# PWDFT.jl Documentation

*Fadjar Fathurrahman*

## 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.

---

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

## 2   Installation

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

- Julia version 0.7 or higher

- LibXC

- SPGLIB

- C compiler to compiler 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"))
```

The second way is to use Julia's development directory located at `$HOME/.julia/dev`. I usually do this by sym-linking my original PWDFT.jl directory to `$HOME/.julia/dev`.

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

```
./configure --prefix=/home/user/softwares/libxc
make
make install
```

### 2.2   SPGLIB

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

## 3   Getting started

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

- Atomic structures: coordinates and unit cell

- Pseudopotentials

This code will initialize a hydrogen molecule (H2) inside a box of $16 \times 16 \times 16$ bohr.

```
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))
```

---

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

## 4 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.

## 5 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 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

## 6   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
```

## 7   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*ecutw`
- `Ns::Tuple{`**`Int64`**`,`**`Int64`**`,`**`Int64`**`}`: parameters defining real-space grid points.
- `LatVecs::Array{`**`Float64`**`,2}`: lattice vectors of unit cell ($3 \times 3$ matrix)
- `RecVecs::Array{`**`Float64`**`,2}`: reciprocal lattice vectors ($3 \times 3$ 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 )
```

---

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

## 7.1 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'.

## 7.2 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

## 7.3 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.

## 8   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 **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`

## 9   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.

## 10   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.

## 11  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.

```julia
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
```

## 12  Hamiltonian operator

Apply Hamiltonian operator to wave function

```julia
for ispin = 1:Nspin
    for ik = 1:Nkpt
        Ham.ispin = ispin
        Ham.ik = ik
        Hpsiks = op_H(Ham, psiks)
    end
end
```

## 13  Solving Kohn-Sham problems

Two main algorithms: SCF and direct minimization

Function names: `KS_solve_*`

### 13.1  Self-consistent field

`KS_solve_SCF`

```julia
"""
Solves Kohn-Sham problem using traditional self-consistent field (SCF)
iterations with density mixing.
"""
function KS_solve_SCF!( Ham::Hamiltonian ;
                        startingwfc=nothing, savewfc=false,
                        betamix = 0.2, NiterMax=100, verbose=true,
                        print_final_ebands=true,
                        print_final_energies=true,
                        check_rhoe=false,
                        use_smearing = false, kT=1e-3,
                        update_psi="LOBPCG", cheby_degree=8,
                        mix_method="simple", MIXDIM=5,
                        print_e_gap=false,
                        ETOT_CONV_THR=1e-6 )

    pw = Ham.pw
    Ngw = pw.gvecw.Ngw
    wk = Ham.pw.gvecw.kpoints.wk
    #
    kpoints = pw.gvecw.kpoints
    Nkpt = kpoints.Nkpt
    #
    Ns = pw.Ns
    Npoints = prod(Ns)
    CellVolume = pw.CellVolume
    dVol = CellVolume/Npoints
    #
    electrons = Ham.electrons
    Nelectrons = electrons.Nelectrons
    Focc = copy(electrons.Focc) # make sure to use the copy
    Nstates = electrons.Nstates
    Nstates_occ = electrons.Nstates_occ
```

```julia
Nspin = electrons.Nspin
#
Nkspin = Nkpt*Nspin

Nstates_occ = electrons.Nstates_occ

#
# Random guess of wave function
#
if startingwfc==nothing
    psiks = rand_BlochWavefunc(pw, electrons)
else
    psiks = startingwfc
end

E_GAP_INFO = false
if Nstates_occ < Nstates
    E_GAP_INFO = true
    if Nspin == 2
        idx_HOMO = max(round(Int64,Nstates_occ/2),1)
        idx_LUMO = idx_HOMO + 1
    else
        idx_HOMO = Nstates_occ
        idx_LUMO = idx_HOMO + 1
    end
end

#
# Calculated electron density from this wave function and update Hamiltonian
#
Rhoe = zeros(Float64,Npoints,Nspin)

if Nspin == 2
    idx_HOMO = max(round(Int64,Nstates_occ/2),1)
    idx_LUMO = idx_HOMO + 1
    Focc[idx_HOMO,1:Nkpt] = Focc[idx_HOMO,1:Nkpt] .+ 0.5
    Focc[idx_HOMO,Nkpt+1:2*Nkpt] = Focc[idx_HOMO,Nkpt+1:2*Nkpt] .- 0.5
end

Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )
if Nspin == 2
    @printf("\nInitial integ Rhoe up = %18.10f\n", sum(Rhoe[:,1])*dVol)
    @printf("\nInitial integ Rhoe dn = %18.10f\n", sum(Rhoe[:,2])*dVol)
    @printf("\nInitial integ magn_den = %18.10f\n", sum(Rhoe[:,1] - Rhoe[:,2])*dVol)
end

update!(Ham, Rhoe)

Etot_old = 0.0

Rhoe_new = zeros(Float64,Npoints,Nspin)

diffRhoe = zeros(Nspin)

evals = zeros(Float64,Nstates,Nkspin)

ETHR_EVALS_LAST = 1e-6

ethr = 0.1

if mix_method == "anderson"
    df = zeros(Float64,Npoints*Nspin,MIXDIM)
    dv = zeros(Float64,Npoints*Nspin,MIXDIM)

elseif mix_method in ("rpulay", "rpulay_kerker", "ppulay", "pulay")
    XX = zeros(Float64,Npoints*Nspin,MIXDIM)
    FF = zeros(Float64,Npoints*Nspin,MIXDIM)
    x_old = zeros(Float64,Npoints*Nspin)
    f_old = zeros(Float64,Npoints*Nspin)
end


@printf("\n")
@printf("Self-consistent iteration begins ...\n")
@printf("update_psi = %s\n", update_psi)
@printf("\n")
@printf("mix_method = %s\n", mix_method)
if mix_method in ("rpulay", "rpulay_kerker", "anderson", "ppulay")
    @printf("MIXDIM = %d\n", MIXDIM)
```

```julia
end
@printf("Density mixing with betamix = %10.5f\n", betamix)
if use_smearing
    @printf("Smearing = %f\n", kT)
end
println("") # blank line before SCF iteration info

# calculate E_NN
Ham.energies.NN = calc_E_NN( Ham.atoms )

# calculate PspCore energy
Ham.energies.PspCore = calc_PspCore_ene( Ham.atoms, Ham.pspots, CellVolume )

CONVERGED = 0

E_fermiSpin = zeros(Nspin)

for iter = 1:NiterMax

    # determine convergence criteria for diagonalization
    if iter == 1
        ethr = 0.1
    elseif iter == 2
        ethr = 0.01
    else
        ethr = ethr/5.0
        ethr = max( ethr, ETHR_EVALS_LAST )
    end

    if update_psi == "LOBPCG"

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

    elseif update_psi == "davidson"

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

    elseif update_psi == "PCG"

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

    elseif update_psi == "CheFSI"
        # evals will be calculated later
        diag_CheFSI!( Ham, psiks, cheby_degree )

    else
        error( @sprintf("Unknown method for update_psi = %s\n", update_psi) )
    end

    if E_GAP_INFO && verbose && print_e_gap
        @printf("E gap = %18.10f\n", minimum(evals[idx_LUMO,:] - evals[idx_HOMO,:]))
    end

    if use_smearing
        Focc, E_fermi = calc_Focc( Nelectrons, wk, kT, evals, Nspin )
        Entropy = calc_entropy( wk, kT, evals, E_fermi, Nspin )
        Ham.electrons.Focc = copy(Focc)
    end

    Rhoe_new[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )
    for ispin = 1:Nspin
        diffRhoe[ispin] = norm(Rhoe_new[:,ispin] - Rhoe[:,ispin])
    end

    # check norm of
    if check_rhoe
        integRhoe = sum(Rhoe_new)*dVol
        @printf("Before: integRhoe_new = %18.10f\n", integRhoe)
        Rhoe_new = Nelectrons/integRhoe * Rhoe_new
        integRhoe = sum(Rhoe_new)*dVol
        @printf("After renormalize Rhoe_new: = %18.10f\n", integRhoe)
    end
```

```julia
if mix_method == "simple"
    for ispin = 1:Nspin
        Rhoe[:,ispin] = betamix*Rhoe_new[:,ispin] + (1-betamix)*Rhoe[:,ispin]
    end

elseif mix_method == "simple_kerker"
    for ispin = 1:Nspin
        Rhoe[:,ispin] = Rhoe[:,ispin] + betamix*precKerker(pw, Rhoe_new[:,ispin] - Rhoe[:,ispin])
    end

elseif mix_method == "pulay"

    Rhoe = reshape( mix_pulay!(
        reshape(Rhoe,(Npoints*Nspin)),
        reshape(Rhoe_new,(Npoints*Nspin)), betamix, XX, FF, iter, MIXDIM, x_old, f_old
        ), (Npoints,Nspin) )

    if Nspin == 2
        magn_den = Rhoe[:,1] - Rhoe[:,2]
    end

elseif mix_method == "rpulay"

    Rhoe = reshape( mix_rpulay!(
        reshape(Rhoe,(Npoints*Nspin)),
        reshape(Rhoe_new,(Npoints*Nspin)), betamix, XX, FF, iter, MIXDIM, x_old, f_old
        ), (Npoints,Nspin) )

    if Nspin == 2
        magn_den = Rhoe[:,1] - Rhoe[:,2]
    end

elseif mix_method == "ppulay"

    Rhoe = reshape( mix_ppulay!(
        reshape(Rhoe,(Npoints*Nspin)),
        reshape(Rhoe_new,(Npoints*Nspin)), betamix, XX, FF, iter, MIXDIM, 3, x_old, f_old
        ), (Npoints,Nspin) )

    if Nspin == 2
        magn_den = Rhoe[:,1] - Rhoe[:,2]
    end

elseif mix_method == "rpulay_kerker"

    Rhoe = reshape( mix_rpulay_kerker!( pw,
        reshape(Rhoe,(Npoints*Nspin)),
        reshape(Rhoe_new,(Npoints*Nspin)), betamix, XX, FF, iter, MIXDIM, x_old, f_old
        ), (Npoints,Nspin) )

    if Nspin == 2
        magn_den = Rhoe[:,1] - Rhoe[:,2]
    end

elseif mix_method == "anderson"
    Rhoe[:,:] = mix_anderson!( Nspin, Rhoe, Rhoe_new, betamix, df, dv, iter, MIXDIM )

else
    error(@sprintf("Unknown mix_method = %s\n", mix_method))

end

for rho in Rhoe
    if rho < eps()
        rho = 0.0
    end
end

# renormalize
if check_rhoe
    integRhoe = sum(Rhoe)*dVol
    #@printf("After mixing: integRhoe = %18.10f\n", integRhoe)
    Rhoe = Nelectrons/integRhoe * Rhoe
    integRhoe = sum(Rhoe)*dVol
    #@printf("After renormalize Rhoe: = %18.10f\n", integRhoe)
end

update!( Ham, Rhoe )
```

```julia
    # Calculate energies
    Ham.energies = calc_energies( Ham, psiks )
    if use_smearing
        Ham.energies.mTS = Entropy
    end
    Etot = sum(Ham.energies)
    diffE = abs( Etot - Etot_old )

    if verbose
        if Nspin == 1
            @printf("\nSCF: %8d %18.10f %18.10e %18.10e\n",
                    iter, Etot, diffE, diffRhoe[1] )
            @printf("integ Rhoe = %18.10f\n", sum(Rhoe)*dVol)
        else
            @printf("SCF: %8d %18.10f %18.10e %18.10e %18.10e\n",
                    iter, Etot, diffE, diffRhoe[1], diffRhoe[2] )
            magn_den = Rhoe[:,1] - Rhoe[:,2]
            @printf("integ Rhoe spin up = %18.10f\n", sum(Rhoe[:,1])*dVol)
            @printf("integ Rhoe spin dn = %18.10f\n", sum(Rhoe[:,2])*dVol)
            @printf("integ magn_den = %18.10f\n", sum(magn_den)*dVol)
        end

    end

    if diffE < ETOT_CONV_THR
        CONVERGED = CONVERGED + 1
    else  # reset CONVERGED
        CONVERGED = 0
    end

    if CONVERGED >= 2
        if verbose
            @printf("SCF is converged: iter: %d , diffE = %10.7e\n", iter, diffE)
        end
        break
    end
    #
    Etot_old = Etot

    flush(stdout)
end

# Eigenvalues are not calculated if using CheFSI.
# We calculate them here.
if update_psi == "CheFSI"
    for ispin = 1:Nspin
    for ik = 1:Nkpt
        Ham.ik = ik
        Ham.ispin = ispin
        ikspin = ik + (ispin - 1)*Nkpt
        Hr = psiks[ikspin]' * op_H( Ham, psiks[ikspin] )
        evals[:,ikspin] = eigvals(Hermitian(Hr))
    end
    end
end

Ham.electrons.ebands = evals

if verbose && print_final_ebands
    @printf("\n")
    @printf("---------------------------\n")
    @printf("Final Kohn-Sham eigenvalues:\n")
    @printf("---------------------------\n")
    @printf("\n")
    print_ebands(Ham.electrons, Ham.pw.gvecw.kpoints)
end

if verbose && print_final_energies
    @printf("\n")
    @printf("------------------------\n")
    @printf("Final Kohn-Sham energies:\n")
    @printf("------------------------\n")
    @printf("\n")
    println(Ham.energies, use_smearing=use_smearing)
end

if savewfc
    for ikspin = 1:Nkpt*Nspin
        wfc_file = open("WFC_ikspin_"*string(ikspin)*".data","w")
```

```
            write( wfc_file, psiks[ikspin] )
            close( wfc_file )
        end
    end

    return

end
```

## 13.2  Eigensolver

Davidson, LOBPCG, and PCG

## 13.3  Direct energy minimization via nonlinear conjugate gradient method

`KS_solve_Emin_PCG`

This method is described by IsmailBeigi-Arias

```
"""
Solves Kohn-Sham problem using direct energy minimization as described
by Ismail-Beigi and Arias.
"""
function KS_solve_Emin_PCG!( Ham::Hamiltonian;
                             startingwfc=nothing, savewfc=false,
                             α_t=3e-5, NiterMax=200, verbose=true,
                             print_final_ebands=true, print_final_energies=true,
                             I_CG_BETA=2, ETOT_CONV_THR=1e-6 )

    pw = Ham.pw
    electrons = Ham.electrons

    Focc = electrons.Focc
    Nstates = electrons.Nstates
    Nelectrons = electrons.Nelectrons

    Ns = pw.Ns
    Npoints = prod(Ns)
    CellVolume = pw.CellVolume
    Ngw = pw.gvecw.Ngw
    Ngwx = pw.gvecw.Ngwx
    Nkpt = pw.gvecw.kpoints.Nkpt

    Nspin = electrons.Nspin
    Nkspin = Nkpt*Nspin

    #
    # Initial wave function
    #
    if startingwfc == nothing
        psiks = rand_BlochWavefunc(pw, electrons)
    else
        psiks = startingwfc
    end

    #
    # Calculated electron density from this wave function and
    # update Hamiltonian (calculate Hartree and XC potential).
    #
    Rhoe = zeros(Float64,Npoints,Nspin)

    Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )

    update!(Ham, Rhoe)

    #
    # Variables for PCG
    #
    g = Array{Array{ComplexF64,2},1}(undef,Nkspin)
    d = Array{Array{ComplexF64,2},1}(undef,Nkspin)
    g_old = Array{Array{ComplexF64,2},1}(undef,Nkspin)
    d_old = Array{Array{ComplexF64,2},1}(undef,Nkspin)
    Kg = Array{Array{ComplexF64,2},1}(undef,Nkspin)
    Kg_old = Array{Array{ComplexF64,2},1}(undef,Nkspin)
    psic = Array{Array{ComplexF64,2},1}(undef,Nkspin)
    gt = Array{Array{ComplexF64,2},1}(undef,Nkspin)
    #
```

```julia
    for ispin = 1:Nspin
    for ik = 1:Nkpt
        ikspin = ik + (ispin - 1)*Nkpt
        g[ikspin] = zeros(ComplexF64, Ngw[ik], Nstates)
        d[ikspin] = zeros(ComplexF64, Ngw[ik], Nstates)
        g_old[ikspin] = zeros(ComplexF64, Ngw[ik], Nstates)
        d_old[ikspin] = zeros(ComplexF64, Ngw[ik], Nstates)
        Kg[ikspin] = zeros(ComplexF64, Ngw[ik], Nstates)
        Kg_old[ikspin] = zeros(ComplexF64, Ngw[ik], Nstates)
        psic[ikspin] = zeros(ComplexF64, Ngw[ik], Nstates)
        gt[ikspin] = zeros(ComplexF64, Ngw[ik], Nstates)
    end
    end

    β = zeros(Nkspin)
    α = zeros(Nkspin)

    Etot_old = 0.0

    # calculate E_NN
    Ham.energies.NN = calc_E_NN( Ham.atoms )

    # calculate PspCore energy
    Ham.energies.PspCore = calc_PspCore_ene( Ham.atoms, Ham.pspots, CellVolume )

    # Calculate energy at this psi
    energies = calc_energies(Ham, psiks)
    Ham.energies = energies
    Etot = sum(energies)

    CONVERGED = 0

    if verbose
        @printf("\n")
        @printf("Minimizing Kohn-Sham energy using PCG\n")
        @printf("-----------------------------------\n")
        @printf("NiterMax  = %d\n", NiterMax)
        @printf("α_t       = %e\n", α_t)
        @printf("conv_trh  = %e\n", ETOT_CONV_THR)
        if I_CG_BETA == 1
            @printf("Using Fletcher-Reeves formula for CG_BETA\n")
        elseif I_CG_BETA == 2
            @printf("Using Polak-Ribiere formula for CG_BETA\n")
        elseif I_CG_BETA == 3
            @printf("Using Hestenes-Stiefeld formula for CG_BETA\n")
        else
            @printf("Using Dai-Yuan formula for CG_BETA\n")
        end
        @printf("\n")
    end


    for iter = 1:NiterMax

        for ispin = 1:Nspin
        for ik = 1:Nkpt

            Ham.ik = ik
            Ham.ispin = ispin
            ikspin = ik + (ispin - 1)*Nkpt

            g[ikspin] = calc_grad( Ham, psiks[ikspin] )
            Kg[ikspin] = Kprec( Ham.ik, pw, g[ikspin] )

            # XXX: define function trace for real(sum(conj(...)))
            if iter != 1
                if I_CG_BETA == 1
                    β[ikspin] =
                    real(sum(conj(g[ikspin]).*Kg[ikspin]))/real(sum(conj(g_old[ikspin]).*Kg_old[ikspin]))
                elseif I_CG_BETA == 2
                    β[ikspin] =
↪   real(sum(conj(g[ikspin]-g_old[ikspin]).*Kg[ikspin]))/real(sum(conj(g_old[ikspin]).*Kg_old[ikspin]))
                elseif I_CG_BETA == 3
                    β[ikspin] =
                    real(sum(conj(g[ik-
↪   spin]-g_old[ikspin]).*Kg[ikspin]))/real(sum(conj(g[ikspin]-g_old[ikspin]).*d[ikspin]))
                else
                    β[ikspin] =
```

```
↪  real(sum(conj(g[ikspin]).*Kg[ikspin]))/real(sum((g[ikspin]-g_old[ikspin]).*conj(d_old[ikspin])))
            end
        end
        if β[ikspin] < 0.0
            β[ikspin] = 0.0
        end

        d[ikspin] = -Kg[ikspin] + β[ikspin] * d_old[ikspin]

        psic[ikspin] = ortho_sqrt(psiks[ikspin] + α_t*d[ikspin])
    end # ik
    end # ispin

    Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )

    update!(Ham, Rhoe)

    for ispin = 1:Nspin
    for ik = 1:Nkpt
        Ham.ik = ik
        Ham.ispin = ispin
        ikspin = ik + (ispin - 1)*Nkpt
        gt[ikspin] = calc_grad(Ham, psic[ikspin])

        denum = real(sum(conj(g[ikspin]-gt[ikspin]).*d[ikspin]))
        if denum != 0.0
            α[ikspin] = abs( α_t*real(sum(conj(g[ikspin]).*d[ikspin]))/denum )
        else
            α[ikspin] = 0.0
        end

        # Update wavefunction
        psiks[ikspin] = psiks[ikspin] + α[ikspin]*d[ikspin]

        # Update potentials
        psiks[ikspin] = ortho_sqrt(psiks[ikspin])
    end
    end

    Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )

    update!(Ham, Rhoe)

    Ham.energies = calc_energies( Ham, psiks )
    Etot = sum(Ham.energies)
    diffE = abs(Etot-Etot_old)

    if verbose
        @printf("CG step %8d = %18.10f %10.7e\n", iter, Etot, diffE)
    end

    if diffE < ETOT_CONV_THR
        CONVERGED = CONVERGED + 1
    else
        CONVERGED = 0
    end

    if CONVERGED >= 2
        if verbose
            @printf("CONVERGENCE ACHIEVED\n")
        end
        break
    end

    g_old = copy(g)
    d_old = copy(d)
    Kg_old = copy(Kg)
    Etot_old = Etot

    flush(stdout)
end

# Calculate eigenvalues
for ispin = 1:Nspin
for ik = 1:Nkpt
    Ham.ik = ik
    Ham.ispin = ispin
    ikspin = ik + (ispin - 1)*Nkpt
```

```julia
        psiks[ikspin] = ortho_sqrt(psiks[ikspin])
        Hr = psiks[ikspin]' * op_H( Ham, psiks[ikspin] )
        evals, evecs = eigen(Hr)
        evals = real(evals[:])
        # We need to sort this
        idx_sorted = sortperm(evals)
        Ham.electrons.ebands[:,ikspin] = evals[idx_sorted]
        psiks[ikspin] = psiks[ikspin]*evecs[:,idx_sorted]
    end
    end

    if verbose && print_final_ebands
        @printf("\n")
        @printf("--------------------------\n")
        @printf("Final Kohn-Sham eigenvalues:\n")
        @printf("--------------------------\n")
        @printf("\n")
        print_ebands(Ham.electrons, Ham.pw.gvecw.kpoints)
    end

    if verbose && print_final_energies
        @printf("\n")
        @printf("-----------------------\n")
        @printf("Final Kohn-Sham energies:\n")
        @printf("-----------------------\n")
        @printf("\n")
        println(Ham.energies)
    end

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


    return

end
```

## 13.4   Direct constrained minimization method of Yang

`KS_solve_DCM`

```julia
"""
Solves Kohn-Sham problem using direct constrined minimization (DCM) as described
by Yang.
"""
function KS_solve_DCM!( Ham::Hamiltonian;
                        NiterMax = 100, startingwfc=nothing,
                        verbose=true,
                        print_final_ebands=true, print_final_energies=true,
                        savewfc=false, ETOT_CONV_THR=1e-6 )


    pw = Ham.pw
    Ngw = pw.gvecw.Ngw
    Ns = pw.Ns
    Npoints = prod(Ns)
    CellVolume = pw.CellVolume
    ΔV = CellVolume/Npoints

    electrons = Ham.electrons
    Nelectrons = electrons.Nelectrons
    Focc = electrons.Focc
    Nocc = electrons.Nstates_occ
    Nstates = electrons.Nstates

    Nkpt = Ham.pw.gvecw.kpoints.Nkpt
    Nspin = electrons.Nspin

    Nkspin = Nkpt*Nspin

    psiks = Array{Array{ComplexF64,2},1}(undef,Nkspin)

    #
```

```julia
# Initial wave function
#
if startingwfc == nothing
    psiks = rand_BlochWavefunc(pw, electrons)
else
    psiks = startingwfc
end


#
# Calculated electron density from this wave function and update Hamiltonian
#
Rhoe = zeros(Float64,Npoints,Nspin)

Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )
update!(Ham, Rhoe)

Rhoe_old = copy(Rhoe)

evals = zeros(Float64,Nstates,Nkspin)

# calculate E_NN
Ham.energies.NN = calc_E_NN( Ham.atoms )

# calculate PspCore energy
Ham.energies.PspCore = calc_PspCore_ene( Ham.atoms, Ham.pspots, CellVolume)

# Starting eigenvalues and psi
for ispin = 1:Nspin
for ik = 1:Nkpt
    Ham.ik = ik
    Ham.ispin = ispin
    ikspin = ik + (ispin - 1)*Nkpt
    evals[:,ikspin], psiks[ikspin] =
    diag_LOBPCG( Ham, psiks[ikspin], verbose_last=false, NiterMax=10 )
end
end

Ham.energies = calc_energies( Ham, psiks )

Etot_old = sum(Ham.energies)

# subspace
Y = Array{Array{ComplexF64,2},1}(undef,Nkspin)
R = Array{Array{ComplexF64,2},1}(undef,Nkspin)
P = Array{Array{ComplexF64,2},1}(undef,Nkspin)
G = Array{Array{ComplexF64,2},1}(undef,Nkspin)
T = Array{Array{Float64,2},1}(undef,Nkspin)
B = Array{Array{Float64,2},1}(undef,Nkspin)
A = Array{Array{Float64,2},1}(undef,Nkspin)
C = Array{Array{Float64,2},1}(undef,Nkspin)
for ispin = 1:Nspin
for ik = 1:Nkpt
    ikspin = ik + (ispin - 1)*Nkpt
    Y[ikspin] = zeros( ComplexF64, Ngw[ik], 3*Nstates )
    R[ikspin] = zeros( ComplexF64, Ngw[ik], Nstates )
    P[ikspin] = zeros( ComplexF64, Ngw[ik], Nstates )
    G[ikspin] = zeros( ComplexF64, 3*Nstates, 3*Nstates )
    T[ikspin] = zeros( Float64, 3*Nstates, 3*Nstates )
    B[ikspin] = zeros( Float64, 3*Nstates, 3*Nstates )
    A[ikspin] = zeros( Float64, 3*Nstates, 3*Nstates )
    C[ikspin] = zeros( Float64, 3*Nstates, 3*Nstates )
end
end

D = zeros(Float64,3*Nstates,Nkspin)  # array for saving eigenvalues of subspace problem

set1 = 1:Nstates
set2 = Nstates+1:2*Nstates
set3 = 2*Nstates+1:3*Nstates
set4 = Nstates+1:3*Nstates
set5 = 1:2*Nstates

MaxInnerSCF = 3

for iter = 1:NiterMax

    for ispin = 1:Nspin
    for ik = 1:Nkpt
        Ham.ik = ik
```

```julia
            Ham.ispin = ispin
            ikspin = ik + (ispin - 1)*Nkpt
            #
            Hpsi = op_H( Ham, psiks[ikspin] )
            #
            psiHpsi = psiks[ikspin]' * Hpsi
            psiHpsi = 0.5*( psiHpsi + psiHpsi' )
            # Calculate residual
            R[ikspin] = Hpsi - psiks[ikspin]*psiHpsi
            R[ikspin] = Kprec( ik, pw, R[ikspin] )
            # Construct subspace
            Y[ikspin][:,set1] = psiks[ikspin]
            Y[ikspin][:,set2] = R[ikspin]
            #
            if iter > 1
                Y[ikspin][:,set3] = P[ikspin]
            end


            #
            # Project kinetic and ionic potential
            #
            if iter > 1
                KY = op_K( Ham, Y[ikspin] ) + op_V_Ps_loc( Ham, Y[ikspin] )
                T[ikspin] = real(Y[ikspin]'*KY)
                B[ikspin] = real(Y[ikspin]'*Y[ikspin])
                B[ikspin] = 0.5*( B[ikspin] + B[ikspin]' )
            else
                # only set5=1:2*Nstates is active for iter=1
                KY = op_K( Ham, Y[ikspin][:,set5] ) + op_V_Ps_loc( Ham, Y[ikspin][:,set5] )
                T[ikspin][set5,set5] = real(Y[ikspin][:,set5]'*KY)
                bb = real(Y[ikspin][set5,set5]'*Y[ikspin][set5,set5])
                B[ikspin][set5,set5] = 0.5*( bb + bb' )
            end

            if iter > 1
                G[ikspin] = Matrix(1.0I, 3*Nstates, 3*Nstates) #eye(3*Nstates)
            else
                G[ikspin] = Matrix(1.0I, 2*Nstates, 2*Nstates)
            end
    end
    end

    @printf("DCM iter: %3d\n", iter)

    for iterscf = 1:MaxInnerSCF

        for ispin = 1:Nspin
        for ik = 1:Nkpt
            #
            Ham.ik = ik
            Ham.ispin = ispin
            ikspin = ik + (ispin - 1)*Nkpt
            #
            V_loc = Ham.potentials.Hartree + Ham.potentials.XC[:,ispin]
            #
            if iter > 1
                yy = Y[ikspin]
            else
                yy = Y[ikspin][:,set5]
            end
            #
            if Ham.pspotNL.NbetaNL > 0
                VY = op_V_Ps_nloc( Ham, yy ) + op_V_loc( ik, pw, V_loc, yy )
            else
                VY = op_V_loc( ik, pw, V_loc, yy )
            end
            #
            if iter > 1
                A[ikspin] = real( T[ikspin] + yy'*VY )
                A[ikspin] = 0.5*( A[ikspin] + A[ikspin]' )
            else
                aa = real( T[ikspin][set5,set5] + yy'*VY )
                A[ikspin] = 0.5*( aa + aa' )
            end
            #
            if iter > 1
                BG = B[ikspin]*G[ikspin][:,1:Nocc]
                C[ikspin] = real( BG*BG' )
                C[ikspin] = 0.5*( C[ikspin] + C[ikspin]' )
```

```julia
            else
                BG = B[ikspin][set5,set5]*G[ikspin][set5,1:Nocc]
                cc = real( BG*BG' )
                C[ikspin][set5,set5] = 0.5*( cc + cc' )
            end
            #
            if iter > 1
                D[:,ikspin], G[ikspin] = eigen( A[ikspin], B[ikspin] )
            else
                D[set5,ikspin], G[ikspin][set5,set5] =
                eigen( A[ikspin][set5,set5], B[ikspin][set5,set5] )
            end
            #
            # update wavefunction
            if iter > 1
                psiks[ikspin] = Y[ikspin]*G[ikspin][:,set1]
                ortho_sqrt!(psiks[ikspin])   # is this necessary ?
            else
                psiks[ikspin] = Y[ikspin][:,set5]*G[ikspin][set5,set1]
                ortho_sqrt!(psiks[ikspin])
            end
        end
        end

        Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )
        update!( Ham, Rhoe )

        Rhoe_old = copy(Rhoe)

        # Calculate energies once again
        Ham.energies = calc_energies( Ham, psiks )
        Etot = sum(Ham.energies)
        diffE = -(Etot - Etot_old)

        @printf("innerSCF: %5d %18.10f %18.10e", iterscf, Etot, diffE)
        # positive value of diffE is taken as reducing
        if diffE < 0.0
            @printf(" : Energy is not reducing !\n")
        else
            @printf("\n")
        end

    end

    # Calculate energies once again
    Ham.energies = calc_energies( Ham, psiks )
    Etot = sum(Ham.energies)
    diffE = abs( Etot - Etot_old )
    @printf("DCM: %5d %18.10f %18.10e\n", iter, Etot, diffE)

    if abs(diffE) < ETOT_CONV_THR
        @printf("DCM is converged: iter: %d , diffE = %10.7e\n", iter, diffE)
        break
    end

    Etot_old = Etot

    # No need to update potential, it is already updated in inner SCF loop
    for ispin = 1:Nspin
    for ik = 1:Nkpt
        ikspin = ik + (ispin - 1)*Nkpt
        if iter > 1
            P[ikspin] = Y[ikspin][:,set4]*G[ikspin][set4,set1]
        else
            P[ikspin] = Y[ikspin][:,set2]*G[ikspin][set2,set1]
        end
    end
    end

    flush(stdout)
end  # end of DCM iteration

Ham.electrons.ebands = evals[:,:]

if verbose && print_final_ebands
    @printf("\n")
    @printf("--------------------------\n")
    @printf("Final Kohn-Sham eigenvalues:\n")
    @printf("--------------------------\n")
```

```julia
        @printf("\n")
        print_ebands(Ham.electrons, Ham.pw.gvecw.kpoints)
    end

    if verbose && print_final_energies
        @printf("\n")
        @printf("-----------------------\n")
        @printf("Final Kohn-Sham energies:\n")
        @printf("-----------------------\n")
        @printf("\n")
        println(Ham.energies)
    end

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

    return
end

function KS_solve_TRDCM!( Ham::Hamiltonian;
                          NiterMax = 100, startingwfc=nothing,
                          verbose=true,
                          print_final_ebands=true, print_final_energies=true,
                          savewfc=false, ETOT_CONV_THR=1e-6 )


    pw = Ham.pw
    Ngw = pw.gvecw.Ngw
    Ns = pw.Ns
    Npoints = prod(Ns)
    CellVolume = pw.CellVolume
    ΔV = CellVolume/Npoints
    electrons = Ham.electrons
    Focc = electrons.Focc
    Nstates = electrons.Nstates
    Nocc = electrons.Nstates_occ
    Nelectrons = electrons.Nelectrons

    Nkpt = Ham.pw.gvecw.kpoints.Nkpt
    Nspin = electrons.Nspin

    Nkspin = Nkpt*Nspin

    psiks = Array{Array{ComplexF64,2},1}(undef,Nkspin)

    #
    # Initial wave function
    #
    if startingwfc == nothing
        psiks = rand_BlochWavefunc(pw, electrons)
    else
        psiks = startingwfc
    end

    #
    # Calculated electron density from this wave function and update Hamiltonian
    #
    Rhoe = zeros(Float64,Npoints,Nspin)

    Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )
    update!(Ham, Rhoe)

    evals = zeros(Float64,Nstates,Nkspin)

    # Starting eigenvalues and psi
    for ispin = 1:Nspin
    for ik = 1:Nkpt
        Ham.ik = ik
        Ham.ispin = ispin
        ikspin = ik + (ispin - 1)*Nkpt
        evals[:,ikspin], psiks[ikspin] =
        diag_LOBPCG( Ham, psiks[ikspin], verbose_last=true, NiterMax=100 )
    end
    end
```

```julia
# calculate E_NN
Ham.energies.NN = calc_E_NN( Ham.atoms )

# calculate PspCore energy
Ham.energies.PspCore = calc_PspCore_ene( Ham.atoms, Ham.pspots, CellVolume )

#
Ham.energies = calc_energies( Ham, psiks )
Etot = sum(Ham.energies)
Etot_old = Etot

# subspace
Y = Array{Array{ComplexF64,2},1}(undef,Nkspin)
R = Array{Array{ComplexF64,2},1}(undef,Nkspin)
P = Array{Array{ComplexF64,2},1}(undef,Nkspin)
G = Array{Array{ComplexF64,2},1}(undef,Nkspin)
T = Array{Array{Float64,2},1}(undef,Nkspin)
B = Array{Array{Float64,2},1}(undef,Nkspin)
A = Array{Array{Float64,2},1}(undef,Nkspin)
C = Array{Array{Float64,2},1}(undef,Nkspin)
for ispin = 1:Nspin
for ik = 1:Nkpt
    ikspin = ik + (ispin - 1)*Nkpt
    Y[ikspin] = zeros( ComplexF64, Ngw[ik], 3*Nstates )
    R[ikspin] = zeros( ComplexF64, Ngw[ik], Nstates )
    P[ikspin] = zeros( ComplexF64, Ngw[ik], Nstates )
    G[ikspin] = zeros( ComplexF64, 3*Nstates, 3*Nstates )
    T[ikspin] = zeros( Float64, 3*Nstates, 3*Nstates )
    B[ikspin] = zeros( Float64, 3*Nstates, 3*Nstates )
    A[ikspin] = zeros( Float64, 3*Nstates, 3*Nstates )
    C[ikspin] = zeros( Float64, 3*Nstates, 3*Nstates )
end
end

D = zeros(Float64,3*Nstates,Nkspin)  # array for saving eigenvalues of subspace problem

#XXX use plain 3d-array for G, T, and B ?

set1 = 1:Nstates
set2 = Nstates+1:2*Nstates
set3 = 2*Nstates+1:3*Nstates
set4 = Nstates+1:3*Nstates
set5 = 1:2*Nstates

MaxInnerSCF = 3
MAXTRY = 10
FUDGE = 1e-12
SMALL = 1e-12

sigma = zeros(Float64,Nkspin)
gapmax = zeros(Float64,Nkspin)

CONVERGED = 0

for iter = 1:NiterMax

    for ispin = 1:Nspin
    for ik = 1:Nkpt
        Ham.ik = ik
        Ham.ispin = ispin
        ikspin = ik + (ispin - 1)*Nkpt
        #
        Hpsi = op_H( Ham, psiks[ikspin] )
        #
        psiHpsi = psiks[ikspin]' * Hpsi
        psiHpsi = 0.5*( psiHpsi + psiHpsi' )
        # Calculate residual
        R[ikspin] = Hpsi - psiks[ikspin]*psiHpsi
        R[ikspin] = Kprec( ik, pw, R[ikspin] )
        # Construct subspace
        Y[ikspin][:,set1] = psiks[ikspin]
        Y[ikspin][:,set2] = R[ikspin]
        #
        if iter > 1
            Y[ikspin][:,set3] = P[ikspin]
        end

        #
        # Project kinetic and ionic potential
```

```julia
        #
        if iter > 1
            KY = op_K( Ham, Y[ikspin] ) + op_V_Ps_loc( Ham, Y[ikspin] )
            T[ikspin] = real(Y[ikspin]'*KY)
            B[ikspin] = real(Y[ikspin]'*Y[ikspin])
            B[ikspin] = 0.5*( B[ikspin] + B[ikspin]' )
        else
            # only set5=1:2*Nstates is active for iter=1
            KY = op_K( Ham, Y[ikspin][:,set5] ) + op_V_Ps_loc( Ham, Y[ikspin][:,set5] )
            T[ikspin][set5,set5] = real(Y[ikspin][:,set5]'*KY)
            bb = real(Y[ikspin][set5,set5]'*Y[ikspin][set5,set5])
            B[ikspin][set5,set5] = 0.5*( bb + bb' )
        end

        if iter > 1
            G[ikspin] = Matrix(1.0I, 3*Nstates, 3*Nstates) #eye(3*Nstates)
        else
            G[ikspin][set5,set5] = Matrix(1.0I, 2*Nstates, 2*Nstates)
        end
    end
end

@printf("TRDCM iter: %3d\n", iter)

sigma[:] .= 0.0  # reset sigma to zero at the beginning of inner SCF iteration
numtry = 0
Etot0 = sum(Ham.energies)

println("Etot0 = ", Etot0)

for iterscf = 1:MaxInnerSCF

    for ispin = 1:Nspin
    for ik = 1:Nkpt
        #
        Ham.ik = ik
        Ham.ispin = ispin
        ikspin = ik + (ispin - 1)*Nkpt
        #
        # Project Hartree, XC potential, and nonlocal pspot if any
        #
        V_loc = Ham.potentials.Hartree + Ham.potentials.XC[:,ispin]
        #
        if iter > 1
            yy = Y[ikspin]
        else
            yy = Y[ikspin][:,set5]
        end
        #
        if Ham.pspotNL.NbetaNL > 0
            VY = op_V_Ps_nloc( Ham, yy ) + op_V_loc( ik, pw, V_loc, yy )
        else
            VY = op_V_loc( ik, pw, V_loc, yy )
        end
        #
        if iter > 1
            A[ikspin] = real( T[ikspin] + yy'*VY )
            A[ikspin] = 0.5*( A[ikspin] + A[ikspin]' )
        else
            aa = real( T[ikspin][set5,set5] + yy'*VY )
            A[ikspin] = 0.5*( aa + aa' )
        end
        #
        if iter > 1
            BG = B[ikspin]*G[ikspin][:,1:Nocc]
            C[ikspin] = real( BG*BG' )
            C[ikspin] = 0.5*( C[ikspin] + C[ikspin]' )
        else
            BG = B[ikspin][set5,set5]*G[ikspin][set5,1:Nocc]
            cc = real( BG*BG' )
            C[ikspin][set5,set5] = 0.5*( cc + cc' )
        end
        #
        # apply trust region if necessary
        if abs(sigma[ikspin]) > SMALL # sigma is not zero
            println("Trust region is imposed")
            if iter > 1
                D[:,ikspin], G[ikspin] =
                eigen( A[ikspin] - sigma[ikspin]*C[ikspin], B[ikspin] )
```

```julia
                    else
                        D[set5,ikspin], G[ikspin][set5,set5] =
                        eigen( A[ikspin][set5,set5] - sigma[ikspin]*C[ikspin][set5,set5],
↪  B[ikspin][set5,set5] )
                    end
                else
                    if iter > 1
                        D[:,ikspin], G[ikspin] = eigen( A[ikspin], B[ikspin] )
                    else
                        D[set5,ikspin], G[ikspin][set5,set5] = eigen( A[ikspin][set5,set5],
↪  B[ikspin][set5,set5] )
                    end
                end
                #
                evals[:,ikspin] = D[1:Nstates,ikspin] .+ sigma[ikspin]
                #
                # update wavefunction
                if iter > 1
                    psiks[ikspin] = Y[ikspin]*G[ikspin][:,set1]
                    ortho_sqrt!(psiks[ikspin])  # is this necessary ?
                else
                    psiks[ikspin] = Y[ikspin][:,set5]*G[ikspin][set5,set1]
                    ortho_sqrt!(psiks[ikspin])
                end
            end
            end

            Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )
            update!( Ham, Rhoe )

            # Calculate energies once again
            Ham.energies = calc_energies( Ham, psiks )
            Etot = sum(Ham.energies)

            println("Etot = ", Etot)

            if Etot > Etot0

                @printf("TRDCM: %f > %f: Trust region will be imposed\n", Etot, Etot0)

                # Total energy is increased, impose trust region
                # Do this for all kspin

                for ikspin = 1:Nkspin

                    if iter == 1
                        gaps = D[2:2*Nstates,ikspin] - D[1:2*Nstates-1,ikspin]
                        gapmax[ikspin] = maximum(gaps)
                    else
                        gaps = D[2:3*Nstates] - D[1:3*Nstates-1]
                        gapmax[ikspin] = maximum(gaps)
                    end
                    gap0 = D[Nocc+1,ikspin] - D[Nocc,ikspin]

                    while (gap0 < 0.9*gapmax[ikspin]) && (numtry < MAXTRY)
                        println("Increase sigma to fix gap0: numtry = ", numtry)
                        @printf("gap0 : %f < %f\n", gap0, 0.9*gapmax[ikspin])
                        if abs(sigma[ikspin]) < SMALL # approx for sigma == 0.0
                            # initial value for sigma
                            sigma[ikspin] = 2*gapmax[ikspin]
                        else
                            sigma[ikspin] = 2*sigma[ikspin]
                        end
                        @printf("fix gap0: ikspin = %d, sigma = %f\n", ikspin, sigma[ikspin])
                        #
                        if iter > 1
                            D[:,ikspin], G[ikspin] = eigen( A[ikspin] - sigma[ikspin]*C[ikspin], B[ikspin]
↪  )
                            gaps = D[2:2*Nstates,ikspin] - D[1:2*Nstates-1,ikspin]
                        else
                            D[set5,ikspin], G[ikspin][set5,set5] =
                            eigen( A[ikspin][set5,set5] - sigma[ikspin]*C[ikspin][set5,set5],
↪  B[ikspin][set5,set5] )
                            gaps = D[2:3*Nstates,ikspin] - D[1:3*Nstates-1,ikspin]
                        end
                        gapmax[ikspin] = maximum(gaps)
                        gap0 = D[Nocc+1,ikspin] - D[Nocc,ikspin]
                        numtry = numtry + 1
                    end
```

```
        end # Nkspin

    end # if Etot > Etot0

    println("sigma = ", sigma)
    numtry = 0  # reset numtry for this while loop

    while (Etot > Etot0) &
        #(abs(Etot-Etot0) > FUDGE*abs(Etot0)) &
        (numtry < MAXTRY)
        @printf("Increase sigma part 2: %f > %f ?\n", Etot, Etot0)
        #
        # update wavefunction
        #
        for ikspin = 1:Nkspin
            if iter > 1
                psiks[ikspin] = Y[ikspin]*G[ikspin][:,set1]
                ortho_gram_schmidt!(psiks[ikspin])
            else
                psiks[ikspin] = Y[ikspin][:,set5]*G[ikspin][set5,set1]
                ortho_gram_schmidt!(psiks[ikspin])
            end
        end
        #
        Rhoe[:,:] = calc_rhoe( Nelectrons, pw, Focc, psiks, Nspin )
        update!( Ham, Rhoe )

        # Calculate energies once again
        Ham.energies = calc_energies( Ham, psiks )
        Etot = sum(Ham.energies)
        #
        if Etot > Etot0
            println("Increase sigma part 2")
            for ikspin = 1:Nkspin
                if abs(sigma[ikspin]) > SMALL # sigma is not 0
                    sigma[ikspin] = 2*sigma[ikspin]
                else
                    sigma[ikspin] = 1.2*gapmax[ikspin]
                end
                @printf("ikspin = %d sigma = %f\n", ikspin, sigma[ikspin])
                if iter > 1
                    D[:,ikspin], G[ikspin] = eigen( A[ikspin] - sigma[ikspin]*C[ikspin], B[ikspin]
↪  )
                else
                    D[set5,ikspin], G[ikspin][set5,set5] = eigen( A[ikspin][set5,set5] -
↪  sigma[ikspin]*C[ikspin][set5,set5], B[ikspin][set5,set5] )
                end
            end
        end
        numtry = numtry + 1  # outside ikspin loop
    end # while

    Etot0 = Etot

end # end of inner SCF iteration

# Calculate energies once again
Ham.energies = calc_energies( Ham, psiks )
Etot = sum(Ham.energies)
diffE = abs( Etot - Etot_old )
@printf("TRDCM: %5d %18.10f %18.10e\n", iter, Etot, diffE)

if diffE < ETOT_CONV_THR
    CONVERGED = CONVERGED + 1
else  # reset CONVERGED
    CONVERGED = 0
end

if CONVERGED >= 2
    @printf("TRDCM is converged: iter: %d , diffE = %10.7e\n", iter, diffE)
    break
end

Etot_old = Etot

# No need to update potential, it is already updated in inner SCF loop
for ispin = 1:Nspin
for ik = 1:Nkpt
    ikspin = ik + (ispin - 1)*Nkpt
```

```julia
            if iter > 1
                P[ikspin] = Y[ikspin][:,set4]*G[ikspin][set4,set1]
            else
                P[ikspin] = Y[ikspin][:,set2]*G[ikspin][set2,set1]
            end
        end
        end

        flush(stdout)
    end  # end of DCM iteration

    Ham.electrons.ebands = evals

    if verbose && print_final_ebands
        @printf("\n")
        @printf("---------------------------\n")
        @printf("Final Kohn-Sham eigenvalues:\n")
        @printf("---------------------------\n")
        @printf("\n")
        print_ebands(Ham.electrons, Ham.pw.gvecw.kpoints)
    end

    if verbose && print_final_energies
        @printf("\n")
        @printf("------------------------\n")
        @printf("Final Kohn-Sham energies:\n")
        @printf("------------------------\n")
        @printf("\n")
        println(Ham.energies)
    end


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


    return
end
```

## 13.5   Chebyshev filtered subspace iteration SCF

```julia
update_psi="CheFSI"

function diag_CheFSI!( Ham::Hamiltonian, psiks::BlochWavefunc, cheby_degree::Int64)
    Nspin = Ham.electrons.Nspin
    Nkpt = Ham.pw.gvecw.kpoints.Nkpt
    Nstates = Ham.electrons.Nstates

    for ispin = 1:Nspin
    for ik = 1:Nkpt
        Ham.ik = ik
        Ham.ispin = ispin
        ikspin = ik + (ispin - 1)*Nkpt
        ub, lb = get_ub_lb_lanczos( Ham, Nstates*2 )
        psiks[ikspin] = chebyfilt( Ham, psiks[ikspin], cheby_degree, lb, ub)
        psiks[ikspin] = ortho_sqrt( psiks[ik] )
    end
    end
end


function chebyfilt( Ham::Hamiltonian, X, degree, lb, ub)
    Ngw_ik  = size(X)[1]
    Nstates = size(X)[2]
    #
    ee = (ub - lb)/2
    c = (ub + lb)/2
    sigma = ee/(lb-ub)
    sigma1 = sigma
    #
    Y = zeros(ComplexF64,Ngw_ik,Nstates)
    Y1 = zeros(ComplexF64,Ngw_ik,Nstates)
    #
    Y = op_H(Ham, X) - X*c
```

```julia
    Y = Y*sigma1/ee
    #
    for i = 2:degree
        sigma2 = 1/(2/sigma1 - sigma)
        Y1 = ( op_H(Ham,Y) - Y*c)*2 * sigma2/ee - X*(sigma*sigma2)
        X = Y
        Y = Y1
        sigma = sigma2
    end
    return Y
end


function get_ub_lb_lanczos( Ham::Hamiltonian, nlancz::Int64 )
    #
    pw = Ham.pw
    ik = Ham.ik
    #
    Ngw_ik = pw.gvecw.Ngw[ik]
    V = zeros(ComplexF64,Ngw_ik,nlancz)
    HV = zeros(ComplexF64,Ngw_ik,nlancz)
    T = zeros(Float64,nlancz,nlancz)
    f = zeros(ComplexF64,Ngw_ik)
    s = zeros(ComplexF64,nlancz)
    h = zeros(ComplexF64,nlancz)
    #
    V[:,1] = randn(Ngw_ik) + im*randn(Ngw_ik)
    beta = norm(V[:,1])
    V[:,1] = V[:,1] ./ beta
    #
    HV[:,1] = op_H( Ham, V[:,1] )
    h[1] = real( V[:,1]' * HV[:,1] )
    #
    T[1,1] = h[1]
    # One-step of reorthogonalization
    f[:] = HV[:,1] - V[:,1]*h[1]
    s[1] = V[:,1]' * f[:]
    h[1] = h[1] + s[1]
    f[:] = f[:] - V[:,1]*s[1]
    # MAIN LOOP
    for j = 2:nlancz
        #@printf("iter lanczos = %d\n", j)
        beta = norm(f)
        T[j,j-1] = beta
        V[:,j] = f[:]/beta
        HV[:,j] = op_H( Ham, V[:,j] )
        #
        for jj = 1:j
            h[jj] = V[:,jj]' * HV[:,j]
        end
        f[:] = HV[:,j] - V[:,1:j]*h[1:j]

        # One-step of reorthogonalization
        #s = V' * f
        #h = h + s
        for jj=1:j
            s[jj] = V[:,jj]' * f[:]
            h[jj] = h[jj] + s[jj]
        end
        f[:] = f[:] - V[:,1:j]*s[1:j]
        #
        T[1:j,j] = real(h[1:j])
    end
    #
    evalsT = eigvals(T)
    #lb = evalsT[Nstates+2]
    #ub = evalsT[2*Nstates]
    lb = evalsT[Int64(nlancz/2)]
    ub = norm_matrix_induced(T) + norm(f)
    #
    return lb, ub
end


function norm_matrix_induced(A::Array{Float64,2})
    N = size(A)[1]
    # FIXME no check for matrix form

    # unit-norm vector
```

```
    d = 1/sqrt(N)
    v1 = ones(N)*d
    #
    v = A*v1
    return norm(v)
end
```