Cornell University

Department of Physics

Phys 480/680, Astro 690 April 8, 2004

Computational Physics, Spring 2004

Homework Assignment # 7

(Due Thursday, April 22)

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))));
```

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

Recap of last week of class:

- 04/06 Lec 18: Spectral methods (use of the DFT/FFT to implement plane wave bases), expressive software notation, octave syntax, **O** and **L** operators
- 04/08 Lec 19: Transform operators ($\mathcal{I}$, $\mathcal{J}$), indexing of three dimensional objects
  Reading: NR 12.0, 12.1
- 04/09 Lab 7: FLOP performance of basic operations; FLOP rates in octave.

Preview of coming two weeks:

- 04/13 Lec 18: Aliasing/Nyquist frequency, Fourier factorization theorem; Fourier inversion theorem; Eigenvalue problems in $d > 1$ dimension, variational principle, analytic continuation for constraints
  Reading: NR 12.0, 12.4
- 04/15 NO CLASS
- 04/16 NO LAB
- 04/20 Lec 19: Algebraic formulation for energy for Schrödinger's equation; Numerical minimization: steepest descents
  Reading: NR 10.0
- 04/22 Lec 20: Gradient of energy for Schrödinger's equation; Numerical minimization: line minimization, preconditioning; difficulties with line minimization; concept of conjugate gradients (directions, actually)
  Reading: NR 10.6
- 04/23 Lab 8: Spectral solution to Poisson's equation in octave

Contents

1 Overview **2**

2 Definition of basic variables: "setup.m" **2**
2.1 Indexing 2
2.2 Sampling points and reciprocal lattice vectors 4

3 Charge distribution: "poisson.m" **5**

4 Operators **5**
4.1 Global variables 6
4.2 O() 6
4.3 L() 7
4.4 Linv() 7
4.5 cI() 8
4.6 cJ() 8

5 Final solution to Poisson's equation: "poisson.m" **9**

A "slice.m" **10**

B "view.m" **11**

C "poisson.m" **12**

D "O.m" **13**

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,

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

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

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.

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
```