Cornell University

Department of Physics

Phys 480/680, Astro 690 May 4, 2000

Computational Physics, Spring 2000

Homework Assignment # 11

(Due Tuesday, May 9 at 8:40 am in Lab.)

Agenda and readings for the remainder of the term

All readings from *Numerical Recipes: The Art of Scientific Computing*, 2nd edition (in Fortran *or* C).

- – Lab10/HW#11, 05/04: DFT solutions of molecules
- – Lec16, 05/02: Theory of conjugate gradients; FFT

Goal for this assignment: Compute the bond-length of the H_2 molecule.

Contents

1 Infrastructure **1**
- 1.1 "physics2.c" . . . 1
- 1.2 get_grad_and_E() . . . 2
- 1.3 conj() . . . 2
- 1.4 excVWN(), excpVWN() . . . 3
- 1.5 exc(), excp() . . . 3

2 Coulomb potential (-Z/r) **3**
- 2.1 evalG() . . . 4
- 2.2 Nuc1(), Nuc2() . . . 4
- 2.3 Debugging . . . 5

3 Kohn-Sham equations: full implementation of density functional theory **5**
- 3.1 Finite-difference test . . . 5
- 3.2 Kohn-Sham equations . . . 6
  - 3.2.1 Debugging . . . 7

4 *Ab initio* calculation of H_2 bond length! **7**

1 Infrastructure

1.1 "physics2.c"

To complete this assignment we will need to evaluate the interaction energy of nuclei in a periodic crystal. Outside of a small Gaussian distribution of charge, the potential is identical to that of a point charge. Thus, we can calculate the energy of a periodic array of charges by solving for the electrostatic energy of Gaussian distributions as we did in Homework Assignment # 9.

To do this, copy your program physics.c into a new program physics2.c and make the following modifications.

- Of the two Gaussians, make one centered at the origin (as we had done before) and the other centered at a point $r = (x, 0, 0)$ along the x-axis. You can accomplish this with the following code fragment in the main program along with an new function GaussR2() modeled appropriately on your original GaussR(). For this exercise, you may "hardwire" the value of the parameter x into the GaussR2() routine.

```
                .
                .
rho1.evalr(GaussR,sigma1);
rho2.evalr(GaussR2,sigma2);
                .
                .
```

- Because these Gaussian charges are to represent the two nuclei in the hydrogen molecule, be sure to *add* them together (not subtract) to form the total, rho.
- Because we are interested only in the energy of interaction of the two Gaussians and not their self energy, you should printout the value of U-Uself where Uself$\equiv 1/(2\sqrt{\pi})(1/\sigma_1 + 1\sigma_2)$.
- You may remove the analytic tests of phi.

Debugging: You should find that at a separation of x=4.00 and widths sigma1=sigma2=0.5 you get results which are the same to within about 0.02 millihartree for (n=32, dx=0.5) as you do for (n=64, dx=0.25). More importantly, your result should change by only about 1 millihartree when you change to sigma1=sigma2=0.25 and evaluate at (n=64, dx=0.25). All of these results should be about -0.094 hartree.

Finally, using the parameters sigma1=sigma2=0.25, and (n=64, dx=0.25), calculate and save in a file for later the values of U-Uself for nuclei at separation x=0.50, 1.00, 1.25, 1.50, 1.75, 2.00, 4.00 and 6.00 bohr.

1.2 get_grad_and_E()

```
pvector get_grad_and_E(double& E,const pvector& u,const pvector& Vtilde);
```

Return value:

- pvector which contains the gradient of the energy function

Output:

- E: value of the energy function.

Input:

- u: expansion coefficients of *non*-normalized wave function
- Vtilde: expansion coefficients of effective representation of the potential, so that the potential energy is POT=u^(Idag(Vtilde*I(u))).

To make the transition to the Kohn-Sham equations easier, it helps to isolate the algorithm (conjugate-gradients) from the physics (computation of E and grad). To do this, make a copy of your "sch.C" program called "sch2.C" and add a function of the above prototype to it.

Debugging: Write "sch2.C" so that it calls get_grad_and_E() to do *all* of the energy and gradient calculations, and verify that it produces *exactly* the same output as your original "sch.C."

1.3 conj()

```
friend pvector conj(const pvector& in1);
```

Return value:

- a pvector identical to in1, but with all of its dat elements complex-conjugated

Input:

- in1: input pvector

Add a function of the above prototype to the pvector class. You may find the function "pvector operator*(Complex c,const pvector& in1)" a useful example from which to work.

1.4 excVWN(), excpVWN()

```
inline double excVWN(double rs)
inline double excpVWN(double rs)
```

Return values:

- excVWN: value of VWN form for $\epsilon_{xc}(r_s)$ from HW#6
- excpVWN: value of Perdew-Zunger form for $\epsilon'_{xc}(r_s) \equiv d\epsilon_{xc}/dr_s$ from HW#6

Input:

- rs: value of $r_s \equiv \left(\frac{3}{4\pi}\right)^{1/3} n^{-1/3}$

Add two functions of the above prototypes to program sch2.C. This should involve simply cutting and pasting the exc() and excp() functions from your atomic code and renaming them excVWN() and excpVWN().

Debugging: These routines will be debugged in Part 3.

1.5 exc(), excp()

```
friend pvector exc(const pvector& in1);
friend pvector excp(const pvector& in1);
```

Return values:

- exc: pvector, each of whose data elements is the result of the application of excVWN() to the components of in1: out.dat[i][j][k]=excVWN(rs), where rs=pow(((4*pi)/3.)*in1.dat[i][j][k], -1./3.) .
- excp: pvector, each of whose data elements is $d\epsilon_{xc}/dn$ for each element of in1, out.dat[i][j][k]==excpVWN(rs)*(-rs/in1.dat[i][j][k]/3.) .

Input:

- in1: input pvector (containing the charge density)

Debugging: These routines will be debugged in Part 3.

2 Coulomb potential (-Z/r)

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

```
phinuc=I(Linv((-4*pi)*O(J(rhonuc))));
```

and our the definition of Vtilde from HW#10,

```
Vtidle=Jdag(O(J(-phinuc)));
```

where the extra '-' in front of phinuc comes from the fact that the charge on the electron is (-e): the potential energy of an electron at $\vec{x}$ is $V(\vec{x}) = (-e)\phi_{nuc}(\vec{x})$. Combining the above two results gives

```
Vtilde=Jdag(O(Linv((4*pi)*O(J(rhonuc)))));
```

where we have used the fact that J(I(x))=x.

Only difficulty with this approach is that rhonuc represents a point charge, whose density is zero everywhere but at a single point where the density is infinite. We can, however, find the expansion coefficients $[\mathrm{J(rhonuc)}]_{\vec{G}} \equiv \hat{\rho}_{\vec{G}}$ mathematically. To do this, we note that

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

so that

$$\begin{aligned} \int e^{-i\vec{G}\cdot\vec{x}} \rho(\vec{x})\, dV &= \int e^{-i\vec{G}\cdot\vec{x}} \sum_{\vec{G}'} \hat{\rho}_{\vec{G}'} e^{i\vec{G}'\cdot\vec{x}}\, dV \\ &= V\hat{\rho}_{\vec{G}} \end{aligned}$$

where V is the volume of the cell. Therefore,

$$\hat{\rho}_{\vec{G}} = \frac{1}{V} \int e^{-i\vec{G}\cdot\vec{x}} \rho(\vec{x})\, dV.$$

In our case, $\rho(\vec{x})$ is a point charge centered at the location of the nucleus whose total integral is Z, the charge on the nucleus. If the nucleus is centered at point $\vec{r}_{nuc}$, then we have simply

$$\hat{\rho}_{\vec{G}} = \frac{Z}{V} e^{-i\vec{G}\cdot\vec{r}_{nuc}}.$$

2.1 evalG()

```
void evalG(Complex (*func)(d3vec,double),double dparam)
```

Input:

- func: function to be evaluated for all G-vectors
- dparam: parameter to be passed to the function

Output:

- dat[i][j][k]: set to values of func(G,dparam) evaluated on coefficient-space point G corresponding to [i][j][k], taking proper account of aliasing and setting the values of dat for the Nyquist frequencies (where i=N.x/2, j=N.y/2 or k=N.z/2) to zero. (This is because G isn't clearly defined at these Nyquist frequencies.)

To be able to compute J(rhonuc), add a function of the above prototype to your definition of the pvector class.

Hint: The general structure of this function is the same as the evalr() function which you wrote for HW#9, but, rather than passing

2.2 Nuc1(), Nuc2()

```
Complex Nuc1(d3vec G,double norm)
Complex Nuc2(d3vec G,double norm)
```

Return values:

- Nuc1: norm*exp(-i*(G.x*r1x+G.y*r1y+G.z*r1z)), where (r1x,r1y,r1z)=(0,0,0)
- Nuc2: norm*exp(-i*(G.x*r2x+G.y*r2y+G.z*r2z)), where (r2x,r2y,r2z)=(1.5,0,0)

Input:

- G: G vector
- norm: input parameter (usually will be set to Z/Vol, where Vol is the volume of the cell and Z is the charge on the nucleus.

Provide the functions of the above two prototypes to be able to initialize J(rhonuc) for nuclei at two different locations.

2.3 Debugging

Change the initialization parts of your sch2.C code to now initialize the potential to that of a hydrogen atom at the origin:

```
// Initialize for H atom
pvector Jrhonuc(i3vec(n,n,n),d3vec(d,d,d));
Jrhonuc.evalG(Nuc1,1./Vol);
Vtilde=Jdag(O(Linv((4*pi)*O(Jrhonuc))));
```

Have the program run Itmx=25 iterations of preconditioned conjugate gradients on a grid of size n=32 and spacing dx=0.5. At the end, have the program normalize u and output the result in real space:

```
u=u*(1./u.dat[0][0][0]);
u=u*(1./sqrt(u^O(u)));
pvector psi=I(u);
psi.printr("u");
```

(The first of these statements makes picks the complex phase of the wave function to be real).

Make a comparison plot of the wave function in file "u" (first column is r, second column is psi, third column is the imaginary part of psi which should be very close to zero) with the analytic solution for the 1s state, $e^{-r}/\sqrt{\pi}$. You should find a quite close match, except for the point right on top of the nucleus (r=0). *Show the plot to the grader to get credit for this problem!*

Note: I find an energy of about -0.321. You should not worry that this isn't -0.5 because Poisson's equation only gives us the potential (and the the electron's energy) up to an arbitrary constant.

3 Kohn-Sham equations: full implementation of density functional theory

3.1 Finite-difference test

To verify your implementation of the Kohn-Sham equations, in the next section you will check with finite differences that your code for the gradient of the energy function is correct.

Not infrequently, when implementing a complicated test like this for debugging purposes, one spends much time looking for a bug in perfectly working code because of a bug in the test *itself*! Thus, it is good programming practice to fist debug the test by running it with code which you know to be working.

To do this for our finite difference test, implement the following code fragment. It is important that the test be put in after u is initialized to random values but before it it normalized so that the gradient formula is tested for non-normalized vectors (the usual case).

```
u.eval(rnd); // Initialize u to random elements
                .
                .
// Finite diff test on gradient:
dir.eval(rnd); // Random elements
{
```

```
double oE,E;
ograd=get_grad_and_E(oE,u,Vtilde);
for (double alpha=1; alpha>=1e-8; alpha/=10) {
  grad=get_grad_and_E(E,u+alpha*dir,Vtilde);
  printf("Gradient test: %20.16f\n",(E-oE)/real(dir^ograd)/alpha);
  printf("    (precision %20.16f)\n",fabs((1e-15*E)/(E-oE)));
}
}
                .
                .
u=u*(1./sqrt(u^O(u))); // Normalize u for starting
                .
                .
// Proceed to rest of preconditioned conjugate-gradient minimization code...
```

When you run this, you should find (just as you did with the test of excp earlier in the term) that the "Gradient test" approaches a value of 1, adding one digit for each new iteration. This pattern should continue for a number of iterations until rounding errors become significant. Note that, in terms of precision, you can only trust those digits in the gradient test which are zero on the "precision" line.

Hint: You will probably find that the formula for the gradient given on HW#10 was missing a factor of 2. This factor canceled out of all of our calculations on that problem set, but you should now put it in the gradient formula so that the gradient test actually does approach 1.

3.2 Kohn-Sham equations

For a single orbital containing f elections, the full density-functional-theory energy function, including the Hartree and exchange-correlation energies, is

$$E = f\frac{\int -\frac{1}{2}u^*(\vec{x})\nabla^2 u(\vec{x})\,dV}{\int u^*(\vec{x})u(\vec{x})\,dV} + \int n(\vec{x})\left(V(\vec{x}) + \frac{1}{2}\phi(\vec{x}) + \epsilon_{xc}(n(\vec{x}))\right)\,dV,$$

where

$$n(\vec{x}) \equiv \frac{f\psi^*(\vec{x})\psi(\vec{x})}{\int u^*(\vec{x})u(\vec{x})\,dV},$$

$\phi(\vec{x})$ is the solution of Poisson's equation, and $\epsilon_{xc}(n)$ is the exchange-correlation energy function.

In our matrix language, this is

$$E = -\frac{1}{2}fu^\dagger Lu/D + n^\dagger\tilde{V} + \frac{1}{2}(Jn)^\dagger O\phi + (Jn)^\dagger OJ\epsilon_{xc}(n),$$

where $D \equiv u^\dagger Ou$,

$$n \equiv f\text{conj}\,(I(u)) * I(u)$$

with the '*' operator defined as point-wise multiplication of pvectors as in HW#10, conj(v) representing complex conjugation of all components the vector v, and $\epsilon_{xc}(n)$ representing a pvector each of whose components is the application of the $\epsilon()$ function applied to the corresponding component of the pvector n. Finally, $\tilde{V} \equiv J^\dagger OJV$, and we can determine ϕ from our previous algorithm for the solution of Poisson's equation, $\phi = L^{-1}(-4\pi OJn)$.

In the same language, the gradient of E is

$$\nabla_u E = 2f\frac{1}{D}\left(Hu - \frac{N}{D}Ou\right).$$

Here, $N \equiv u^\dagger Hu$ where

$$Hu \equiv -\frac{1}{2}Lu + I^\dagger(V * Iu),$$

where

$$V \equiv \tilde{V} + J^\dagger O\phi + J^\dagger O J \epsilon_{xc}(n) + \epsilon'_{xc}(n) * J^\dagger O J n.$$

3.2.1 Debugging

Make a new copy of "sch2.C" called "kohnsham.C". Implement the new equations for E and grad in your get_grad_and_E() function in the "kohnsham.C" program.

Debugging: Running your newly developed density-functional theory program with f=2 electrons in the 1s state of Hydrogen, verify that it passes the finite-difference gradient test. Note that this depends on your get_grad_and_E() function as well as your exc() and excp() functions. If you fail the finite-difference test, you can isolate the bug by separately turning on and off the Hartree and exchange-correlation terms in both the energy and gradient terms at the same time.

4 *Ab initio* calculation of H_2 bond length!

Use the following code fragment to initialize Vtilde for a molecule (two nuclei) instead of just one:

```
// Initialize for H2 molecule
pvector Jrhonuc1(i3vec(n,n,n),d3vec(d,d,d));
Jrhonuc1.evalG(Nuc1,1./Vol);
pvector Jrhonuc2(i3vec(n,n,n),d3vec(d,d,d));
Jrhonuc2.evalG(Nuc2,1./Vol);
Vtilde=Jdag(O(Linv((4*pi)*O(Jrhonuc1+Jrhonuc2))));
```

Because of your coding for Nuc2, this represents a hydrogen molecule with the nuclei separated by 1.5 bohr. Running at n=32 and spacing dx=0.5 you should find (after about 25 iterations) a total energy for the electrons of about -1.443 237 hartree. Adding this to the result for the nuclear energy U in Part 1.1 (0.313 589 hartree) gives a total energy for the molecule of E=-1.129 648 hartree.

Using this approach, compute the total energy of the H_2 molecule for the separations 0.50, 1.00, 1.25, 1.50, 1.75, 2.00, 4.00 and 6.00 bohr.

Make a plot of your results. From the plot, determine the bond-length of H_2 and compare with the experimental result from a standard reference such as the CRC handbook. *Show the plot to the grader to get credit for this problem!*