Cornell University

Department of Physics

Phys 480/680, Astro 690 May 4, 2005

Computational Physics, Spring 2005

Homework Assignment # 10

(Due Thursday, May 5)

Agenda and readings for the week of May 2-5:

Goal: DFT calculations of solids!!!

Readings marked NR are from *Numerical Recipes: The Art of Scientific Computing*, 2^{nd} edition (in C). Readings marked LN are from the course lecture notes to be found at http://www.ccmr.cornell.edu/~muchomas/P480.

Recap of last week of class:

- 04/26 Lec 24: Further optimizations for spectral methods; *ab initio* pseudopotentials
- 04/28 Lab 9: Performance of numerical minimization techniques (from Lab 8); preconditioner for quantum calculations; quantum states of harmonic oscillator (Lab 7), physics of hydrogen molecule (Lab 9).

Preview of final week of class:

- 05/03 Lec 26: Variational principle for general eigenvalue problems in $d > 1$ dimension; extraction of eigenstates from variational principle
- 05/05 Lec 27: DFT++ expressions for general case of variable fillings
  Reading: Secs. 1–4 of "New algebraic formulation of density functional calculation," by Sohrab Ismail-Beigi and T.A. Arias, Computer Physics Communications 128:1-2, 1-45 (June 2000). Preprint: http://arxiv.org/abs/cond-mat/9909130.
- 05/05 Lab 10: Physics of quantum dot (Lab 8), electronic structure of crystalline germanium solid (Lab 10).

Contents

1 Minimal, isotropic spectral representation **3**
- 1.1 "setup.m" . . . 3
  - 1.1.1 Radius of inscribed sphere . . . 3
  - 1.1.2 Determining and communicating active basis functions . . . 3
  - 1.1.3 Debugging . . . 4
- 1.2 Operators . . . 4
  - 1.2.1 "L.m" and "K.m" . . . 4
  - 1.2.2 "cI.m" . . . 4
  - 1.2.3 "cIdag.m" . . . 5
  - 1.2.4 Debugging . . . 5
- 1.3 Full benefit from minimal representation . . . 5
  - 1.3.1 getn(), getE(), H() . . . 5
  - 1.3.2 H() . . . 6

2 Variable fillings **7**
- 2.1 getn() . . . 7
- 2.2 getE() . . . 7
- 2.3 Q() . . . 7
- 2.4 getgrad() . . . 8
  - 2.4.1 Debugging . . . 8

3 Pseudopotential for Ge **9**
- 3.1 Electronic structure of Ge . . . 9
- 3.2 Electronic structure of pseudo-Ge . . . 11

4 Three-dimensional pseudopotential calculation **12**
- 4.1 "Ge.m" . . . 12
  - 4.1.1 Debugging . . . 13
- 4.2 "poissonGe.m" . . . 13
  - 4.2.1 Debugging . . . 13
- 4.3 Final debugging . . . 13

5 Germanium Solid **14**
- 5.1 "setup.m" . . . 14
  - 5.1.1 Debugging: "poissonGeSolid.m" . . . 15
- 5.2 "GeSolid.m" . . . 15
  - 5.2.1 Debugging . . . 15

A "Q.m" **17**

B "pseudo.m" **18**

C "atoms8.m" **19**

D "ppm.m" **20**

E "smooth.m" **21**

1 Minimal, isotropic spectral representation

We will be making major modifications to your DFT software. At this point, you should create a new copy of all of your octave software in a new directory (called something like "Solids") in which to do your work for this assignment.

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

1.1 "setup.m"

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

1.1.1 Radius of inscribed sphere

To allow quick viewing and debugging, temporarily modify "setup.m" to work with a problem size of S=[4;6;8;]. Insert the following code block right after the computation of the lengths of the G vectors (where your code says "G2=..."),

```
%# Locate edges (assume S's are even!) and determine max 'ok' G2
  printf("Odd dimension in S, cannot continue!\n");
  return;
endif
```

This block just checks, so that we can assume that the sizes S are always even. After this check, insert the code block

```
edges=find( ... );
G2mx=min(G2(edges));
```

where "..." is a logical statement identifying the edges of the Fourier box.
Hints: Run "setup.m" with S=[4;6;8;] and inspect the basis indices in N. When the S's are even, the edges occur whenever any of N(:,k) equals either S/2 or -S/2+1. You can make a matrix of the values of S/2 of size equal to N by doing ones(size(N,1),1)*S'/2. In octave, the logical "or" operation is "|". Finally, if you have a matrix L of logical values and want to know if any element in each row is true, just use any(L,2). The final "2" here means to look for any true value along dimension 2.

1.1.2 Determining and communicating active basis functions

Once "setup.m" determines G2mx, have it determine the list of active basis functions and corresponding values of G2, and print out some debugging information with the statements

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

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

where "..." is an appropriate formula in terms of G2mx. Finally, to communicate the pertinent information to the rest of the software, declare gbl_active and gbl_G2c as global variables at the top of "setup.m" and be sure to set gbl_active and gbl_G2c to the values you find above.
Note: for the test case S=[4;6;8], there will be only one active element, active=[1].

1.1.3 Debugging

Set S=[32; 32; 32] and R=diag([8 8 8]) in "setup.m" and run "setup.m". You should find a basis set compression ratio of about 18.3, a little larger than the theoretical value. (We still have relatively few points so that the number of points actually falling within a sphere isn't exactly the volume of the sphere.)

1.2 Operators

For debugging purposes, you should now run your software for DFT calculations in a simple-harmonic oscillator potential ("dft.m") with the parameters S=[32; 32; 32] and R=diag([8 8 8]). For convenience, you may wish to comment out the parts of "dft.m" generating the plots, as we will not be looking at them. (Do keep the call to getPsi(), however.) Take note of your final converged values, as we will be checking that your reduced basis calculation gives essentially the same result. (I get E=46.594, epsilon=[6.320;7.742;7.742;7.742].)

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

1.2.1 "L.m" and "K.m"

Both L() and K() are diagonal operators based on the values of G2. We thus need only check whether the input has a size corresponding to G2 or G2c and to then use the values from the corresponding array. This can be accomplished by adding the statement "global gbl_G2c" to the top of each of the above functions and employing statements along the lines of "if size(in,1)==length(gbl_G2c)" with copies of your code with G2 replaced by G2c. (Be sure to keep your code functioning for the case "size(in,1)==length(gbl_G2)" as well!)
Note: Here and below, when we say "along the lines of", the names of some variables (especially internal working variables of a function) in the example may not correspond exactly to those in your routine.

1.2.2 "cI.m"

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

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

Finally, to perform the mapping $\mathbf{B}$ when needed, you should use a code block along the lines of

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

1.2.3 "cIdag.m"

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

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

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

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

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

1.2.4 Debugging

Modify your "dft.m" routine to repeat the simple-harmonic oscillator calculation with your minimal spectral representation, and verify that you reproduce your previous results to within 0.001 hartree!
Hints: The modification to "dft.m" should involve only changing the initialization of W to that of a random array of size length(gbl_active)×Ns.
Hint: For debugging the above operators (most bugs will involve various "nonconformant" errors), make liberal use the size() function to print out the sizes of the relevant objects from within the functions that generate errors.

1.3 Full benefit from minimal representation

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

1.3.1 getn(), getE(), H()

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

Input:

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

Output:

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

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

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

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

Debugging: Rerun "dft.m" to convince yourself that the output is identical. (You need not rerun the entire calculation; identical results through the end of the finite-difference test should be sufficient.)

1.3.2 H()

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

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

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

Debugging: Again, rerun "dft.m" to convince yourself that the output is unchanged.

2 Variable fillings

An important test case of the pseudopotential software which we shall develop is the case of an isolated atom of Ge. The valence electrons of Ge have the configuration $4s^2 4p^2$. The $4p^2$ means that the three p states share two electrons equally. This means that the fillings vector $\vec{f}$ should be f=[2;2/3;2/3;2/3]. Our current software, however, *assumes* constant fillings, f=[2;2;2;...]. We must modify the current DFT software to include the appropriate general expressions.

2.1 getn()

`function n=getn(psi,f)`

Input:

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

Output:

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

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

2.2 getE()

`function E=getE(W)`

Input:

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

Global variables:

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

Output:

- E: Energies summed over N_s states

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

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

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

2.3 Q()

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

2.4 getgrad()

`function grad=getgrad(W)`

Input:

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

Output:

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

Global variable:

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

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

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

where $N \equiv W^\dagger O(W)$, $\tilde{H} \equiv N^{-\frac{1}{2}}\left(W^\dagger H(W)\right) N^{-\frac{1}{2}}$, $F \equiv \mathbf{Diag}\,\vec{f}$, and $Q_N(\mathbf{A})$ is the operator which the function in Appendix A computes.[1] Note that the parentheses have been arranged here again to minimize the size and cost of the calculation of intermediate expressions and that $N^{-\frac{1}{2}}$ may be computed in octave as sqrtm(inv(N)).

2.4.1 Debugging

First, for basic debugging, delete the statement "gbl_f=2;" from "dft.m" and add the statement

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

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

Finally, for a non-trivial test, replace the above specification for gbl_f with

```
gbl_f=[2;2/3;2/3;2/3]; %# Variable filling test
```

Your code will now produce different output, but should pass the finite-difference test. Running to convergence, you should find a lower total energy (There are fewer electrons and less mutual repulsion.), but still a degeneracy pattern consistent with one s state and three p states. I find E=18.380, epsilon=[4.683; 6.339; 6.339; 6.339].

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

3 Pseudopotential for Ge

This problem explores the suitability of

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

as a pseudopotential for germanium (Ge) when $Z_v \equiv 4$, $r_c \equiv 1.052$ bohr, and $\lambda \equiv 18.5$ bohr^{-1}. This will not involve much development of new code but mostly quickly modifying ("hacking") your atomic solver in "broyden.c"

3.1 Electronic structure of Ge

The electronic configuration of the Ge atom ($Z = 32$) is $[1s^2 2s^2 2p^6 3s^2 3p^6 3d^{10}]4s^2 4p^2$. Verify that your atomic solver functions properly by running it with Itmx=20, N=4000, and the following specification for the above electronic configuration.

```
 /* Specs for Ge */
Z=32.;
lmax=2;

nmax=ivector(0,lmax);
nmax[0]=3;
nmax[1]=2;
nmax[2]=0;

nmaxmax=0;
for (l=0; l<=lmax; l++)
  if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

F=dmatrix(0,lmax,0,nmaxmax);
F[0][0]=2.;
F[0][1]=2.;
F[0][2]=2.;
F[0][3]=2.;
F[1][0]=6.;
F[1][1]=6.;
F[1][2]=2.;
F[2][0]=10.;
```

Confirm your results by comparing with the table at the NIST web site.

To ease the change from the true potential to the pseudopotential, add the following function to the top of the file containing your function getg(),

```
double Vnuc(double r, double Z)
{
  if (r==0.)
    return 0.;
  else
    return -Z/r;
}
```

and replace all occurrences along the lines of `-Z/r[k]` in your code with `Vnuc(r[k],Z)`.
Hint: There is likely only one occurrence, the computation of the total self-consistent potential V[] in getg(). Rerun "broyden.c" and verify that the results are unchanged.

More detailed information than simply the total energy is needed to verify the pseudopotential. Modify your code to provide this information as follows:

1. Change the function prototype of getg() to include an additional argument "flag" that we will use to get an optional detailed report of the electronic structure. The new prototype will be

```
double getg(double Rhoerr[], double Rho[],
 double Z, int lmax, int nmax[], int nmaxmax, double **F,
 double r[], double dr[], int N,int flag)
```

2. Add to getg() the following code block that produces a detailed report of the electronic structure only if flag>0. You should add this block after all of the computations but before the freeing of the dvectors and various other objects.

```
  /* Get detailed report if flag>0 */
  if (flag>0) {
    char str[6]="spdfg"; /* Strings for angular momentum states */
    FILE *fp;

    /* Output various total energy contributions to stdout */
    printf("Etot : %12.6f\n",Etot);
    for (k=0; k<=N; k++)
      integrand[k]=(phi[k]/2+Vnuc(r[k],Z)+Depsxc[k]+Vxc[k])*Rho[k];
    printf("Ekin : %12.6f\n",Etot-simpint(integrand,r,dr,N));
    for (k=0; k<=N; k++)
      integrand[k]=phi[k]*Rho[k]/2;
    printf("Ecoul: %12.6f\n",simpint(integrand,r,dr,N));
    for (k=0; k<=N; k++)
      integrand[k]=Vnuc(r[k],Z)*Rho[k];
    printf("Eenuc: %12.6f\n",simpint(integrand,r,dr,N));
    for (k=0; k<=N; k++)
      integrand[k]=(Depsxc[k]+Vxc[k])*Rho[k];
    printf("Exc  : %12.6f\n",simpint(integrand,r,dr,N));

    /* Output state energies to stdout */
    for (l=0; l<=lmax; l++)
      for (n=0; n<=nmax[l]; n++)
printf("%d%c[%.0f]   : %12.6f\n",n+l+1,str[l],F[l][n],E[l][n]);

    /* Output wave functions to file "psiout" */
    /*   First number is r, the rest are the wave functions in the same */
    /*   order as the states above come out. */
    if ( (fp=fopen("psis","w"))==NULL ) {
      printf("\nError: Cannot open file \"psis\" for output!\n\n");
      exit(1);
    }
    for (k=0; k<N; k++) {
      fprintf(fp,"%le  ",r[k]);
      for (l=0; l<=lmax; l++)
for (n=0; n<=nmax[l]; n++)
  fprintf(fp," %le",Psi[l][n][k]);
      fprintf(fp,"\n");
    }
```

```
  fclose(fp);
}
```

3. To every current call to getg() in main(), add an additional argument with value "0", so that your current code is unaffected by these changes.

4. Add a final call to getg() to get the detailed report. This call will be along the lines of

   ```
   getg(g[Itmx+1],Rho[Itmx+1],Z,lmax,nmax,nmaxmax,F,r,dr,N,1);
   ```

   It should use the most updated version of Rho[], and be placed in main() after all of the computations, outside of all of the loops, and just before the freeing of the various data objects.

Rerun "broyden.c" with these modifications, and rename the file containing the output wave functions ("psis") to "psis_real" with a statement along the lines of `cp psis psis_real`. To view a plot of these functions, run the octave file "pseudo.m" from Appendix B. This program will crash looking for "psis_pseudo", but will first provide a plot of all of the electronic wave functions of Ge.

3.2 Electronic structure of pseudo-Ge

To test the results of the pseudopotential, modify Vnuc() to return the value of Eq. (1), and run "broyden.c" with the electronic configuration

```
 /* Specs for pseudo-Ge */
Z=4.;
lmax=1;

nmax=ivector(0,lmax);
nmax[0]=0;
nmax[1]=0;

nmaxmax=0;
for (l=0; l<=lmax; l++)
  if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

F=dmatrix(0,lmax,0,nmaxmax);
F[0][0]=2.;
F[1][0]=2.;
```

This configuration corresponds to a core (nucleus plus core electrons) with a net charge of four, and 2 valence electrons in the lowest s state ("1s") and two in the lowest p state ("2p").
Hint: To handle the case $r = 0$, you may wish to use $\lim_{r\to 0} V_{ps}(r) = -\frac{Z_v \lambda}{1+e^{\lambda r_c}}$.

To verify the suitability of the pseudopotential, compare the pseudo 1s eigenvalue with the all-electron 4s eigenvalue and the pseudo 2p with the all-electron 4p eigenvalues. Compare the corresponding pseudo- and all-electron wave functions by copying the output file "psis" to "psis_pseudo" and running "pseudo.m" from within octave. Finally, take note of the pseudo total energy and the difference between the pseudo 1s and pseudo 2p energies. The next problem will compare these last two quantities with the output of the $d = 3$ dimensional code.

4 Three-dimensional pseudopotential calculation

As an important test of our implementation of pseudopotentials in the three-dimensional code, we shall perform a calculation of a single atom using that code and then compare the results directly with the the previous problem.

4.1 "Ge.m"

Begin by making a new copy of "dft.m" called "Ge.m" and confirming that "setup.m" is currently set to run with a problem size of S=[32; 32; 32]; R=diag([8 8 8]). Because "dft.m" was already running with fillings f=[2; 2/3; 2/3; 2/3], the only change needed to calculate Ge is to replace the simple harmonic oscillator potential with the pseudopotential.

The final form used for the potential in the code is the vector of dual coefficients, Vdual=cJdag(O(cJ(V))), where V is the vector of sample values of the potential. For the pseudopotential, our strategy will be to compute the expansion coefficients $\hat{V}_{ps}$=cJ(V) directly by analytically Fourier transforming the pseudopotential in Eq. (1).

Keeping in mind that we have a periodic copy of the atom at all positions $\mathbf{R}\vec{m}$, where $\vec{m}$ is any vector of three integers, the actual total pseudopotential is $\sum_{\vec{m}} V_{ps}(\vec{r}-\mathbf{R}\vec{m})$. Thus, the relation between the expansion coefficients Vps and the pseudopotential is

$$\sum_{\vec{G}} (\hat{V}_{ps})_{\vec{G}}\, e^{i\vec{G}\cdot\vec{r}} = \sum_{\vec{m}} V_{ps}(\vec{r}-\mathbf{R}\vec{m}),$$

which, upon multiplication of both sides by $e^{-i\vec{G}\cdot\vec{r}}$ and integration over the supercell, gives

$$\begin{aligned}(\hat{V}_{ps})_{\vec{G}} &= \frac{1}{\det \mathbf{R}} \sum_{\vec{m}} \int_{cell} e^{-i\vec{G}\cdot(\vec{r}-\mathbf{R}\vec{m})} V_{ps}(\vec{r}-\mathbf{R}\vec{m})\, d^3r \\ &= \frac{1}{\det \mathbf{R}} \int_{\text{all space}} e^{-i\vec{G}\cdot\vec{x}} V_{ps}(\vec{x}), \end{aligned} \tag{2}$$

where on the first line we use the fact that always $e^{i\vec{G}\cdot\mathbf{R}\vec{m}} = 1$ and, in going to the second line, we note that the range of points covered by all $\vec{r}-\mathbf{R}\vec{m}$, where $\vec{r}$ ranges over one supercell, is in fact all of space. Finally, the integral over all of space can be done almost analytically. Arias's (unfamous) result for this integral is in terms of a rapidly convergent asymptotic series (four terms suffice to give 26 digits!),

$$\begin{aligned}\int e^{-i\vec{G}\cdot\vec{x}} V_{ps}(\vec{x}) &= -\frac{4\pi Z_v}{G^2} + \frac{4\pi Z_v(1+e^{-\lambda r_c})}{G^2} \left(-\frac{2\pi e^{-\pi G/\lambda} \cos(G r_c)\frac{G}{\lambda}}{1-e^{-2\pi G/\lambda}} \right. \\ &\qquad \left. + \sum_{n=0}^{\infty} (-1)^n \frac{e^{-\lambda r_c n}}{1+\left(\frac{n\lambda}{G}\right)^2} \right) \end{aligned} \tag{3}$$

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

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

Combining Eqs. (2-4) gives $\hat{V}_{ps}$, from which we then get the final result needed by the code Vdual=cJdag(O($\hat{V}_{ps}$)). To avoid needless heartache, feel free to cut and past the octave form for the resulting formula,

```
%# Pseudopotential
Z=4;
lambda=18.5;
```

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

n=[1:4];
Vps(1)=4*pi*Z*(1+exp(-lambda*rc))*(rc^2/2+1/lambda^2* ...
                     (pi^2/6+sum((-1).^n.*exp(-lambda*rc*n)./n.^2)));
Vps=Vps/det(R);
Vdual=cJdag(O(Vps)); %# Dual coefficients
gbl_Vdual=Vdual; %# Export value of global variable
```

4.1.1 Debugging

After switching from the simple harmonic oscillator potential to the pseudopotential for Ge, run "Ge.m" and taking note of the final total energy and also the final eigenvalues stored in the variable epsilon. You should find a degeneracy pattern consistent with one s and three p states. You should find about 20% agreement with the s-p energy difference from the atomic code. You cannot compare the total energy yet (you should have a value of about -0.873 hartree), however, because you do not yet have the nuclear-nuclear contribution, which we will find in the next part.

4.2 "poissonGe.m"

Make a copy of "poisson2.m" called "poissonGe.m". In "poissonGe.m", remove all references to the second Gaussian at (diag(R)'/2+[x,0,0]), keeping all terms pertaining to the first Gaussian in all mathematical expressions. Be sure to keep the printouts checking the normalization of the Gaussian and the total charge in the cell, and to keep accounting for the self-energy part for the remaining Gaussian. Finally, change the norm on the remaining Gaussian at (diag(R)'/2) so that it represents a net charge of four instead of one.

4.2.1 Debugging

Run "poissonGe.m" and add the resulting nuclear-nuclear contribution to the energy from Problem 4.1.1. You should find a total energy for the Ge atom in agreement with your atomic code to about 1%!

4.3 Final debugging

Repeat the above calculations in a larger cell, R=diag([12 12 12]), with a corresponding larger number of sampling points, S=[48; 48; 48]. You should find significantly improved agreement for both the total energy and the s-p energy difference.
Hint: Don't forget to rerun "poissonGe.m" for the larger cell as well!

5 Germanium Solid

All that remains for calculations in solids is to add capability for multiple atoms to both "Ge.m" and "poissonGe.m" and to specify the locations of the atoms in the crystal.

The strategy for including multiple atoms which is most easily implemented from a coding point of view is the use of a "structure factor". The basic idea is that if

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

is a potential or a density function (such as $V_ps(\vec{x})$ in "Ge.m" or $n(\vec{x})$ in "poissonGe.m"), then if we shift the center of $f(\vec{x})$ to a new point $\vec{x}_0$, it's expansion will change to

$$\begin{aligned} f(\vec{x}-\vec{x}_0) &= \sum_{\alpha} \hat{f}_{\alpha} e^{i\vec{G}_{\alpha}\cdot(\vec{x}-\vec{x}_0)} \\ &= \sum_{\alpha} \left(\hat{f}_{\alpha} e^{-i\vec{G}_{\alpha}\cdot\vec{x}_0}\right) e^{i\vec{G}_{\alpha}\cdot\vec{x}}. \end{aligned}$$

Thus, to move the center of the function $f(\vec{x})$ over to the point $\vec{x}_0$, we need only multiply the expansion coefficients $\hat{f}_{\alpha}$ by the phase factors $e^{-i\vec{G}_{\alpha}\cdot\vec{x}_0}$. If we have several copies of $f(\vec{x})$ at different locations $\vec{x}_i$ and wish to add them together, such as to compute the total pseudopotential $V_{ps}(\vec{x})$ or nuclear density $n(\vec{x})$ from multiple atoms, we thus multiply the corresponding expansion coefficients by the sum of such factors, the so called "structure factor,"

$$S_f(\vec{G}) \equiv \sum_{i} e^{-i\vec{G}_{\alpha}\cdot\vec{x}_i}. \tag{5}$$

Note that, in octave, if X is a matrix, each of whose columns i contains the location $\vec{x}_i$ of one of the atoms, then the above dot products can all be evaluated in the single BLAS operation G*X, where G is our matrix of $\vec{G}_{\alpha}$'s.

5.1 "setup.m"

Make the following changes to "setup.m" to import the atomic locations and periodic cell size for solid Ge and to compute the structure factor.

1. Copy "atoms8.m" from Appendix C into your current working directory. This program loads the locations of the atoms in crystalline Ge into the array X and the appropriate periodic boundary conditions into the array R. The basic supercell we shall use for the crystal is of cubic shape and contains 8 atoms.

2. Place the statement

   ```
   atoms8;
   ```

   near the top of "setup.m", immediately after where you set S and R but before evaluation of any quantities such as N, M, r, G, etc. This will load the atomic locations and give the appropriate value to R to the result of "setup.m".

3. At the very end of "setup.m", place a statement of the form

   ```
   Sf= ... G*X ...
   ```

   which will place the structure factors into the column vector Sf according to the definition in Eq. (5). Note that we do not need to export Sf as a global variable as the programs "poissonGe.m" and "Ge.m" can access it directly.
   Hint: In octave, for any matrix A, analogously to our previous hint regarding the octave statement "any(...,2)", "sum(A,2)" produces a column vector equal to the sum of all elements in each row of A.

5.1.1 Debugging: "poissonGeSolid.m"

Make a copy of "poissonGe.m" called "poissonGeSolid.m" in which the expansion coefficients of the Gaussian (obtained by applying cJ() to the sample values) are, value by value, multiplied by Sf before being used in any future expressions. This makes copies of the Gaussian representing the nuclear charge at all of the atomic locations listed in X. Keeping the initialization S=[48; 48; 48], run "poissonGeSolid.m". This, then, gives the nuclear-nuclear interaction energy for the germanium crystal. I find a value of about $V_{nuc-nuc} = -31.904$ hartree. Finally, make note of the value you get for S=[64; 64; 64], which should be accurate to about 10^{-6} hartree.

5.2 "GeSolid.m"

Make a copy of "Ge.m" called "GeSolid.m", and modify it as follows to carry out a calculation of solid germanium.

1. Immediately after the evaluation of the expansion coefficients Vps is complete and before they are used in any expressions, multiply Vps, value by value, by Sf. This makes the total potential equal to the sum of contributions for all of the atoms listed in X.

2. Replace the statement "Ns=4;" with a statement giving the appropriate number of states.
   Hints: The number of atoms is size(X,2), there are four electrons per atom, and two electrons per state.

3. Make certain that the fillings are set to a column vector of length Ns filled with all 2's.

4. To display two-dimensional slices through the final charge density, add the following statements to the very end of "GeSolid.m",

```
%# Get density
n=getn(Psi,gbl_f);

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

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

   and copy the utility files "ppm.m" and "smooth.m" from Appendices D-E into your current directory.

5.2.1 Debugging

Run "GeSolid.m" with S=[48; 48; 48]. (This will take about 1/2 hour!) The code should produce (and will try to display) two graphics, "100.ppm" and "110.ppm". If the display doesn't work on your computer, you can try to view the images with some other application.

The graphics show a brightness proportional to the total charge density at each point in a plane cutting through the solid. The sky-blue graphic ("100.ppm") shows a slice across the top of the cubic cell. The atoms should appear as rings of round charge distributions with low density (dark regions) centered on the nuclei. The reason why there are no electrons on the nuclei is that we are using a pseudopotential and are not computing the core electrons. You should see one atom in the center of the 100 slice and four atoms cutoff at the corners. This reflects the basic faced-centered cubic (fcc) structure of the germanium crystal. The

110 graphic is more exciting. This cut slices through the centers of neighboring atoms, showing significant buildup of charge right in between them. These are the *bonds* which hold the crystal together!!!

Finally, adding your final energy from "GeSolid.m" to the 8 atom result of "poissonGeSolid.m" and subtracting 8 times the energy that you got for an isolated pseudoatom (calculated with the same S and R is best) gives your result for the binding energy of an 8 atom cell of germanium. (Your result should be negative: because the crystal is stable, it's energy should be lower than that of the same number of isolated atoms.) Divide this number by 8, convert units (1 hartree=27.21 eV), and compare to the *experimental* cohesive energy of germanium of 3.85 eV/atom!

A "Q.m"

```
function out=Q(A,N)
  [V,mu]=eig(N); mu=diag(mu);

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

B "pseudo.m"

```
load psis_real
psis=psis_real;

%# Plot all-electron atom wave functions
figure(0);
axis([0 5]);

r=psis(:,1);
plot(r,psis(:,2),';1s;',r,psis(:,3),';2s;',r,psis(:,4),';3s;', ...
     r,psis(:,5),';4s;',r,psis(:,6),';2p;',r,psis(:,7),';3p;', ...
     r,psis(:,8),';4p;',r,psis(:,9),';3d;');

%# Compare pseudo and all-electron wave functions
figure(1);
load psis_pseudo
plot(r,psis(:,5),'r-;4s;',r,psis(:,8),'g-;4p;', ...
     r,psis_pseudo(:,2),'rx;1s(ps);',r,psis_pseudo(:,3),'gx;2p(ps);');
```

C "atoms8.m"

```
%# Ge cubic lattice constant (first in Angstrom, converted to atomic units)
a=5.66/0.52917721;

%# Standard cubic cell crystalline coordinates (as columns) for Ge
%#   ("diamond" lattice)
dl=[0.00 0.25 0.00 0.25 0.50 0.75 0.50 0.75
    0.00 0.25 0.50 0.75 0.00 0.25 0.50 0.75
    0.00 0.25 0.50 0.75 0.50 0.75 0.00 0.25];

%# Scale atom locations to actual cell size and form lattice vectors for cell
X=dl*a;
R=diag([a,a,a]);
```

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 if less than 200x200
  while(size(red,2)<200)
    red=smooth(red); green=smooth(green); blue=smooth(blue);
  end

  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);
end
```

E "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
```