Cornell University

Department of Physics

Phys 480/680, Astro 690 April 3, 2005

Computational Physics, Spring 2005

Homework Assignment # 7

(Due Thursday, April 14)

Agenda and readings for the weeks of April 6–23:

Goal: Learn to use the `octave` linear algebraic language to express scientific algorithms succinctly; solution of Poisson's equation in a single line of code

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

solution of Schrodinger's equation by steepest descents.

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

recap of last week of class:

- 03/29 Lec 17: Blocked matrix multiplies; Solution of Poisson's equation in d¿1 dimensions; expressive software; solution to Poisson's equation in a single line of code; choice of planewave (complex exponential) basis for periodic boundary conditions.
- 03/31 Lec 18: Form of operators and transforms in a planewave basis; octave/matlab notations; standard ordering of indices for Discrete Fourier Transforms (DFTs).

Preview of coming two weeks:

- 04/05 Lec 19: Discrete Fourier Transforms: aliasing/Nyquist frequency, Fourier factorization theorem, Fourier inversion theorem
  Reading: NR 12.0, 12.1, 12.4
- 04/07 Lec 20: Eigenvalue problems in $d > 1$ dimension: variational principle, analytic continuation for constraints, minimization by steepest descents
  Reading: NR 10.0
- 04/07 Lab 7: FLOP performance of basic operations; FLOP rates in octave; Automated empirical optimization of software (AEOS): automatically tuned linear algebra software (ATLAS), fastest Fourier Transform in the West (FFTW).
- 04/12 Lec 21: DFT++ expressions for energy and gradient for multiple states
- 04/14 Lec 22: Converge of steepest descents; preconditioning; line minimization
  Reading: NR 10.2, 10.3
- 04/14 Lab 8: Spectral solution of Poisson's and Schrödinger's equations

Contents

1 Overview **3**

2 Definition of basic variables: "setup.m" **3**
2.1 Indexing . . . 3
2.2 Sampling points and reciprocal lattice vectors . . . 5

3 Charge distribution: "poisson.m" **6**

4 Operators **6**
4.1 Global variables . . . 7
4.2 O() . . . 7
4.3 L() . . . 8
4.4 Linv() . . . 8
4.5 cI() . . . 9
4.6 cJ() . . . 9

5 Final solution to Poisson's equation: "poisson.m" **10**

6 Background for Schrödinger's equation **11**

7 Updated Operators **12**
7.1 cI() . . . 12
7.2 cJ() . . . 13
7.3 O() . . . 13
7.4 L() . . . 13
7.5 cIdag() . . . 14
7.6 cJdag() . . . 14
7.7 Debugging . . . 14
7.7.1 Single column cases . . . 14
7.7.2 Multiple column cases . . . 15

8 Energy calculation: "sch.m" and "getE.m" **15**
8.1 Setup of the potential . . . 15
8.2 diagouter() . . . 16
8.2.1 Debugging . . . 16
8.3 getE() . . . 16

9 Gradient calculation **17**
9.1 Diagprod() . . . 17
9.1.1 Debugging . . . 17
9.2 H() . . . 18
9.2.1 Debugging . . . 18
9.3 getgrad() . . . 18
9.3.1 Debugging . . . 19

10 Solution of Schrödinger's equation using steepest descents: sd() **19**
10.1 Initialize W . . . 20
10.1.1 Debugging . . . 20
10.2 sd() . . . 20
10.2.1 Debugging . . . 20
10.3 getPsi() . . . 20

10.3.1 Debugging 21

11 Final solution: "sch.m" 21

A "slice.m" 22

B "view.m" 23

C "poisson.m" 24

D "O.m" 25

E "viewmid.m" 26

F "fdtest.m" 27

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 the remainder of this course.

2 Definition of basic variables: "setup.m"

2.1 Indexing

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

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

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

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & 0 \\ 1 & 2 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 2 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 1 & 1 \\ 0 & 2 & 1 \\ 1 & 2 & 1 \\ 2 & 2 & 1 \end{bmatrix} \quad (3)$$

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 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 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));
```

Rerun "setup.m" and inspect the result by entering first just "m1" and then just "m2" 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.

Following the logic which led to m1 and m2, add a corresponding line to compute m3. Verify by inspection that your formula for m3 functions properly. Finally, combine each of your columns into a matrix using the octave block matrix formation

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

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

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

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

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

and then combine your result into the the final matrix.

2.2 Sampling points and reciprocal lattice vectors

To specify the periodicity vectors (generally called 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,

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

$$\mathbf{G} = 2\pi\mathbf{N}\mathbf{R}^{-1} \quad (5)$$

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 your octave directory. (To understand more how the view() and slice() functions work, you may enter "help slice" and "help view" at the octave prompt after you have copied the files.) Once you have these 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 $\mathbf{r}$ by running "view(r(:,k),S)" for k=1,2,3 and verifying that you see the correct behavior.

3 Charge distribution: "poisson.m"

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

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

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

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

As an exercise to show to the grader, copy the code "poisson.m" from Appendix C to your octave directory, and complete the program by replace "..." 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 $\mathbf{r}$ and the center of the cell,

```
[R(1,1)/2; R(2,2)/2; R(3,3)/2]=diag(R)'/2
```

Hints: Consider the matrix

```
ones(prod(S),1)*diag(R)'
```

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

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

4 Operators

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

4.1 Global variables

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

4.2 O()

`function out=O(in)`

Input:

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

Global variables:

- gbl_R: lattice vectors **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 "O.m" (in your octave directory) 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)
```

4.3 L()

`function out=L(in)`

Input:

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

4.4 Linv()

`function out=Linv(in)`

Input:

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

4.5 cI()

`function out=cI(in)`

Input:

- in: $d = 3$ dimensional data stored as an $\prod S_k \times 1$ 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 you will need to copy the file /home/muchomas/P480/Octave/fftw3.oct on the physics educational cluster into your octave directory. To use this function, note that

```
fftw3(dat,Nx,Ny,Nz,1);
```

returns the discrete Fourier sum with sign $+i$ in the exponential for data of dimension Nx×Ny×Nz. (For more information, enter "help fftw3" at the octave prompt once you've copied fftw3.oct into your octave directory.)
NOTE: The remote login machine remote.physics.cornell.edu runs an older version of `octave` that can't read fftw3.oct. 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.

4.6 cJ()

`function out=cJ(in)`

Input:

- in: $d = 3$ dimensional data stored as an $\prod S_k \times 1$ 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 fftw3 to compute the inverse Fourier transform.

Hint: Do not forget the normalization factor $\prod S_k$.
To test your transforms execute the following commands at the octave prompt:

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

5 Final 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 Background for Schrödinger's equation

Lecture defined the unconstrained objective function for finding multiple solutions to Schrödinger's equation as

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

where

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

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

$$\begin{aligned} E &= -\frac{1}{2}\mathrm{Tr}\,\left(W^\dagger L W N^{-1}\right) + \tilde{V}^\dagger \vec{n} \\ N &\equiv W^\dagger \mathcal{O} W \\ \vec{n} &\equiv \mathrm{diag}\,\left(\left(\mathcal{I} W N^{-1}\right)(\mathcal{I} W)^\dagger\right) \\ \tilde{V} &\equiv \mathcal{J}^\dagger \mathcal{O} \mathcal{J} \vec{V}, \end{aligned} \tag{8}$$

where we have regrouped quantities under †'s, used the fact that $\mathcal{O}$ is Hermitian ($\mathcal{O}^\dagger = \mathcal{O}$), 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,

$$[\nabla_W E]_{\alpha,n} \equiv \frac{\partial E}{\partial W^*_{\alpha,n}}. \tag{9}$$

Using this notation, as we shall derive in a later lecture, the gradient of E becomes

$$\nabla_W E = \left(HW - \mathcal{O} W N^{-1} W^\dagger H W\right) N^{-1}, \tag{10}$$

where N is defined as above and

$$H \equiv -\frac{1}{2} L + \mathcal{I}^\dagger \left(\mathrm{Diag}\,\tilde{V}\right) \mathcal{I}. \tag{11}$$

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 eignensolutions Ψ is

$$Y \equiv W N^{-1/2} \tag{12}$$
$$\Psi \equiv Y U, \tag{13}$$

where U diagonalizes the matrix

$$\mu \equiv Y^\dagger H Y, \tag{14}$$

according to

$$U^\dagger \mu U = \mathrm{diag}\,\vec{\epsilon}, \tag{15}$$

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

Finally, note that with gradients expressed as matrices as in Eq. (10), the following (ultimately!) simple expression may be used to compute the directional derivative of the energy E along the direction dW,

$$\begin{aligned} dE &\equiv \sum_{\alpha,n} \left[\mathrm{Re}\left(dW_{\alpha,n}\right) \frac{\partial E}{\partial \mathrm{Re}\left(dW_{\alpha,n}\right)} + \mathrm{Im}\left(dW_{\alpha,n}\right) \frac{\partial E}{\partial \mathrm{Im}\left(dW_{\alpha,n}\right)} \right] \\ &= \sum_{\alpha,n} \mathrm{Re}\left[\left(\mathrm{Re}\left(dW_{\alpha,n}\right) - i\mathrm{Im}\left(dW_{\alpha,n}\right)\right) \left(\frac{\partial E}{\partial \mathrm{Re}\left(W_{\alpha,n}\right)} + i \frac{\partial E}{\partial \mathrm{Im}\left(W_{\alpha,n}\right)} \right) \right] \\ &= \mathrm{Re} \sum_{\alpha,n} dW^*_{\alpha,n} 2 \left(\frac{\partial E}{\partial W^*_{\alpha,n}} \right) \\ &= 2\,\mathrm{Re}\,\mathrm{Tr}\, dW^\dagger \nabla_W E. \end{aligned} \tag{16}$$

7 Updated Operators

Two new operators appear in the above expressions, $\mathcal{I}^\dagger$ and $\mathcal{J}^\dagger$. 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.

7.1 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,

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

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 $\mathcal{I}$ 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 avoids extra calls to malloc() 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 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".

7.3 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".

7.4 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!

7.5 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 fftw.

7.6 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 fftw.

7.7 Debugging

7.7.1 Single column cases

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

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

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

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)
```

7.7.2 Multiple column cases

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

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

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

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

8.1 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 E, 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. (8) 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.

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

$$\vec{c} = \text{diag}\left(AB^\dagger\right). \tag{17}$$

Because A and B are of dimension $\prod S_k \times N_s$, it is extremely wasteful (of time and memory) to directly form 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. (17) 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".

8.2.1 Debugging

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

```
A=randn(10,3)+i*randn(10,3); %# Form random N and M 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!

8.3 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. (8) 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.

9 Gradient calculation

9.1 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. (10) ultimately involves products of the form

$$C = (\mathrm{Diag}\,\vec{a})\,B. \tag{18}$$

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. (18) 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".

9.1.1 Debugging

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

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

9.2 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 (which represents "LHS(W)")

Using the expression in Eq. (11) and your Diagprod() operator above, produce a function of the above prototype in the file 'H.m"

9.2.1 Debugging

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

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

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

9.3 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. (10) 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 $(\mathcal{O}WN^{-1}W^\dagger)(HW)$ is the product of a $\prod S_k \times \prod S_k$ matrix with a $\prod S_k \times N_s$ matrix, while $(\mathcal{O}WN^{-1/2})(W^\dagger HW)$ 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. (10) in octave, you may wish to include some "extra" parentheses.

9.3.1 Debugging

Copy the function fdtest() from Appendix F 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. (16). 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 correct behavior.

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 "ftest.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.

10 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

10.1 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. (12), being sure to store your result back in W.

Hint: In octave, inv() and sqrtm(), respectively, invert and take the square root of a square matrix.

10.1.1 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).

10.2 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.)

10.2.1 Debugging

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

10.3 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. (12,13).

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

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

produces the matrix U and the vector $\vec{\epsilon}$ in Eq. (15). (Again, we take a real part because rounding errors sometimes lead to tiny imaginary parts.)

10.3.1 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
```

11 Final solution: "sch.m"

Place the following code block at the end of your "sch.m" and run. The code should then start from random functions, perform a finite difference test, orthonormalize, run 400 iterations of your sd(), and compute the final results with your getPsi(). Finally, the code will display the energy of each state along with mesh plots of the values of $|\Psi_n(\vec{r})|^2$ on planes cutting through the center of your cell.

```
%# 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));
  viewmid(abs(cI(Psi(:,st))).^2,S)
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. Note that s states are spherically symmetric and p states have two lumps separated by a plane of zeros.

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)

  if dir==3
    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
    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
    dat=reshape(dat,N(1),N(2)*N(3)); %# Group to expose N(1)
    dat=conj(dat'); %# dat is now N(2)*N(3),N(1)
    out=reshape(dat(:,n),N(2),N(3));
  else
    printf("\nError in slice(): invalid choice for dir.  dir=%f\n\n",dir);
  endif

endfunction
```

B "view.m"

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

function view(dat,S)

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

for k=1:3
  if k==1
    fprintf('m1=0 slice (m3, m2 along left-, right- axes):\n');
  elseif k==2
    fprintf('m2=0 slice (m3, m1 along left-, right- axes):\n');
  elseif k==3
    fprintf('m3=0 slice (m2, m1 along left-, right- axes):\n');
  end

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

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
  mesh(slice(n,S,S(dir)/2,dir));
  fprintf('n%d=%d slice\n',dir,S(dir)/2); 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
%#  mesh(slice(phi,S,S(dir)/2,dir));
%#  fprintf('n%d=%d slice of phi\n',dir,S(dir)/2); 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);
```

D "O.m"

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

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

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

E "viewmid.m"

```
% Function to view slices through center point of three dimensional data sets
%
% Usage: viewmid(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
    if k==1
      fprintf('m1=%d slice (m2, m3 along left, right axes):\n',S(1)/2-1);
    elseif k==2
      fprintf('m2=%d slice (m1, m3 along left, right axes):\n',S(2)/2-1);
    elseif k==3
      fprintf('m3=%d slice (m1, m2 along left, right axes):\n',S(3)/2-1);
    end

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

F "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
```