PWDFT.jl Documentation

Fadjar Fathurrahman

Contents

Part I. User Guide

1 Overview

PWDFT.jl uses Hartree atomic units.

2 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.

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

Part II. Implementation Notes

In this note, I will describe several user-defined 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 tried to always use type annotations.

3 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] )
  ```

- The above mentioned `Atoms` constructor is actually a wrapper for the functions `init_atoms_xyz` and `init_atoms_xyz_string` 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.

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

4 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 [1])

```
Hpsi = H*psi
```

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

5 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: `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)
- `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 instace of `GVectors`: for potentials and density expansion
- `gvecw::GVectorsW`: an instace of `GVectorsW`, for wave function expansion
- `planfw::FFTW.cFFTWPlan{Complex{Float64},-1,false,3}`: FFTW forward plan
- `planbw::AbstractFFTs.ScaledPlan{ComplexF64,FFTW.cFFTWPlan{ComplexF64,1,false,3- Float64}}`: 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 )
```

5.1 G-vectors for potentials and densities expansion

G-vectors are described by type ‘GVectors‘. It is defined in 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'.

5.2 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

5.3 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 internally to generate reduce number of k-points.

6 Wave functions

Using ‘`Array{ComplexF64,2}`.
General wavefunction on kpoints

7 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.

8 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 in the real space representation.

9 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
```

10 Solving Kohn-Sham problems

Two main algorithms: SCF and direct minimization
Function names: `KS_solve_*`

10.1 Self-consistent field

`KS_solve_SCF`

10.2 Eigensolver

Davidson, LOBPCG, and PCG

10.3 Direct energy minimization via nonlinear conjugate gradient method

`KS_solve_Emin_PCG`
This method is described by IsmailBeigi-Arias

10.4 Direct constrained minimization method of Yang

`KS_solve_DCM`
Chebyshev filtered subspace iteration SCF
`update_psi="CheFSI"`