Cornell University

Department of Physics

Practical DFT February 22, 2014

Phys 7654, Module #2, Spring 2014

Practical Density-Functional Theory

Homework Assignment # 3

(Due Tuesday, March 7 at 11:59pm)

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 $\hat{\vec{n}}$.
  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 Advanced techniques for numerical minimization

Begin by copying all of your ".m" files from "∼/A2a" into a new directory "∼/A3" and proceed to work there.

In this problem, we implement line minimization, preconditioning, and conjugate gradients. To verify the convergence properties of the various techniques, each function will return, as an optional second argument, the list of energies at each stage of the minimization. We begin by adding this capability to sd().

1.1 sd()

```
function [out, Elist]=sd(W,Nit)
```

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions
- Nit: Number of iterations desired

Output:

- out: Result of Nit iterations of steepest descents
- Elist: Nit×1 column vector containing result of getE() after each iteration
- DISPLAY: result of each getE() also displayed

Generalize your function in "sd.m" to provide the above functionality. It practice, you can accomplish this by including "Elist" in the function prototype at the top of the file and adding the two indicated statements to your loop as in the example below.

```
% Minimization with steepest descent algorithm using getE() and getgrad()
%
% Usage: [out, Elist]=sd(W,Nit)

function [out, Elist]=sd(W,Nit)
          .
          .
          .
  for it=1:Nit
    out=out-alpha*getgrad(out);
    Elist(it)=getE(out); %# <= New statements
    Elist(it)
  end
          .
          .
          .
```

1.2 lm()

```
function [out, Elist]=lm(W,Nit)
```

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions
- Nit: Number of iterations desired

Output:

- out: Result of Nit iterations of line minimization
- Elist: Nit$\times$1 column vector containing result of getE() after each iteration
- DISPLAY: result of each getE(), "linmin test" (angle cosine between previous search direction and current gradient)

Implement the line minimization algorithm discussed in lecture,

1. Initialize W_0
2. Evaluate gradient: $g = \nabla_W E(W_n)$
3. If not first iteration, do "linmin test": check the angle cosine $(g \cdot d)/\sqrt{(g \cdot g)(d \cdot d)}$, where d is the previous search direction
4. Set search direction: $d = -g$
5. Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d)$
6. Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - gt] \cdot d)$
7. $W_{n+1} = W_n + \alpha d$
8. Repeat (2-7)

with a function of the above prototype in the file "lm.m". You may wish to experiment with the trial step size. You will find that the algorithm is not very sensitive to this parameter. A trial step size of $\alpha_t = 3 \times 10^{-5}$ works well.
Hint: Because of our trick of coding gradients as complex matrices, we must evaluate the dot products according to the formula

$$a \cdot b = \mathrm{Re}\,\mathrm{Tr}\,\left(a^\dagger b\right).$$

1.2.1 Debugging

Replace the call to sd() in your "dft.m" script file with lm(), and run. If you have implemented steepest descents correctly, you should find that the energy always decreases and that the angle-cosine test gives nearly zero, indicating that the new gradient is orthogonal to the previous search direction. Note that at the start, when you are far from the minimum and the function is not very quadratic, the approximate minimization is not perfect and you may see angle cosines near 0.1. However, after about 20 iterations, the algorithm begins to approach the minimum, and the angle cosines will become much smaller ($\sim 10^{-4}$) and continue to improve as the minimization proceeds. Feel free to interrupt the program before all 400 iterations are complete.

1.3 K()

```
function out=K(W)
```

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions

Global variables:

- gbl_G2: Square magnitude of G vectors

Output:

- out: Preconditioner 1/(1+G2) applied to each column of W

Implement in the file "K.m" the preconditioner function of the above prototype.
Hint: Your code should closely resemble that for Linv().

1.4 pclm()

```
function [out, Elist]=pclm(W,Nit)
```

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions
- Nit: Number of iterations desired

Output:

- out: Result of Nit iterations of preconditioned line minimization
- Elist: Nit$\times$1 column vector containing result of getE() after each iteration
- DISPLAY: result of each getE(), "linmin test" (angle cosine between previous search direction and current gradient)

Implement the preconditioned line minimization algorithm discussed in lecture,

1. Initialize W_0
2. Evaluate gradient: $g = \nabla_W E(W_n)$
3. If not first iteration, do "linmin test": check $(g \cdot d)/\sqrt{(g \cdot g)(d \cdot d)}$ for d being the previous search direction
4. Set search direction: $d = -K(g)$
5. Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d)$
6. Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - gt] \cdot d)$
7. $W_{n+1} = W_n + \alpha d$
8. Repeat (2-7)

with a function of the above prototype in the file "pclm.m".
Hint: You should have to change only a single line of "lm.m" to generate "pclm.m"

1.4.1 Debugging

The ultimate test of a good preconditioner is whether it actually improves the rate of convergence. You may note, however, that if you replace your current call to lm() in "dft.m" with pclm(), the energy may begin to increase momentarily. This is not uncommon when starting a minimization far from the minimum. To make a more appropriate comparison replace your current call to lm() with

```
%# Converge
W=sd(W,20); %# 20 iterations of simple sd() to get nearer to the minimum
W=W*inv(sqrtm(W'*O(W))); %# Restart as orthonormal functions

[Wlm, Elm]=lm(W,50); %# 50 iterations of lm() from W, while recording results
[Wpclm, Epclm]=pclm(W,50); %# 50 iterations of pclm from same W

%# Plot results on log scale with nice labels
grid on;
xlabel("Iteration (#) ->"); ylabel("Error (H)");
semilogy([1:length(Elm)],Elm-43.3371147782040,'r-;lm;', ...
         [1:length(Epclm)],Epclm-43.3371147782040,'b-;pclm;');
pause; %# Wait for keyboard input before continuing
```

This code block first "relaxes" W with 20 iterations and orthonormalizes the result to give a good starting point relatively near the minimum. The code block then runs both lm() and pclm() from that this starting point, each for 50 iterations, and plots on a log scale the difference of each algorithm from the fully relaxed result, with line minimization in red, and preconditioned line minimization in blue. If the preconditioner is working, you should notice *both* some initial difference coming from the first iteration where the preconditioner eliminates nearly all of the high frequency errors *and* a difference in slope (exponential convergence rate) at the higher iteration numbers.
Note: After the plot, the program will halt at the "pause;" statement, giving the option of exiting by hitting Ctrl-C before generating the plots.

1.5 pccg()

```
function [out, Elist]=pccg(W,Nit,cgform)
```

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions
- Nit: Number of iterations desired
- cgform: Version of cg: 1=Fletcher-Reeves, 2=Polak-Ribiere, 3=Hestenes-Stiefel

Output:

- out: Result of Nit iterations of preconditioned conjugate gradients
- Elist: Nit$\times$1 column vector containing result of getE() after each iteration
- DISPLAY: result of each getE(), "linmin test" (angle cosine between previous search direction and current gradient), "conjugate gradient test" (angle cosine, measured through the preconditioner, of the current gradient and the previous gradient)

Implement the preconditioned conjugate-gradients algorithm discussed in lecture,

1. Initialize W_0
2. Evaluate gradient: $g_n = \nabla_W E(W_n)$
3. If not first iteration, do "linmin test": check $(g_n \cdot d_{n-1})/\sqrt{(g_n \cdot g_n)(d_{n-1} \cdot d_{n-1})}$ for d being the previous search direction
4. If not first iteration, do "cg test": check $(g_n \cdot Kg_{n-1})/\sqrt{(g_n \cdot Kg_n)(g_{n-1} \cdot Kg_{n-1})}$ for d being the previous search direction
5. Set search direction: $d_n = -Kg_n + \beta d_{n-1}$, where
   - Fletcher-Reeves: $\beta = (g_n \cdot Kg_n)/(g_{n-1} \cdot Kg_{n-1})$
   - Polak-Ribiere: $\beta = ([g_n - g_{n-1}] \cdot Kg_n)/(g_{n-1} \cdot Kg_{n-1})$
   - Hestenes-Stiefel: $\beta = ([g_n - g_{n-1}] \cdot Kg_n)/([g_n - g_{n-1}] \cdot d_{n-1})$
6. Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d_n)$
7. Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - g_t t] \cdot d_n)$
8. $W_{n+1} = W_n + \alpha d_n$
9. Repeat (2-8)

with a function of the above prototype in the file "pccg.m".
Hint: You may wish to use the "`if ... elseif ... elseif ... end`" octave construct.

1.5.1 Debugging

Analogously to debugging for line minimization, the best test of your implementation is the gradient-orthogonality theorem, the "cg test". Depending upon the form used for β, this test will not function well until quite close to the minimum. Thus, begin your testing with the least sensitive version (Hestenes-Stiefel), running it from the wave functions determined by preconditioned line minimization by entering "`Wout=pccg(Wpclm,50,3);`" at the command prompt after having just run your "dft.m". You should observe that "cg test" gives results which track but are somewhat larger than those of the "linmin test". Also, after a number of iterations, the "cg test" should become quite good.

Once this test passes, Wout will contain wave functions sufficiently close to the minimum that the other forms for β will work well when properly implemented. You may then verify those versions using "`pccg(Wout,10,1);`" and "`pccg(Wout,10,2)`;".

1.6 Final comparison of techniques

To complete the comparison of the techniques, add the lines

```
[Wcg1, Ecg1]=pccg(W,50,1); %# 50 iters from W of cg versions 1, 2, 3 ...
[Wcg2, Ecg2]=pccg(W,50,2);
[Wcg3, Ecg3]=pccg(W,50,3);
```

immediately after the call to pclm(W,50) in your current version of "sch.m" and replace the current plotting call to semilogy(...) with

```
semilogy([1:length(Elm)],Elm-43.3371147782040,'-;lm;', ...
         [1:length(Epclm)],Epclm-43.3371147782040,'-;pclm;', ...
         [1:length(Ecg1)],Ecg1-43.3371147782040,'-;pccg-FR;', ...
         [1:length(Ecg2)],Ecg2-43.3371147782040,'-;pccg-PR;', ...
         [1:length(Ecg3)],Ecg3-43.3371147782040,'-;pccg-HS;');
%# Set W to best result for plotting
W=Wcg3;
```

2 Density functional calculation of molecules

In this problem, you will use your software to evaluate the bond-length of the H_2 molecule directly from first principles using your own software.

2.1 Ionic potential

2.1.1 Background

Your program "dft.m" now solves the Kohn-Sham equations in the potential coming from a simple harmonic oscillator. For most electronic structure calculations, however, the potential comes from the interaction between electrons and ions with a form like

$$V(\vec{r}) = -\sum_I \frac{Z_I}{|\vec{r} - \vec{X}_I|}, \tag{1}$$

where Z_I and $\vec{X}_I$ are the charge and location of each nucleus. We now proceed to determine the value Vdual which corresponds to this potential, including all relevant signs and normalizations.

Our approach to finding the potential Vdual from the nuclei for our periodic calculations is based on our standard approach to solving Poisson's equation,

$$\vec{\phi}_{nuc} = \mathbf{I}L^{-1}\left(-4\pi\mathbf{OJ}\vec{\rho}_{nuc}\right)$$

and the definition of Vdual from the previous exercise on the Harmonic oscillator,

$$\tilde{V} = \mathbf{J}^\dagger \mathbf{O} \mathbf{J} \left(-\vec{\phi}_{nuc} \right),$$

where the extra '-' in front of ϕ_{nuc} corresponds to the minus sign in Eq. (1). Combining the above two results gives

$$\tilde{V} = \mathbf{J}^\dagger \mathbf{O} \mathbf{L}^{-1} (4\pi \mathbf{O} \mathbf{J} \vec{\rho}_{nuc}) \tag{2}$$

where we have used the fact that $\mathbf{J}(\mathbf{I}(\vec{x})) = \vec{x}$.

Now, the expansion coefficients $[\mathbf{J}\vec{\rho}_{nuc})]_{\vec{G}} \equiv \hat{\rho}_{\vec{G}}$ are normalized so that

$$\rho(\vec{r}) \equiv \sum_{\vec{G}} \hat{\rho}_{\vec{G}} e^{i\vec{G}\cdot\vec{r}}.$$

Therefore,

$$\begin{aligned} \int_{\text{cell}} e^{-i\vec{G}\cdot\vec{r}} \rho(\vec{r})\, dV &= \int_{\text{cell}} e^{-i\vec{G}\cdot\vec{r}} \sum_{\vec{G}'} \hat{\rho}_{\vec{G}'} e^{i\vec{G}'\cdot\vec{r}}\, dV \\ &= \det(\mathbf{R})\, \hat{\rho}_{\vec{G}}, \end{aligned}$$

where $\det(\mathbf{R})$ is the volume of the cell and the integral extends over that volume. This means

$$\hat{\rho}_{\vec{G}} = \frac{1}{\det(\mathbf{R})} \int_{\text{cell}} e^{-i\vec{G}\cdot\vec{r}} \rho(\vec{r})\, dV.$$

Finally, we have one copy of each nucleus in the cell, so that (over the range of the integral) $\rho(\vec{r}) = \sum_I Z\delta^{(3)}\left(\vec{r} - \vec{X}_I\right)$, where $\delta^{(3)}(\vec{x})$ is the Dirac-delta function in three-dimensions, and Z is the charge of each nucleus. Thus,

$$\hat{\rho}_{\vec{G}} = \frac{Z}{\det(\mathbf{R})} \sum_I e^{-i\vec{G}\cdot\vec{X}_I} = \frac{Z}{\det(\mathbf{R})} S_f(\vec{G}),$$

where $S_f(\vec{G})$ is the same structure factor

$$S_f(\vec{G}) \equiv \sum_I e^{-i\vec{G}\cdot\vec{X}_I}, \tag{3}$$

from the previous problem on computing Ewald energies. Inserting this into Eq. (2) and using the normalizations associated with the various operators gives

$$\tilde{V} = \mathbf{J}^\dagger \hat{v}, \tag{4}$$

where

$$\hat{v}_{\vec{G}} = \left(\frac{-4\pi Z}{G^2} \right) S_f(\vec{G}). \tag{5}$$

Here, the term $-4\pi Z/G^2$ is just the Fourier-space version of the ionic potential $-Z/|\vec{r}|$, the structure factor $S_f(\vec{G})$ replicates each of the ions in the appropriate locations, and $\mathbf{J}$ Fourier transforms back to real space with the appropriate normalization.

2.1.2 Implementation

First, make a copy of "∼/A3/dft.m" called "∼/A3/Hatoms.m". Then, remove from "Hatoms.m" the code block which displays the wave functions, and replace the code block that relaxes the wave functions with

```
format long

%# Converge
W=sd(W,20); %# 20 iterations of simple sd() to get nearer to the minimum
W=W*inv(sqrtm(W'*O(W))); %# Restart as orthonormal functions
[W, Elist]=pccg(W,50,1); %# 50 iterations of pclm from same W
```

At this point, you may wish to rerun "setup.m" and "Hatoms.m" to verify that the code functions properly as before.

Next, delete the computation of V and Vdual in "Hatoms.m", and replace it with the following. First, compute the ionic potential in Fourier space as

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

Note that this code is careful to set the infinite $\vec{G} = 0$ component of the ionic potential to zero, a term which exactly balances with the $\vec{G} = 0$ components of the Ewald and Hartree energies. Also, we are calling this Fourier-transformed potential from a single ion "Vps" because it will become the Fourier transform of something called a "pseudopotential" when we do solid germanium. After computing Vps, next evaluate $\tilde{V}$ according to Eqs. (4-5) as

```
Vdual=cJ(Vps.*Sf);
```

Finally, be sure to store the result in gbl_Vdual for communication to the various functions.

2.1.3 Debugging

As a test, perform a three-dimensional calculation of a hydrogen atom as follows.

1. Using S=[64;64;64], R=diag([16; 16; 16])), X=[0 0 0], Z=1 in "setup.m" and sigma1=0.25 in "ewald.m", compute the Ewald energy of a single proton in a cell of size (16 bohr)^3. You should find a value for the Ewald energy of -0.0885440851224253 H.

2. Then, run "Hatoms.m" for a single state with one electron (gbl_f=1, Ns=1) using the same parameters in "setup.m" as you used for the ewald calculation. You should find a value (to within about 0.001 H due to possibly different levels of convergence between you and me) for the electronic energy of -0.356725655980680 H.

When your code functions properly (and you add the Ewald energy), you should find a total energy within 1 millihartree of the "official" LDA total energy published on the NIST web site[1] for hydrogen, -0.445671 H.

2.2 Hydrogen molecule

Finally, modifying X in "setup.m" to [0 0 0; x 0 0] for x=0.50, 1.00, 1.25, 1.50, 1.75, 2.00, 4.00, and 6.00 bohr, and running "setup.m", "ewald.m" and "Hatoms.m" (be sure to use gbl_f=2!) each time, compute the total energy of the H_2 molecule at the above separations.

Debugging: At a separation of 1.5 bohr (X=[0 0 0; 1.5 0 0] in "setup.m" and gbl_f=2 in "setup.m"), you should find a total energy of -1.136 hartree, corresponding to an Ewald energy of 0.3135897639081198 H and an electronic energy of -1.4495422745929 H. (Again your electronic energy may vary slightly due to different levels of convergence.)

From this data, you should be able to extract the LDA predictions for the bond length, energy and effective spring constant for H_2. See the file "H2 Bond" under the "Computer Files" folder for the course for an analysis of the results with a comparison to data from the CRC handbook!

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