Cornell University

Department of Physics

Phys 480/680, Astro 690 April 23, 2004

Computational Physics, Spring 2004

Homework Assignment # 8

(Due Thursday, April 29)

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

Goals: Solution of single particle Schrödinger equation using *unconstrained* minimization techniques; minimization using steepest descents; solution of the harmonic oscillator in $d = 3$ dimensions.

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 two weeks of class:

- 04/13 Lec 18: Details of implementation of spectral methods: mappings to d=3 dimensions, Aliasing/Nyquist frequency, Fourier factorization theorem
  Reading: NR 12.0, 12.4
- 04/15 NO CLASS
- 04/16 NO LAB
- 04/20 Lec 19: Fourier inversion theorem; variational principle for eigenvalue problems in $d > 1$ dimension
- 04/22 Lec 20: Analytic continuation for constrains; algebraic formulation for energy for Schrödinger's equation; Numerical minimization: steepest descents
  Reading: NR 10.0
- 04/23 NO LAB

Preview of final two weeks:

- 04/27 Lec 21: Numerical minimization: line minimization, preconditioning; difficulties with line minimization; concept of conjugate gradients (directions, actually)
  Reading: NR 10.6
- 04/29 Lec 22: Conjugate gradient algorithm
- 04/30 Lab 8: Spectral solution to Poisson's equation in octave, solution to Schrödinger's equation via steepest descent minimization
- 05/04 Lec 23: Derivation of gradients in algebraic notation, gradients for DFT
- 05/06 Lec 24: FFT algorithm
- 05/07 Lab 9: DFT solutions for molecules

Contents

1 Background **3**

2 Operators **4**
- 2.1 cI() . . . 4
- 2.2 cJ() . . . 5
- 2.3 O() . . . 5
- 2.4 L() . . . 5
- 2.5 cIdag() . . . 6
- 2.6 cJdag() . . . 6
- 2.7 Debugging . . . 6
  - 2.7.1 Single column cases . . . 6
  - 2.7.2 Multiple column cases . . . 7

3 Energy calculation: "sch.m" and "getE.m" **7**
- 3.1 Setup of the potential . . . 7
- 3.2 diagouter() . . . 8
  - 3.2.1 Debugging . . . 8
- 3.3 getE() . . . 8

4 Gradient calculation **9**
- 4.1 Diagprod() . . . 9
  - 4.1.1 Debugging . . . 9
- 4.2 H() . . . 10
  - 4.2.1 Debugging . . . 10
- 4.3 getgrad() . . . 10
  - 4.3.1 Debugging . . . 11

5 Solution of Schrödinger's equation using steepest descents: sd() **11**
- 5.1 Initialize W . . . 12
  - 5.1.1 Debugging . . . 12
- 5.2 sd() . . . 12
  - 5.2.1 Debugging . . . 12
- 5.3 getPsi() . . . 12
  - 5.3.1 Debugging . . . 13

6 Final solution: "sch.m" **13**

A "viewmid.m" **14**

B "fdtest.m" **15**

1 Background

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

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

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,

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

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

$$\nabla_W E = \left(HW - \mathcal{O} W N^{-1} W^\dagger H W\right) N^{-1}, \tag{3}$$

where N is defined as above and

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

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 eignensolutions Ψ is

$$Y \equiv W N^{-1/2} \tag{5}$$
$$\Psi \equiv Y U, \tag{6}$$

where U diagonalizes the matrix

$$\mu \equiv Y^\dagger H Y, \tag{7}$$

according to

$$U^\dagger \mu U = \mathrm{diag}\,\vec{\epsilon}, \tag{8}$$

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

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

$$\begin{aligned} dE &\equiv \sum_{\alpha,n} \left[\mathrm{Re}\left(dW_{\alpha,n}\right) \frac{\partial E}{\partial \mathrm{Re}\left(dW_{\alpha,n}\right)} + \mathrm{Im}\left(dW_{\alpha,n}\right) \frac{\partial E}{\partial \mathrm{Im}\left(dW_{\alpha,n}\right)} \right] \\ &= \sum_{\alpha,n} \mathrm{Re}\left[\left(\mathrm{Re}\left(dW_{\alpha,n}\right) - i\mathrm{Im}\left(dW_{\alpha,n}\right)\right) \left(\frac{\partial E}{\partial \mathrm{Re}\left(W_{\alpha,n}\right)} + i \frac{\partial E}{\partial \mathrm{Im}\left(W_{\alpha,n}\right)} \right) \right] \\ &= \mathrm{Re} \sum_{\alpha,n} dW^*_{\alpha,n} 2 \left(\frac{\partial E}{\partial W^*_{\alpha,n}} \right) \\ &= 2\,\mathrm{Re}\,\mathrm{Tr}\, dW^\dagger \nabla_W E. \end{aligned} \tag{9}$$

2 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.

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,

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

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.

2.2 cJ()

`function out=cJ(in)`

Input:

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

Global variables:

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

Output:

- out: cJ operator applied to in

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

2.3 O()

`function out=O(in)`

Input:

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

Global variables:

- gbl_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".

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!

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 fftw.

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 fftw.

2.7 Debugging

2.7.1 Single column cases

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

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

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

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)
```

2.7.2 Multiple column cases

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

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

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

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

3.1 Setup of the potential

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

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

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

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

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

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

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

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

immediately after your computation of V.

3.2 diagouter()

`function out=diagouter(A,B)`

Input:

- A,B: $N \times M$ matrices

Output:

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

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

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

Because A and B are of dimension $\prod S_k \times N_s$, it is extremely wasteful (of time and memory) to 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. (10) is

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

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

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

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

3.2.1 Debugging

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

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

3.3 getE()

`function E=getE(W)`

Input:

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

Global variables:

- gbl_Vdual: Dual potential coefficients stored as a $S_k \times 1$ column vector.

Output:

- E: Energies summed over N_s states

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

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

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

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

4 Gradient calculation

4.1 Diagprod()

`function out=Diagprod(a,B)`

Input:

- a: $N \times 1$ column vector
- B: $N \times M$ matrix

Output:

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

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

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

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

In terms of components, Eq. (11) becomes

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

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

4.1.1 Debugging

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

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

4.2 H()

`function out=H(W)`

Input:

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

Global variables:

- gbl_Vdual: Dual potential coefficients stored as a $S_k \times 1$ column vector.

Output:

- out: HW (which represents "LHS(W)")

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

4.2.1 Debugging

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

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

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

4.3 getgrad()

`function grad=getgrad(W)`

Input:

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

Output:

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

Using the expression in Eq. (3) and your H() operator above, produce a function of the above prototype in the file "getgrad.m".
Hint: Matrix multiplication is associative, so that the final result of a matrix product ABC does not depend on the order in which the product is formed, $A(BC)$ or $(AB)C$. However, the dimensions of the intermediate values can be quite different. For instance $(\mathcal{O}WN^{-1}W^\dagger)(HW)$ 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. (3) in octave, you may wish to include some "extra" parentheses.

4.3.1 Debugging

Copy the function fdtest() from Appendix B into the file "fdtest.m". This program takes an initial W containing Ns wave functions and computes the energy and gradient for that W using your functions getE() and getgrad(). The function then forms a random direction and steps different distances along that direction, printing at each step the ratio of the actual change in energy to the change in energy expected from your gradient according to the formula Eq. (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 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.

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

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

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

5.1 Initialize W

It is helpful to at least start with orthonormal wave functions. Thus, immediately after the call to fdtest in your "sch.m" file, orthonormalize W according to equation Eq. (5), 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.

5.1.1 Debugging

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

5.2 sd()

`function out=sd(W,Nit)`

Input:

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

Output:

- out: Result of Nit iterations of steepest descents
- DISPLAY: with each iteration, print the result of getE() on current solutions

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

5.2.1 Debugging

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

5.3 getPsi()

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

Input:

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

Output:

- Psi: eigensolutions
- epsilon: eigenvalues

Provide a function of the above prototype in the file "getPsi.m" which computes the final solutions from the non-orthonormal W according to the formulas Eqs. (5,6).

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

5.3.1 Debugging

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

6 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 "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
```

B "fdtest.m"

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

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

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

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

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