Cornell University

Department of Physics

Practical DFT February 10, 2014

Phys 7654, Module #2, Spring 2014

Practical Density-Functional Theory

Homework Assignment # 1

(Due Friday, February 21 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 and transforms for planewaves; Introduction to octave/matlab notations.
  Reading:4.1 and 4.2 of "DFT++-Derivations".

- **Lecture 3: An expressive software language for density-functional theory.** Aliasing/Nyquist frequency; minimization by steepest descents; Analytic continuation for constraints (generalization of Rayleigh-Ritz principle to multople states and density functional theory); DFT++ expressions for total energy in density-functional theory.
  Reading: Section 4.3 of "DFT++-Derivations".

- **Lecture 4: Expressions/code for minimization of density-functional energy.** Derivation of gradient of density-functional theory energy for multiple states with constraints. **Reading: Sections 4.4-4.7 of "DFT++-Derivations".**

- **Lecture 5: Computer lab on Poisson's equation and Improved minimization algorithms** Exploration of quality of solution of Poisson's equation with spectral methods and practical considerations. Further improvements in time/memory efficiency to allow DFT calculations in solids. 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.

Contents

1 Overview

Solution of Poisson's equation as the single line of code

```
phi=cI(Linv(-4*pi*O(cJ(n))));
```

requires two capabilities not common in C programs. First, in the above expression, the calls to functions such as cJ(...) return not single numbers but entire arrays. Second, the expression -4*pi*O(...) represents not just simple scalar multiplications but rather multiplication of the scalars -4 and pi with the *entire* array returned by O(...). We therefore need the ability to work with *objects* such as arrays as easily as we work with numbers in `C`.

The `C++` language supports the capability of writing software which allows for such operations very generally. However, the `octave` language already automatically includes these operations, tuned automatically for peak performance, for the typical sort of linear algebraic operations common in scientific computing. We thus shall use `octave` for this course.

2 Definition of basic variables: "setup.m"

2.1 Indexing

Computational implementation of problems in $d = 3$ dimensions requires careful mapping of three-dimensional objects into a linear memory space. In lecture, we developed an approach to this indexing through the formation of two index matrices $\mathbf{M}$ and $\mathbf{N}$, where

$$\mathbf{M} \equiv \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ \vdots & 0 & 0 \\ S_1-1 & 0 & 0 \\ S_1 & 0 & 0 \\ \vdots & 1 & 0 \\ \vdots & \vdots & 0 \\ \vdots & S_2-1 & 0 \\ \vdots & S_2 & 0 \\ \vdots & \vdots & 1 \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & S_3-1 \\ \vdots & \vdots & S_3 \end{bmatrix} \quad (1) \qquad \mathbf{N} \equiv \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ \vdots & 0 & 0 \\ -2 & 0 & 0 \\ -1 & 0 & 0 \\ \vdots & 1 & 0 \\ \vdots & \vdots & 0 \\ \vdots & -2 & 0 \\ \vdots & -1 & 0 \\ \vdots & \vdots & 1 \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & -2 \\ \vdots & \vdots & -1 \end{bmatrix} \quad (2)$$

As a specific example, if S=[3; 3; 2], then

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & 0 \\ 1 & 2 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 2 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 1 & 1 \\ 0 & 2 & 1 \\ 1 & 2 & 1 \\ 2 & 2 & 1 \end{bmatrix} \tag{3}$$

This problem guides you step by step through the setup of these matrices while exploiting the interactive aspects of the `octave` language.

Begin by creating a directory "∼/A1" in which you shall put all of the work for this first assignment. Working in "∼/A1", create a file "setup.m" containing a single line of octave code:

```
S=[3; 4; 5]
```

Verify that you can run octave by starting octave (in the same directory "∼A1" with "setup.m") and entering "setup" at the octave prompt. (Don't forget to hit the enter key!) This will run the octave commands in the file "setup.m" Octave should respond:

```
S =

  3
  4
  5
```

Next, create the first two columns of **M** by adding the following lines to "setup.m" immediately after the statement declaring S. (**Note:** You should be able to cut and paste the code block below and the other code blocks in these assignments right out of the PDF documents and into your code editor!!!)

```
%# Code fragment to create columns of m1, m2, m3 indices and the matrix M
ms=[0:prod(S)-1]'; %# Count from zero to S1*S2*s3-1 in a column vector
m1=rem(ms,S(1));
m2=rem(floor(ms/S(1)),S(2));
m3=rem(floor(ms/(S(1)*S(2))),S(3));
```

Rerun "setup.m" and inspect and verify the result by entering first just "`m1`", then just "`m2`", and finally just "`m3`" at the octave command prompt. Note that you can inspect any part of the above expressions, such as "`floor(ms/S(1))`", by entering it at the command prompt. Also, by entering "whos", you can get a report of all of the variables currently defined and, very usefully, their dimensions. (The function size() may be used to find the size of any single object.) Finally, combine each of your columns into a matrix using the octave block matrix formation

```
M=[m1, m2, m3];
```

Inspect and verify your final result by rerunning "setup.m" and entering "M" at the command prompt.

Finally, write a block of code to compute **N**, verifying your results in a similar way.

Hint: Compute the columns of **N** from the corresponding columns of **M** using forms such as

```
n1=m1-(m1>S(1)/2)*S(1);
```

and then combine your result into the the final matrix.

2.2 Sampling points and reciprocal lattice vectors

To specify the lattice vectors, add the following statement immediately following the statement specifying **S** and before the statements computing **M** and **N**,

```
R=diag([6; 6; 6])
```

Note that this means that we'll be working with a cubic unit cell of side length 6 bohr.

Next, add a code block to the bottom of your "setup.m" file to form the lists of real-space sample points **r** and reciprocal lattice vectors **G** from the formulae derived in lecture,

$$\mathbf{r} = \mathbf{M}\left(\mathbf{Diag}\left(S\right)\right)^{-1}\mathbf{R}^T \tag{4}$$

$$\mathbf{G} = 2\pi\mathbf{N}\mathbf{R}^{-1} \tag{5}$$

Finally, add a statement to create a column vector G2 containing the square-magnitude of each G-vector.

Hint: For this final part, note that `G.^2` produces the squares of each element of G and that `sum(A,d)` sums the matrix A along its d-th dimension.

To check your final results from "setup.m", you should copy the two files "slice.m" and "view.m" from Appendix A and Appendix B, respectively, into "∼A1"[1]. To understand more how the view() and slice() functions work, you may enter "`help slice`" and "`help view`" at the octave prompt after you have copied the files.

Once you have the above functions in your directory, change the size specification at the top of "setup.m" to that of a $20 \times 25 \times 30$ problem (`S=[20; 25; 30]`), rerun "setup.m", and then enter "`view(G2,S)`" at the octave prompt. This will then generate plots of slices of G^2 in the n1=0 plane, n2=0, and n3=0 planes, respectively. In each case, you should see parabolic surfaces centered on the origin.

Finally, check your result for **r** by running "view(r(:,k),S)" for k=1,2,3 and verifying that you see the correct behavior.

3 Charge distribution: "poisson.m"

The charge density for our example solution to Poisson's equation will be

$$n = g_1(r) - g_2(r) \tag{6}$$

$$\equiv \frac{e^{-r^2/(2\sigma_2^2)}}{(2\pi\sigma_2^2)^{3/2}} - \frac{e^{-r^2/(2\sigma_1^2)}}{(2\pi\sigma_1^2)^{3/2}} \tag{7}$$

where r is the distance from the *center* of the cell, $g_1(r)$ and $g_2(r)$ are normalized three-dimensional Gaussian distributions (each containing a net charge of unity), and $\sigma_1 = 0.75$ bohr and $\sigma_2 = 0.50$ bohr, respectively.

[1]Note that copies of these and all program files in a zip file under "2011 Phys 7654 Practical DFT⇒Computer Files" at http://TomasArias.com .

Copy the code "poisson.m" from Appendix C to "~/A1" (or download directly from http://TomasArias.com), and complete the program by replacing "..." on the line labelled "CODE INSERTION # 1" with a code fragment to evaluate in the column vector dr the distance between each sampling point in **r** and the center of the cell,

```
sum(R,2)/2
```

Hints: Consider the matrix

```
ones(prod(S),1)*sum(R,2)'
```

and use a similar procedure to what you used to compute G2. Feel free to type `ones(prod(S),1)*sum(R,2)'` at the command prompt to see the value of this expression. Finally, note that in `octave`, "sqrt(A)" creates an object of the same dimensions as "A" and with elements equal to the square root of each corresponding element of "A".

To verify your code fragment, run "poisson.m". This octave program will use the distances which you compute in dr to evaluate g_1, g_2 and n according to Eq. (6). The program then checks your distributions by integrating each of these functions by a simple Riemann sum $\int f(\vec{r})\, d^3r \approx \sum_i f(r_i)\Delta V$, where the volume per sample point is $\Delta V = \det R / \prod_{k=1}^{3} S_k$. For these tests, your results should be good to at least 3 decimal places. Finally, to aid in debugging, the program will plot the charge density in three perpendicular planes passing through the *center* of your cell. Each of these should appear as a positive Gaussian with a noticeable negative ripple as you move away from the main peak.

4 Operators

With the indexing and charge density constructed, the next step is to provide software for the various operators. Although there are a number of operators to provide and debug, each operator comes more quickly as you become more familiar with octave. With the completion of these operators, you will be a single line of code away from a general algorithm for the solution of Poisson's equation in three dimensions!

4.1 Global variables

Each of the operators, O(), L(), Linv(), cI(), cJ() require certain basic data, such as R, S or G2, to carry out its operation. In normal C programming style, we would pass this information as additional arguments to the functions. However, our solution to Poisson's equation would then cease to resemble the formal expression $\vec{\phi} = \mathbf{I}\,\mathbf{L}^{-1}\left(-4\pi\mathbf{O}\,\mathbf{J}\vec{n}\right)$ and become the quite ugly expression

```
phi=cI(Linv(-4*pi*O(cJ(n,S),R),R,G2),S);
```

`C++` provides an elegant solution to this problem. With full control over object types, one could ensure that each vector includes a pointer to the information in R, S and G2, thus giving each operator access to the relevant information through its input vector. Octave does provide a primitive capacity to build and pass structures, but there is not enough control over these structures to provide the capabilities we would require.

Our solution to this dilemma is less elegant than the `C++` solution but workable within the limits of octave. The alternative to passing information to a function through an argument is to pass it through a "global" variable. Generally, we strongly discourage the use of such variables because one easily looses track of which functions may, or may not, unexpectedly change the values of such variables. Under such circumstances, debugging becomes extremely difficult.

Fortunately, in our case, the information we wish to pass via global variables (R, S, G2) remains constant throughout our calculations, thus averting most of the dangers associated with use of global variables. Also, we will take the extra precaution of prefixing all global variables with "`gbl_`" so as to to mitigate the chances of inadvertent modification of global variables with common variable names.

To make the setup information available as global variables, declare the corresponding variables as global by adding the following lines to the very top of "setup.m"

```
%# Make setup info globally accessible (ugh!)
global gbl_S; global gbl_R; global gbl_G2;
```

and set these variables to the appropriate values by adding the following lines to the very bottom of "setup.m"

```
%# Assign computed values to the global variables
gbl_S=S; gbl_R=R; gbl_G2=G2;
```

4.2 O()

`function out=O(in)`

Input:

- in: $d = 3$ dimensional data stored as an $\prod_k S_k \times 1$ vector

Global variables:

- gbl_R: lattice vectors $\mathbf{R}$

Output:

- out: O operator applied to in, where $\mathbf{O} = (\det \mathbf{R})\mathbf{1}$, with $\mathbf{1}$ being the identity matrix.

To produce a function of the above prototype, copy the program in Appendix D into the file "~/A1/O.m" and replace the "..." on the line labeled "YOUR CODE HERE" with code to compute out according to the definition of the O() operator in the function specification given above.

Hints: Don't forget that, to access R, you will need to use the variable name gbl_R. Also, be sure to include a semicolon at the end of your line for computing out. Otherwise, you'll be plagued by a large printout of your results! Finally, be more clever in your implementation than wasting lots of time and space by actually forming the matrix $\mathbf{O}$ explicitly and then multiplying by it – your only responsibility is to make sure that the return value of O() is correct!!!

To verify your O() operator, execute the following at the octave prompt (after running your new "setup.m", of course),

```
in=randn(10,1) %# Create a random (normally distributed) 10x1 column vector
out=O(in); %# Apply O to in and store result in out
out./in %# Check ratio of each element of out to each element of in
det(R) %# Compare to det(R)
```

4.3 L()

`function out=L(in)`

Input:

- in: $d = 3$ dimensional data stored as an $\prod_k S_k \times 1$ vector

Global variables:

- gbl_R: lattice vectors $\mathbf{R}$
- gbl_G2: lengths squared of G vectors

Output:

- out: L operator applied to in, where $\mathbf{L} = -(\det \mathbf{R})(\mathbf{Diag}\, G2)$

Using your software for "O.m" as an example, create a file called "L.m" containing software for a function L() of the above specification.

Hint: Do not form the matrix **L** directly. Rather, use the fact that, in octave, a.*b produces a vector of the products of the corresponding elements of a and b.

To verify your L() operator, execute the following at the octave prompt,

```
in=randn(prod(S),1); %# Create a random d=3 dimensional column vector
out=L(in); %# Apply O to in and store result in out
[out./in, -det(R)*G2] %# Compare ratio of out to in to -det(R)*G2
```

4.4 Linv()

`function out=Linv(in)`

Input:

- in: $d = 3$ dimensional data stored as an $\prod_k S_k \times 1$ vector

Global variables:

- gbl_R: lattice vectors **R**
- gbl_G2: lengths squared of G vectors

Output:

- out: inverse of L operator applied to in, where $\mathbf{L} = -(\det \mathbf{R})(\mathbf{Diag}\, G2)$ and, by convention, out(1)≡0.

Using your software for "L.m" as an example, create a file called "Linv.m" containing software for a function Linv() of the above specification.

Hint: Do not form the matrix $\mathbf{L}^{-1}$ directly. Rather, use the fact that a./b produces a vector of the ratios of the corresponding elements of a and b.

To verify your Linv() operator, execute the following at the octave prompt,

```
in=randn(prod(S),1); %# Create a random d=3 dimensional column vector
Linv(L(in))./in %# Check ratio of Linv applied to L(in) to in
```

4.5 cI()

`function out=cI(in)`

Input:

- in: $d = 3$ dimensional data stored as an $\prod_k S_k \times 1$ vector

Global variables:

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

Output:

- out: cI operator applied to in

Write a file "cI.m" containing a function of the above specification. To carry out the forward transform using FFTW, copy the file "fft3.m" from Appendix E into "∼/A1". To use this function, note that `fft3(dat,S,1)` returns the discrete Fourier sum with sign $+i$ in the exponential for data of dimension S(1)×S(2)×S(3). (For more information, enter "help fft3"once you've copied "fft3.m".)

4.6 cJ()

```
function out=cJ(in)
```

Input:

- in: $d = 3$ dimensional data stored as an $\prod_k S_k \times 1$ vector

Global variables:

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

Output:

- out: cJ operator applied to in, where cJ≡cI^{-1}

Write a file "cJ.m" containing a function of the above specification which uses FFTW to compute the inverse Fourier transform.

Hint: Do not forget the normalization factor $1/\prod_k S_k$.

To test your transforms execute the following commands at the octave prompt:

```
in=randn(prod(S),1); %# Random input vector
cJ(cI(in))./in %# Check ratio of cJ applied to cI(in) to in
```

5 Solution to Poisson's equation: "poisson.m"

With the operators coded, you are one line away from the solution to Poisson's equation!

Uncomment the remaining lines in "poisson.m" and replace the "..." on the line labeled "CODE INSERTION # 2" with your single line solution to Poisson's equation. You may then run "poisson.m" to check your results!

To confirm your solution, "poisson.m" first takes the real part (due to rounding errors and that fact that the Fourier transformation is complex, tiny imaginary parts creep into your solution), and then plots slices of your solution through planes passing through the center point of the cell. Finally, "poisson.m" compares the known analytic result with the integral for the total Coulomb energy, $U = (1/2)\int n\phi$, obtained with your numerical solution for ϕ.

IMPORTANT NOTE for MATLAB users: The software for this course was designed using Octave. The Matlab mesh() function (which we use in the view() function provided in the appendix) calls view() internally. Matlab, though, will end up having mesh() call our user defined view() instead of its internal function, resulting in a bug. Consequently, Matlab users should rename our view() function to something less generic like view7654(), change the name of the function definition at the top of the view, rename the file "view.m" to "view7654.m" and be sure to replace all calls to view() in our programs (such as "poisson.m") to view7654().
Hint: The comparison of energies should agree to four significant figures.

6 Ewald energy calculator

Molecules and solids, by definition, have more than one atom. Thus, the nuclear-nuclear interaction U_{NN}, which is zero for an *isolated* atom, will soon become for us an important term in the energy. When using periodic boundary conditions, we are effectively repeating each atom in our cell infinitely many times in a periodic arrangement throughout space. Summing the Coulomb energy,

$$U_{NN} \equiv \frac{1}{2}\sum_{I\neq J}\frac{Z_I Z_J}{||\vec{R}_I - \vec{R}_J||},$$

directly then becomes a very tricky task.

A very powerful technique for computing this term is the "Ewald summation" technique. This technique is based on the fact that, outside a spherically symmetric charge distribution, the distribution acts just like a point charge concentrated at the center. Thus, we can compute U_{NN} with our Poisson solver by simply replacing each of the nuclear charges with a Gaussian containing the same net charge. The only subtlety is that the Gaussians will inevitably overlap somewhat and not be entirely "outside" of each other. The full Ewald algorithm handles this correction explicitly. For us, by choosing the Gaussians to narrow enough that the overlaps are insignificant, we can make a workable (but not optimal) implementation of the Ewald technique. Because your "poisson.m" script already gives the energy of continuous, periodic charge distributions, you are a few changes away from having a quick-and-dirty Ewald summation program.

From the above discussion, we see that the Ewald technique hinges on construction of a density of the form

$$n(\vec{x}) = \sum_I g(\vec{x} - \vec{R}_I), \tag{8}$$

where $g(\vec{x})$ is a Gaussian normalized to the charge of each nucleus Z (assumded here to be the same for all nuclei), and the $\vec{R}_I$ are the locations of the nuclei. Notice that the expansion coefficients $\hat{n_\alpha}$ are easy to compute once we have the expansion coefficients $\hat{g}_\alpha$ for $g(x)$,

$$g(\vec{x}) = \sum_\alpha \hat{g}_\alpha e^{i\vec{G}_\alpha \cdot \vec{x}}. \tag{9}$$

Inserting (9) into (8) gives,

$$\begin{aligned} n(\vec{x}) &= \sum_I \sum_\alpha \hat{g}_\alpha e^{i\vec{G}_\alpha \cdot (\vec{x} - \vec{R}_I} \\ &= \sum_\alpha \left(\hat{g}_\alpha \sum_I e^{-\vec{G}_\alpha \cdot \vec{R}_I} \right) e^{i\vec{G}_\alpha \cdot \vec{x}}, \end{aligned}$$

where we have used the product rule for addition of exponents and rearranged the terms in the sums. Comparing the result to the expansion for $n(\vec{x})$,

$$n(\vec{x}) = \sum_\alpha \hat{n}_\alpha e^{i\vec{G}_\alpha \cdot \vec{x}},$$

we see immediately that the expansion coefficients which we need for $n(\vec{x})$ are just

$$\hat{n}_\alpha = \hat{g}_\alpha S_{\vec{G}_\alpha},$$

where the so-called "structure factor" is defined as

$$S_{\vec{G}_\alpha} \equiv \sum_I e^{-\vec{G}_\alpha \cdot \vec{R}_I}.$$

6.1 Additions to "setup.m"

The script "setup.m" is an appropriate place to put the locations of the ionic cores, which we shall store in an $n \times 3$ matrix $\mathbf{X}$, and their charges, which we shall call Z. In "setup.m", immediately after your statements defining the FFT box size S and the lattice vectors $\mathbf{R}$, place the statements

```
%# Define atomic locations and nuclear charge
X=[0 0 0; 1.75 0 0]; Z=1;
```

which specifies two protons ($Z = 1$) at a distance of 1.75 bohr, an example we will consider shortly when we compute the bond length of H_2 within density-functional theory.

Finally, the information about the atomic locations is best conveyed to the other parts of our program through the “structure factor” derived above

$$S_f(\vec{G}) \equiv \sum_I e^{-i\vec{G}\cdot\vec{X}_I}, \tag{10}$$

where I ranges over the atomic cores and $\vec{X}_I$ is the location of the I^{th} core. To evaluate this quantity for the remainder of the program, place the statements

```
%# Computation of structure factor
Sf=sum( exp(-i*G*X'), 2);
```

immediately after the computation of the matrix G and the vector G2.

Debugging: Confirm that “setup.m” continues to run without error.

6.2 “ewald.m”

Make a copy of “poisson.m” called “ewald.m” and modify “ewald.m” as follows.

1. Modify the calculation of the first Gaussian g1 so that (a) its width is 0.25 bohr instead of 0.75 bohr and (b) its norm is Z instead of unity. (To avoid real-space truncation effects at the cell edges, g1 should remain at the center of the cell.)

2. Remove all references to g2.

3. The total charge should be evaluated as

   ```
   %# Use structure factor to create all atoms
   n=cI(cJ(g1).*Sf); n=real(n);
   ```

   rather than `g2-g1`. (The statement `n=real(n);` simply removes tiny imaginary parts due to rounding.)

4. Keep the printouts which check the normalization of the Gaussian g1 and the total charge n in the cell.

5. Replace the computation of Uanal with the following statement that computes the self-energies of the Gaussians,

   ```
   Uself=Z^2/(2*sqrt(pi))*(1/sigma1)*size(X,1);
   ```

6. The final printout should give Unum-Uself as the final result for the Ewald energy.

Debugging: The visualizations should show a “dimer” oriented along the x direction, with one “bump” in the n1-slice and two in the n2- and n3- slices. Finally, the Ewald energy should evaluate to approximately -0.333 hartree. Once you have completed this quick verification of “ewald.m”, you should comment out the two visualization code blocks as they dramatically slow the run time for larger calculations.

As final confirmation, you should find that at a separation of 4.00 bohr along the x-direction and Gaussian width sigma1=0.5, you get results which are the same to within about 0.025 millihartree for setup parameters S=[32; 32; 32] and R=diag([16; 16; 16]) as you do for setup parameters S=[64; 64; 64] and R=diag([16; 16; 16]). (Be sure to modify *and rerun* “setup.m” and “ewald.m” to change these parameters and to make sure that the changes take effect!) More importantly, your result should change by only about 1 millihartree when you change to sigma1=0.25 and evaluate at S=[64; 64; 64], R=diag([16; 16; 16]). All of these results should be about -0.094 hartree.

To ensure that everyone works with the same parameters for future problems, be sure, after running these tests, to restore S and R in “setup.m” to their original values of S=[20; 25; 30], R=diag([6 6 6]), and sigma1 in “ewald.m” to its original value of sigma1=0.25.

A “slice.m”

```
% Function to extract two dimensional slices from a 3d data set
%
% Usage: out=slice(dat,N,n,dir)
%
% out: n-th dir-plane of dat (lower remaining dimension leading)
% n: desired slice number from data; 1 <= n <= N(dir)
% dir: direction perpendicular to slice --- dir=1,2,3 gives yz,xz,yz planes
% dat: 3d data set (any shape) of total size prod(N)=N(1)*N(2)*N(3)
% N: dimensions of dat in a 3-vector

function out=slice(dat,N,n,dir)
  n=floor(n); %# Be sure to take integer part to avoid errors
  if n<1
    printf("\nAsking for non-existent slice, n=%f.\n\n",n);
    return
  endif

  if dir==3
    if n>N(3)
      printf("\nAsking for non-existent slice, n=%f.\n\n",n);
      return
    endif
    dat=reshape(dat,N(1)*N(2),N(3)); %# Group into matrix with dir=3 as cols
    out=reshape(dat(:,n),N(1),N(2)); %# Take n-th col and reshape as slice
  elseif dir==2
    if n>N(2)
      printf("\nAsking for non-existent slice, n=%f.\n\n",n);
      return
    endif
    dat=reshape(dat,N(1)*N(2),N(3)); %# Group to expose N(2)
    dat=conj(dat'); %# dat is now in order N(3),N(1)*N(2)
    dat=reshape(dat,N(3)*N(1),N(2)); %# Form with dir=2 as cols
    out=reshape(dat(:,n),N(3),N(1)); %# Shape into slice
    out=conj(out'); %# Reorder as N(1),N(3);
  elseif dir==1
    if n>N(1)
      printf("\nAsking for non-existent slice, n=%f.\n\n",n);
      return
    endif
    dat=reshape(dat,N(1),N(2)*N(3)); %# Group to expose N(1)
    dat=conj(dat'); %# dat is now N(2)*N(3),N(1)
    out=reshape(dat(:,n),N(2),N(3));
  else
    printf("\nError in slice(): invalid choice for dir.  dir=%f\n\n",dir);
  endif

endfunction
```

B “view.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 view(dat,S)

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

  for k=1:3
    if k==1
      fprintf('m1=0 slice...\n');
      title("m1=0 slice"); xlabel("m3 ->"); ylabel("m2 ->");
    elseif k==2
      fprintf('m2=0 slice...\n');
      title("m2=0 slice"); xlabel("m3 ->"); ylabel("m1 ->");
    elseif k==3
      fprintf('m3=0 slice...\n');
      title("m3=0 slice"); xlabel("m2 ->"); ylabel("m1 ->");
    end

    mesh(slice(dat,S,1,k)); pause;
  end

endfunction
```

C "poisson.m"

```
%# Code to solve Poisson's equation

%# Compute distances dr to center point in cell
dr= ... %# <=== CODE INSERTION # 1

%# Compute two normalized Gaussians (widths 0.50 and 0.75)
sigma1=0.75;
g1=exp(-dr.^2/(2*sigma1^2))/sqrt(2*pi*sigma1^2)^3;

sigma2=0.50;
g2=exp(-dr.^2/(2*sigma2^2))/sqrt(2*pi*sigma2^2)^3;


%# Define charge density as the difference
n=g2-g1;

%# Check norms and integral (should be near 1 and 0, respectively)
fprintf('Normalization check on g1: %20.16f\n',sum(g1)*det(R)/prod(S));
fprintf('Normalization check on g2: %20.16f\n',sum(g2)*det(R)/prod(S));
fprintf('Total charge check: %20.16f\n',sum(n)*det(R)/prod(S));

%# Visualize slices through center of cell
for dir=1:3
  text=sprintf("n%d=%d slice of n",dir,S(dir)/2);
  title(text);
  fprintf("%s (Hit <enter>... )\n",text);
  mesh(slice(n,S,S(dir)/2,dir)); pause;
end

%#%# Solve Poisson's equation
%#phi= ... %# <=== CODE INSERTION # 2
%#
%#%#Due to rounding, tiny imaginary parts creep into the solution.  Eliminate
%#%#by taking the real part.
%#phi=real(phi);
%#
%#%# Visualize slices through center of cell
%#for dir=1:3
%#  text=sprintf("n%d=%d slice of phi",dir,S(dir)/2);
%#  title(text);
%#  fprintf("%s (Hit <enter>... )\n",text);
%#  mesh(slice(phi,S,S(dir)/2,dir)); pause;
%#end
%#
%#%# Check total Coulomb energy -- must use ' NOT transpose() because ' includes the needed complex con
%#Unum=0.5*real(cJ(phi)'*O(cJ(n)));
%#Uanal=((1/sigma1+1/sigma2)/2-sqrt(2)/sqrt(sigma1^2+sigma2^2))/sqrt(pi);
%#fprintf('Numeric, analytic Coulomb energy: %20.16f,%20.16f\n',Unum,Uanal);
```

D "O.m"

```
% Overlap operator (acting on 3d data sets)
%
% Usage: out=O(in)
%
% in: input 3d data set
% out: output 3d data set
%
% Uses GLOBAL variable(s) ---
% gbl_R: Lattice vectors

function out=O(in)
  global gbl_R; %# Must declare all globals with such statements to access them

  %# Operator definition (multiplication by volume)
  out= ... %# <=== YOUR CODE HERE
endfunction
```

E “fft3.m”

```
%# out=fft3(dat,N,s) - computes 3d fft (dimensions in N) of sign s,
%# out(l,m,n)=sum_{a,b,c} exp(2 pi s i *(a*l/N(1)+b*m/N(2)+c*n/N(3))*in(a,b,c).
%# Notes: 1) fortran/matlab ordering assumed, ordering in mem is out(1,1,1),
%#           out(2,1,1), ..., out(N(1),1,1), out(1,2,1), ...
%#        2) a=fft3(b,N,1) => b=fft3(a,N,-1)/prod(N); ie., fft(dat,N,1) and
%#           fft(dat,N,-1) are inverses except for normalization by
%#           N(1)*N(2)*N(3)
function out=fft3(dat,N,s)
%  tic;
  if s==1
    out=reshape(ifftn(reshape(dat,N(1),N(2),N(3)))*prod(N),size(dat));
  else
    out=reshape(fftn(reshape(dat,N(1),N(2),N(3))),size(dat));
  end

%  Ntot=prod(size(dat));
%  MFLOPS=5*Ntot*log2(Ntot)/1e6/toc
endfunction
```