{Float64,2}`: lattice vectors of unit cell (3×3 matrix)

[3] The operator * is actually implemented as function in Julia

- `RecVecs::Array{Float64,2}`:
  reciprocal lattice vectors (3×3 matrix)
- `CellVolume::Float64`:
  the volume of real-space unit cell
- `r::Array{Float64,2}`:
  real-space grid points. Its shape is `(3,Npoints)`
- `gvec::GVectors`:
  an instance of `GVectors`, for potentials and density expansion
- `gvecw::GVectorsW`:
  an instace of `GVectorsW`, for wave function expansion
- `planfw`:
  FFTW forward plan
- `planbw`:
  FFTW backward plan

The following constructor can be used to create an instance of `PWGrid`:

```
function PWGrid( ecutwfc::Float64,
                 LatVecs::Array{Float64,2};
                 kpoints=nothing )
```

6.1 G-vectors for potentials and densities expansion

G-vectors are described by type `GVectors`. It is defined in the file `PWGrid.jl`. It has the following fields:

- `Ng::Int64`: total number of **G**-vectors
- `G::Array{Float64,2}` The array containing the actual **G**-vectors. Its shape is `(3,Ng)`.
- `G2::Array{Float64,1}`: The array containing magnitude of **G**-vectors. Its shape is `(Ng,)`.
- `idx_g2r::Array{Int64,1}` The array containing mapping between **G**-vectors to real space grid points. Its shape is `(Ng,)`.

The following function is used as the constructor:

```
function init_gvec( Ns, RecVecs, ecutrho )
```

This function takes the following arguments

- `Ns`: a tuple of three `Int64`'s specifying sampling points along the 1st, 2nd, and 3rd lattice vector directions.
- `RecVecs`: 3 by 3 matrix describing reciprocal lattice vectors
- `ecutrho`: cutoff energy (in Hartree). For norm-conserving PP, it is 4 times `ecutwfc`.

6.2 G-vectorsW

The **G**-vectors for wave function expansion is described by the type `GVectorsW`. They are a subset of `GVectors`. It has the following fields:

- `Ngwx::Int64`: maximum number of **G**-vectors for all kpoints.
- `Ngw::Array{Int64,1}`: number of **G**-vectors for each kpoints
- `idx_gw2g::Array{Array{Int64,1},1}`: mapping between indices of `GVectorsW` to indices of `GVectors`.
- `idx_gw2r::Array{Array{Int64,1},1}`: mapping between indices of `GVectorsW` to indices of real space grid points `PWGrid.r`
- `kpoints::KPoints`: an instance of `KPoints`

Constructor: TODO

6.3 Bloch wave vector

The type describing Bloch wave vectors, or commonly referred to as k-points, is `KPoints`. It is defined in the file `KPoints.jl`. It has the following fields

- `Nkpt::Int64`: total number of k-points.
- `k::Array{Float64,2}`: the actual k-points. Its shape is `(3,Nkpt)`.
- `wk::Array{Float64,1}`: the weight of each k-points needed for integration over Brillouin zone
- `RecVecs::Array{Float64,2}`: a copy of `PWGrid.RecVecs`.

Monkhorst-Pack grid points:

$$k_i = \frac{2n_i - N_i - 1}{2N_i}\mathbf{b}_i. \quad i = 1, 2, 3 \tag{1}$$

where $n_i = 1, 2, ..., N_i$ size $= (N_1, N_2, N_3)$ and the $\mathbf{b}_i$'s are reciprocal lattice vectors.
Constructor:

```
function KPoints( atoms::Atoms, mesh::Array{Int64,1},
                  is_shift::Array{Int64,1};
                  time_reversal=1 )
```

SPGLIB is used to generate reduced k-points within the first Brillouin zone.

7 Wave functions

In the present implementation, I have chosen to use the simplest data structure for representing wave functions, namely `Array{ComplexF64,2}`. In PWDFT.jl, an alias has been defined:

```
const Wavefunc = Array{ComplexF64,2}
```

I also differentiate between wave function and Bloch wave function. A Bloch wave function is composed of several wave functions having different Bloch wave vectors $\mathbf{k}$ or k-points. The following alias also has been defined

```
const BlochWavefunc = Array{Array{ComplexF64,2},1}
```

Spin index is merged with kpoint index

Conventions: `psi` for an instance of `Wavefunc` and `psiks` for an instance of `BlochWavefunc`

Functions related to wave functions:

- `rand_Wavefunc`
- `zeros_Wavefunc`
- `rand_BlochWavefunc`
- `zeros_BlochWavefunc`

Wave functions in real and reciprocal space.

Converting between wave functions in real and reciprocal space

Orthonormalization functions `ortho_gram_schmidt` and `ortho_sqrt`

8 Total energy components

Total energy components are stored in the type `Energies`. Currently its fields are as follows.

- `Kinetic`: kinetic energy
- `Ps_loc`: local pseudopotential energy
- `Ps_nloc`: nonlocal pseudopotential energy
- `Hartree`: classical electrostatic energy
- `XC`: exchange correlation energy
- `NN`: nuclear-nuclear (repulsive) interaction energy
- `PspCore`: core (screened) pseudopotential energy
- `mTS`: electronic entropy contribution (with minus sign). This is used for calculation with partial occupations such as metals.

9 Local potentials

Various local potentials are stored in the type `Potentials`. Currently its fields are as follows.

- `Ps_loc` local pseudopotential components. Its shape is `(Npoints,)`.
- `Hartree`: classical electrostatic potential. Its shape is `(Npoints,)`.
- `XC`: exchange-correlation potential. This potential can be spin dependent so its shape is `(Npoints,Nspin)`.

All of these potentials are stored in the real space representation.

10 Pseudopotentials

Currently, only GTH pseudopotentials with no core-correction are supported. The type for handling GTH pseudopotential for a species is `PsPot_GTH`. It is declared as follows.

```
struct PsPot_GTH
    pspfile::String
    atsymb::String
    zval::Int64
    rlocal::Float64
    rc::Array{Float64,1}
    c::Array{Float64,1}
    h::Array{Float64,3}
    lmax::Int64
    Nproj_l::Array{Int64,1}
    rcut_NL::Array{Float64,1}
end
```

11 Hamiltonian operator

Apply Hamiltonian operator to wave function

```
for ispin = 1:Nspin
    for ik = 1:Nkpt
        # Important: need to set internal variables within ::Hamiltonian
        Ham.ispin = ispin
        Ham.ik = ik
        Hpsiks = op_H(Ham, psiks)
    end
end
```

12 Eigensolver

Three methods are available:

- LOBPCG method

- blocked Davidson method
- preconditioned conjugate gradient by minizing trace of Hamiltonian

These eigensolvers can be used to find eigenvalues and eigenvectors of Hamiltonian at fixed potential.

13 Solving Kohn-Sham problems

Two main algorithms are available:

- self-consistent field (SCF)
- direct minimization of Kohn-Sham total energy functional

The solvers of Kohn-Sham problems start with prefix `KS_solve_`. Currently implemented solvers are

- `KS_solve_SCF`
- `KS_solve_Emin_PCG`
- `KS_solve_DCM`
- `KS_solve_TRDCM`

13.1 Self-consistent field

`KS_solve_SCF`

13.2 Direct energy minimization via nonlinear conjugate gradient method

`KS_solve_Emin_PCG`

This method is described by Ismail-Beigi and Arias

Presently only working for systems with band gap (insulators and semiconductors).

13.3 Direct constrained minimization method of Yang (experimental feature)

`KS_solve_DCM`

13.4 Chebyshev filtered subspace iteration SCF (experimental feature)

```
update_psi="CheFSI"
```

Part III

Howtos

Initializing `Atoms`

From xyz file: supply the path to xyz file as string and set the lattice vectors:

```
atoms = Atoms(xyz_file="file.xyz", LatVecs=gen_lattice_sc(16.0))
```

In extended xyz file, the lattice vectors information is included (along with several others information, if any):

```
atoms = Atoms(ext_xyz_file="file.xyz")
```

For crystalline systems, using keyword argument `xyz_string_frac` is sometimes convenient:

```
atoms = Atoms(xyz_string_frac=
        """
        2

        Si  0.0  0.0  0.0
        Si  0.25  0.25  0.25
        """, in_bohr=true,
        LatVecs=gen_lattice_fcc(10.2631))
```

IMPORTANT We need to be careful to also specify `in_bohr` keyword to get the correct coordinates in bohr (which is used internally in `PWDFT.jl`).

Referring or including files in `sandbox` (or other dirs in `PWDFT.jl`)

```
using PWDFT
const DIR_PWDFT = joinpath(dirname(pathof(PWDFT)),"..")
const DIR_PSP = joinpath(DIR_PWDFT,"pseudopotentials","pade_gth")
const DIR_STRUCTURES = joinpath(DIR_PWDFT, "structures")

pspfiles = [joinpath(DIR_PSP,"Ag-q11.gth")]
```

Generating lattice vectors

Lattice vectors are simply 3x3 array. We can set it manually or use one of functions defined in `gen_lattice_pwscf.jl`. for generating lattice vectors for Bravais lattices that used in Quantum ESPRESSO's PWSCF.

Using Babel to generate xyz file from SMILES

```
babel file.smi file.sdf
babel file.sdf file.xyz
```

Use `babel -h` to autogenerate hydrogens.

Setting up pseudopotentials

One can use the function `get_default_psp(::Atoms)` to get default pseudopotentials set for a given instance of `Atoms`.
Currently, it is not part of main `PWDFT.jl` package. It is located under `sandbox` subdirectory of `PWDFT.jl` distribution.

```
using PWDFT

DIR_PWDFT = jointpath(dirname(pathof(PWDFT)),"..")
include(jointpath(DIRPWDFT,"sandbox","get_default_psp.jl"))

atoms = Atoms(ext_xyz_file="atoms.xyz")
pspfiles = get_default_psp(atoms)
```

Alternatively, one can set `pspfiles` manually because it is simply an array of `String`:

```
pspfiles = ["Al-q3.gth", "O-q6.gth"]
```

IMPORTANT Be careful to set the order of species to be same as `atoms.SpeciesSymbols`. For example, if

```
atoms.SpeciesSymbols = ["Al", "O", "H"]
```

then

```
pspfiles = ["Al-q3.gth", "O-q6.gth", "H-q1.gth"]
```

Initializing Hamiltonian

For molecular systems:

```
Ham = Hamiltonian( atoms, pspfiles, ecutwfc )
```

For insulator and semiconductor solids:

```
Ham = Hamiltonian( atoms, pspfiles, ecutwfc, meshk=[3,3,3] )
```

For metallic systems:

```
Ham = Hamiltonian( atoms, pspfiles, ecutwfc, meshk=[3,3,3], extra_states=4 )
```

Empty extra states can be specified by using `extra_states` keyword.
For spin-polarized systems, `Nspin` keyword can be used.

Calculating electron density

Several ways:

```
Rhoe = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )
# or
Rhoe = calc_rhoe( Ham, psiks )
# or
calc_rhoe!( Ham, psiks, Rhoe )
```

Read and write array (binary file)

Write to binary files:

```
for ikspin = 1:Nkpt*Nspin
    wfc_file = open("WFC_ikspin_"*string(ikspin)*".data","w")
    write( wfc_file, psiks[ikspin] )
    close( wfc_file )
end
```

Read from binary files:

```
psiks = BlochWavefunc(undef,Nkpt)
for ispin = 1:Nspin
for ik = 1:Nkpt
    ikspin = ik + (ispin-1)*Nkpt
    # Don't forget to use read mode
    wfc_file = open("WFC_ikspin_"*string(ikspin)*".data","r")
    psiks[ikspin] = Array{ComplexF64}(undef,Ngw[ik],Nstates)
    psiks[ikspin] = read!( wfc_file, psiks[ikspin] )
    close( wfc_file )
end
end
```

Subspace rotation

In case need sorting:

```
Hr = psiks[ikspin]' * op_H( Ham, psiks[ikspin] )
evals, evecs = eigen(Hr)
evals = real(evals[:])

# Sort in ascending order based on evals
idx_sorted = sortperm(evals)

# Copy to Hamiltonian
Ham.electrons.ebands[:,ikspin] = evals[idx_sorted]

# and rotate
psiks[ikspin] = psiks[ikspin]*evecs[:,idx_sorted]
```

Status

28 May 2018 The following features are working now:

- LDA and GGA, spin-paired and spin polarized calculations
- Calculation with k-points (for periodic solids). SPGLIB is used to reduce the Monkhorst-Pack grid points for integration over Brillouin zone.

Band structure calculation is possible in principle as this can be done by simply solving Schrodinger equation with converged Kohn-Sham potentials, however there is currently no tidy script or function to do that.

Total energy result for isolated systems (atoms and molecules) agrees quite well with ABINIT and PWSCF results.

~~Total energy result for periodic solid is quite different from ABINIT and PWSCF. I suspect that this is related to treatment of electrostatic terms in periodic system.~~

These discrepancies have been minimized. For several systems the agreement is very good even though I did not use the same algorithm as ABINIT.

~~SCF is rather shaky for several systems, however it is working in quite well in nonmetallic system.~~

SCF stability has been improved with Pulay mixing and its variants.