Cornell University

Department of Physics



# Phys 7654, Module #2, Spring 2014

# Practical Density-Functional Theory

## Homework Assignment # 2

(Due Tuesday, Februrary 28 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 Background for Schrödinger's equation

Taking all of the occupancies to be $f = 1$, lecture defined the unconstrained objective function for finding multiple solutions to Schrödinger's equation as

$$E = -\frac{1}{2}\mathrm{Tr}\left(W^\dagger L W U^{-1}\right) + \vec{V}^\dagger \mathbf{J}^\dagger \mathbf{O}\mathbf{J}\vec{n},$$

where

$$U \equiv W^\dagger \mathbf{O} W,$$
$$\vec{n} \equiv \mathrm{diag}\left(\mathbf{I} W U^{-1} W^\dagger \mathbf{I}^\dagger\right),$$

and $\vec{V}$ is the vector of sample values of the potential on the real space grid.

A minor rearrangement of these expressions, more convenient for our purposes, is

$$\begin{aligned} E &= -\frac{1}{2}\mathrm{Tr}\left(W^\dagger L W U^{-1}\right) + \tilde{V}^\dagger \vec{n} \\ U &\equiv W^\dagger \mathbf{O} W \\ \vec{n} &\equiv \mathrm{diag}\left(\left(\mathbf{I} W U^{-1}\right)(\mathbf{I}W)^\dagger\right) \\ \tilde{V} &\equiv \mathbf{J}^\dagger \mathbf{O}\mathbf{J}\vec{V}, \end{aligned} \tag{1}$$

where we have regrouped quantities under †'s, used the fact that $\mathbf{O}$ is Hermitian ($\mathbf{O}^\dagger = \mathbf{O}$), and defined a "dual" set of potential coefficients $\tilde{V}$ which can be directly combined with $\vec{n}$ to form the potential energy.

Because $W$ is a complex matrix rather than a real vector, the most convenient form for expressing the gradient of $E$ with respect to $W$ is to form a matrix of the same dimensions as $W$, with each element set equal to the partial derivative of $E$ with respect to the complex conjugate of the corresponding element of $W$,

$$\left[\nabla_W E\right]_{\alpha,n} \equiv \frac{\partial E}{\partial W^*_{\alpha,n}}. \tag{2}$$

As derived in lecture, the gradient of $E$ then becomes

$$\nabla_W E = \left(HW - \mathbf{O}WU^{-1}W^\dagger HW\right)U^{-1}, \tag{3}$$

where $U$ is defined as above and

$$H \equiv -\frac{1}{2}L + \mathbf{I}^\dagger\left(\mathrm{Diag}\,\tilde{V}\right)\mathbf{I}. \tag{4}$$

Note that with gradients expressed as matrices as in Eq. (3), the following (ultimately!) simple expression may be used to compute the directional derivative of the energy $E$ along the direction $dW$,

$$\begin{aligned} dE &\equiv \sum_{\alpha,n}\left[\mathrm{Re}\left(dW_{\alpha,n}\right)\frac{\partial E}{\partial \mathrm{Re}\left(dW_{\alpha,n}\right)} + \mathrm{Im}\left(dW_{\alpha,n}\right)\frac{\partial E}{\partial \mathrm{Im}\left(dW_{\alpha,n}\right)}\right] \\ &= \sum_{\alpha,n}\mathrm{Re}\left[\left(\mathrm{Re}\left(dW_{\alpha,n}\right) - i\mathrm{Im}\left(dW_{\alpha,n}\right)\right)\left(\frac{\partial E}{\partial \mathrm{Re}\left(W_{\alpha,n}\right)} + i\frac{\partial E}{\partial \mathrm{Im}\left(W_{\alpha,n}\right)}\right)\right] \\ &= \mathrm{Re}\sum_{\alpha,n} dW^*_{\alpha,n} 2\left(\frac{\partial E}{\partial W^*_{\alpha,n}}\right) \\ &= 2\,\mathrm{Re}\,\mathrm{Tr}\,dW^\dagger \nabla_W E. \end{aligned} \tag{5}$$

In the last week of class (time permitting), we will explain the following final note. This isn't needed for computing energies but only if you are interested in the actual eigenstates. In the above, H is a matrix representation of the "Hamiltonian," so that the product $HW$ corresponds to the action of the left-hand

side of the eigenvalue equation, "LHS(W)". The transformation which turns the unnormalized $W$ into the normalized $Y$ and then the final eigensolutions $\Psi$ is

$$Y \equiv WU^{-1/2} \tag{6}$$

$$\Psi \equiv YD, \tag{7}$$

where $D$ diagonalizes the matrix

$$\mu \equiv Y^\dagger HY, \tag{8}$$

according to

$$D^\dagger \mu D = \operatorname{diag} \vec{\epsilon}, \tag{9}$$

where $\vec{\epsilon}$ are the final eigenvalues.

# 2 Updated Operators

Begin this assignment (Assignment #2) by making a directory "∼/A2", changing into that directory and copying everything from the previous assignment into your new directory with "`cd ∼; mkdir ∼/A2; cd ∼/A2; cp ../A1/*.m .`". Be sure that the parameters are set in "setup.m" to S=[20; 25; 30] and R=diag([6 6 6]) and in "ewald.m" to sigma1=0.25.

Two new operators appear in the expressions, $\mathbf{I}^\dagger$ and $\mathbf{J}^\dagger$. Also, operators now sometimes act on matrices, which we view as collections of column vectors, rather than on single column vectors. We must therefore provide some new operators and generalize the ones we already have.

## 2.1 cI()

`function out=cI(in)`

Input:

- in: $N_s$ sets of $d = 3$ dimensional data, stored as an $\prod S_k \times N_s$ matrix

Global variables:

- gbl_S: dimensions of $d = 3$ dimensional data sets

Output:

- out: cI operator applied to in

Generalize the software in your file "cI.m" to have the above capability. To do this, note that by block matrix multiplication,

$$\mathbf{I}\,\mathrm{in} = \mathbf{I}\,[\mathrm{in}(:,1), \mathrm{in}(:,2), \ldots, \mathrm{in}(:,\mathrm{Ns})] = [\mathbf{I}\,\mathrm{in}(:,1), \mathbf{I}\,\mathrm{in}(:,2), \ldots, \mathbf{I}\,\mathrm{in}(:,\mathrm{Ns})]\,,$$

where in(:,k) is octave notation for the k-th column of the matrix in. The above result simply states that you need only apply the action of $\mathbf{I}$ independently to each column of in and to store the result in out.

In octave this type of operation is most efficient when you first form the output matrix with data of the appropriate size with a statement like

```
out=zeros(size(in));
```

Doing this improves efficiency by avoiding extra system calls to malloc() to allocate memory as the data for out is actually computed. Then, you should loop over the columns of in with a code fragment of the form

```
for col=1:size(in,2) %# size(in,2) gives 2nd dimension (# of columns) of in
  out(:,col)=fft3(in(:,col), ... ); %# <= Same operation you had before
end
```

where you compute out(:,col) from in(:,col) in the same was as you previously computed out from in.

## 2.2 cJ()

`function out=cJ(in)`

Input:

- in: $N_s$ sets of $d = 3$ dimensional data, stored as an $\prod S_k \times N_s$ matrix

Global variables:

- gbl_S: dimensions of $d = 3$ dimensional data sets

Output:

- out: cJ operator applied to in

Generalize the software in your file "cJ.m" according to the above prototype by following the same procedure you used to generalize "cI.m".

## 2.3 O()

`function out=O(in)`

Input:

- in: $N_s$ sets of $d = 3$ dimensional data, stored as an $\prod S_k \times N_s$ matrix

Global variables:

- gbl_R: lattice-vector matrix

Output:

- out: O operator applied to in

Actually, because the action of O() is simply multiplication by a constant, an operation already defined properly in octave for matrices, your software for O() should *probably* function fine "as is".

## 2.4 L()

`function out=L(in)`

Input:

- in: $N_s$ sets of $d = 3$ dimensional data, stored as an $\prod S_k \times N_s$ matrix

Global variables:

- gbl_R: lattice-vector matrix
- gbl_G2: square-magnitudes of G vectors

Output:

- out: L operator applied to in

Generalize the software in your file "L.m" according to the above prototype. One option would be to follow the same procedure you used to generalize "cI.m". However, a computationally quicker option (but one which uses more memory) is to use BLAS operations to expand G2 into a matrix each of whose columns contains a copy of G2, as may be accomplished with "`gbl_G2*ones(1,size(in,2))`". Then you may use the "`.*`" operator to compute the output using a statement containing the fragment "`gbl_G2*ones(1,size(in,2)).*in`". **Hint:** Don't forget all of the other important factors!

## 2.5 cIdag()

`function out=cIdag(in)`

Input:

- in: $N_s$ sets of $d = 3$ dimensional data, stored as an $\prod S_k \times N_s$ matrix

Global variables:

- gbl_S: dimensions of $d = 3$ dimensional data sets

Output:

- out: Hermitian conjugate of cI operator applied to in

Produce a function of the above prototype in the file "cIdag.m".
**Hint:** Because the discrete Fourier transform kernel, $\exp 2\pi i \left(n_1 m_1/S_1 + n_2 m_2/S_2 + n_3 m_3/S_3\right)$, is symmetric in $n$ and $m$, the only difference between your codes for cI and cIdag should be the sign of i in the call to fft3.

## 2.6 cJdag()

`function out=cJdag(in)`

Input:

- in: $N_s$ sets of $d = 3$ dimensional data, stored as an $\prod S_k \times N_s$ matrix

Global variables:

- gbl_S: dimensions of $d = 3$ dimensional data sets

Output:

- out: Hermitian conjugate of cJ operator applied to in

Produce a function of the above prototype in the file "cJdag.m".
**Hint:** Again, the only difference between your codes for cJ and cJdag should be the sign of i in the call to fft3.

## 2.7 Debugging

### 2.7.1 Single column cases

To debug the codes which you generalized for cI(), cJ(), L(), and O(), simply rerun your Poisson solver (don't forget to run "setup.m" first!) and verify that you have the same results. This verifies that these operators function properly in the case of a single column.

To verify the new operators cIdag() and cJdag(), you should check the identities which actually define the mathematic meaning of the Hermitian conjugate of an operator,

$$\begin{aligned} \left(a^\dagger \mathbf{I} b\right)^* &= b^\dagger \mathbf{I}^\dagger a \\ \left(a^\dagger \mathbf{J} b\right)^* &= b^\dagger \mathbf{J}^\dagger a, \end{aligned}$$

for all vectors $a$ and $b$. Cut and paste the fragment below into the octave prompt to check the above identities:

```
a=randn(prod(S),1)+i*randn(prod(S),1); %# Single columns of random complex data
b=randn(prod(S),1)+i*randn(prod(S),1);

conj(a'*cI(b))
b'*cIdag(a)

conj(a'*cJ(b))
b'*cJdag(a)
```

#### 2.7.2 Multiple column cases

Now that each operator is verified for single column vectors, all that remains is to verify proper action on multiple columns of input. To check cI()'s action on multiple columns, perform the following test at the octave prompt:

```
in=randn(prod(S),3)+i*randn(prod(S),3); %# Form random input with 3 columns
out1=cI(in); %# Output of new operator
out2=[cI(in(:,1)), cI(in(:,2)), cI(in(:,3))]; %# Output using debugged case
max(abs(out2-out1)) %# Check maximum value for discrepancy in each column
```

Finally, you should repeat the test, substituting each of the remaining operators for cI in the code fragment above.

# 3 Energy calculation: "sch.m" and "getE.m"

## 3.1 Setup of the potential

We will continue to use the same "setup.m" to initialize the basic variables needed for the spectral method. As this part of the setup is fully general, no changes need be made to "setup.m". For this problem we will use the same parameters as for the Poisson solution, S=[20; 25; 30]; R=diag([6 6 6]). Before proceeding, double check that you have the same values for these parameters set at the top of "setup.m".

Next, create a new file "sch.m" where we will place our solution to Schrödinger's equation. Begin by setting the variable V to the sample values of a simple harmonic oscillator potential of frequency $\omega = 2$,

$$V(\vec{r}) = \frac{1}{2}\omega^2 |\vec{dr}|^2 = 2|\vec{dr}|^2,$$

where $\vec{dr}$ is the distance to the center of the cell.

To verify your result, cut and paste the file "viewmid.m" from Appendix A (which is simply the visualization code block from poisson.m encapsulated as a function), run your "sch.m" and then enter the command "viewmid(V,S);". The function viewmid(V,S) will draw mesh plots of your potential as viewed in planes slicing through the center of the cell, which in this case should yield parabolic surfaces with minima in the center of each plane and maxima of approximately 35.

**Hint:** You may borrow your computation of dr from your "poisson.m".

Because it is always the combination $\tilde{V}$ from Eq. (1) which appears in expressions, it is most convenient to compute this "dual" representation once and then export it as a global variable. Include a statement at the top of "sch.m" declaring gbl_Vdual as a global variable, and set its value by including the statement

```
gbl_Vdual=cJdag(O(cJ(V)));
```

immediately after your computation of V.

## 3.2 diagouter()

`function out=diagouter(A,B)`

Input:

- A,B: $n \times m$ matrices

Output:

- out: diag $\left(AB^\dagger\right)$

The expression for the density in Eq. (1) is in the form of taking as a column vector the diagonal elements of the "outer product" of two matrices,

$$\vec{c} = \text{diag}\left(AB^\dagger\right). \tag{10}$$

Because $A$ and $B$ are of dimension $\prod S_k \times N_s$, it is extremely wasteful (of time and memory) to form directly the matrix $\prod S_k \times \prod S_k$ matrix $AB^\dagger$ only to then take its diagonal elements. In this case, it is better to provide our own function, diagouter(), to perform the using other BLAS operations. In terms of components Eq. (10) is

$$c_i = \sum_n A_{i,n} B^*_{i,n}.$$

Thus, we can perform this operation by first forming the matrix elements $C_{i,n} \equiv A_{i,n} B^*_{i,n}$ using octave's ".*" and "conj()" operators, and then summing along the rows (second index). In octave notation, this becomes simply

```
c=sum(A.*conj(B),2);
```

Use this approach to produce a function of the above prototype in the file "diagouter.m".

### 3.2.1 Debugging

Verify your diagouter() on a small test case as follows

```
A=randn(10,3)+i*randn(10,3); %# Form random A and B matrices
B=randn(10,3)+i*randn(10,3);
diag(A*B') %# Direct calculation from definition and octave operators
diagouter(A,B) %# Your routine
```

Recall that "'" in octave represents complex-conjugate transpose. Also, diag() takes the diagonal elements of a matrix just as in the notation from class.

**Note:** *Do not* try this with your actual full-sized data sets – you will likely crash octave!

## 3.3 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.

Output:

- E: Energies summed over $N_s$ states

Using the expressions in Eq. (1) and your diagouter() operator above, produce a function of the above prototype in the file "getE.m".

As a quick test, although you will generally find machine-precision sized imaginary parts due to rounding, your output should always be real. You may verify this behavior with

```
setup; sch; %# Make sure your global variables are all set
W=randn(prod(S),4)+i*randn(prod(S),4); %# Put 4 random wavefunctions in W
getE(W)
```

*After* verifying that your output is indeed real to machine precision for a few different random input $W$'s, it is best to modify your code to take the real part of E using real() before returning, so as to avoid dealing with complex numbers in inappropriate places later.

# 4 Gradient calculation

## 4.1 Diagprod()

`function out=Diagprod(a,B)`

Input:

- a: $n \times 1$ column vector
- B: $n \times m$ matrix

Output:

- out: $(\mathrm{Diag}\,\vec{a})\,B$

The expression for the gradient in Eq. (3) ultimately involves products of the form

$$C = (\mathrm{Diag}\,\vec{a})\,B. \tag{11}$$

Here, Diag takes a vector of length $\prod S_k$ and forms a very large, diagonal $\prod S_k \times \prod S_k$ matrix, which then multiples the matrix $B$. Again, this direct evaluation of the expression is extremely wasteful of both time and space, and so we shall provide an function which performs the operation in terms of a more efficient selection of BLAS routines.

In terms of components, Eq. (11) becomes

$$C_{i,n} = \sum_j a_j \delta_{j,i} B_{i,n} = a_i B_{i,n}.$$

Thus, each column of $C$ may be computed independently as the "`.*`" product of $\vec{a}$ with the corresponding column of $B$, a series of BLAS1 operations. Alternately, by using a little more memory, the same operation may be carried out with BLAS2 operations (which in this case run about 2 times faster) by first forming a matrix of the same size as $B$ with copies of $\vec{a}$ in each column and then taking the "`.*`" product, `c=(a*ones(1,size(B,2))).*B`. Using this strategy provide a function of the above prototype in the file "Diagprod.m".

### 4.1.1 Debugging

Again, verify your Diagprod() on small test cases as below. (Don't try the full sized case!)

```
a=randn(10,1); %# Random column vector
B=randn(10,3); %# Random matrix
diag(a)*B %# Direct calculation from definition and octave operators
Diagprod(a,B) %# Your routine
```

## 4.2 H()

`function out=H(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.

Output:

- out: $HW$

Using the expression in Eq. (4) and your Diagprod() operator above, produce a function of the above prototype in the file 'H.m"

### 4.2.1 Debugging

As a quick test, the operator H() should be Hermitian. This means that $(\vec{a}^\dagger H\vec{b})^* = \vec{b}^\dagger H\vec{a}$ for any vectors $\vec{a}$ and $\vec{b}$. You may check this with

```
a=randn(prod(S),1)+i*randn(prod(S),1); %# Two random vectors
b=randn(prod(S),1)+i*randn(prod(S),1); %# Two random vectors
conj(a'*H(b))
b'*H(a)
```

**Hint:** Be sure to have run "setup.m" and "sch.m" recently so that all needed variables are defined.

## 4.3 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 $\partial E/\partial W^*_{i,n}$

Using the expression in Eq. (3) and your H() operator above, produce a function of the above prototype in the file "getgrad.m".
**Hint:** Matrix multiplication is associative, so that the final result of a matrix product $ABC$ does not depend on the order in which the product is formed, $A(BC)$ or $(AB)C$. However, the dimensions of the intermediate values can be quite different. For instance $(\mathbf{O}WU^{-1}W^\dagger)(HW)$ is the product of a $\prod S_k \times \prod S_k$ matrix with a $\prod S_k \times N_s$ matrix, while $(\mathbf{O}WU^{-1/2})(W^\dagger HW)$ is the product of a $\prod S_k \times N_s$ matrix with an $N_s \times N_s$ matrix. The former will likely overflow the memory, whereas the latter will fit nicely. Thus, in evaluating Eq. (3) in octave, you may wish to include some "extra" parentheses.

### 4.3.1 Debugging

Copy the function fdtest() from Appendix B into the file "fdtest.m". This program takes an initial $W$ containing Ns wave functions and computes the energy and gradient for that $W$ using your functions getE() and getgrad(). The function then forms a random direction and steps different distances along that direction, printing at each step the ratio of the actual change in energy to the change in energy expected from your gradient according to the formula Eq. (5). The number printed immediately below this ratio is a *rough* estimate of the amount of rounding error you can expect in this quantity.

To run this test, add the code block

```
%# Finite difference test
Ns=4; %# Number of states

randn('seed',0.2004);
W=(randn(prod(S),Ns)+i*randn(prod(S),Ns));

more off; %# View output as it is computed
fdtest(W);
```

to the bottom of your file "sch.m". Then run "setup.m" and "sch.m" and verify that you observe the ratio approach unity, gaining one order of magnitude per iteration before rounding error dominates the ratio.

**Notes:** The above code block sets the number of states for our problem to Ns=4, "seeds" the random number generator so that we all will get the same results, provides an initial random complex $W$ for "fdtest.m", does "more off" so that you can immediately see outputs as they are computed, and then calls fdtest.

**Hints if your code fails the above test:**

If your code fails the test, then it is helpful to debug the kinetic and potential energy parts separately. To test the kinetic energy part alone just delete (or comment out) the Vdual parts in *both* getE() and H(), and rerun. Then, repeat for the potential energy by putting back the Vdual parts and commenting out the L parts in *both* getE() and H(). If one of these works, but not the other, then you have isolated the problem.

If neither the potential nor the kinetic parts work, then the problem is likely in the algebra in your getgrad(). One way to test for this and to be able to debug H() and getE() independently of this extra algebra is to start with an initially orthonormal $W$. You can create such by including the statement "`W=W*inv(sqrtm(W'*O(W)));`" immediately before the call to fdtest. Be certain, however, to remove this statement and test again once you have identified your bug(s). For the rest of this problem set to function, it is critical that getgrad() and getE() work with non-orthonormal functions.

# 5 Solution of Schrödinger's equation using steepest descents: sd()

With the completion of get() and getgrad(), you are ready to solve Schrödinger's equation with the simple steepest descents algorithm described in lecture:

1. Initialize $W$
2. $W \leftarrow W - \alpha \nabla_W E$
3. Display $E$
4. Repeat (2 & 3) until converged

## 5.1 Initialize W

It is helpful to at least start with orthonormal wave functions. Thus, immediately after the call to fdtest in your "sch.m" file, orthonormalize $W$ according to equation Eq. (6), being sure to store your result back in $W$.

**Hint:** In octave, inv() and sqrtm(), respectively, return the *matrix* inverse and square root of a matrix, as opposed to simply inverting or taking the square root of each matrix element separately.

### 5.1.1 Debugging

You may check your formula by running "sch.m" and then typing `W'*O(W)`, which should now be the $4 \times 4$ identity matrix (to within machine precision).

## 5.2 sd()

`function out=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
- DISPLAY: with each iteration, print the result of getE() on current solutions

Provide a function of the above prototype which performs Nit iterations of the steepest descents algorithm with a step size of $\alpha = 3 \times 10^{-5}$. (You may wish to play with $\alpha$ later to see if you can find a better value.)

### 5.2.1 Debugging

After running "sch.m", have sd() improve $W$ with a relatively low number of iterations, `W=sd(W,20);`. You should be able to confirm that the energy decreases with each iteration. To check that the return value is correct, verify that `getE(W)` gives the same result as the most recent printout from sd(). Finally, run sd() with 250 iterations and verify that your result is converging to the analytic answer, E=18.

## 5.3 getPsi()

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

Input:

- W: $\prod S \times N_s$ matrix of non-orthonormal functions minimizing E

Output:

- Psi: eigensolutions
- epsilon: eigenvalues

Provide a function of the above prototype in the file "getPsi.m" which computes the final solutions from the non-orthonormal $W$ according to the formulas Eqs. (6,7).

**Hint:** Given the matrix "`mu=Y'*H(Y)`", the code fragment

```
[D, epsilon]=eig(mu); epsilon=real(diag(epsilon));
```

produces the matrix $D$ and the vector $\vec{\epsilon}$ in Eq. (9). (Again, we take a real part because rounding errors sometimes lead to tiny imaginary parts.)

### 5.3.1 Debugging

Check that your output states are orthonormal and diagonalize $\mu$ by executing

```
[Psi, epsilon]=getPsi(W); %# Run getPsi
Psi'*O(Psi) %# Should be the identity
Psi'*H(Psi) %# Should be diagonal
epsilon %# Should match the diagonal elements of previous matrix
```

# 6 Final solution: "sch.m"

Place the following code block at the end of your "sch.m" and run. (**Note:** if you wish to regain control before all of the iterations are complete, you have the option of hitting Ctrl-C once (and then <enter> if there is no immediate response. Be patient: if you hit Ctrl-C more than once, `octave` will likely crash, destroying your octave session and creating a large octave-core file.)

The code block below starts from random functions, performs a finite difference test, orthonormalizes the starting functions, runs 400 iterations of your sd(), and computes the final results with your getPsi(). Finally, the code displays the energy of each state along with grey-scale density plots of the values of $|\Psi_n(\vec{r})|^2$ on planes cutting through the center of your cell. For the graphics to function, be sure to copy "smooth.m" and "ppm.m" from Appendices C and D into your octave directory.

```
%# Allow for more digits in printouts
format long

%# Converge using steepest descents
W=sd(W,400);

%# Extract and display final results
[Psi, epsilon]=getPsi(W);

for st=1:Ns
  printf('=== State # %d, Energy = %f ===\n',st,epsilon(st));
  dat=abs(cI(Psi(:,st))).^2;
  for k=1:3
    sl=slice(dat,S,S(k)/2,k);
    name=sprintf("psi%d_m%d.ppm",st,k);
    ppm(name,sl*0.3,sl,sl); system(["display " name "&"]);
  end
end
```

For your reference, with an angular frequency of the oscillator of $\omega = 2$, the analytic result is that the first lowest four states have energies 3, 5, 5 and 5, with spatial character s, p, p and p, respectively. To view all of the electron density slices, place the mouse over the graphics window. (You may have to click on the title bar to get the graphics window's "attention".) Then, to flip through the pictures, hit the space bar while the mouse is over the image.

# 7 Density functional theory: "dft.m"

## 7.1 Background

At this stage, the only differences between your Schrödinger solver and a density functional solver are the energy and gradient functions, getE(), getgrad() and getH(), respectively.

To see this, we begin by noting that one important difference between the energy functions is that, now, each wave function $\psi_i$ has an associated occupancy, or "filling factor," $f_i$. For simplicity we shall here always work in cases where all states have the same occupancy $f_i = f$. (Many physical systems have this property; usually $f = 2$. The generalization to difference occupancies for different orbitals is conceptually straight forward but involves algebra beyond the scope of this mini-course.)

Under the conditions of the previous paragraph, three of the quantities from density-functional theory have very similar forms to those from the Schrödinger case. For instance, the density-functional form for the kinetic energy $(-1/2)\sum_i f_i \int \psi_i^* \nabla^2 \psi_i$ simply picks up an extra factor of $f$,

$$T = -\frac{1}{2}\sum_i f_i \int \psi_i^* \nabla^2 \psi_i = f \cdot \frac{1}{2}\mathrm{Tr}\,(W^\dagger L W U^{-1/2}). \tag{12}$$

The total electron density $\sum_i f_i |\psi_i|^2$ picks up a similar factor of $f$,

$$\vec{n} = f \cdot \mathrm{diag}\left[\mathbf{I}WU^{-1}\left(\mathbf{I}W\right)^\dagger\right]. \tag{13}$$

Finally, in terms of the vector of sample values $\vec{n}$, the formula for the electron-nuclear potential energy $PE = \int V(r)n(r)$ is identical to the corresponding formula in the Schrödinger case and thus leads to the same result,

$$U = \tilde{V}^\dagger \vec{n}, \tag{14}$$

where

$$\tilde{V} \equiv \mathbf{J}^\dagger \mathbf{O} \mathbf{J} \vec{V}, \tag{15}$$

with $\vec{V}$ being the sample values of the potential.

The first new term appearing in density-functional theory is the electron-electron potential energy $U_{ee} = \frac{1}{2}\int n(r)^* \phi(r)$, where $\nabla^2 \phi(r) = -4\pi n(r)$. (As in a similar derivation in lecture, the "*" on $n(r)$ doesn't change the integral because $n(r)$ is real. We include it, however, so that we may use our currently defined operator $\mathbf{O}$.) Inserting the expansions $n(r) = \sum_\alpha b_\alpha(r)\hat{n}_\alpha$ and $\phi(r) = \sum_\alpha b_\alpha(r)\hat{\phi}_\alpha$ into the integral for $U_{ee}$, using our solution for Poisson's equation from Problem Set 7 and rearranging quantities within †'s, we find

$$U_{ee} = \frac{1}{2}\vec{n}^\dagger \mathbf{J}^\dagger \mathbf{O} \hat{\phi}, \tag{16}$$

where the expansion coefficients for the solution to Poisson's equation are

$$\hat{\phi} = -4\pi L^{-1} \mathbf{O} \mathbf{J} \vec{n}.$$

The final term we require is the exchange correlation energy $\int n_(r)^* \epsilon_{xc}(n(r))$. Defining the operator $\epsilon_{xc}(\vec{n})$ as returning a vector each of whose components is the evaluation of the exchange-correlation energy for the corresponding component of $\vec{n}$ and inserting appropriate expansions gives the result,

$$E_{xc} = \vec{n}^\dagger \mathbf{J}^\dagger \mathbf{O} \mathbf{J} \epsilon_{xc}(\vec{n}). \tag{17}$$

In summary, in terms of unconstrained wave function coefficients $W$, the total energy within density functional theory is

$$\begin{aligned} E_{LDA} &= -f \cdot \frac{1}{2}\mathrm{Tr}\,\left(W^\dagger L W U^{-1}\right) + \tilde{V}^\dagger n \\ &\quad + \frac{1}{2}\vec{n}^\dagger \mathbf{J}^\dagger \mathbf{O} \hat{\phi} + \vec{n}^\dagger \mathbf{J}^\dagger \mathbf{O} \mathbf{J} \epsilon_{xc}(\vec{n}), \end{aligned} \tag{18}$$

where

$$\begin{aligned} U &\equiv W^\dagger \mathbf{O} W \\ \vec{n} &\equiv f \cdot \mathrm{diag}\left[\mathbf{I}WU^{-1}\left(\mathbf{I}W\right)^\dagger\right] \\ \hat{\phi} &\equiv -4\pi L^{-1} \mathbf{O} \mathbf{J} \vec{n}. \end{aligned}$$

As with the Schrödinger case, two effects contribute to the gradient. First, there are contributions due to the constraints, which turn out to be of an identical form as we had previously. Including the appropriate factors for $f$, we thus have

$$\nabla_W E = f\left(HW - \mathbf{O}WU^{-1}W^\dagger HW\right)U^{-1}. \tag{19}$$

Finally, the operator $H()$ encapsulates the remaining contributions to the gradient, which come from the basic structure of the energy and has the form,

$$H \equiv -\frac{1}{2}L + \mathbf{I}^\dagger\left(\mathrm{Diag}\, V_{\mathrm{eff}}\right)\mathbf{I}, \tag{20}$$

where

$$V_{\mathrm{eff}} \equiv \tilde{V} + \mathbf{J}^\dagger\mathbf{O}\hat{\phi} + \mathbf{J}^\dagger\mathbf{O}\mathbf{J}\epsilon_{xc}(\vec{n}) + \mathrm{Diag}\,\left[\epsilon'_{xc}(\vec{n})\right]\mathbf{J}^\dagger\mathbf{O}\mathbf{J}\vec{n},$$

with the operator $\epsilon'_{xc}(\vec{n})$ defined similarly to $\epsilon_{xc}(\vec{n})$ but now evaluating $\partial\epsilon_{xc}/\partial n$ on each component of $\vec{n}$.

## 7.2 Implementation strategy: "dft.m"

In principle, we need now only modify getE(), getgrad() and H() to perform density functional calculations. In order not to destroy your Schrödinger solver, copy each of the ".m" files from "∼/A2" to a new directory called "∼/A2a" where we shall now work. Once this is done, go to your new directory, rename "sch.m" to "dft.m", start a new octave session, and run "dft.m" from within octave to confirm that it behaves just as did "sch.m". (Don't forget to run "setup.m" first!)

Next, we shall make the appropriate changes to getE(), getgrad() and getH(). Because the minimization algorithm is already debugged, we need first only confirm that we have an appropriate energy and gradient with the finite difference test, and then we are ready for full density-functional calculations!

To prevent proliferation of output as we run these tests, you may wish to put a "`pause;`" statement immediately after the call to fdtest() in "dft.m". Again, you should run "dft.m" one last time to confirm all is well. Hitting Ctrl-C (and then <enter> if there is no immediate response) when the program hits the pause statement will regain the octave prompt.

## 7.3 Occupancies

The various energy and gradient functions require access to the filling filling factor $f$. Rather than passing $f$ as an argument down through all of the various functions, we will allow ourselves one final global variable, "gbl_f", a blemish which a `C++` implementation could easily eliminate. Add the following lines to the very top of "dft.m",

```
%# Set the orbital occupancies
global gbl_f;
gbl_f=2; %# The usual case of a constant filling of two electrons per orbital
```

Your original Schrödinger solver implicitly assumed state occupancies of $f = 1$. The above energy and gradient formulas for density-functional theory should work just as well for general values of $f$, even if the $\phi$ and $\epsilon_{xc}$ terms are set to zero. Thus, before adding any of the new terms, add the factors of $f$ to the appropriate places in getE() and getgrad().

**Hints:** Don't forget to declare gbl_f as a global variable in each function which needs "f" and to include it in both the density and the kinetic energy parts.

### 7.3.1 Debugging

Run "dft.m" and verify proper functioning of the finite-difference test.

## 7.4 Hartree theory

Ignoring the exchange-correlation terms represents the first mean-field many-electron theory, "Hartree theory." We will begin with this theory as it involves energy terms with which we are already familiar.

Add all terms to getE() and H() from Eqs. (18,20) which involve $\phi$.

**Hints:** Note that you will have to build the density n and solve Poisson's both in get() and in H(). Don't forget to orthonormalize the wave functions before computing n!

### 7.4.1 Debugging

Because of possible cancellations, the most convincing test is to first zero out all terms except the Hartree terms in the energy and gradient before running the finite difference test. (Any easy way to do this is without much typing is to just multiply the unwanted terms temporarily by zero with "*0".) Once you have confirmed your Hartree terms, then turn all of the other terms back on and run the test again to make sure everything is turned on properly.

## 7.5 Full density-functional theory

Add the remaining terms in Eqs. (18,20), those involving $\epsilon_{xc}$ and $\epsilon'_{xc}$, to get() and H(). You may use the functions excVWN() and excpVWN() in Appendices E and F, which implement the operators $\epsilon_{xc}(\vec{n})$ and $\epsilon'_{xc}(\vec{n})$, respectively, using the Vosko-Wilk-Nusair parameterization.

### 7.5.1 Debugging

Run the finite difference test, first zeroing out all terms but those involving $\epsilon_{xc}$ and $\epsilon'_{xc}$ and then including all terms.

## 7.6 Quantum dot

A common approximation for a "quantum dot," a relatively small number of electrons trapped in a potential, is to model the trapping potential as a harmonic oscillator (which actually is the potential V(r) which you have currently in "dft.m") and to treat the electron-electron interactions within density-functional theory. To perform such a calculation, run "dft.m" through to its end. (Be sure to remove any "`pause;`" statements so that the code runs through the part which gets the final wave functions and plots them.)

Run your code. You should find a converged value of the total energy for this quantum dot of 8 electrons (4 states, with two electrons each) near Etot=43.337 H. You should also find states with similar s and p characters as to what you had before, but now with eigenvalues of approximately 5.509 H, 6.949 H, 6.949 H, 6.949 H. (The lesser upward shift of the p states reflects centrifugal repulsion from the center, a common feature in the shell structure of quantum dots.)

# A "viewmid.m"

```
% Function to view slices of three dimensional data sets
%
% Usage: view(dat,S)
%
% dat: 3d data set (any shape) of total size prod(S)=S(1)*S(2)*S(3)
% S: dimensions of dat in a 3-vector

function viewmid(dat,S)

  fprintf('\nRemember to hit <enter> or <spacebar> after each plot!\n\n');

  for k=1:3
    text=sprintf("m%d=%d slice",k,floor(S(k)/2));
    fprintf("%s...\n",text);
    title(text);
    if k==1
      xlabel("m3 ->"); ylabel("m2 ->");
    elseif k==2
      xlabel("m3 ->"); ylabel("m1 ->");
    elseif k==3
      xlabel("m2 ->"); ylabel("m1 ->");
    end

    mesh(slice(dat,S,S(k)/2,k)); pause;
  end
```

## B "fdtest.m"

```
% Performs finite difference test of getE() and getgrad()
%
% Usage: fdtest(W,S)
%
% W: starting point for test (size: prod(S) x Ns)
% S: Dimensions of 3d data

function fdtest(W)
  %# Compute intial energy and gradient
  E0=getE(W)
  g0=getgrad(W);

  %# Choose a random direction to explore
  dW=randn(size(W))+i*randn(size(W));

  %# Explore a range of step sizes decreasing by powers of ten
  for delta=10.^[1:-1:-9]
    %# Directional derivative formula
    dE=2*real(trace(g0'*delta*dW));

    %# Print ratio of actual change to expected change, along with estimate
    %#   of the error in this quantity due to rounding
    printf(' %20.16f\n %20.16f\n\n', ...
    (getE(W+delta*dW)-E0)/dE, sqrt(size(W,1))*eps/abs(dE) );
  end
endfunction
```

## C "smooth.m"

```
%# Double data density by linear interpolation
function out=smooth(in)
  out=colsmooth(in);
  out=colsmooth(out')';
endfunction

%# Double row density by smoothing columns
function out=colsmooth(dat)
  nc=size(dat,1);
  out=zeros(2*nc-1,size(dat,2));
  out(1:2:2*nc-1,:)=dat;
  out(2:2:2*nc-2,:)=(out(1:2:2*nc-3,:)+out(3:2:2*nc-1,:))/2;
endfunction
```

# D "ppm.m"

```
%# Usage: ppm(fname,red,green,blue)
%#
%# Output- color ppm image in file "fname" (view with "xli fname").
%# Input- red, green, blue: 2d data of red, green, blue intensities

function ppm(fname,red,green,blue)
  %# Enlarge image
  for en=1:4
    red=smooth(red); green=smooth(green); blue=smooth(blue);
  endfor

  pixmx=255;
  height=size(red,1); width=size(red,2);
  mx=max(max([red, green, blue]));
  mn=min(min([red, green, blue]));

  fid=fopen(fname,'w');
  fprintf(fid,'P3\n');
  fprintf(fid,'%d %d\n',width,height);
  fprintf(fid,'%d\n',pixmx);

  dat=([reshape(red',1,width*height); ...
        reshape(green',1,width*height); ...
        reshape(blue',1,width*height)] ...
           -mn)/(mx-mn)*pixmx;

  fprintf(fid,'%d ',dat);
  fclose(fid);
endfunction
```

# E "excVWN.m"

```
% VWN parameterization of the exchange correlation energy
function out=excVWN(n)
  %# Constants
  X1 = 0.75*(3.0/(2.0*pi))^(2.0/3.0);
  A  =  0.0310907;
  x0 = -0.10498;
  b  = 3.72744;
  c  = 12.9352;
  Q  = sqrt(4*c-b*b);
  X0 = x0*x0+b*x0+c;

  rs=(4*pi/3*n).^(-1/3); %# Added internal conversion to rs

  x=sqrt(rs); X=x.*x+b*x+c;

  out=-X1./rs ...
    + A*( ...
+log(x.*x./X)+2*b/Q*atan(Q./(2*x+b)) ...
-(b*x0)/X0*( ...
    log((x-x0).*(x-x0)./X)+2*(2*x0+b)/Q*atan(Q./(2*x+b)) ...
    ) ...
);
endfunction
```

## F "excpVWN.m"

```
% d/dn deriv of VWN parameterization of the exchange correlation energy
function out=excpVWN(n)
  %# Constants
  X1 = 0.75*(3.0/(2.0*pi))^(2.0/3.0);
  A  =  0.0310907;
  x0 = -0.10498;
  b  = 3.72744;
  c  = 12.9352;
  Q  = sqrt(4*c-b*b);
  X0 = x0*x0+b*x0+c;

  rs=(4*pi/3*n).^(-1/3); %# Added internal conversion to rs

  x=sqrt(rs); X=x.*x+b*x+c;


  dx=(0.5)./x; %# Chain rule needs dx/drho!

  out=dx.*( ...
     2*X1./(rs.*x)+A*( ...
    (2)./x-(2*x+b)./X-4*b./(Q*Q+(2*x+b).*(2*x+b)) ...
    -(b*x0)/X0*((2)./(x-x0)-(2*x+b)./X-4*(2*x0+b)./ ...
                                                    (Q*Q+(2*x+b).*(2*x+b)) ) ...
    ) ...
                              );

  out=(-rs./(3*n)).*out; %# Added d(rs)/dn from chain rule from rs to n conv

endfunction
```