Cornell University

Department of Physics

Practical DFT February 22, 2014

Phys 7654, Module #2, Spring 2014

Practical Density-Functional Theory

Homework Assignment # 4

(Optional... but rewarding!!!)

Agenda and readings:

Goals: Implement variational solution to Schrödinger's equation and Kohn -Sham equations using steepest descents.

Readings: The discussion in the course is meant to be self-contained. The readings are given for those who would like more background and/or who like to learn from readings as well. All readings are available on the course web site under "2011 Phys 7654: Practicl DFT" at TomasArias.com . Readings from Numerical Recipes are from *Numerical Recipes, 2nd ed*, which is available online also under "2011 Phys 7654 Practical DFT⇒Readings" at TomasArias.com .

Lecture overviews:

- **Lecture 1: Density functional theory introduction and Expressive Software Concept.** Course overview; Quick review of density-functional theory formalism and successes of density functional theory; How to take a computational approach to problems in physics, best practices; Expressive software.
Reading: "Intro-to-DFT" and Sections 1, 2, (3 is optional), 4.1 and 4.2 of "DFT++-Derivations".

- **Lecture 2: Expressive software for Poisson's equation and Spectral Methods** Solution to Poisson's equation in a single line of code; Spectral methods and Choice of plane-wave (complex exponential) basis for Poisson's equation; Form of operators for planewaves; Introduction to octave/matlab notations; Selection of sample points in periodic boundary conditions.
Reading:4.1 and 4.2 of "DFT++-Derivations".

- **Lecture 3: An expressive software language for density-functional theory.** Choice of wave vectors for periodic boundary conditions, $\mathbf{N}$ and $\mathbf{M}$ matrices; expression of density-functional theory in terms of computational representation for wave functions $\mathbf{C}$ and density $\hat{\tilde{n}}$.
Reading: Section 4.3 of "DFT++-Derivations".

- **Lecture 4: Expressions/code for minimization of density-functional energy.** Expression of density and orthonormality constraints in terms of wave-function representation $\mathbf{C}$; minimization by steepest descents; analytic continuation for constraints (generalization of Rayleigh-Ritz principle to multople states and density functional theory); differential matrix calculus, relation of gradient with respect to real and imaginary components; gradient of kinetic energy in general form involving density matrix. **Reading: Sections 4.4-4.7 of "DFT++-Derivations".**

- **Lecture 5: Improved minimization algorithms** Complete derivation of gradient of DFT energy expressions for total energy in density-functional theory; numerical analysis of minimization algoritms and how to improve them; improved minimization algorithms including preconditioned conjugate gradients. **Reading:** ***Numerical Recipes, 2nd ed*** **10.2, 10.3. 10.5, 10.6**
- **Lecture 6: Further improvements/optimizations.** Art of preconditioning; Line minimization algorithms; Extraction of eigenstates from minimized wave functions. Minimal (isotropic) representations in plane waves; implemention of operators; pseudopotentials; extraction of eigenstates.

Contents

1 Minimal, isotropic spectral representation

Expanding the wave functions with an appropriate subset of the full get of basis functions reduces the memory and run-time requirements by a factor of about sixteen in the limit of a large calculation. Fortunately, our software design limits all knowledge of the details of the basis set to the operators O(), L(), cI(), cIdag(), cJ(), cJdag(). Thus, modifying our code to take advantage of reducing the basis set requires only modification of the corresponding functions (if needed) and of the initialization file "setup.m".

This optimization represents major modifications to your basic operators. Copy all of the current ".m" files from "∼/A3" into a new working directory "∼/A3a".

1.1 "setup.m"

As discussed in lecture, the active basis functions which we shall use are those contained within a sphere of radius $G_n/2$, where G_n is the radius of the sphere inscribed within the edges of our Fourier box. Modify "setup.m" as follows to determine the associated information and communicate it to the rest of the software as follows.

1.1.1 Radius of inscribed sphere

To locate the edges of the Fourier box, insert the following code block immediately after your computation of G2 in "setup.m",

```
%# Locate edges (assume S's are even!) and determine max 'ok' G2
if any(rem(S,2)!=0)
  printf("Odd dimension in S, cannot continue...\n");
  return;
endif
eS=S/2+0.5;
edges=find(any(abs(M-ones(size(M,1),1)*eS')<1,2));
```

This block first verifies the assumption that the FFT box sizes are all even. After execution of this code block, `edges` contains a list of the indices of all rows of M with any element equal to S/2 or S/2+1, namely those with any component of $\vec{G}$ of largest possible absolute value.

Next, to find a list of the active rows of G (those corresponding to wave vectors within the Fourier sphere), first compute the minimum magnitude reciprocal lattice vector along the edges, and then find all vectors with magnitude less than one-half of that minimum magnitude (one-fourth of the square magnitude), by adding the code block

```
%# Compute active list and corresponding G2's
G2mx=min(G2(edges));
active=find(G2<G2mx/4); %# Sphere is 1/2 size (but looking at G^2!)
G2c=G2(active);

printf("Compression: %f (theoretical: %f)\n", ...
                    length(G2)/length(G2c), 1/(4*pi*(1/4)^3/3));
```

immediately after the block determining the edges. Note that this will produce in `G2c` a sequential list (without gaps!) of the values of G^2 associated with the rows listed in `active`.

Finally, be sure to make the values of `G2c` and `active` available to the various operators by getting up a global variable `gbl_G2c` in just the same way `gbl_G` and the other global variables are handled.

1.1.2 Debugging

Run the modified “setup.m” for parameters for a small hydrogen atom calculation, S=[32; 32; 32], R=diag([8 8 8]), X=[0 0 0], Z=1. You should find a basis set compression ratio of about 18.296, a bit larger than the theoretical value. (We still have relatively few points so that the number of points actually falling within a sphere isn’t exactly the volume of the sphere. Also, the number is larger because we have “rounded” down in all cases.)

1.2 Operators

For debugging purposes, you should now rerun your Hatoms.m calculation with the same parameters as above for “setup.m” (S=[32; 32; 32], R=diag([8 8 8]), X=[0 0 0]) *and* gbl_f=1, Ns=1 in “Hatoms.m” . Take note of your final converged value, as we will be checking that your reduced basis calculation gives essentially the same result. (I find E=-0.2841 H for the electronic part of the energy.)

Of the operators, it turns out that Linv(), cJ() and cJdag() are never called to act on the wave functions, and so need no modification. (Ultimately, the fact that your software runs unmodified without octave generating any “nonconformant” size errors confirms this.) Also, as shown in lecture, the overlap operator is still just multiplication by the same constant, $\mathcal{O} = \det(R)\mathbf{1}$, and so requires no modification. This, then, leaves the following operators to be modified.

1.2.1 “L.m” and “K.m”

Both L() and K() are diagonal operators based on the values of G2. We thus need only check whether the input has a size corresponding to G2 or G2c and to then use the values from the corresponding array. This can be accomplished by adding the statement “global gbl_G2c” to the top of each of the above functions and employing constructions “`if size(in,1)==length(gbl_G2c) [code block using G2c] else [code block using G2] endif`” which execute, as appropriate, either your current code or your current code with G2c replacing G2.

1.2.2 “cI.m”

In lecture, we found that the I() operator becomes $\mathbf{I} = \mathbf{F_{(3)}}\mathbf{B}$, where “$\mathbf{F}_{(3)}$” is the three-dimensional Fourier transform kernel and “**B**” maps data from sphere of active basis functions into the Fourier box. To implement this, include “global gbl_active” at the top of cI(). Note that, regardless of the number of rows of the input data, the output always has the same size of data, the full FFT box. Thus, you should change your initial allocation of the output to

```
out=zeros(prod(gbl_S),size(in,2));
```

Finally, to perform the mapping **B** when needed, you should use a code block along the lines of

```
if size(in,1)==prod(gbl_S)
  out(:,col)= ... %# previous code
else
  full=zeros(prod(gbl_S),1); full(gbl_active)=in(:,col);
  out(:,col)= ... %# previous code with "full" replacing "in(:,col)"
end
```

Note that, here and below, “along the lines of” indicates that the names of some variables (especially internal working variables of a function which you designed) in the example may not correspond exactly to those in your routine.

1.2.3 “cIdag.m”

The Idag() operator now becomes $\mathbf{I}^\dagger = \left(\mathbf{F}_{(3)}\mathbf{B}\right)^\dagger = \mathbf{B}^\dagger\mathbf{F}^\dagger_{(3)}$. Again, to implement this, you will need “global gbl_active” at the top of the function. In this case, it turns out that the output cIdag() is always a wave-function-like object (again, the lack of “nonconformant” errors ultimately confirms this) and the output should be allocated as

```
out=zeros(length(gbl_active),size(in,2));
```

Finally, the operation $\mathbf{B}^\dagger \left(\mathcal{F}^{(3)}\right)^\dagger$ may be implemented using a statement along the lines of

```
full=fft3(in(:,col),gbl_S,-1); out(:,col)=full(gbl_active);
```

Note that, because the output of cIdag() is always in the reduced, active space, the function requires no if statements in its implementation.

1.2.4 Debugging

Now, modify and rerun your hydrogen calculation in “Hatoms.m” and verify that you reproduce your previous results to within better than 0.005 hartree! The modification to “Hatoms.m” should involve only changing the initialization of W to that of a random array of size length(gbl_active)$\times$Ns. You should also notice a decrease in the iteration and an improvement in the number of iterations needed to reach convergence.
Hints: For debugging the above operators (most bugs will involve various “nonconformant” errors), make liberal use the size() function in your various operators in order to print out the sizes of the relevant objects from within the functions that generate errors.

2 Full benefit from minimal representation

Currently, the wave functions are stored in the minimal representation. However, when transformed to sample values with I(), as occurs in getE() and H(), the output lives in the full FFT box and consumes an unnecessarily large amount of memory (sixteen times more than is needed), whose allocation and deallocation also tends to slow the calculation. To avoid this, we should apply I() only to one column of the wave functions at a time. To achieve this, make the following modifications.

2.1 getn(), getE(), H()

```
function n=getn(psi,f)
```

Input:

- psi: expansion coefficients of N_s *orthonormal* wave functions
- f: fillings of the orbitals (typically f=2).

Output:

- n: sample values of electron density, $n_i = f \sum_k \psi^*_{ik}\psi_{ik}$

Both getE() and H() compute the electron density with a call to I() on the wave functions. Rather than updating two nearly identical blocks of code, produce a function of the above prototype which calculates the density *with an explicit loop over the columns of psi rather than using calls to diagouter().* Note that you should not need to use any global variables for this function.

Finally, replace the current expressions which calculate the density n in getE() and H() with calls to getn(). Note that, depending on your implementation, you may have to modify getE() and H() somewhat to first obtain the orthonormal wave functions from the input W. The following statement will do this for you,

```
Y=W*inv(sqrtm(W'*O(W))); %# Orthonormal wave functions
```

Debugging: Rerun "Hatoms.m" to convince yourself that the output is identical. (You need not rerun the entire calculation; because we always seed the random number generator with the same number, finding identical results through the end of the finite-difference test should be sufficient.)

2.2 H()

After the above modifications, the only remaining call of cI() on a wave-function object is in the application of the local potential in H(), in the form cIdag(Diagprod(gbl_Vdual+Vsc,cI(C))). Rewrite this contribution to H() using an explicit loop over the columns of C with a code fragment along the lines of

```
for col=1:size(C,2)
 ... cIdag((gbl_Vdual+Vsc).*cI(C(:,col))) ...
end
```

Note: You may have used a different internal variable name for Vsc, the sum of the nuclear, Hartree and exchange-correlation potentials.

Debugging: Rerun "Hatoms.m" to convince yourself that the output is unchanged.

3 Calculation of solid Ge

To prepare for your calculations of solid Ge, creating a new copy of all of your ".m" files in "∼/A3a" in a new directory, "∼/A3b" and begin working there.

3.1 "setup.m"

To prepare "setup.m" to run a calculation of an eight atom cell of Ge (valence charge Z=4), use the following code block to initialize S, R, X, and Z,

```
S=[48; 48; 48];

a=5.66/0.52917721; %# Lattice constant (converted from angstroms to bohrs)

R=a*diag(ones(3,1)); %# Cubic lattice
X=a*[0.00 0.00 0.00 %# diamond lattice in cubic cell
0.25 0.25 0.25
0.00 0.50 0.50
0.25 0.75 0.75
0.50 0.00 0.50
0.75 0.25 0.75
0.50 0.50 0.00
0.75 0.75 0.25];
Z=4; %# Valence charge
```

3.2 "Geatoms.m"

To calculate the electronic structure of the eight atom cell of germanium, make a copy of "Hatoms.m" called "Geatoms.m". The eight atom cell of germanium has 32 electrons distributed among 16 states; thus, modify "Geatoms.m" so that gbl_f=2 and Ns=16.

Next, we need a Fourier-transformed ionic potential for Vps. The Starkloff-Joannopoulos local pseudopotential for germanium is

$$V_{ps}(r) = -\frac{Z}{r}\frac{1-e^{-\lambda r}}{1+e^{-\lambda(r-r_c)}},$$

where $Z \equiv 4$, $r_c \equiv 1.052$ bohr, and $\lambda \equiv 18.5$ bohr^{-1}. Arias's (unfamous) analytic result for the Fourier transform of this potential is the rapidly convergent asymptotic series (four terms suffice to give 26 digits!),

$$\begin{aligned}\int e^{-i\vec{G}\cdot\vec{x}} V_{ps}(\vec{x}) \quad &= \quad -\frac{4\pi Z}{G^2} + \frac{4\pi Z(1+e^{-\lambda r_c})}{G^2}\left(-\frac{2\pi e^{-\pi G/\lambda}\cos(Gr_c)\frac{G}{\lambda}}{1-e^{-2\pi G/\lambda}}\right. \\ &\qquad \left. + \sum_{n=0}^{\infty}(-1)^n \frac{e^{-\lambda r_c n}}{1+\left(\frac{n\lambda}{G}\right)^2}\right)\end{aligned} \tag{1}$$

Along with this formula, we need the following limit to use for the case $\vec{G}=0$ (the first element in the FFT box),

$$\lim_{G\to 0}\left(\frac{4\pi Z}{G^2} + \int e^{-i\vec{G}\cdot\vec{x}} V_{ps}(\vec{x})\right) = 4\pi Z\left(1+e^{-\lambda r_c}\right)\left(\frac{r_c^2}{2} + \frac{1}{\lambda^2}\left(\frac{\pi^2}{6} + \sum_{n=1}^{\infty}(-1)^n\frac{e^{-\lambda r_c n}}{n^2}\right)\right) \tag{2}$$

To avoid needless heartache, feel free to replace the original computation of `Vps` for the hydrogen atom,

```
Vps=-4*pi*Z./G2; Vps(1)=0.;
```

by cutting and pasting the octave form for the resulting formula below directly into "Geatoms.m",

```
%# Ge pseudopotential
Z=4;
lambda=18.5;
rc=1.052;
Gm=sqrt(G2);
Vps=-2*pi*exp(-pi*Gm/lambda).*cos(Gm*rc).*(Gm/lambda)./(1-exp(-2*pi*Gm/lambda));
for n=0:4
  Vps=Vps+(-1)^n*exp(-lambda*rc*n)./(1+(n*lambda./Gm).^2);
end
Vps=Vps.*4*pi*Z./Gm.^2*(1+exp(-lambda*rc))-4*pi*Z./Gm.^2;

n=[1:4];
Vps(1)=4*pi*Z*(1+exp(-lambda*rc))*(rc^2/2+1/lambda^2* ...
                     (pi^2/6+sum((-1).^n.*exp(-lambda*rc*n)./n.^2)));
```

Finally, to allow for viewing of the total charge density, add the code block below to the very end of "Geatoms.m" (deleting any "pause" statements that may remain in your code or statements for viewing individual wave functions).

```
%# Compute density for final viewing
Y=W*inv(sqrtm(W'*O(W))); %# Orthonormal wave functions
n=getn(Y,gbl_f); %# Charge density

%# Basic slice from ''100'' edge of cell
sl=reshape(n(1:S(1)*S(2)),S(1),S(2));
%# Make and view image
ppm("100.ppm",sl*0.3,sl,sl); system("display 100.ppm &");

%# 110 slice cutting bonds (assumes cube)
sl=reshape(n(find(M(:,2)==M(:,3))),S(1),S(2));
%# Expand by 2, drop data to restore (approximate) aspect ratio
li=find(rem([1:size(sl,1)],3)!=0); sl=sl(li,:);
%# Make and view image
ppm("110.ppm",sl,sl,sl*0.3); system("display 110.ppm &");
```

3.2.1 Debugging

Run "Geatoms.m". (This will take about 1/2 hour!) The code should produce (and will try to display) two graphics, "100.ppm" and "110.ppm". If the `display` executable doesn't work on your computer, you can try to view the image file directly for your directory using some other application.

The graphics show a brightness proportional to the total charge density at each point in a plane cutting through the solid. The sky-blue graphic ("100.ppm") shows a slice across the top of the cubic cell. The atoms should appear as rings of round charge distributions with low density (dark regions) centered on the nuclei. The reason why there are no electrons on the nuclei is that we are using a pseudopotential and are not computing the core electrons. You should see one atom in the center of the 100 slice and four atoms cut off at the corners. This reflects the basic faced-centered cubic (fcc) structure of the germanium crystal. The 110 graphic is more exciting. This cut slices through the centers of neighboring atoms, showing significant buildup of charge right in between them. These are the *bonds* which hold the crystal together!!!

Run "ewald.m", and compute the total energy *per atom* of the crystal. (I find an Ewald energy of -31.9038 H and an electronic energy of 1.0561 H.)

Finally, compare this to the energy of the pseudoatom calculated at the same spatial resolution, -3.7301 H. (See Section 5 if you'd like to compute this for yourself.) Finally, using the conversion 1 hartree=27.21 eV compute your prediction for the cohesive energy of germanium. (The experimental value is 3.85 eV/atom!)

4 Variable fillings and verification of pseudopotential

Our final tasks are to compute the total energy of an isolated atom of germanium as described by our pseudopotential and to convince ourselves that the pseudopotential we are using reproduces the influence on the valence electrons of the nucleus and core electrons of the germanium atom. We shall do this by using our software to compute the electronic structure of an isolated atom of germanium.

For germanium, the calculation of an isolated atom requires variable fillings. The valence electrons of Ge have the configuration $4s^2 4p^2$. The $4p^2$ means that the three p states share two electrons equally. This means that the fillings vector $\vec{f}$ should be f=[2;2/3;2/3;2/3]. Our current software, however, *assumes* constant fillings, f=[2;2;2;...]. We must modify the current DFT software to include the appropriate general expressions.

To prepare for this create a final copy of all of your ".m" files in "∼/A4" in a new directory, "∼/A4" and begin working there.

4.1 getn()

```
function n=getn(psi,f)
```

Input:

- psi: sample values of N_s orthonormal wave functions
- f: $Ns \times 1$ column vector of fillings

Output:

- n: sample values of electron density, $n_i = \sum_k f_k \psi_{ik}^* \psi_{ik}$

Modify getn() to accept a vector $\vec{f}$ of fillings as above and to compute the appropriate sum to form the density.

4.2 getE()

```
function E=getE(W)
```

Input:

- W: Expansion coefficients for N_s unconstrained wave functions, stored as an $\prod S_k \times N_s$ matrix

Global variables:

- gbl_Vdual: Dual potential coefficients stored as a $S_k \times 1$ column vector.
- gbl_f: State fillings stored as an $N_s \times 1$ column vector.

Output:

- E: Energies summed over N_s states

Modify getE() to accept variable fillings as above. Note that, because getE() now simply passes gbl_f to getn(), there is nothing to change for the calculation of the electron density and associated terms. The calculation of the kinetic energy, however, must change. In our matrix language, the new expression is

$$T = -\frac{1}{2}\mathrm{Tr}\left((\mathbf{Diag}\,\vec{f}) \left(Y^\dagger L Y \right) \right),$$

where Y is the matrix of expansion coefficients for the *orthonormal* wave functions.
Hint: The extra parentheses in the above expression limit the size of and number computations required in evaluating intermediate expressions.

4.3 Q()

Copy the function in Appendix A into the file "Q.m". It evaluates an operator which is needed for the expression for the gradient in the general case of non-constant $\vec{f}$. Note that the subscript "U" to the operator "$Q_U(X)$" is sent as the second argument to the function. (Type "help Q" at the octave prompt after installing the file "Q.m" into your working directory.)

4.4 getgrad()

```
function grad=getgrad(W)
```

Input:

- W: Expansion coefficients for N_s unconstrained wave functions, stored as an $\prod S_k \times N_s$ matrix

Output:

- grad: $\prod S_k \times N_s$ matrix containing the derivatives $\partial E/\partial W^*_{i,n}$

Global variable:

- gbl_f: State fillings stored as an $N_s \times 1$ column vector.

Modify getgrad() as above to employ the appropriate formula for the case of variable fillings,

$$\nabla_{W^\dagger} E = \left(H(W) - O(W)U^{-1}\left(W^\dagger H(W)\right)\right)\left(U^{-\frac{1}{2}} F U^{-\frac{1}{2}}\right) + O(W)\left(U^{-\frac{1}{2}} Q_U(\tilde{H}F - F\tilde{H})\right),$$

where $U \equiv W^\dagger O(W)$, $\tilde{H} \equiv U^{-\frac{1}{2}}\left(W^\dagger H(W)\right)U^{-\frac{1}{2}}$, $F \equiv \mathbf{Diag}\,\vec{f}$, and $Q_U(\mathbf{A})$ is the operator which the function in Appendix A computes.[1] Note that the parentheses have been arranged here again to minimize the size and cost of the calculation of intermediate expressions and that $U^{-\frac{1}{2}}$ may be computed in octave as `sqrtm(inv(U))`. (Don't forget the "m" in "sqrtm"! It specifies the matrix-square root as opposed to simply taking the square root of each element of the matrix.)

4.4.1 Debugging

For basic debugging, delete the statement "gbl_f=2;" from "Geatoms.m" and add the statement

```
gbl_f=2*ones(Ns,1); %# Constant fillings as a vector
```

just after the definition of Ns in "Geatoms.m". Rerun "Geatoms.m" and verify that your output (with perhaps changes in the fourteenth or fifteenth digits) is identical to before, at least through the finite-difference test.

5 Isolated Ge atom

As a non-trivial test of your variable fillings code, and to compute the energy (given in Section 3.2.1) of an isolated Ge atom at the same resolution as your solid state calculation, set the parameter `X=a*[0.00 0.00 0.00]` in "setup.m" and the parameters `Ns=4` and `gbl_f=[2;2/3;2/3;2/3]` in "Geatoms.m" and rerun "setup.m" and "ewald.m" and "Geatoms.m". Passing the finite-difference test should be sufficient to verify your new code, and summing the resulting Ewald and electronic energies should give the value of an isolated Ge atom quoted in Section 3.2.1).

[1] We will derive this expression on the last day of class, if there is time. You may verify for yourself that if $\vec{f}$ is constant, this reduces to your current expression.

Finally, as (partial) verification that the pseudopotential indeed gives states which correspond to the valence states of the germanium atom, compare the s-p energy difference (the energy difference between the s and p states of the atom) with those from your atomic code, or equivalently, from the table at NIST [2] To do this, put the statements

```
[Psi, epsilon]=getPsi(W);
epsilon
```

at the end of "Geatoms.m" (and/or run them at the command prompt immediately following your run of "Geatoms.m"). (Note that only the 4s-4p energy difference is physical because of the arbitrary choice in zero in the definition of potential energy.) You should find agreement to within about 3 mH!

[2]http://physics.nist.gov/PhysRefData/DFTdata/Tables/ptable.html .

A “Q.m”

```
function out=Q(in,U)
  [V,mu]=eig(U); mu=diag(mu);

  denom=sqrt(mu)*ones(1,length(mu)); denom=denom+denom';
  out=V*( (V'*in*V)./denom )*V';
end
```