DFT++ Minicourse

Tomás A. Arias
Department of Physics, Cornell University

June 27, 2005

This material is based upon work supported by the National Science Foundation under Grant No. 0113670. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

Part I

Notes

Chapter 1

Notation

1.1 General

1. "$\vec{\ }$", "$\hat{\ }$": all vectors are taken to be column vectors, with "$\vec{\ }$" denoting either a generic vector or a collection of real-space sample values and "$\hat{\ }$" denoting a collection of expansion coefficients
2. **X**: matrices are often (but not always) denoted with capital letters and **bold face**
3. "[. . .]": formation of a matrix from elements or sub-blocks
4. "[**A**; **B**]": vertical placement of sub-blocks (or elements) on top of each other; *e.g.*, $[a;b] \equiv \begin{bmatrix} a \\ b \end{bmatrix}$, a column vector.
5. "[**A**, **B**]": horizontal placement of sub-blocks or elements on top of each other; *e.g.*, $[a,b] \equiv [ab]$, a row vector
6. diag : produce a (column) vector from the diagonal elements of a matrix; *e.g.,* $\vec{a} = \mathrm{diag}\,\mathbf{A}$ implies $a_i = A_{ii}$
7. Diag : produce a diagonal matrix from a (column) vector; *e.g.,* $\mathbf{A} = \mathrm{Diag}\,\vec{a}$ implies $A_{ij} = a_i\delta_{ij}$

1.2 DFT++

1. $b_\alpha(\vec{x})$: α^{th} basis function
2. $\vec{x}_i$: i^{th} sample point
3. **O**: overlap matrix, $O_{\beta,\alpha} \equiv \int b_\beta(\vec{x})^* b_\alpha(\vec{x})\,dV$, coded as function `O()`
4. **L**: Laplacian matrix, $L_{\beta,\alpha} \equiv \int b_\beta(\vec{x})^* \nabla^2 b_\alpha(\vec{x})\,dV$, coded as function `L()`
5. $\mathbf{L}^{-1}$: Inverse of Laplacian matrix, $L_{\beta,\alpha} \equiv \int b_\beta(\vec{x})^* \nabla^2 b_\alpha(\vec{x})\,dV$, coded as function `Linv()`
6. **I**: sample-value or "forward transform" matrix, $I_{i,\alpha} \equiv b_\alpha(x_i)$, coded as function `cI()`
7. $\mathbf{I}^d agger$: Hermitian conjugate of sample-value matrix, coded as function `cIdag()`
8. **J**: "inverse transform" matrix, $cJ \equiv cI^{-1}$, coded as function `cJ()`
9. $\mathbf{J}^\dagger$: Hermitian conjugate of inverse transform matrix, coded as function `cJdag()`

1.3 Plane waves

1. $\mathbf{R}$: 3×3 matrix, each of whose columns is a lattice vector $\vec{R}_k$, $\mathbf{R} \equiv [\vec{R}_1, \vec{R}_2, \vec{R}_3]$
2. $\vec{S}$: 3×1 vector containing the number of sample points along each lattice vector
3. M: $\prod_k S_k \times 3$ matrix, each of whose rows contains the indices of a three-dimensional array, in standard Fortran order (first index fastest); *e.g.,* $M = [0,0,0;1,0,0;\ldots S_1 - 1,0,0;0,1,0;\ldots]$
4. N: $\prod_k S_k \times 3$ matrix, each of whose rows contains the indices of a three-dimensional array, in standard Fortran order and with standard Fourier aliasing; *e.g.,* $N = [0,0,0;1,0,0;\ldots -1,0,0;0,1,0;\ldots]$
5. r: $\prod_k S_k \times 3$ matrix, each of whose rows contains the real-space coordinates of a sample point in standard order, $\mathbf{r} \equiv \mathbf{M}\left(\mathrm{Diag}\,\vec{S}\right)^{-1}\mathbf{R}^T$
6. G: $\prod_k S_k \times 3$ matrix, each of whose rows contains the components of a reciprocal lattice vector in standard order, $\mathbf{G} \equiv 2\pi\mathbf{N}\mathbf{R}^{-1}$
7. $\vec{G}^2$: $\prod_k S_k \times 1$ vector, each of whose elements is the square magnitude of the vector in the corresponding row of G.
8. $\mathbf{O}$: overlap matrix, $\mathbf{O} \equiv \det(\mathbf{R})\mathbf{1}$, where $\mathbf{1}$ is the identity
9. $\mathbf{L}$: Laplacian matrix, $\mathbf{L} \equiv -\det(\mathbf{R})\mathrm{Diag}\,\vec{G}^2$
10. $\mathbf{L}^{-1}$: Inverse Laplacian matrix, $\mathbf{L}^{-1} \equiv -\frac{1}{\det(\mathbf{R})}\left(\mathrm{Diag}\,\vec{G}^2\right)^{-1}$ with the specification that $[L^{-1}]_{1,1} \equiv 0$, where $[\ldots]_{1,1}$ indicates the upper- and left- most element.
11. $\mathbf{K}$: Preconditioner, $\mathbf{K} \equiv \left(\mathrm{Diag}\left(\vec{1} + \vec{G}^2\right)\right)^{-1}$, where $\vec{1}$ is defined as a column vector, each of whose elements is unity
12. $\mathbf{F}_3$: Standard three-dimensional discrete Fourier transform kernel (plus sign in the exponent, no normalization factor)
13. $\mathbf{F}_3^\dagger$: Conjugated three-dimensional discrete Fourier transform kernel (minus sign in the exponent, no normalization factor)
14. $\mathbf{I}$: Forward transform, $\mathbf{I} \equiv \mathbf{F}_{(3)}$
15. $\mathbf{I}^\dagger$: Forward transform, $\mathbf{I}^\dagger \equiv \mathbf{F}_{(3)}^\dagger$
16. $\mathbf{J}$: Inverse transform, $\mathbf{J} \equiv \frac{1}{\prod_k S_k}\mathbf{F}_{(3)}^\dagger$
17. $\mathbf{J}^\dagger$: Conjugate inverse transform, $\mathbf{J}^\dagger \equiv \frac{1}{\prod_k S_k}\mathbf{F}_{(3)}$
18. $\mathbf{B}$: Injection from cut-off sphere to Fourier box, $B_{\alpha,s} \equiv \delta_{\alpha,s}$, where α is the index of any G vector in the Fourier box and s any vector within the cut-off sphere
19. $\mathbf{B}^\dagger$: Projection from cut-off sphere to Fourier box, $B_{s,alpha} \equiv \delta_{\alpha,s}$, where α is the index of any G vector in the Fourier box and s any vector within the cut-off sphere
20. $\mathbf{X}$: atomic locations, $N_{at} \times 3$ matrix, each of whose rows contains the real-space coordinates of an atom (in bohrs)
21. $\vec{S}_f$: $\prod_k S_k \times 1$ column vector containing the structure factor associated with the wave vector of the corresponding column of $\mathbf{G}$.

Chapter 2

DFT++ Expressions

2.1 Poisson's equation

1. Problem: $\nabla^2\phi(\vec{x}) = -4\pi n(\vec{x})$ within periodic boundary conditions
2. $\vec{n}$: sample values of density $n(\vec{x})$, $n_i \equiv n(\vec{x}_i)$
3. $\vec{\phi}$: sample values of the potential $\phi(\vec{x}_i)$
4. $\hat{\phi}$: expansion coefficients of the potential, $\phi(\vec{x}) \equiv \sum_\alpha \hat{\phi}_\alpha b_\alpha(\vec{x})$, $\vec{\phi} = \mathbf{I}\hat{\phi}$
5. $\hat{n}$: expansion coefficients of the density, $n(\vec{x}) \equiv \sum_\alpha \hat{n}_\alpha b_\alpha(\vec{x})$, $\hat{n} = \mathbf{J}\vec{n}$
6. Galerkin representation: $\mathbf{L}\hat{\phi} = -4\pi\mathbf{O}\hat{n}$
7. Solution: $\vec{\phi} = \mathbf{I}\mathbf{L}^{-1}\left(-4\pi\mathbf{O}\mathbf{J}\vec{n}\right)$
8. U: total electrostatic energy, $U \equiv \frac{1}{2}\int n(\vec{x})^*\phi(\vec{x})\,dV = (-2\phi)\hat{n}^\dagger\mathbf{O}\hat{\phi} = \vec{n}\mathbf{J}^\dagger\mathbf{O}\mathbf{L}^{-1}\mathbf{O}\mathbf{J}\vec{n}$

2.2 Schrödinger and Kohn-Sham equations

1. Problem: Under the constraints
$$\int \psi_k^*(\vec{x})\psi_{k'}(\vec{x})\,dV \equiv \delta_{k,k'},$$
minimize
$$E \equiv U_{nn} + \int\left(-\frac{1}{2}f\sum_k \psi_k^*\nabla^2\psi_k + V(\vec{x})n(\vec{x}) + \frac{1}{2}n(\vec{x})\phi(\vec{x}) + n(\vec{x})\epsilon_{xc}(n(\vec{x}))\right)dV,$$
where f is the occupancy of each orbital (usually 2), U_{nn} is the Ewald energy, $V(\vec{x})$ is the potential from the nuclei/ionic cores and
$$n(\vec{x}) \equiv f\sum_k \psi_k^*(\vec{x})\psi_k(\vec{x}),$$
$$\nabla^2\phi(\vec{x}) = -4\phi n(\vec{x}).$$
Note: For the Schrödinger equation, set $f \equiv 1$ and drop all terms involving $\phi(\vec{x})$ and $\epsilon_{xc}(n(\vec{x}))$.
2. **W**: expansion coefficients of unconstrained wave functions arranged into a matrix, each of whose columns contains the expansion coefficients for a particular wave function, $w_k(\vec{x}) \equiv \sum_\alpha w_{\alpha,k}b_\alpha(\vec{x})$.

3. $\mathbf{U}$: matrix of wave function overlaps, $\mathbf{U} \equiv \mathbf{W}^\dagger \mathbf{O} \mathbf{W}$

4. $\mathbf{Y}$: expansion coefficients of orthonormalized wave functions arranged as in $\mathbf{W}$, $\mathbf{Y} \equiv \mathbf{W} \left(\mathbf{W}^\dagger \mathbf{O} \mathbf{W} \right)^{-\frac{1}{2}}$

5. Energy:

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

   where

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

   (For Schrödinger's equation, drop all terms with ϵ_{xc} or ϕ.)

6. Gradient: $\left[\nabla_W E \right]_{\alpha,n} \equiv \frac{\partial E}{\partial W^*_{\alpha,n}}$, so that $dE = 2 \, \mathrm{Re} \, \mathrm{Tr} \, dW^\dagger \nabla_W E$. Then,

$$\nabla_W E = f \left(H W - \mathbf{O} W U^{-1} W^\dagger H W \right) U^{-1},$$

   where

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

   with

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

7. $\mathbf{\Psi}$: expansion coefficients of orthonormalized eigenstates, $\mathbf{\Psi} \equiv \mathbf{Y} \mathbf{D}$, where $\mathbf{D}$ is a unitary matrix such that $\mathbf{D}^\dagger \left(\mathbf{Y}^\dagger \mathbf{H} \mathbf{Y} \right) \mathbf{D}$ is diagonal, where H is the same operator defined for the gradient expression above.

Chapter 3

Minimization Methods

1. Steepest descents
   (a) Initialize W
   (b) $W \leftarrow W - \alpha \nabla_W E$
   (c) Display E
   (d) Repeat (b & c) until converged
2. Line minimization
   (a) Initialize W_0
   (b) Evaluate gradient: $g = \nabla_W E(W_n)$
   (c) If not first iteration, do "linmin test": check the angle cosine $(g \cdot d)/\sqrt{(g \cdot g)(d \cdot d)}$, where d is the previous search direction
   (d) Set search direction: $d = -g$
   (e) Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d)$
   (f) Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - gt] \cdot d)$
   (g) $W_{n+1} = W_n + \alpha d$
   (h) Repeat (b-g)
3. Preconditioned line minimization
   (a) Initialize W_0
   (b) Evaluate gradient: $g = \nabla_W E(W_n)$
   (c) If not first iteration, do "linmin test": check $(g \cdot d)/\sqrt{(g \cdot g)(d \cdot d)}$ for d being the previous search direction
   (d) Set search direction: $d = -K(g)$
   (e) Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d)$
   (f) Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - gt] \cdot d)$
   (g) $W_{n+1} = W_n + \alpha d$
   (h) Repeat (b-7)
4. Preconditioned conjugate gradients
   (a) Initialize W_0

(b) Evaluate gradient: $g_n = \nabla_W E(W_n)$

(c) If not first iteration, do "linmin test": check $(g_n \cdot d_{n-1})/\sqrt{(g_n \cdot g_n)(d_{n-1} \cdot d_{n-1})}$ for d being the previous search direction

(d) If not first iteration, do "cg test": check $(g_n \cdot Kg_{n-1})/\sqrt{(g_n \cdot Kg_n)(g_{n-1} \cdot Kg_{n-1})}$ for d being the previous search direction

(e) Set search direction: $d_n = -Kg_n + \beta d_{n-1}$, where

- Fletcher-Reeves: $\beta = (g_n \cdot Kg_n)/(g_{n-1} \cdot Kg_{n-1})$

(f) Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d_n)$

(g) Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - g_t t] \cdot d_n)$

(h) $W_{n+1} = W_n + \alpha d_n$

(i) Repeat (b-h)

Chapter 4

Octave Quick Reference

- Program control
  - `help fname`: provide documentation of (built-in or user-provided) function `fname`
  - `more off`, `more on`: lets long output scroll past on screen, or pipes output through more, respectively
  - `format`, `format long`: displays output values in shortened or full precision format
  - `pause`: halts execution until user hits a key in the command window
  - "`;`": Command lines ending in "`;`" will produce no output, whereas lines ending without "`;`" will print the value of the last expression. "`;`" also serves as a separator for multiple statements on a single line
- Matrix construction
  - `[A; B]`, `[A, B]`: composition of a matrix from sub-blocks, as defined in Section 1.1
  - `[n1:n2:n3]`: produces a list of numbers (as a row vector) starting from `n1`, proceeding in strides of `n2`, and ending at `n3`
  - `zeros(n,m)`: produces an `n`×`m` matrix, all of whose elements are zero
  - `ones(n,m)`: produces an `n`×`m` matrix, all of whose elements are unity
  - `diag(x)`: if `x` is a vector, produces a diagonal matrix with the elements of `x` along the diagonal, if `x` is a matrix, produces a column vector consisting of the diagonal elements of `x`
  - "`x'`": Hermitian conjugate of `x`
- Matrix manipulation
  - `whos`: gives a list of all variables, their types and sizes
  - `sum(A)`: sum of columns of `A`, resulting in row vector, if `A` is a vector (row or column), gives total of all elements
  - `sum(A,d)`: sum of `A` along dimension `d`; for `d`=1 (`d`=2) sums all elements in each column (row) producing a row (column) vector
  - `prod()`: works just as `sum()` but takes product instead
  - `trace(A)`, `det(A)`, `inv(A)`, `sqrtm(A)`, `size(A)`: trace, determinant, inverse, matrix square-root, dimensions of `A`, respectively.
  - `floor(A)`, `sqrt(A)`, `sin(A)`, `real(A)`, `rem(A,m)`: returns a matrix each of whose elements are the greatest integer below, the square root, sin, real part, or remainder modulo m or the corresponding elements of `A`. Note that `sqrt(A)` is not generally the same as `sqrtm(A)`.

- `A.*B, A./B, A.^2`, *etc.*: element-wise operations, return a matrix, each of whose elements are the pairwise products or quotients of the corresponding elements of `A` and `B` (which must have the same size) or are the squares of the corresponding elements of `A`. Note that `sqrt(A).*sqrt(A)` gives `A`
- `*`: standard matrix product. Note that `sqrtm(A)*sqrtm(A)` gives `A`.
- `[U, epsilon]=eig(H)`: diagonalizes H, producing a diagonal matrix of eigenvalues (`epsilon`) and unitary matrix `U` such that `U'*H*U` gives `epsilon`.

- Logical and indexing functions
  - `>, <, <=, ==, !=`, *etc.*: standard logical statements evaluate to zero when false and unity when true
  - `any()`: a functional analogous to `sum()` and `prod` but which takes the logical "or" of rows or columns; *i.e.,* determines if "any" elements were true
  - `list=find(A)`: makes a list of the elements of `A` which are true (have non-zero value). Thus, `find(x<0)` will give a list of integers specifying the elements of `x` which are less than zero.
  - `negatives=x(list)`: makes a vector consisting of the elements of `x` with indices specified in `list`. Thus `x(find(x<0))` produces a list of negative numbers only.
  - `x(list)=zeros(size(list))`: fills those elements specified by `list` on the left-hand size with the values on the right-hand side.

- Programming
  - "...": line continuator, indicates that the next line should be interpreted as a continuation of the current line
  - "`.m`" files: files ending with "`.m`" which contain a list of statements will be executed as scripts from the command line (or within another script) when the file name (without the "`.m`" extension) is used as a statement
  - "`#`", "`%`": comment line
  - `function [out1,out2,out3,...]=fname(in1,in2,in3,...)...endfunction`: declares a function fname() which takes inputs in1,in2,in3,... and gives outputs out1,out2,out3... . Such declarations should be made in a separate file of the corresponding name; *e.g.,* "`fname.m`". The body of the function appears between the `function` and `endfunction` statements, and the function is called as in the declaration, [out1,out2,out3,...]=fname(in1,in2,in3,...); In the (usual) case of a single output, the shorthand out=fname(in1,in2,in3,...) may be used both to invoke and declare the function. The return value of a function with multiple outputs is the first in the list. Thus, if out=fname() is used to call a function with multiple outputs, out will take on the value of the first listed.
  - `global X`: declare `X` as a global variable. This declaration must appear *both* in the main program and in each function which uses the value of `X`. Otherwise, all variables (except input and output) are private to each function.

- Flow control
  - `if logical_statement ... elseif ... else ... endif`: if-then-else construct
  - `for n=N1:N2:N3 ... endfor`: for-loop construct, with n taking values [N1:N2:N3]

Chapter 5

Additional Resources

- Octave manual:
  www.octave.org/doc/octave_toc.html
- `C++` production code:
  abinitio.org.
- Parallelization, DFT++ formulas for other techniques, and additional details:
  "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: arXiv.org/abs/cond-mat/9909130 .
- Information on the discrete Fourier transform:
  Numerical Recipes in C: The Art of Scientific Computing, 2^{nd} Edition (Cambridge University Press, 1992), chapter 12.
- Information on minimization algorithms:
  Numerical Recipes, chapter 10.

Part II

Assignments

Chapter 6

DFT++ Operators & Poisson Equation

6.1 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 mini-course.

6.2 Definition of basic variables: "setup.m"

6.2.1 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

$$\mathbf{M} \equiv \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ \vdots & 0 & 0 \\ S_1 - 1 & 0 & 0 \\ S_1 & 0 & 0 \\ \vdots & 1 & 0 \\ \vdots & \vdots & 0 \\ \vdots & S_2 - 1 & 0 \\ \vdots & S_2 & 0 \\ \vdots & \vdots & 1 \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & S_3 - 1 \\ \vdots & \vdots & S_3 \end{bmatrix} \tag{6.1}$$

$$\mathbf{N} \equiv \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ \vdots & 0 & 0 \\ -2 & 0 & 0 \\ -1 & 0 & 0 \\ \vdots & 1 & 0 \\ \vdots & \vdots & 0 \\ \vdots & -2 & 0 \\ \vdots & -1 & 0 \\ \vdots & \vdots & 1 \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & -2 \\ \vdots & \vdots & -1 \end{bmatrix} \tag{6.2}$$

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

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & 0 \\ 1 & 2 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 2 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 1 & 1 \\ 0 & 2 & 1 \\ 1 & 2 & 1 \\ 2 & 2 & 1 \end{bmatrix} \tag{6.3}$$

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

```
%# 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.

6.2.2 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,

$$\mathbf{r} = \mathbf{M}\left(\mathbf{Diag}\left(S\right)\right)^{-1}\mathbf{R}^{T} \tag{6.4}$$

$$\mathbf{G} = 2\pi\mathbf{N}\mathbf{R}^{-1} \tag{6.5}$$

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.

[1]Note that copies of these and all program files in the appendices are also available on the Cornell Physics Educational Computing Facility at "/home/muchomas/Minicourse".

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.

6.3 Charge distribution: "poisson.m"

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

$$n = g_1(r) - g_2(r) \tag{6.6}$$

$$\equiv \frac{e^{-r^2/(2\sigma_2^2)}}{(2\pi\sigma_2^2)^{3/2}} - \frac{e^{-r^2/(2\sigma_1^2)}}{(2\pi\sigma_1^2)^{3/2}} \tag{6.7}$$

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.

Copy the code "poisson.m" from Appendix C to "∼/A1", 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. (6.6). 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.

6.4 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!

6.4.1 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 $\vec{\phi} = \mathbf{I}\,\mathbf{L}^{-1}\,(-4\pi\mathbf{O}\,\mathbf{J}\vec{n})$ 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 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;
```

6.4.2 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 $\mathbf{O} = (\det \mathbf{R})\mathbf{I}$

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!

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)
```

6.4.3 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 $\mathbf{L}$ 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
```

6.4.4 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 $\mathbf{R}$
- 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
```

6.4.5 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 "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".)
NOTE: The remote login machine remote.physics.cornell.edu runs an older version of `octave` that doesn't have FFTW installed. If you work remotely, you will have to first ssh into one of the workstations ws01, ws02, ... If you log directly into a workstation, this is not an issue.

6.4.6 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
```

6.5 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.

Hint: The comparison of energies should agree to four significant figures.

6.6 Ewald energy calculator

As mentioned in lecture, the "Ewald summation" technique computes the energy of a periodic array of point charges by treating each charge as a continuous Gaussian distribution and then correcting for the overlaps of the Gaussian tails via a sum in real-space. Choosing the Gaussians to be so narrow that the overlaps are insignificant and just computing the energy of the array of Gaussians thus represents one special, 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.

6.6.1 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 core 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

$$S_f(\vec{G}) \equiv \sum_I e^{-i\vec{G}\cdot\vec{X}_I}, \tag{6.8}$$

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.

6.6.2 "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.

Chapter 7

Simple Harmonic Oscillator & Quantum Dot

7.1 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

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

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

$$\begin{aligned} E &= -\frac{1}{2}\mathrm{Tr}\left(W^\dagger L W U^{-1}\right) + \tilde{V}^\dagger \vec{n} \\ U &\equiv W^\dagger \mathbf{O} W \\ \vec{n} &\equiv \mathrm{diag}\left(\left(\mathbf{I}WU^{-1}\right)\left(\mathbf{I}W\right)^\dagger\right) \\ \tilde{V} &\equiv \mathbf{J}^\dagger \mathbf{O}\mathbf{J}\vec{V}, \end{aligned} \tag{7.1}$$

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,

$$\left[\nabla_W E\right]_{\alpha,n} \equiv \frac{\partial E}{\partial W^*_{\alpha,n}}. \tag{7.2}$$

As derived in lecture, the gradient of E then becomes

$$\nabla_W E = \left(HW - \mathbf{O}WU^{-1}W^\dagger HW\right)U^{-1}, \tag{7.3}$$

where U is defined as above and

$$H \equiv -\frac{1}{2}L + \mathbf{I}^\dagger\left(\mathrm{Diag}\,\tilde{V}\right)\mathbf{I}. \tag{7.4}$$

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 Ψ is

$$Y \equiv WU^{-1/2} \tag{7.5}$$
$$\Psi \equiv YD, \tag{7.6}$$

where D diagonalizes the matrix

$$\mu \equiv Y^\dagger H Y, \tag{7.7}$$

according to

$$D^\dagger \mu D = \mathrm{diag}\,\vec{\epsilon}, \tag{7.8}$$

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

Finally, note that with gradients expressed as matrices as in Eq. (7.3), the following (ultimately!) simple expression may be used to compute the directional derivative of the energy E along the direction dW,

$$\begin{aligned} dE &\equiv \sum_{\alpha,n} \left[\mathrm{Re}\left(dW_{\alpha,n}\right) \frac{\partial E}{\partial \mathrm{Re}\left(dW_{\alpha,n}\right)} + \mathrm{Im}\left(dW_{\alpha,n}\right) \frac{\partial E}{\partial \mathrm{Im}\left(dW_{\alpha,n}\right)} \right] \\ &= \sum_{\alpha,n} \mathrm{Re}\left[\left(\mathrm{Re}\left(dW_{\alpha,n}\right) - i\mathrm{Im}\left(dW_{\alpha,n}\right)\right) \left(\frac{\partial E}{\partial \mathrm{Re}\left(W_{\alpha,n}\right)} + i \frac{\partial E}{\partial \mathrm{Im}\left(W_{\alpha,n}\right)} \right) \right] \\ &= \mathrm{Re} \sum_{\alpha,n} dW^*_{\alpha,n} 2 \left(\frac{\partial E}{\partial W^*_{\alpha,n}} \right) \\ &= 2\,\mathrm{Re}\,\mathrm{Tr}\, dW^\dagger \nabla_W E. \end{aligned} \tag{7.9}$$

7.2 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.

7.2.1 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,

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

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)=fftw3(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.

7.2.2 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".

7.2.3 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_S: dimensions of $d = 3$ dimensional data sets

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".

7.2.4 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_S: dimensions of $d = 3$ dimensional data sets

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!

7.2.5 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.

7.2.6 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.

7.2.7 Debugging

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)
```

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.

7.3 Energy calculation: "sch.m" and "getE.m"

7.3.1 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 F (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. (7.1) 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.

7.3.2 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. (7.1) is in the form of taking as a column vector the diagonal elements of the "outer product" of two matrices,

$$\vec{c} = \text{diag}\left(AB^\dagger\right). \tag{7.10}$$

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. (7.10) 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".

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!

7.3.3 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. (7.1) 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.

7.4 Gradient calculation

7.4.1 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. (7.3) ultimately involves products of the form

$$C = (\mathrm{Diag}\,\vec{a})\,B. \tag{7.11}$$

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. (7.11) 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".

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
```

7.4.2 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. (7.4) and your Diagprod() operator above, produce a function of the above prototype in the file 'H.m"

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.

7.4.3 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. (7.3) 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. (7.3) in octave, you may wish to include some "extra" parentheses.

Debugging

Copy the function fdtest() from Appendix G 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. (7.9). 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.

7.5 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

7.5.1 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. (7.5), 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.

Debugging

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

7.5.2 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 α later to see if you can find a better value.)

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.

7.5.3 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. (7.5,7.6).

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. (7.8). (Again, we take a real part because rounding errors sometimes lead to tiny imaginary parts.)

Debugging

Check that your output states are orthonormal and diagonalize μ 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
```

7.6 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 H and I 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.

7.7 Density functional theory: "dft.m"

7.7.1 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 ψ_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,

$$T = -\frac{1}{2}\sum_i f_i \int \psi_i^* \nabla^2 \psi_i = f \cdot \frac{1}{2}\mathrm{Tr}\,(W^\dagger L W U^{-1/2}). \tag{7.12}$$

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

$$\vec{n} = f \cdot \mathrm{diag}\left[\mathbf{I}WU^{-1}\left(\mathbf{I}W\right)^\dagger\right]. \tag{7.13}$$

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,

$$U = \tilde{V}^\dagger \vec{n}, \tag{7.14}$$

where

$$\tilde{V} \equiv \mathbf{J}^\dagger \mathbf{O} \mathbf{J} \vec{V}, \tag{7.15}$$

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

$$U_{ee} = \frac{1}{2}\vec{n}^\dagger \mathbf{J}^\dagger \mathbf{O} \hat{\phi}, \tag{7.16}$$

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,

$$E_{xc} = \vec{n}^\dagger \mathbf{J}^\dagger \mathbf{O} \mathbf{J} \epsilon_{xc}(\mathbf{I}\vec{n}). \tag{7.17}$$

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

$$\begin{aligned} E_{LDA} &= -f \cdot \frac{1}{2} \mathrm{Tr}\left(W^\dagger L W U^{-1}\right) + \tilde{V}^\dagger n \\ &\quad + \frac{1}{2} \vec{n}^\dagger \mathbf{J}^\dagger \mathbf{O} \hat{\phi} + \vec{n}^\dagger \mathbf{J}^\dagger \mathbf{O} \mathbf{J} \epsilon_{xc}(\vec{n}), \end{aligned} \tag{7.18}$$

where

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

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

$$\nabla_W E = f\left(HW - \mathbf{O} W U^{-1} W^\dagger H W\right) U^{-1}. \tag{7.19}$$

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

$$H \equiv -\frac{1}{2} L + \mathbf{I}^\dagger \left(\mathrm{Diag}\, V_{\mathrm{eff}}\right) \mathbf{I}, \tag{7.20}$$

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 $\vec{n}$.

7.7.2 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.

7.7.3 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 ϕ and ϵ_{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.

Debugging

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

7.7.4 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. (7.18,7.20) which involve ϕ.
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!

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.

7.7.5 Full density-functional theory

Add the remaining terms in Eqs. (7.18,7.20), those involving ϵ_{xc} and ϵ'_{xc}, to get() and H(). You may use the functions excVWN() and excpVWN() in Appendices J and K, which implement the operators $\epsilon_{xc}(\vec{n})$ and $\epsilon'_{xc}(\vec{n})$, respectively, using the Vosko-Wilk-Nusair parameterization.

Debugging

Run the finite difference test, first zeroing out all terms but those involving ϵ_{xc} and ϵ'_{xc} and then including all terms.

7.7.6 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.)

Chapter 8

Optimizations, Hydrogen Molecule & Solid Ge

8.1 Advanced techniques for numerical minimization

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

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

8.1.1 sd()

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

Input:

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

Output:

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

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

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

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

8.1.2 lm()

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

Input:

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

Output:

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

Implement the line minimization algorithm discussed in lecture,

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

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

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

Debugging

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

8.1.3 K()

`function out=K(W)`

Input:

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

Global variables:

- gbl_G2: Square magnitude of G vectors

Output:

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

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

8.1.4 pclm()

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

Input:

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

Output:

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

Implement the preconditioned line minimization algorithm discussed in lecture,

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

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

Debugging

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

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

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

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

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

8.1.5 pccg()

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

Input:

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

Output:

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

Implement the preconditioned conjugate-gradients algorithm discussed in lecture,

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

with a function of the above prototype in the file "pccg.m".

Debugging

Analogously to debugging for line minimization, the best test of your implementation is the gradient-orthogonality theorem, the "cg test". Thus, immediately after running "dft.m", approach the minimum even more closely by running 50 iterations of the conjugate-gradient algorithm with "`Wout=pccg(Wpclm,50);`" at the command prompt. You should observe that, because the Fletcher-Reeves form for β depends on a set of mathematical identities which assume the minimum is a pure quadratic form, "cg test" gives results which track but are somewhat larger than those of the largest preceding "linmin test". Finally, to convince yourself completely that pccg() functions properly, send the output of pccg back into pccg with "`pccg(Wout,50);`". At this point, you are close enough to the minimum to find that the "cg test" gives results on the order of 10^{-7}.

8.1.6 Final comparison of techniques

To complete the comparison of the techniques, add the line

```
[W, Epccg]=pccg(W,50); %# 50 iterations of pccg from same W
```

immediately after the line in "dft.m" calling pclm(). Note that this statement puts the final result back into W so that the code below the pause statement will work from these much improved wave functions. Finally, replace the current call to semilogy() with

```
semilogy([1:length(Elm)],Elm-43.3371147782040,'r-;lm;', ...
         [1:length(Epclm)],Epclm-43.3371147782040,'b-;pclm;', ...
         [1:length(Epccg)],Epccg-43.3371147782040,'g-;pccg;');
```

8.2 Density functional calculation of molecules

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

8.2.1 Ionic potential

Background

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

$$V(\vec{r}) = -\sum_I \frac{Z_I}{|\vec{r} - \vec{X}_I|}, \tag{8.1}$$

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

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

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

and the definition of Vdual from Section 7,

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

where the extra '-' in front of ϕ_{nuc} corresponds to the minus sign in Eq. (8.1). Combining the above two results gives

$$\tilde{V} = \mathbf{J}^\dagger \mathbf{O}\mathbf{L}^{-1}(4\pi \mathbf{O}\mathbf{J}\vec{\rho}_{nuc}) \tag{8.2}$$

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

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

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

Therefore,

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

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

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

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

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

where $S_f(\vec{G})$ is the same structure factor from Eq. (6.8). Inserting this into Eq. (8.2) and using the normalizations associated with the various operators gives

$$\tilde{V} = \mathbf{J}^\dagger \hat{v}, \tag{8.3}$$

where

$$\hat{v}_{\vec{G}} = \left(\frac{-4\pi Z}{G^2}\right) S_f(\vec{G}). \tag{8.4}$$

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

Implementation

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

```
format long

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

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

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

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

Note that this code is careful to set the infinite $\vec{G} = 0$ component of the ionic potential to zero, a term which exactly balances with the $\vec{G} = 0$ components of the Ewald and Hartree energies. Also, we are calling this Fourier-transformed potential from a single ion "Vps" because it will become the Fourier transform of the corresponding pseudopotential when we do solid germanium. After computing Vps, next evaluate $\tilde{V}$ according to Eqs. (8.3-8.4) as

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

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

Debugging

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

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

2. Then, run "Hatoms.m" for a single state with one electron (gbl_f=1, Ns=1) using the same parameters in "setup.m" as you used for the ewald calculation.

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

8.2.2 Hydrogen molecule

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

Debugging: At a separation of 1.5 bohr, you should find a total energy of -1.136 hartree.

From this data, you should be able to extract the LDA predictions for the bond length, energy and effective spring constant for H_2. See the files "/home/muchomas/Minicourse/H2*" for a plot and summary of an analysis of the results with a comparison to data from the CRC handbook!

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

8.3 Minimal, isotropic spectral representation

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

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

8.3.1 "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.

Radius of inscribed sphere

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

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

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

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

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

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

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

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

Debugging

Run the modified "setup.m" for parameters S=[32; 32; 32], R=diag([8 8 8]) and X=[0 0 0]. 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. Also, the number is larger because we have "rounded" down in all cases.)

8.3.2 Operators

For debugging purposes, you should now rerun your hydrogen atom calculation with parameters S=[32; 32; 32], R=diag([8 8 8]) in "setup.m" and gbl_f=1, Ns=1, X=[0 0 0] in "Hatoms.m" . Take note of your final converged values, as we will be checking that your reduced basis calculation gives essentially the same result. (I find E=-0.5216 H for the electronic part of the energy.)

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

"L.m" and "K.m"

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

"cI.m"

In lecture, we found that the I() operator becomes $\mathbf{I} = \mathbf{F}_{(\mathbf{3})}\mathbf{B}$, where "$\mathbf{F}_{(3)}$" is the three-dimensional Fourier transform kernel and "$\mathbf{B}$" 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
```

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

"cIdag.m"

The Idag() operator now becomes $\mathbf{I}^\dagger = \left(\mathbf{F}_{(\mathbf{3})}\mathbf{B}\right)^\dagger = \mathbf{B}^\dagger\mathbf{F}^\dagger_{(3)}$. Again, to implement this, you will need "global gbl_active" at the top of the function. In this case, it turns out that the output cIdag() is always a wavefunction-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.

Debugging

Now, rerun your hydrogen calculation in "Hatoms.m" and verify that you reproduce your previous results to within 0.006 hartree! You should also notice a decrease in the iteration and an improvement in the number of iterations needed to reach convergence.
Hints: The modification to "Hatoms.m" should involve only changing the initialization of W to that of a random array of size length(gbl_active)×Ns. For debugging the above operators (most bugs will involve various "nonconformant" errors), make liberal use the size() function in your various operators in order to print out the sizes of the relevant objects from within the functions that generate errors.

8.4 Calculation of solid Ge

8.4.1 "setup.m"

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

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

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

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

8.4.2 "Geatoms.m"

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

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

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

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

$$\begin{aligned}\int e^{-i\vec{G}\cdot\vec{x}}V_{ps}(\vec{x}) &= -\frac{4\pi Z}{G^2} + \frac{4\pi Z(1+e^{-\lambda r_c})}{G^2}\left(-\frac{2\pi e^{-\pi G/\lambda}\cos(Gr_c)\frac{G}{\lambda}}{1-e^{-2\pi G/\lambda}}\right. \\ &\left. + \sum_{n=0}^{\infty}(-1)^n\frac{e^{-\lambda r_c n}}{1+\left(\frac{n\lambda}{G}\right)^2}\right)\end{aligned} \tag{8.5}$$

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

$$\lim_{G\to 0}\left(\frac{4\pi Z}{G^2} + \int e^{-i\vec{G}\cdot\vec{x}}V_{ps}(\vec{x})\right) = 4\pi Z\left(1+e^{-\lambda r_c}\right)\left(\frac{r_c^2}{2} + \frac{1}{\lambda^2}\left(\frac{\pi^2}{6} + \sum_{n=1}^{\infty}(-1)^n\frac{e^{-\lambda r_c n}}{n^2}\right)\right) \tag{8.6}$$

To avoid needless heartache, feel free to cut and past the octave form for the resulting formula,

```
%# Ge pseudopotential
Z=4;
lambda=18.5;
rc=1.052;
```

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

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

Finally, to allow for viewing of the total charge density, add the code block below to the very end of "Geatoms.m",

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

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

Debugging

Run "Geatoms.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` executable doesn't work on your computer, you can try to view the image file directly for your directory using some other application.

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

Finally, run "ewald.m", and compute the total energy *per atom* of the crystal. Compare this to the energy of the pseudoatom calculated at the same spatial resolution, -3.7301 H. (See Section 9.3 if you'd like to compute this for yourself.) Finally, using the conversion 1 hartree=27.21 eV compute your prediction for the cohesive energy of germanium. (The experimental value is 3.85 eV/atom!)

Chapter 9

Additional Optimizations and Features

9.1 Full benefit from minimal representation

Currently, the wave functions are stored in the minimal representation. However, when transformed to sample values with I(), as occurs in getE() and H(), the output lives in the full FFT box and consumes an unnecessarily large amount of memory (sixteen times more than is needed), whose allocation and deallocation also tends to slow the calculation. To avoid this, we should apply I() only to one column of the wave functions at a time. To achieve this, make the following modifications **after creating a new copy of all of your ".m" files in "∼/A3a" in a new directory, "∼/A4"**.

9.1.1 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 "Geatoms.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.) You should find for the Ge calculation a noticeable improvement in run time of about 25% and peak memory (which you can monitor with the Linux "top" command) and in memory consumption by about a factor of two.

9.1.2 H()

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

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

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

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

9.2 Variable fillings

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

9.2.1 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.

9.2.2 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.

9.2.3 Q()

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

9.2.4 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,

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

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

Debugging

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

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

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

9.3 Isolated Ge atom

As a non-trivial test of your variable fillings code, and to compute the energy (given in Section 8.4.2) of an isolated Ge atom at the same resolution as your solid state calculation, set the parameter `X=a*[0.00 0.00 0.00]` in "setup.m" and the parameters `Ns=4` and `gbl_f=[2;2/3;2/3;2/3]` in "Geatoms.m" and rerun "setup.m" and "ewald.m" and "Geatoms.m". Passing the finite-difference test should be sufficient to verify your new code, and summing the resulting Ewald and electronic energies should give the value of an isolated Ge atom quoted in Section 8.4.2).

[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.

Part III

Appendices

Appendix 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
```

Appendix 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
```

Appendix 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
%#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);
```

Appendix 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
```

Appendix 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
```

Appendix F

“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
```

Appendix G

"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
```

Appendix H

“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
```

Appendix I

"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
```

Appendix J

"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
```

Appendix K

"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
```

Appendix L

"Q.m"

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

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