

# ***PWDFT.jl***

## ***Electronic Structure Calculations with Julia***

---

**Fadjar Fathurrahman**

*Program Studi Teknik Fisika*

*Fakultas Teknologi Industri*

*Institut Teknologi Bandung*

*E-mail:* [fadjar.fathurrahman@gmail.com](mailto:fadjar.fathurrahman@gmail.com), [fadjar@tf.itb.ac.id](mailto:fadjar@tf.itb.ac.id)

---

# Contents

<b>I</b>	<b>Introduction</b>	<b>1</b>
1	Installation	1
2	Overview	4
<b>II</b>	<b>Implementation</b>	<b>6</b>
3	Atomic structure	6
4	Plane wave basis set, real space grid, and k-points	7
5	Electronic states	10
6	Potentials and energies	10
6.1	Electron density	11
6.2	Kinetic energy	11
6.3	Local and nonlocal pseudopotential energy	11
6.4	Hartree energy	11
6.5	XC energy and potential	12
6.5.1	Calculation of $E_{xc}$ in PWDFT.jl using Libxc	12
6.5.2	Calculation of $V_{xc}$ in PWDFT.jl using Libxc	14
7	Pseudopotentials	17
7.1	Local pseudopotential	18
7.2	Nonlocal pseudopotential	18
8	Hamiltonian operators	20
8.1	Iterative diagonalization of Hamiltonian	20
9	Self-consistent field	20
10	Direct minimization	21
11	Calculation of atomic forces	21

<b>III</b>	<b>Other things</b>	<b>22</b>
<b>A</b>	<b>Howtos</b>	<b>22</b>
A.1	Referring or including files in sandbox (or other dirs in <code>PWDFT.jl</code> )	22
A.2	Using Babel to generate xyz file from SMILES	22
A.3	Setting up pseudopotentials	22
A.4	Initializing Hamiltonian	23
A.5	Iterative diagonalization of Hamiltonian	23
A.6	Calculating electron density	23
A.7	Read and write array (binary file)	23

---

# Introduction

PWDFT.jl is a package to solve electronic structure problems based on density functional theory (DFT) and Kohn-Sham equations. It is written in Julia programming language.

The Kohn-Sham orbitals are expanded using plane wave basis. This basis set is very popular within solid-state community and is also used in several electronic structure packages such as Quantum ESPRESSO, ABINIT, VASP, etc.

Features

- Total energy calculation of molecules, surfaces, and crystalline system within periodic unit cell (however, no corrections are implemented for non-periodic systems yet).
- SCF with electron density mixing (for semiconducting and metallic systems)
- Direct minimization method using conjugate gradient (for semiconducting systems)
- GTH pseudopotentials (included in the repository)
- LDA-VWN and GGA-PBE functionals (via 'Libxc.jl')

## 1 Installation

Requirements

- [Julia: <https://julialang.org/downloads> version  $\geq 0.7$ , with the following packages installed: - FFTW - SpecialFunctions - Libxc (a wrapper to Libxc <https://gitlab.com/libxc/libxc> - LibSymspg (a wrapper to spglib <https://github.com/atztogo/spglib>)

These packages are registered so they can be installed by using Julia's package manager.

```
using Pkg
Pkg.add("FFTW")
Pkg.add("SpecialFunctions")
Pkg.add("Libxc")
Pkg.add("LibSymspg")
```

These packages should be automatically installed 'PWDFT.jl' is installed as local package (see below).

Currently, this package is not yet registered. So, 'Pkg.add("PWDFT")' will not work (yet).

We have several alternatives:

1. Using Julia's package manager to install directly from the repository URL:

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

2. Using Julia development directory. We will use  $\$HOME/.julia/dev$  for this. To enable  $\$HOME/.julia/dev$  directory, we need to modify the Julia's  $LOAD\_PATH$  variable. Add the following line in your  $\$HOME/.julia/config/startup.jl$ .

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

After this has been set, you can download the package as zip file (using Github) or clone this repository to your computer.

If you download the zip file, extract the zip file under  $\$HOME/.julia/dev$ . You need to rename the extracted directory to 'PWDFT' (with no 'jl' extension).

Alternatively, create symlink under  $\$HOME/.julia/dev$  to point to you cloned (or extracted) 'PWDFT.jl' directory. The link name should not contain the 'jl' part. For example:

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

3. Install PWDFT.jl as local package. Firstly, get into Pkg's REPL mode by tapping ']', and activate a independent environment 'activate .'.

Install the PWDFT.jl package in this environment:

```
(PWDFT) pkg> develop <path/to/PWDFT.jl>
```

To make sure that the package is installed correctly, you can load the package and verify that there are no error messages during precompilation step. You can do this by typing the following in the Julia console.

```
using PWDFT
```

Change directory to examples/Si\_fcc and run the following in the terminal.

```
julia run.jl
```

The above command will calculate total energy of hydrogen atom by SCF method.

The script will calculate total energy per unit cell of silicon crystal using self-consistent field iteration and direct energy minimization.

Units

'PWDFT.jl' internally uses Hartree atomic units (energy in Hartree and length in bohr).

A simple work flow

- create an instance of 'Atoms':

```
atoms = Atoms(xyz_file="CH4.xyz", LatVecs=gen_lattice_sc(16.0))
```

- create an instance of 'Hamiltonian':

```
ecutwfc = 15.0 # in Hartree
pspfiles = ["../pseudopotentials/pade_gth/C-q4.gth",
            "../pseudopotentials/pade_gth/H-q1.gth"]
Ham = Hamiltonian( atoms, pspfiles, ecutwfc )
```

- solve the Kohn-Sham problem

```
KS_solve_SCF!( Ham, betamix=0.2 ) # using SCF (self-consistent field) method
# or
KS_solve_Emin_PCG!( Ham ) # direct minimization using preconditioned conjugate
↪ gradient
```

More examples on creating an instance of Atoms

GaAs crystal (primitive unit cell), using keyword xyz\_string\_frac:

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

    Ga  0.0  0.0  0.0
    As  0.25 0.25 0.25
    """,
    in_bohr=true,
    LatVecs = gen_lattice_fcc(10.6839444516)
)
```

Hydrazine molecule in extended xyz file

```
atoms = Atoms(ext_xyz_file="N2H4.xyz")
```

with the following 'N2H4.xyz' file (generated using [ASE](https://wiki.fysik.dtu.dk/ase/)):

```
6
Lattice="11.896428 0.0 0.0 0.0 12.185504 0.0 0.0 0.0 11.151965"
↳ Properties=species:S:1:pos:R:3:Z:I:1 pbc="T T T"
N      5.94821400      6.81171100      5.22639100      7
N      5.94821400      5.37379300      5.22639100      7
H      6.15929600      7.18550400      6.15196500      1
H      5.00000000      7.09777800      5.00000000      1
H      5.73713200      5.00000000      6.15196500      1
H      6.89642800      5.08772600      5.00000000      1
```

Lattice vectors information is taken from the xyz file.

More examples on creating an instance of 'Hamiltonian'

Using 3x3x3 Monkhorst-Pack kpoint grid (usually used for crystalline systems):

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

Include 4 extra states:

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

Using spin-polarized ('Nspin=2 '):

```
Ham = Hamiltonian( atoms, pspfiles, ecutwfc, meshk=[3,3,3],
    Nspin=2, extra_states=4 )
```

NOTES: Currently spin-polarized calculations are only supported by specifying calculations with smearing scheme (no fixed magnetization yet), so `extra_states` should also be specified.

Using PBE exchange-correlation functional:

```
Ham = Hamiltonian( atoms, pspfiles, ecutwfc, meshk=[3,3,3],
    Nspin=2, extra_states=4, xcfunc="PBE" )
```

Currently, only two XC functional is supported, namely 'xcfunc="VWN"' (default) and 'xcfunc="PBE"'. Future developments should support all functionals included in LibXC.

More examples on solving the Kohn-Sham problem

Several solvers are available:

- `KS_solve_SCF!`: SCF algorithm with density mixing
- `KS_solve_SCF_potmix!`: SCF algorithm with XC and Hartree potential mixing
- `KS_solve_Emin_PCG!`: using direct total energy minimization by preconditioned conjugate gradient method (proposed by Prof. Arias, et al.). Only the version which works with systems with band gap is implemented.

Stopping criteria is based on difference in total energy.

The following example will use `Emin_PCG`. It will stop if the difference in total energy is less than `etot_conv_thr` and it occurs twice in a row.

```
KS_solve_Emin_PCG!( Ham, etot_conv_thr=1e-6, NiterMax=150 )
```

Using SCF with `betamix` (mixing parameter) 0.1:

```
KS_solve_SCF!( Ham, betamix=0.1 )
```

Smaller ‘betamix’ usually will lead to slower convergence but more stable. Larger ‘betamix’ will give faster convergence but might result in unstable SCF.

Several mixing methods are available in `KS_solve_SCF!`:

- ‘simple’ or linear mixing
- `linear_adaptive`
- ‘anderson’
- ‘broyden’
- ‘pulay’
- ‘ppulay’: periodic Pulay mixing
- ‘rpulay’: restarted Pulay mixing

For metallic system, we use Fermi smearing scheme for occupation numbers of electrons. This is activated by setting `use_smearing=true` and specifying a small smearing parameter ‘kT’ (in Hartree, default ‘kT=0.001’).

```
KS_solve_SCF!( Ham, mix_method="rpulay", use_smearing=true, kT=0.001 )
```

Citation

- Fadjat Fathurrahman, Mohammad Kemal Agusta, Adhitya Gandaryus Saputro, Hermawan Kresno Dipojono PWDFT.jl : A Julia package for electronic structure calculation using density functional theory and plane wave basis](<https://doi.org/10.1016/j.cpc.2020.107372>). Comp. Phys. Comm. **256** 107372 (2020).

**This document is a work in progress**

In this part I will describe my design choices in implementing PWDFT.jl. This design is by no means perfect and it might change in the future to accomodate more complex use cases.

## 2 Overview

The design of PWDFT.jl is intended to be rather simple. One constraint that is set to the code is that it should be possible to perform application of Hamiltonian operator to wave function as simple as:

```
Hpsi = Ham*psi # or  
Hpsi = op_H(Ham, psi)
```

where `psi` is, currently, of type `Array{ComplexF64,2}`<sup>1</sup>. This comes with an important consequences: all other pieces of information about how this operation is done should be present in the type of `Ham`.<sup>2</sup>

In PWDFT.jl, the type of `Ham` is `Hamiltonian`. Several important fields of `Hamiltonian` are instances of the following types (please refer to the source code for more details about this):

- `Atoms`: contains information about atomic structure: cell vectors, atomic species and atomic coordinates.
- `PSPot_GTH`: contains information about atomic pseudopotentials.
- `Electrons`: contains information about electronic states.
- `PWGrid`: contains information about plane wave basis set.

---

<sup>1</sup>This function may be extended take other types other than plain Julia array for more complex case.

<sup>2</sup>We will also see some quirks related to this design choice later, such as applying Hamiltonian to several k-points or spin-polarized case

- Potentials: contains information about local potentials such as local pseudopotential, Hartree and exchange-correlation potential.
- PsPotNL: contains information about nonlocal pseudopotential terms.
- Energies: contains information about components of Kohn-Sham energy.
- SymmetryInfo: contains information about symmetry operations.



# Implementation

## 3 Atomic structure

The type `Atoms` contains the following information:

- Number of atoms: `Natoms::Int64`
- Number of atomic species: `Nspecies::Int64`
- Atomic coordinates: `positions::Array{Float64,2}`
- Unit cell vectors (lattice vectors): `LatVecs::Array{Float64,2}`

`Atoms` also contains several other fields such as `Zvals` which will be set according to the pseudopotentials assigned to the instance of `Atoms`.<sup>3</sup>

### Atoms struct definition

```
mutable struct Atoms
  Natoms::Int64
  Nspecies::Int64
  positions::Array{Float64,2}
  atm2species::Array{Int64,1}
  atsyms::Array{String,1}
  SpeciesSymbols::Array{String,1}
  LatVecs::Array{Float64,2}
  Zvals::Array{Float64,1}
end
```

**Figure 1.** Definition of `Atoms` struct which contains variables needed to describe crystalline of molecular structure.

`LatVecs` is a  $3 \times 3$  matrix. The vectors are stored column-wise which is opposite to the PWSCF input convention. Several convenience functions to generate lattice vectors for Bravais lattices are provided in `PWDFT.jl`. These functions adopt PWSCF definition. Some examples are listed below.

- `gen_lattice_sc` or `gen_lattice_cubic` for generating simple cubic lattice vectors.
- `gen_lattice_fcc`: for fcc structure
- `gen_lattice_bcc`: for bcc structure
- `gen_lattice_hcp`: for hcp structure

Please see file `gen_lattice.jl` for more information.

There are several ways to initialize an instance of `Atoms`. The following are typical cases.

- From xyz file. We need to 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))
```

<sup>3</sup>Not all fields of `Atoms` (or any custom types defined in `PWDFT.jl`) are listed. The most up to date definition can be consulted in the corresponding source code.

- 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`).

- From extended xyz file, the lattice vectors information is included along with several others information, if any, however they are ignored):

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

## 4 Plane wave basis set, real space grid, and k-points

The type `PWGrid` wraps various variables related to plane wave basis set. This has two fields of type `GVectors` and `GVectorsW` for storing information about G-vectors that are used in potential and wave functions, respectively.

### PWGrid struct definition

```
struct PWGrid
    ecutwfc::Float64
    ecutrho::Float64
    Ns::Tuple{Int64,Int64,Int64}
    LatVecs::Array{Float64,2}
    RecVecs::Array{Float64,2}
    CellVolume::Float64
    gvec::GVectors
    gvecw::GVectorsW
    planfw
    planbw
end
```

**Figure 2.** Definition of `PWGrid`. The type annotation of `planfw` and `planbw` is omitted because they are too long.

We can define grid points over unit cell as:

$$\mathbf{r} = \frac{i}{N_{s1}} \mathbf{a}_1 + \frac{j}{N_{s2}} \mathbf{a}_2 + \frac{k}{N_{s3}} \mathbf{a}_3$$

where  $i = 0, 1, \dots, N_{s1} - 1$ ,  $j = 0, 1, \dots, N_{s2} - 1$ ,  $k = 0, 1, \dots, N_{s3} - 1$

#### GVectors struct definition

```

struct GVectors
  Ng::Int64
  G::Array{Float64,2}
  G2::Array{Float64,1}
  idx_g2r::Array{Int64,1}
  G2_shells::Array{Float64,1}
  idx_g2shells::Array{Int64,1}
end

```

Figure 3. Definition of GVectors.

#### GVectorsW struct definition

```

struct GVectorsW
  Ngwx::Int64
  Ngw::Array{Int64,1}
  idx_gw2g::Array{Array{Int64,1},1}
  idx_gw2r::Array{Array{Int64,1},1}
  kpoints::KPoints
end

```

Figure 4. Definition of GVectorsW.

The G-vectors can be defined as:

$$\mathbf{G} = n_1 \mathbf{b}_1 + n_2 \mathbf{b}_2 + n_3 \mathbf{b}_3 \quad (4.1)$$

where  $n_1, n_2, n_3$  are integer numbers and  $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$  are three vectors describing unit cell of reciprocal lattice or *unit reciprocal lattice vectors*. They satisfy the following relations:

$$\mathbf{a}_1 = 2\pi \frac{\mathbf{a}_2 \times \mathbf{a}_3}{\Omega} \mathbf{a}_2 = 2\pi \frac{\mathbf{a}_3 \times \mathbf{a}_1}{\Omega} \mathbf{a}_3 = 2\pi \frac{\mathbf{a}_1 \times \mathbf{a}_2}{\Omega} \quad (4.2)$$

A periodic function

$$f(\mathbf{r}) = f(\mathbf{r} + \mathbf{L}), \quad \mathbf{L} = n_1 \mathbf{a}_1 + n_2 \mathbf{a}_2 + n_3 \mathbf{a}_3 \quad (4.3)$$

can be expanded using plane wave basis functions as:

$$f(\mathbf{r}) = \frac{1}{\sqrt{\Omega}} \sum_{\mathbf{G}} C_{\mathbf{G}} \exp(i\mathbf{G} \cdot \mathbf{r}) \quad (4.4)$$

where  $C_{\mathbf{G}}$  are expansion coefficients. This sum is usually truncated at a certain maximum value of G-vector,  $G_{\max}$ .

Kohn-Sham wave function:

$$\psi_{i,\mathbf{k}}(\mathbf{r}) = u_{i,\mathbf{k}}(\mathbf{r}) \exp[i\mathbf{k} \cdot \mathbf{r}] \quad (4.5)$$

where  $u_{i,\mathbf{k}}(\mathbf{r}) = u_{i,\mathbf{k}}(\mathbf{r} + \mathbf{L})$

Using plane wave expansion:

$$u_{i,\mathbf{k}}(\mathbf{r}) = \frac{1}{\sqrt{\Omega}} \sum_{\mathbf{G}} C_{i,\mathbf{k},\mathbf{G}} \exp(i\mathbf{G} \cdot \mathbf{r}), \quad (4.6)$$

we have:

$$\psi_{i,\mathbf{k}}(\mathbf{r}) = \frac{1}{\sqrt{\Omega}} \sum_{\mathbf{G}} C_{i,\mathbf{G}+\mathbf{k}} \exp [i(\mathbf{G} + \mathbf{k}) \cdot \mathbf{r}] \quad (4.7)$$

With this expression we can expand electronic density in plane wave basis:

$$\begin{aligned} \rho(\mathbf{r}) &= \sum_i \int f_{i,\mathbf{k}} \psi_{i,\mathbf{k}}^*(\mathbf{r}) \psi_{i,\mathbf{k}}(\mathbf{r}) d\mathbf{k} \\ &= \frac{1}{\Omega} \sum_i \int f_{i,\mathbf{k}} \left( \sum_{\mathbf{G}'} C_{i,\mathbf{G}'+\mathbf{k}} \exp [-i(\mathbf{G}' + \mathbf{k}) \cdot \mathbf{r}] \right) \left( \sum_{\mathbf{G}} C_{i,\mathbf{G}+\mathbf{k}} \exp [i(\mathbf{G} + \mathbf{k}) \cdot \mathbf{r}] \right) d\mathbf{k} \\ &= \frac{1}{\Omega} \sum_i \int f_{i,\mathbf{k}} \sum_{\mathbf{G}} \sum_{\mathbf{G}'} C_{i,\mathbf{G}+\mathbf{k}} C_{i,\mathbf{G}'+\mathbf{k}} \exp [i(\mathbf{G} - \mathbf{G}') \cdot \mathbf{r}] d\mathbf{k} \\ &= \frac{1}{\Omega} \sum_{\mathbf{G}''} C_{\mathbf{G}''} \exp [i\mathbf{G}'' \cdot \mathbf{r}] d\mathbf{k} \end{aligned}$$

The sum over  $\mathbf{G}''$  extends twice the range over the range needed by the wave function expansion.

For wave function expansion we use plane wave expansion over  $\mathbf{G}$  vectors defined by:

$$\frac{1}{2} |\mathbf{G} + \mathbf{k}|^2 \leq E_{\text{cut}} \quad (4.8)$$

where  $E_{\text{cut}}$  is a given cutoff energy which corresponds to `ecutwfc` field of `PWGrid`. For electronic density (and potentials) we have:

$$\frac{1}{2} \mathbf{G}^2 \leq 4E_{\text{cut}} \quad (4.9)$$

The value of  $4E_{\text{cut}}$  corresponds to `ecutrho` field of `PWGrid`.

In the implementation, we first generate a set of  $\mathbf{G}$ -vectors which satisfies Equation (4.9) and derives several subsets from it which satisfy Equation (4.8) for a given  $\mathbf{k}$ -points.

An instance of `PWGrid` can be initialized by using its constructor which has the following signature:

```
function PWGrid( ecutwfc::Float64, LatVecs::Array{Float64,2};
    kpoints=nothing, Ns_=(0,0,0) )
```

There are two mandatory arguments: `ecutwfc` and `LatVecs`. `ecutwfc` is cutoff energy for kinetic energy (in Hartree) and `LatVecs` is usually correspond to the one used in an instance of `Atoms`.

Structure factor for atomic species  $I_s$  is calculated as

$$S_f(\mathbf{G}, I_s) = \sum_I \exp [\mathbf{G} \cdot \mathbf{R}_{I_s}] \quad (4.10)$$

where the summation is done over all atoms of species  $I_s$ .

Fast Fourier transforms are used to change the representation of a quantity from real space to reciprocal space and *vice versa*. They are:

- `R_to_G`
- `G_to_R`

and also their inplace counterparts (`R_to_G!` and `G_to_R!`).

operators `op nabla op nabla 2`

The `KPoints` struct stores variables related to  $\mathbf{k}$ -points list.

#### KPoints struct definition

```
struct KPoints
  Nkpt::Int64
  mesh::Tuple{Int64,Int64,Int64}
  k::Array{Float64,2}
  wk::Array{Float64,1}
  RecVecs::Array{Float64,2}
end
```

Figure 5. Definition of KPoints.

For a total energy calculation, the list of **k**-points is generated using Monkhorst-Pack scheme.

## 5 Electronic states

#### Electrons struct definition

```
mutable struct Electrons
  Nelectrons::Float64
  Nstates::Int64
  Nstates_occ::Int64
  Focc::Array{Float64,2}
  ebands::Array{Float64,2}
  Nspin::Int64
end
```

Figure 6. Definition of Electrons.

## 6 Potentials and energies

Total energy per unit cell system  $E_{\text{total}}^{\text{KS}}$  can be written as

$$E_{\text{total}}^{\text{KS}} = E_{\text{kin}} + E_{\text{ele-nuc}} + E_{\text{Ha}} + E_{\text{xc}} + E_{\text{nuc-nuc}} \quad (6.1)$$

Kohn-Sham equations:

$$H_{\text{KS}}\psi_{ik}(\mathbf{r}) = \epsilon_{ik}\psi_{ik}(\mathbf{r}) \quad (6.2)$$

#### Potentials struct definition

```
mutable struct Potentials
  Ps_loc::Array{Float64,1}
  Hartree::Array{Float64,1}
  XC::Array{Float64,2}
  Total::Array{Float64,2}
end
```

Figure 7. Definition of Potentials.

### Energies struct definition

```
mutable struct Energies
  Kinetic::Float64
  Ps_loc::Float64
  Ps_nloc::Float64
  Hartree::Float64
  XC::Float64
  NN::Float64
  PspCore::Float64
  mTS::Float64
end
```

Figure 8. Definition of Energies.

## 6.1 Electron density

Electron density  $\rho(\mathbf{r})$  is calculated as:

$$\rho(\mathbf{r}) = \sum_{i=1}^{N_{\text{occ}}} f_i \psi_i^*(\mathbf{r}) \psi_i(\mathbf{r}) \quad (6.3)$$

Function: `calc_rhoe!` and `calc_rhoe`.

## 6.2 Kinetic energy

Kinetic energy:

$$E_{\text{kin}} = -\frac{1}{2} \sum_{\mathbf{k}} \sum_i w_{\mathbf{k}} f_{i\mathbf{k}} \int_{\Omega} \psi_{i\mathbf{k}}^*(\mathbf{r}) \nabla^2 \psi_{i\mathbf{k}}(\mathbf{r}) d\mathbf{r} \quad (6.4)$$

In reciprocal space:

$$E_{\text{kin}} = \frac{1}{2} \sum_{\mathbf{k}} \sum_{i=1}^{N_{\text{occ}}} w_{\mathbf{k}} f_{i\mathbf{k}} \sum_{\mathbf{G}} |\mathbf{G} + \mathbf{k}|^2 |c_{i, \mathbf{G}+\mathbf{k}}|^2 \quad (6.5)$$

## 6.3 Local and nonlocal pseudopotential energy

The local pseudopotential contribution is

$$E_{\text{loc}}^{\text{PS}} = \int_{\Omega} \rho(\mathbf{r}) V_{\text{loc}}^{\text{PS}}(\mathbf{r}) d\mathbf{r} \quad (6.6)$$

and the non-local contribution is

$$E_{\text{nloc}}^{\text{PS}} = \sum_{\mathbf{k}} \sum_i w_{\mathbf{k}} f_{i\mathbf{k}} \int_{\Omega} \psi_{i\mathbf{k}}^*(\mathbf{r}) \left[ \sum_I \sum_{l=0}^{l_{\text{max}}} V_l^{\text{PS}}(\mathbf{r} - \mathbf{R}_I, \mathbf{r}' - \mathbf{R}_I) \right] \psi_{i\mathbf{k}}(\mathbf{r}) d\mathbf{r}. \quad (6.7)$$

## 6.4 Hartree energy

$$E_{\text{Ha}} = \frac{1}{2} \int_{\Omega} V_{\text{Ha}}(\mathbf{r}) \rho(\mathbf{r}) d\mathbf{r} \quad (6.8)$$

where the Hartree potential  $V_{\text{Ha}}$  is defined as

$$V_{\text{Ha}}(\mathbf{r}) = \int_{\Omega} \frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d\mathbf{r}' \quad (6.9)$$

Alternatively,  $V_{\text{Ha}}$  can be calculated as the solution of Poisson equation

$$\nabla^2 V_{\text{Ha}}(\mathbf{r}) = -4\pi\rho(\mathbf{r}) \quad (6.10)$$

## 6.5 XC energy and potential

PWDFT.jl uses Libxc.jl[1], a Julia wrapper to Libxc[2, 3], to calculate exchange correlation energy and potentials.

For LDA we have:

$$E_{\text{xc}}[\rho_\sigma] = \int \epsilon_{\text{xc}}^{\text{HEG}}[\rho_\sigma(\mathbf{r})] \rho_{\text{tot}}(\mathbf{r}) \, \text{d}\mathbf{r} \quad (6.11)$$

$$= \int \{ \epsilon_{\text{x}}^{\text{HEG}}[\rho_\sigma(\mathbf{r})] + \epsilon_{\text{c}}^{\text{HEG}}[\rho_\sigma(\mathbf{r})] \} \rho(\mathbf{r}) \, \text{d}\mathbf{r} \quad (6.12)$$

$$\delta E_{\text{xc}}[\rho_\sigma] = \sum_{\sigma} \int \left( \epsilon_{\text{xc}}^{\text{HEG}} + \rho_{\text{tot}} \frac{\partial}{\partial \rho_{\sigma}} \epsilon_{\text{xc}}^{\text{HEG}} \right) \, \text{d}\mathbf{r} \delta \rho_{\sigma} \quad (6.13)$$

### 6.5.1 Calculation of $E_{\text{xc}}$ in PWDFT.jl using Libxc

Note:

For VWN functional (should be applicable to other LDA functionals), we have the following for non-spin-polarized case:

```
function calc_epsxc_VWN( Rhoe::Array{Float64,1} )
    Npoints = size(Rhoe)[1]
    Nspin = 1
    eps_x = zeros(Float64,Npoints)
    eps_c = zeros(Float64,Npoints)

    ptr = Libxc.xc_func_alloc()

    # exchange part
    Libxc.xc_func_init(ptr, Libxc.LDA_X, Nspin)
    Libxc.xc_lda_exc!(ptr, Npoints, Rhoe, eps_x)
    Libxc.xc_func_end(ptr)

    # correlation part
    Libxc.xc_func_init(ptr, Libxc.LDA_C_VWN, Nspin)
    Libxc.xc_lda_exc!(ptr, Npoints, Rhoe, eps_c)
    Libxc.xc_func_end(ptr)

    Libxc.xc_func_free(ptr)

    return eps_x + eps_c
end

function calc_epsxc_VWN( Rhoe::Array{Float64,2} )
    Nspin = size(Rhoe)[2]
    Npoints = size(Rhoe)[1]
    if Nspin == 1
        return calc_epsxc_VWN( Rhoe[:,1] )
    end

    # Do the transpose manually
    Rhoe_tmp = zeros(2*Npoints)
    ipp = 0
    for ip = 1:2:2*Npoints
        ipp = ipp + 1
        Rhoe_tmp[ip] = Rhoe[ipp,1]
    end
end
```

```

        Rhoe_tmp[ip+1] = Rhoe[ipp,2]
    end

    # ....
    # The rest of the code are similar to non-spin polarized case, however now
    # we use 'Nspin=2' and pass 'Rhoe_tmp' instead of 'Rhoe'
end

```

For PBE:

```

function calc_epsxc_PBE( pw::PWGrid, Rhoe::Array{Float64,1} )
    Npoints = size(Rhoe)[1]
    Nspin = 1

    # calculate gRhoe2
    gRhoe = op_nabla( pw, Rhoe )
    gRhoe2 = zeros( Float64, Npoints )
    for ip = 1:Npoints
        gRhoe2[ip] = dot( gRhoe[:,ip], gRhoe[:,ip] )
    end

    eps_x = zeros(Float64,Npoints)
    eps_c = zeros(Float64,Npoints)

    ptr = Libxc.xc_func_alloc()

    # exchange part
    Libxc.xc_func_init(ptr, Libxc.GGA_X_PBE, Nspin)
    Libxc.xc_gga_exc!(ptr, Npoints, Rhoe, gRhoe2, eps_x)
    Libxc.xc_func_end(ptr)

    # correlation part
    Libxc.xc_func_init(ptr, Libxc.GGA_C_PBE, Nspin)
    Libxc.xc_gga_exc!(ptr, Npoints, Rhoe, gRhoe2, eps_c)
    Libxc.xc_func_end(ptr)

    Libxc.xc_func_free(ptr)

    return eps_x + eps_c
end

```

For PBE spin-polarized case

```

function calc_epsxc_PBE( pw::PWGrid, Rhoe::Array{Float64,2} )
    Nspin = size(Rhoe)[2]
    if Nspin == 1
        return calc_epsxc_PBE( pw, Rhoe[:,1] )
    end

    Npoints = size(Rhoe)[1]

    # calculate gRhoe2
    gRhoe_up = op_nabla( pw, Rhoe[:,1] )
    gRhoe_dn = op_nabla( pw, Rhoe[:,2] )
    gRhoe2 = zeros( Float64, 3*Npoints )
    ipp = 0
    for ip = 1:3:Npoints
        ipp = ipp + 1
        gRhoe2[ip] = dot( gRhoe_up[:,ipp], gRhoe_up[:,ipp] )
        gRhoe2[ip+1] = dot( gRhoe_up[:,ipp], gRhoe_dn[:,ipp] )
        gRhoe2[ip+2] = dot( gRhoe_dn[:,ipp], gRhoe_dn[:,ipp] )
    end

    Rhoe_tmp = zeros(2*Npoints)
    ipp = 0

```



```

for ip = 1:2:2*Npoints
    ipp = ipp + 1
    Rhoe_tmp[ip] = Rhoe[ipp,1]
    Rhoe_tmp[ip+1] = Rhoe[ipp,2]
end

# ....
# The rest of the code are similar to non-spin polarized case, however now
# we use 'Nspin=2' and pass 'Rhoe_tmp' instead of 'Rhoe'
end

```

### 6.5.2 Calculation of $V_{xc}$ in PWDFT.jl using Libxc

VWN non-spin polarized:

```

function calc_Vxc_VWN( Rhoe::Array{Float64,1} )
    Npoints = size(Rhoe)[1]
    Nspin = 1
    v_x = zeros(Float64,Npoints)
    v_c = zeros(Float64,Npoints)

    ptr = Libxc.xc_func_alloc()

    # exchange part
    Libxc.xc_func_init(ptr, Libxc.LDA_X, Nspin)
    Libxc.xc_lda_vxc!(ptr, Npoints, Rhoe, v_x)
    Libxc.xc_func_end(ptr)

    # correlation part
    Libxc.xc_func_init(ptr, Libxc.LDA_C_VWN, Nspin)
    Libxc.xc_lda_vxc!(ptr, Npoints, Rhoe, v_c)
    Libxc.xc_func_end(ptr)

    Libxc.xc_func_free(ptr)

    return v_x + v_c
end

```

VWN spin-polarized:

```

function calc_Vxc_VWN( Rhoe::Array{Float64,2} )
    Nspin = size(Rhoe)[2]
    if Nspin == 1
        return calc_Vxc_VWN( Rhoe[:,1] )
    end

    Npoints = size(Rhoe)[1]

    Vxc = zeros( Float64, Npoints, 2 )
    V_x = zeros( Float64, 2*Npoints )
    V_c = zeros( Float64, 2*Npoints )

    # This is the transposed version of Rhoe, use copy
    Rhoe_tmp = zeros(2*Npoints)
    ipp = 0
    for ip = 1:2:2*Npoints
        ipp = ipp + 1
        Rhoe_tmp[ip] = Rhoe[ipp,1]
        Rhoe_tmp[ip+1] = Rhoe[ipp,2]
    end

    ptr = Libxc.xc_func_alloc()

    # exchange part
    Libxc.xc_func_init(ptr, Libxc.LDA_X, Nspin)

```

```

Libxc.xc_lda_vxc!(ptr, Npoints, Rhoe_tmp, V_x)
Libxc.xc_func_end(ptr)

# correlation part
Libxc.xc_func_init(ptr, Libxc.LDA_C_VWN, Nspin)
Libxc.xc_lda_vxc!(ptr, Npoints, Rhoe_tmp, V_c)
Libxc.xc_func_end(ptr)

Libxc.xc_func_free(ptr)

ipp = 0
for ip = 1:2:2*Npoints
    ipp = ipp + 1
    Vxc[ipp,1] = V_x[ip] + V_c[ip]
    Vxc[ipp,2] = V_x[ip+1] + V_c[ip+1]
end
return Vxc
end

PBE non-spin-polarized:

function calc_Vxc_PBE( pw::PWGrid, Rhoe::Array{Float64,1} )
    Npoints = size(Rhoe)[1]
    Nspin = 1

    # calculate gRhoe2
    gRhoe = op_nabla( pw, Rhoe )
    gRhoe2 = zeros( Float64, Npoints )
    for ip = 1:Npoints
        gRhoe2[ip] = dot( gRhoe[:,ip], gRhoe[:,ip] )
    end

    V_x = zeros(Float64,Npoints)
    V_c = zeros(Float64,Npoints)
    V_xc = zeros(Float64,Npoints)

    Vg_x = zeros(Float64,Npoints)
    Vg_c = zeros(Float64,Npoints)
    Vg_xc = zeros(Float64,Npoints)

    ptr = Libxc.xc_func_alloc()

    # exchange part
    Libxc.xc_func_init(ptr, Libxc.GGA_X_PBE, Nspin)
    Libxc.xc_gga_vxc!(ptr, Npoints, Rhoe, gRhoe2, V_x, Vg_x)
    Libxc.xc_func_end(ptr)

    # correlation part
    Libxc.xc_func_init(ptr, Libxc.GGA_C_PBE, Nspin)
    Libxc.xc_gga_vxc!(ptr, Npoints, Rhoe, gRhoe2, V_c, Vg_c)
    Libxc.xc_func_end(ptr)

    V_xc = V_x + V_c
    Vg_xc = Vg_x + Vg_c

    # gradient correction
    h = zeros(Float64,3,Npoints)
    divh = zeros(Float64,Npoints)
    for ip = 1:Npoints
        h[1,ip] = Vg_xc[ip] * gRhoe[1,ip]
        h[2,ip] = Vg_xc[ip] * gRhoe[2,ip]
        h[3,ip] = Vg_xc[ip] * gRhoe[3,ip]
    end
    # div ( vgrho * gRhoe )
    divh = op_nabla_dot( pw, h )
    for ip = 1:Npoints

```

```

        V_xc[ip] = V_xc[ip] - 2.0*divh[ip]
    end

    return V_xc
end

PBE spin-polarized:

function calc_Vxc_PBE( pw::PWGrid, Rho::Array{Float64,2} )
    Nspin = size(Rho)[2]
    if Nspin == 1
        return calc_Vxc_PBE( pw, Rho[:,1] )
    end

    Npoints = size(Rho)[1]

    # calculate gRhoe2
    gRhoe_up = op_nabla( pw, Rho[:,1] ) # gRhoe = ∇•Rhoe
    gRhoe_dn = op_nabla( pw, Rho[:,2] )
    gRhoe2 = zeros( Float64, 3*Npoints )
    ipp = 0
    for ip = 1:3:3*Npoints
        ipp = ipp + 1
        gRhoe2[ip] = dot( gRhoe_up[:,ipp], gRhoe_up[:,ipp] )
        gRhoe2[ip+1] = dot( gRhoe_up[:,ipp], gRhoe_dn[:,ipp] )
        gRhoe2[ip+2] = dot( gRhoe_dn[:,ipp], gRhoe_dn[:,ipp] )
    end

    V_xc = zeros(Float64, Npoints, 2)
    V_x = zeros(Float64, Npoints*2)
    V_c = zeros(Float64, Npoints*2)

    Vg_xc = zeros(Float64, 3, Npoints)
    Vg_x = zeros(Float64, 3*Npoints)
    Vg_c = zeros(Float64, 3*Npoints)

    Rhoe_tmp = zeros(2*Npoints)
    ipp = 0
    for ip = 1:2:2*Npoints
        ipp = ipp + 1
        Rhoe_tmp[ip] = Rho[ipp,1]
        Rhoe_tmp[ip+1] = Rho[ipp,2]
    end

    ptr = Libxc.xc_func_alloc()

    # exchange part
    Libxc.xc_func_init(ptr, Libxc.GGA_X_PBE, Nspin)
    Libxc.xc_gga_vxc!(ptr, Npoints, Rhoe_tmp, gRhoe2, V_x, Vg_x)
    Libxc.xc_func_end(ptr)

    # correlation part
    Libxc.xc_func_init(ptr, Libxc.GGA_C_PBE, Nspin)
    Libxc.xc_gga_vxc!(ptr, Npoints, Rhoe_tmp, gRhoe2, V_c, Vg_c)
    Libxc.xc_func_end(ptr)

    ipp = 0
    for ip = 1:2:2*Npoints
        ipp = ipp + 1
        V_xc[ipp,1] = V_x[ip] + V_c[ip]
        V_xc[ipp,2] = V_x[ip+1] + V_c[ip+1]
    end

    Vg_xc = reshape(Vg_x + Vg_c, (3,Npoints))

    h = zeros(Float64,3,Npoints)

```

```

divh = zeros(Float64,Npoints)

# spin up
for ip = 1:Npoints
    h[1,ip] = 2*Vg_xc[1,ip]*gRhoe_up[1,ip] + Vg_xc[2,ip]*gRhoe_dn[1,ip]
    h[2,ip] = 2*Vg_xc[1,ip]*gRhoe_up[2,ip] + Vg_xc[2,ip]*gRhoe_dn[2,ip]
    h[3,ip] = 2*Vg_xc[1,ip]*gRhoe_up[3,ip] + Vg_xc[2,ip]*gRhoe_dn[3,ip]
end
divh = op_nabla_dot( pw, h )
for ip = 1:Npoints
    V_xc[ip,1] = V_xc[ip,1] - divh[ip]
end

# Spin down
for ip = 1:Npoints
    h[1,ip] = 2*Vg_xc[3,ip]*gRhoe_dn[1,ip] + Vg_xc[2,ip]*gRhoe_up[1,ip]
    h[2,ip] = 2*Vg_xc[3,ip]*gRhoe_dn[2,ip] + Vg_xc[2,ip]*gRhoe_up[2,ip]
    h[3,ip] = 2*Vg_xc[3,ip]*gRhoe_dn[3,ip] + Vg_xc[2,ip]*gRhoe_up[3,ip]
end
divh = op_nabla_dot( pw, h )
for ip = 1:Npoints
    V_xc[ip,2] = V_xc[ip,2] - divh[ip]
end

return V_xc
end

```

## 7 Pseudopotentials

Currently, PWDFT.jl supports a subset of GTH (Goedecker-Teter-Hutter) pseudopotentials. This type of pseudopotential is analytic and thus is somewhat easier to program. PWDFT.jl distribution contains several parameters of GTH pseudopotentials for LDA and GGA functionals.

These pseudopotentials can be written in terms of local  $V_{\text{loc}}^{\text{PS}}$  and angular momentum  $l$  dependent nonlocal components  $\Delta V_l^{\text{PS}}$ :

$$V_{\text{ene-nuc}}(\mathbf{r}) = \sum_I \left[ V_{\text{loc}}^{\text{PS}}(\mathbf{r} - \mathbf{R}_I) + \sum_{l=0}^{l_{\text{max}}} V_l^{\text{PS}}(\mathbf{r} - \mathbf{R}_I, \mathbf{r}' - \mathbf{R}_I) \right] \quad (7.1)$$

### PsPot\_GTH struct definition

```

struct PsPot_GTH
    pspfile::String
    atsymp::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

```

**Figure 9.** Definition of PsPot\_GTH.

## 7.1 Local pseudopotential

The local pseudopotential for  $I$ -th atom,  $V_{\text{loc}}^{\text{PS}}(\mathbf{r} - \mathbf{R}_I)$ , is radially symmetric function with the following radial form

$$V_{\text{loc}}^{\text{PS}}(r) = -\frac{Z_{\text{val}}}{r} \text{erf} \left[ \frac{\bar{r}}{\sqrt{2}} \right] + \exp \left[ -\frac{1}{2} \bar{r}^2 \right] (C_1 + C_2 \bar{r}^2 + C_3 \bar{r}^4 + C_4 \bar{r}^6) \quad (7.2)$$

with  $\bar{r} = r/r_{\text{loc}}$  and  $r_{\text{loc}}$ ,  $Z_{\text{val}}$ ,  $C_1$ ,  $C_2$ ,  $C_3$  and  $C_4$  are the corresponding pseudopotential parameters. In  $\mathbf{G}$ -space, the GTH local pseudopotential can be written as

$$V_{\text{loc}}^{\text{PS}}(\mathbf{G}) = -\frac{4\pi}{\Omega} \frac{Z_{\text{val}}}{G^2} \exp \left[ -\frac{x^2}{2} \right] + \sqrt{8\pi^3} \frac{r_{\text{loc}}^3}{\Omega} \exp \left[ -\frac{x^2}{2} \right] \times \\ (C_1 + C_2(3 - x^2) + C_3(15 - 10x^2 + x^4) + C_4(105 - 105x^2 + 21x^4 - x^6)) \quad (7.3)$$

where  $x = G r_{\text{loc}}$ .

## 7.2 Nonlocal pseudopotential

PsPotNL struct definition

```
struct PsPotNL
  NbetaNL::Int64
  prj2beta::Array{Int64,4}
  betaNL::Array{ComplexF64,3}
end
```

Figure 10. Definition of PsPotNL.

The nonlocal component of GTH pseudopotential can be written in real space as

$$V_I^{\text{PS}}(\mathbf{r} - \mathbf{R}_I, \mathbf{r}' - \mathbf{R}_I) = \sum_{\mu=1}^{N_I} \sum_{\nu=1}^{N_I} \sum_{m=-l}^l \beta_{\mu lm}(\mathbf{r} - \mathbf{R}_I) h_{\mu\nu}^l \beta_{\nu lm}^*(\mathbf{r}' - \mathbf{R}_I) \quad (7.4)$$

where  $\beta_{\mu lm}(\mathbf{r})$  are atomic-centered projector functions

$$\beta_{\mu lm}(\mathbf{r}) = p_{\mu}^l(r) Y_{lm}(\hat{\mathbf{r}}) \quad (7.5)$$

and  $h_{\mu\nu}^l$  are the pseudopotential parameters and  $Y_{lm}$  are the spherical harmonics. Number of projectors per angular momentum  $N_l$  may take value up to 3 projectors. In  $\mathbf{G}$ -space, the nonlocal part of GTH pseudopotential can be described by the following equation.

$$V_I^{\text{PS}}(\mathbf{G}, \mathbf{G}') = (-1)^l \sum_{\mu}^3 \sum_{\nu}^3 \sum_{m=-l}^l \beta_{\mu lm}(\mathbf{G}) h_{\mu\nu}^l \beta_{\nu lm}^*(\mathbf{G}') \quad (7.6)$$

with the projector functions

$$\beta_{\mu lm}(\mathbf{G}) = p_{\mu}^l(G) Y_{lm}(\hat{\mathbf{G}}) \quad (7.7)$$

The radial part of projector functions take the following form

$$p_{\mu}^l(G) = q_{\mu}^l(G r_l) \frac{\pi^{5/4} G^l \sqrt{r_l^{2l+3}}}{\sqrt{\Omega}} \exp \left[ -\frac{1}{2} G^2 r_l^2 \right] \quad (7.8)$$

For  $l = 0$ , we consider up to  $N_l = 3$  projectors:

$$q_1^0(x) = 4\sqrt{2} \quad (7.9)$$

$$q_2^0(x) = 8\sqrt{\frac{2}{15}}(3 - x^2) \quad (7.10)$$

$$q_3^0(x) = \frac{16}{3}\sqrt{\frac{2}{105}}(15 - 20x^2 + 4x^4) \quad (7.11)$$

For  $l = 1$ , we consider up to  $N_l = 3$  projectors:

$$q_1^1(x) = 8\sqrt{\frac{1}{3}} \quad (7.12)$$

$$q_2^1(x) = 16\sqrt{\frac{1}{105}}(5 - x^2) \quad (7.13)$$

$$q_3^1(x) = 8\sqrt{\frac{1}{1155}}(35 - 28x^2 + 4x^4) \quad (7.14)$$

For  $l = 2$ , we consider up to  $N_l = 2$  projectors:

$$q_1^2(x) = 8\sqrt{\frac{2}{15}} \quad (7.15)$$

$$q_2^2(x) = \frac{16}{3}\sqrt{\frac{2}{105}}(7 - x^2) \quad (7.16)$$

For  $l = 3$ , we only consider up to  $N_l = 1$  projector:

$$q_1^3(x) = 16\sqrt{\frac{1}{105}} \quad (7.17)$$

In the present implementation, we construct the local and nonlocal components of pseudopotential in the G-space using their Fourier-transformed expressions and transformed them back to real space if needed. We refer the readers to the original reference [4] and the book [5] for more information about GTH pseudopotentials.

Due to the separation of local and non-local components of electrons-nuclei interaction, Equation (??) can be written as

$$E_{\text{ele-nuc}} = E_{\text{loc}}^{\text{PS}} + E_{\text{nloc}}^{\text{PS}} \quad (7.18)$$

The local pseudopotential contribution is

$$E_{\text{loc}}^{\text{PS}} = \int_{\Omega} \rho(\mathbf{r}) V_{\text{loc}}^{\text{PS}}(\mathbf{r}) d\mathbf{r} \quad (7.19)$$

and the non-local contribution is

$$E_{\text{nloc}}^{\text{PS}} = \sum_{\mathbf{k}} \sum_i w_{\mathbf{k}} f_{i\mathbf{k}} \int_{\Omega} \psi_{i\mathbf{k}}^*(\mathbf{r}) \left[ \sum_I \sum_{l=0}^{l_{\text{max}}} V_l^{\text{PS}}(\mathbf{r} - \mathbf{R}_I, \mathbf{r}' - \mathbf{R}_I) \right] \psi_{i\mathbf{k}}(\mathbf{r}) d\mathbf{r}. \quad (7.20)$$

## 8 Hamiltonian operators

### Hamiltonian struct definition

```
mutable struct Hamiltonian
  pw::PWGrid
  potentials::Potentials
  energies::Energies
  rhoe::Array{Float64,2}
  electrons::Electrons
  atoms::Atoms
  sym_info::SymmetryInfo
  rhoe_symmetrizer::RhoeSymmetrizer
  pspots::Array{PsPot_GTH,1}
  pspotNL::PsPotNL
  xcfunc::String
  ik::Int64
  ispin::Int64
end
```

Figure 11. Definition of Hamiltonian.

Operators:

- op\_H
- op\_K
- op\_V\_loc
- op\_V\_Ps\_loc
- op\_V\_Ps\_nloc

### 8.1 Iterative diagonalization of Hamiltonian

- diag\_LOBPCG
- diag\_davidson
- diag\_Emin\_PCG

## 9 Self-consistent field

Density vs potential mix

KS\_solve\_SCF

KS\_solve\_SCF\_potmix

In the mean time, they are separated. They might be combined into one function in the future development.

Mixing algorithms:

- Simple or linear mixing
- Adaptive linear mixing
- Anderson mixing
- Broyden mixing

- Pulay mixing
- Restarted Pulay mixing
- Periodic Pulay mixing

## 10 Direct minimization

KS\_solve\_Emin\_PCG

## 11 Calculation of atomic forces

These notes are adapted from Kohanoff2006.

The forces on the nuclei of atomic species  $s$  are given by

$$\mathbf{F}_I^s = -\frac{\partial E_{\text{KS}}}{\partial \mathbf{R}_I^s} \quad (11.1)$$

Specifically:

$$\mathbf{F}_I^s = \int \rho(\mathbf{r}) \left( V_s^{\text{Ps,loc}} \right)' (|\mathbf{r} - \mathbf{R}_I|) \frac{\mathbf{r} - \mathbf{R}_I}{|\mathbf{r} - \mathbf{R}_I|} d\mathbf{r} + \frac{Z_I}{2} \sum_{J \neq I} Z_J \frac{\mathbf{R}_I - \mathbf{R}_J}{|\mathbf{R}_I - \mathbf{R}_J|^3} + \mathbf{F}_{s,I}^{\text{Ps,nloc}} \quad (11.2)$$

$\left( V_s^{\text{Ps,loc}} \right)'$  is the derivative of the local part of the pseudopotential

$\mathbf{F}_{\text{Ps,nloc}}^s$ : nonlocal pseudopotential contribution to the force

Using plane wave basis set.

Nuclear-nuclear contribution:

$$F_{s,I}^{NN} = \frac{Z_I}{2} \sum_{J \neq I}^P Z_J \sum_{L=-L_{\text{max}}}^{L_{\text{max}}} (\mathbf{R}_I + \mathbf{l} - \mathbf{R}_J) \times \left[ \frac{\text{erfc}(|\mathbf{R}_I + \mathbf{l} - \mathbf{R}_J| \eta)}{|\mathbf{R}_I + \mathbf{l} - \mathbf{R}_J|^3} + \frac{\eta e^{-\eta^2 |\mathbf{R}_I + \mathbf{l} - \mathbf{R}_J|^2}}{|\mathbf{R}_I + \mathbf{l} - \mathbf{R}_J|} \right] \quad (11.3)$$

Local pseudopotential contribution:

$$F_{s,I}^{\text{Ps,loc}} = \Omega \sum_{\mathbf{G} \neq 0} i\mathbf{G} e^{i\mathbf{G} \cdot \mathbf{R}_I} V_s^{\text{Ps,loc}}(\mathbf{G}) \rho(\mathbf{G}) \quad (11.4)$$

Nonlocal pseudopotential contribution:

$$F_{s,I}^{\text{Ps,nloc}} = 2\text{Re} \left\{ \sum_{\mathbf{k}} \omega_{\mathbf{k}} \sum_{lm} \sum_{i=1}^{N_{\mathbf{k}}} f_i^{(\mathbf{k})} \alpha_{lm}^s F_{I,i}^{lm,s*} D_{I,i}^{lm,s*} \right\} \quad (11.5)$$

where

$$D_{I,i}^{lm,s*} = \sum_{\mathbf{G}} i\mathbf{G} e^{i\mathbf{G} \cdot \mathbf{R}_I} f_{lm}^s(\mathbf{G} + \mathbf{k}) C_{i\mathbf{k}}(\mathbf{G}) \quad (11.6)$$



# Other things

## A Howtos

This part contains miscellaneous info.

TODO: Some of the should be moved into main text.

### A.1 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")]
```

### A.2 Using Babel to generate xyz file from SMILES

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

Use `babel -h` to autogenerate hydrogens.

### A.3 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 = joinpath(dirname(pathof(PWDFT)), "..")
include(joinpath(DIR_PWDFT, "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"]
```

## A.4 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.

## A.5 Iterative diagonalization of Hamiltonian

```
evals = diag_LOBPCG!( Ham, psiks, verbose=false, verbose_last=false,  
                     Nstates_conv=Nstates_occ )
```

## A.6 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 )
```

## A.7 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, 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
```

### 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]
```

Usually we don't need to sort the eigenvalues if we use Hermitian matrix. We can calculate the subspace Hamiltonian by:

```
evals, evecs = eigen(Hermitian(Hr))
```

## Status

**29 July 2019** Total energy results are now similar to ABINIT and Quantum ESPRESSO. A rather comprehensive test has been added for SCF and Emin PCG for several simple systems.

**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.

## References

- [1] Libxc.jl. <https://github.com/unkcpz/Libxc.jl>.
- [2] Miguel A.L. Marques, Micael J.T. Oliveira, and Tobias Burnus. Libxc: A library of exchange and correlation functionals for density functional theory. *Comput. Phys. Commun.*, 183(10):2272 – 2281, 2012.
- [3] Susi Lehtola, Conrad Steigemann, Micael J.T. Oliveira, and Miguel A.L. Marques. Recent developments in Libxc — a comprehensive library of functionals for density functional theory. *SoftwareX*, 7:1–5, 2018.
- [4] S. Goedecker, M. Teter, and J. Hutter. Separable dual-space Gaussian pseudopotentials. *Phys. Rev. B*, 54:1703–1710, 1996.
- [5] Dominik Marx and Jürg Hutter. *Ab Initio Molecular Dynamics: Basic Theory and Advanced Methods*. Cambridge University Press, 2009.