Electronic Structure Calculation using Plane Wave Basis Set

Fadjar Fathurrahman

Contents

Chapter 1

Introduction

This document is part of package `ffr-ElectronicStructure.jl` using plane wave basis set.
WARNING: This document is under heavy construction

1.1 Outline

First, I will describe the top level description about what the a typical electronic structure calculation based on density functional theory is carried out. After that, I will break it apart into smaller pieces which are hopefully easier to implement and be understood.
In a typical DFT calculation we are trying to solve the so-called Kohn-Sham equations:

$$H_{KS}\psi_{KS} = e_{KS}\psi_{KS} \tag{1.1}$$

Tasks:

- Setting up data structure for PW basis set
- Solving Poisson equation using FFT
- Solving Schrodinger equation: diagonalization, energy minimization
- Solving KS equation: SCF and energy minimzation

Chapter 2

Solving Poisson and Schrodinger equation

1. plane wave basis set
2. solving Poisson equation
3. solving Schrodinger equation
4. solving Kohn-Sham equations

2.1 First implementation of plane wave basis: `PWGrid_01.jl`

To describe a plane wave basis, we need to define our periodic simulation box by specifying three lattice vectors. We also need to specify number sampling points for each lattice vector. We will store the lattice vector in 3×3 matrix. (add convention for lattice vectors, probably using the same convention as PWSCF input file).
In file `PWGrid_01.jl`, we give an implementation of plane wave basis set, which is encapsulated in a user-defined type `PWGrid`. An instance of `PWGrid` can be initialize via code like this:

```
Ns = [40, 40, 40]  # sampling points
LatVecs = 10*diagm(ones(3))  # lattice vectors for cubic system
pw = PWGrid( Ns, LatVecs )
```

2.1.1 Details of `PWGrid`

Let's look into details of `PWGrid`.
`PWGrid` is defined like this:

```
type PWGrid
  Ns::Array{Int64}
  LatVecs::Array{Float64,2}
  RecVecs::Array{Float64,2}
  Npoints::Int
  Ω::Float64
  r::Array{Float64,2}
  G::Array{Float64,2}
  G2::Array{Float64}
end
```

Some explanation about these fields follow:

- `Ns` is an integer array which defines number of sampling points in each lattice vectors.
- `LatVecs` is 3×3 matrix which defines lattice vectors of unit cell in real space.
- `RecVecs` is 3×3 matrix which defines lattice vectors of unit cell in reciprocal space. It is calculated according to (3.2).

- `Npoints` Total number of sampling points
- `Ω` Unit cell volume in real space
- `r` Real space grid points
- `G` **G**-vectors
- `G2` Magnitude of **G**-vectors

The constructor for `PWGrid` is defined as follow.

```
function PWGrid( Ns::Array{Int,1}, LatVecs::Array{Float64,2} )
  Npoints = prod(Ns)
  RecVecs = 2*pi*inv(LatVecs')
  Ω = det(LatVecs)
  R,G,G2 = init_grids( Ns, LatVecs, RecVecs )
  return PWGrid( Ns, LatVecs, RecVecs, Npoints, Ω, R, G, G2 )
end
```

The function `init_grid()` is defined as follow. It takes `Ns`, `LatVecs`, and `RecVecs` as the arguments.

```
function init_grids( Ns, LatVecs, RecVecs )
```

First, grid points in real space are initialized:

```
  Npoints = prod(Ns)
  r = Array(Float64,3,Npoints)
  ip = 0
  for k in 0:Ns[3]-1
  for j in 0:Ns[2]-1
  for i in 0:Ns[1]-1
    ip = ip + 1
    r[1,ip] = LatVecs[1,1]*i/Ns[1] + LatVecs[2,1]*j/Ns[2]
              + LatVecs[3,1]*k/Ns[3]
    r[2,ip] = LatVecs[1,2]*i/Ns[1] + LatVecs[2,2]*j/Ns[2]
              + LatVecs[3,2]*k/Ns[3]
    r[3,ip] = LatVecs[1,3]*i/Ns[1] + LatVecs[2,3]*j/Ns[2]
              + LatVecs[3,3]*k/Ns[3]
  end
  end
  end
```

In the next step, grid points in reciprocal space, or **G**-vectors and also their squared values are initialized

```
  G  = Array(Float64,3,Npoints)
  G2 = Array(Float64,Npoints)
  ip    = 0
  for k in 0:Ns[3]-1
  for j in 0:Ns[2]-1
  for i in 0:Ns[1]-1
    gi = mm_to_nn( i, Ns[1] )
    gj = mm_to_nn( j, Ns[2] )
    gk = mm_to_nn( k, Ns[3] )
    ip = ip + 1
    G[1,ip] = RecVecs[1,1]*gi + RecVecs[2,1]*gj + RecVecs[3,1]*gk
    G[2,ip] = RecVecs[1,2]*gi + RecVecs[2,2]*gj + RecVecs[3,2]*gk
    G[3,ip] = RecVecs[1,3]*gi + RecVecs[2,3]*gj + RecVecs[3,3]*gk
    G2[ip] = G[1,ip]^2 + G[2,ip]^2 + G[3,ip]^2
  end
  end
  end
```

The function `mm_to_nn` defines mapping from real space to Fourier space:

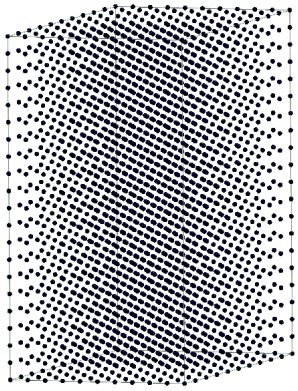

Figure 2.1: Visualization of real space grid points of a hexagonal unit cell.

```
function mm_to_nn(mm::Int,S::Int)
  if mm > S/2
    return mm - S
  else
    return mm
  end
end
```

Finally, the variables `r`, `G`, and `G2` are returned.

```
  return r,G,G2
```

2.1.2 Visualizing real-space grid points

In the directory `pwgrid_01`, we visualize grid points in real space using Xcrysden program. Originally Xcrysden, is meant to visualize crystalline structure, however, we also can use it to visualize grid points, taking periodic boundary conditions into consideration. This is useful to check whether grid points are generated correctly or not. An example of such visualization is shown in Figure 2.1

2.2 Solving Poisson equation

Poisson equation is relatively easy to solve in periodic boundary condition using Fourier method (FFT). This situtation is different from other discretization method, such as finite difference or Lagrange function.
Poisson equation in **G**-space:

$$-G^2\tilde{V}_{\mathrm{Ha}}(\mathbf{G}) = -4\pi\tilde{\rho}(\mathbf{G}) \tag{2.1}$$

Hartree potential can be solve directly, for $\mathbf{G} \neq \mathbf{0}$:

$$\tilde{V}_{\mathrm{Ha}}(\mathbf{G}) = \frac{4\pi\tilde{\rho}(\mathbf{G})}{G^2} \tag{2.2}$$

An example a program to solve Poisson equation is given in directory `poisson_01`. In this program, a charge density is constructed from difference between two Gaussian charge density. Total charge (integrated charge density) is restricted to zero. From this charge density, we calculate the electrostatic (Hartree) potential by solving Poisson equation.
Function to generate vector `dr`:

```
function gen_dr( r, center )
  Npoints = size(r)[2]
  dr = Array(Float64,Npoints)
  for ip=1:Npoints
    dx2 = ( r[1,ip] - center[1] )^2
```

```
    dy2 = ( r[2,ip] - center[2] )^2
    dz2 = ( r[3,ip] - center[3] )^2
    dr[ip] = sqrt( dx2 + dy2 + dz2 )
  end
  return dr
end
```

Function to generate charge density:

```
function gen_rho( dr, σ1, σ2 )
  Npoints = size(dr)[1]
  rho = Array( Float64, Npoints )
  c1 = 2*σ1^2
  c2 = 2*σ2^2
  cc1 = sqrt(2*pi*σ1^2)^3
  cc2 = sqrt(2*pi*σ2^2)^3
  for ip=1:Npoints
    g1 = exp(-dr[ip]^2/c1)/cc1
    g2 = exp(-dr[ip]^2/c2)/cc2
    rho[ip] = g2 - g1
  end
  return rho
end
```

Function to solve Poisson equation:

```
function solve_poisson( pw_grid::PWGrid, rho )
  Ω  = pw_grid.Ω
  G2 = pw_grid.G2
  Ns = pw_grid.Ns
  Npoints = pw_grid.Npoints
  ctmp = 4.0*pi*R_to_G( Ns, rho )
  for ip = 2:Npoints
    ctmp[ip] = ctmp[ip] / G2[ip]
  end
  ctmp[1] = 0.0
  phi = real( G_to_R( Ns, ctmp ) )
  return phi
end
```

2.3 Calculation of structure factor and Ewald energy: first version

Structure factor

```
# Calculate structure factor
# special case: only for 1 species with Z = 1
function structure_factor( Xpos::Array{Float64,2}, G::Array{Float64,2} )
  Ng = size(G)[2]
  Na = size(Xpos)[2]
  Sf = zeros(Complex128,Ng)
  for ia = 1:Na
    for ig = 1:Ng
      GX = Xpos[1,ia]*G[1,ig] +
           Xpos[2,ia]*G[2,ig] +
           Xpos[3,ia]*G[3,ig]
      Sf[ig] = Sf[ig] + cos(GX) - im*sin(GX)
    end
  end
  return Sf
end
```

A simple method to calculate Ewald energy

```
function calc_ewald( pw::PWGrid, Xpos, Sf; sigma=0.25 )
  #
  const Npoints = pw.Npoints
  const Ω  = pw.Ω
  const r  = pw.r
  const Ns = pw.Ns
  const G2 = pw.G2
  #
  # Generate array of distances
  center = sum(pw.LatVecs,2)/2
  dr = gen_dr( r, center )
  #
  # Generate charge density
  rho = gen_rho( Ns, dr, sigma, Sf )
  intrho = sum(rho)*Ω/Npoints
  #
  # Solve Poisson equation and calculate Hartree energy
  ctmp = 4.0*pi*R_to_G( Ns, rho )
  ctmp[1] = 0.0
  for ip = 2:Npoints
    ctmp[ip] = ctmp[ip] / G2[ip]
  end
  phi = real( G_to_R( Ns, ctmp ) )
  Ehartree = 0.5*dot( phi, rho ) * Ω/Npoints
  #
  Eself = 1.0/(2*sqrt(pi))*(1.0/sigma)*size(Xpos,2)
  return Ehartree - Eself
end
```

2.4 Solving Schrodinger equation

2.4.1 Operators

Kinetic energy operators (multicolumns):

```
function op_K( pw::PWGrid, psi::Array{Complex128,2} )
  out = zeros(Complex128,size(psi))
  Ncol = size(psi,2)
  Ω = pw.Ω
  G2 = pw.G2
  Npoints = pw.Npoints
  for is = 1:Ncol
    for ip = 1:Npoints
      out[ip,is] = psi[ip,is]*G2[ip]
    end
  end
  return 0.5*out
end
```

Applying potential

```
function op_Vpot( pw::PWGrid, Vpot, psi::Array{Complex128,2} )
  Ns = pw.Ns
  Ω = pw.Ω
  Npoints = prod(Ns)
  # get values of psi in real space grid via forward transform
  ctmp = G_to_R( Ns, psi )
  return R_to_G( Ns, Diagprod(Vpot, ctmp) )
end
```

Function `Diagprod`:

```
function Diagprod( a,B )
  Ncol    = size(B)[2]
  Npoints = size(B)[1]
  out = zeros( Complex128, size(B) )
  for ic = 1:Ncol
    for ip = 1:Npoints
      out[ip,ic] = a[ip]*B[ip,ic]
    end
  end
  return out
end
```

Hamiltonian operator:

```
function op_H( pw, Vpot, psi )
  return op_K( pw, psi ) + op_Vpot( pw, Vpot, psi )
end
```

2.4.2 Gradient calculation

Gradient of energy with respect to wave function
Not using occupation number

```
function calc_grad( pw::PWGrid, Vpot, psi::Array{Complex128,2} )
  Npoints = size(psi)[1]
  Nstates = size(psi)[2]
  Ω = pw.Ω
  Ns = pw.Ns
  #
  grad = zeros( Complex128, Npoints, Nstates )
  H_psi = op_H( pw, Vpot, psi )
  for i = 1:Nstates
    grad[:,i] = H_psi[:,i]
    for j = 1:Nstates
      grad[:,i] = grad[:,i] - dot( psi[:,j], H_psi[:,i] ) * psi[:,j]
    end
  end
  return grad
end
```

2.4.3 Calculation of charge density

```
function calc_rho( pw::PWGrid, psi::Array{Complex128,2} )
  Ω = pw.Ω
  Ns = pw.Ns
  Npoints = pw.Npoints
  Nstates = size(psi)[2]
  #
  ρ = zeros(Complex128,Npoints)
  # Transform to real space
  psiR = G_to_R(Ns,psi)
  # orthonormalization in real space
  ortho_gram_schmidt!(Nstates,psiR); scale!(sqrt(Npoints/Ω),psiR)
  for is = 1:Nstates
    for ip = 1:Npoints
      ρ[ip] = ρ[ip] + conj(psiR[ip,is])*psiR[ip,is]
    end
  end
  return real(ρ)
end
```

2.4.4 Calculation of total energy

```
function calc_Etot( pw::PWGrid, Vpot, psi::Array{Complex128,2} )
  Ω = pw.Ω
  Npoints = pw.Npoints
  Nstates = size(psi)[2]
  Kpsi = op_K( pw, psi )
  Ekin = 0.0
  for is = 1:Nstates
    Ekin = Ekin + real( dot( psi[:,is], Kpsi[:,is] ) )
  end
  # Calculate in real space
  rho = calc_rho( pw, psi )
  Epot = dot( rho, Vpot ) * Ω/Npoints
  Etot = Ekin + Epot
  return Etot
end
```

2.4.5 Energy minimization with steepest descent

```
function Sch_solve_Emin_sd( pw::PWGrid, Vpot, psi::Array{Complex128,2};
                            NiterMax=1000 )
  α = 3e-5
  Etot_old = 0.0
  Etot = 0.0
  for iter = 1:NiterMax
    psi = psi - α*calc_grad( pw, Vpot, psi )
    psi  = ortho_gram_schmidt(psi)
    Etot = calc_Etot( pw, Vpot, psi )
    conv = abs(Etot-Etot_old)
    if conv < 1e-6
      break
    end
    Etot_old = Etot
  end
  return psi, Etot
end
```

2.4.6 Energy minimization with conjugate gradient

```
function Sch_solve_Emin_cg( pw::PWGrid, Vpot, psi::Array{Complex128,2};
                            NiterMax=1000 )
  #
  Npoints = size(psi)[1]
  Nstates = size(psi)[2]
  d = zeros(Complex128, Npoints, Nstates)
  g_old  = zeros(Complex128, Npoints, Nstates)
  d_old  = zeros(Complex128, Npoints, Nstates)
  Kg     = zeros(Complex128, Npoints, Nstates)
  Kg_old = zeros(Complex128, Npoints, Nstates)
  #
  α_t = 1.e-5
  β = 0.0
  Etot_old = 0.0
  Etot = 0.0
  #
  for iter = 1:NiterMax
    g = calc_grad( pw, Vpot,  psi)
    nrm = 0.0
    for is = 1:Nstates
      nrm = nrm + real( dot( g[:,is], g[:,is] ) )
    end
    Kg = Kprec(pw,g)
```

```
    if iter != 1
      β = real( sum( conj(g) .* Kg ) ) / real( sum( conj(g_old) .* Kg_old ) )
    end
    d = -Kg + β * d_old
    psic = ortho_gram_schmidt(psi + α_t*d)
    gt = calc_grad( pw, Vpot, psic )
    if real(trace((g-gt)'*d)) != 0.0
      α = abs(α_t*real(sum(conj(g).*d))/real(sum(conj(g-gt).*d)))
    else
      α = 0.0
    end
    # Update wavefunction
    psi = psi[:,:] + α*d[:,:]
    psi = ortho_gram_schmidt(psi)
    Etot = calc_Etot( pw, Vpot, psi )
    diff = abs(Etot-Etot_old)
    @printf("E step %8d = %18.10f %18.10f %18.10f\n", iter, Etot, diff, nrm/Nstates)
    if diff < 1e-6
      @printf("CONVERGENCE ACHIEVED\n")
      break
    end
    g_old = copy(g)
    d_old = copy(d)
    Kg_old = copy(Kg)
    Etot_old = Etot
  end
  return psi, Etot
end
```

Using energy minimization:
Introduction to minimization
simple 2D minimization, using steepest-descent and conjugate gradient method
Using iterative diagonalization: Davidson and LOBPCG
background information about iterative diagonalization
Eigenvalue problems

Chapter 3

Formulae

This chapter may be not required

Plane wave basis $b_\alpha(\mathbf{r})$:

$$b_\alpha(\mathbf{r}) = \frac{1}{\sqrt{\Omega}} e^{\mathbf{G}_\alpha \cdot \mathbf{r}} \tag{3.1}$$

Lattice vectors of unit cell in reciprocal space:

$$\mathbf{b} = 2\pi \left(a^T\right)^{-1} \tag{3.2}$$

G-vectors:

$$\mathbf{G} = i\mathbf{b}_1 + j\mathbf{b}_2 + k\mathbf{b}_3 \tag{3.3}$$

Structure factor:

$$S_I(\mathbf{G}) = \sum_{\mathbf{G}} e^{-\mathbf{G}\cdot\mathbf{X}_I} \tag{3.4}$$