Cornell University

Department of Physics

Phys 480/680, Astro 690 March 12, 2004

Computational Physics, Spring 2002

Homework Assignment # 5

(Due Thu, March 18)

Agenda and readings for the week of March 5:

Goal: Building upon the components you have built, you will perform your own full density-functional-theory calculations of atoms. You will reproduce the accepted values to at least as much precision as published by the US National Institute of Standards and Technology, the micro-hartree level!!!

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:

- 03/02 NO LECTURE
- 03/04 Lec 11: Numerical interpolation; exchange and correlation energy functionals; solution of many non-linear equations in many unknowns via self-consistent iteration and mixing.
  Readings: NR 3.0-3.2, 9.6-9.7; Sections "Introduction," "Procedure" (you can skip the relativistic part), "Results," and "The LDA approximation," from http://physics.nist.gov/PhysRefData/DFTdata/contents.html
- 05/05 NO LAB
- 03/09 Lec 12: Mixing as a general algorithm; analysis of convergence for mixing, condition numbers; sophisticated algorithms, Netwon-Raphson, Broyden; need for an alternative for large systems.
  Reading: NR 9.6–9.7
- 03/11 Lec 13: Line optimization; modified Broyden method; Introduction to Numerical linear algebra; benefits of LAPACK; solution of linear equations through LU decomposition.
  Reading: NR 2.0, 2.3
- 03/12 Lab 4: Convergence of energies; Energiess of simple harmonic oscillator; "hot-wired" Hartree solver for atoms!

Preview of coming week:

- 03/16 Lec 14: Crout's algoritm for LU decomposition
  Reading: NR 2.0, 2.3
- 03/18 Lec 15: Optimization: Amdahl's Law; inner loop tricks; Solution of Poisson's equation in d¿1 dimensions; expressive software/DFT++
- 03/19 Lab 5: Full, self-consistent solutions for density functional theory of atoms; using gprof; avoiding malloc; inlining optimizations.

Contents

1 Exchange-correlation energy **3**
- 1.1 Review of background . . . 3
- 1.2 "excPZ" and "excpPZ" . . . 4
  - 1.2.1 Debugging . . . 4
- 1.3 "getVxc" and "getDepsxc" . . . 5
  - 1.3.1 Debugging . . . 5
- 1.4 Vosko, Wilk, and Nusair form for exchange-correlation . . . 5
  - 1.4.1 Debugging . . . 5

2 Putting it all together **5**
- 2.1 "test_atoms.c" . . . 6
  - 2.1.1 Debugging . . . 6
- 2.2 "test_atomsC.c": Carbon . . . 7
- 2.3 "test_atomsX.c": an atom of your very own . . . 7

3 Packaging **8**
- 3.1 "getg" . . . 8
  - 3.1.1 Debugging . . . 8
- 3.2 "ludcmpp480" and "lubksbp480" . . . 8
  - 3.2.1 Debugging . . . 9

4 Initial Performance **9**
- 4.1 Convergence Rate . . . 9
- 4.2 Run time . . . 9

5 Broyden Solver **9**
- 5.1 Bookkeeping . . . 9
  - 5.1.1 Debugging . . . 10
- 5.2 Improved initial guess . . . 10
- 5.3 "broyden" . . . 11
  - 5.3.1 Matrix G and vector u . . . 11
  - 5.3.2 Solution of **G** b = u . . . 12
  - 5.3.3 Formation of final Rho[it+1] . . . 12
- 5.4 Convergence rate . . . 12

A "test_excpPZ.c" **13**

B "test_getxc.c" **14**

C VMN forms for exchange correlation **17**

D "test_getg.c" **18**

E "test_lu.c" **20**

1 Exchange-correlation energy

1.1 Review of background

Within the local density approximation to density functional theory, the total energy of an electronic system is

$$E_{tot} = T + E_{el-nuc} + E_{Hartree} + E_{xc},$$

with

$$\begin{aligned} T &\equiv \sum_i f_i \int \psi_i^*(\vec{x}) \left(-\frac{1}{2}\nabla^2 \psi_i(\vec{x}) \right) \, dV \\ E_{el-nuc} &\equiv \int V_{nuc}(\vec{x}) n(\vec{x}) \, dV \\ E_{Hartree} &\equiv \frac{1}{2} \int \phi(\vec{x}) n(\vec{x}) \, dV \\ E_{xc} &\equiv \int f_{xc}(n(\vec{x})) \, dV, \end{aligned}$$

where

$$n(\vec{x}) \equiv \sum_i f_i |\psi_i(\vec{x})|^2$$

and

$$\nabla^2 \phi(\vec{x}) = -4\pi n(\vec{x}).$$

Minimizing the above total energy is equivalent to solving the Schrödinger equation

$$-\frac{1}{2}\nabla^2 \psi_i(\vec{x}) + V(\vec{x})\psi_i(\vec{x}) = E_i \psi_i(\vec{x}),$$

with

$$V(\vec{x}) \equiv V_{nuc}(\vec{x}) + \phi(\vec{x}) + V_{xc}(\vec{x}),$$

where

$$V_{xc}(\vec{x}) \equiv f'_{xc}(n(\vec{x})).$$

In terms of the electron energies E_i above, an alternative form for the total energy is

$$E_{tot} = \sum_i f_i E_i - E_{Hatree} + \int \left[f_{xc}(n(\vec{x})) - f'_{xc}(n(\vec{x}))n(\vec{x}) \right] \, dV.$$

This first problem of this homework develops the software to compute the exchange-correlation terms in the above equations. The literature always gives $f_{xc}(n)$ in the form $f_{xc}(n) = \epsilon_{xc}(n)n$. In terms of ϵ_{xc}, the quantities we shall need are

$$\begin{aligned} V_{xc}(\vec{x}) &\equiv f'_{xc}(n(\vec{x})) \\ &= \epsilon'_{xc}(n(\vec{x}))n(\vec{x}) + \epsilon_{xc}(n(\vec{x})) \end{aligned} \tag{1}$$

and the exchange-correlation correction energy,

$$\begin{aligned} \Delta E_{xc} &\equiv \int \left[f_{xc}(n(\vec{x})) - f'_{xc}(n(\vec{x}))n(\vec{x}) \right] \, dV \\ &\equiv \int \Delta\epsilon_{xc}(n(\vec{x}))n(\vec{x}) \, dV, \end{aligned}$$

α	$\frac{3}{4}\left(\frac{3}{2\pi}\right)^{2/3}$
a	0.0311
b	-0.0480
c	0.0020
d	-0.0116
A	-0.1423
B	1.0529
C	0.3334

Table 1: Table of coefficients for Perdew-Zunger form (3) of the local exchange correlation functional.

where

$$\begin{aligned}\Delta\epsilon_{xc}(n(\vec{x})) &\equiv \frac{1}{n(\vec{x})}\left[f_{xc}(n(\vec{x})) - f'_{xc}(n(\vec{x}))n(\vec{x})\right] \\ &= \epsilon_{xc}(n(\vec{x})) - f'_{xc}(n(\vec{x})) \\ &= \epsilon_{xc}(n(\vec{x})) - (\epsilon'_{xc}(n(\vec{x}))n(\vec{x}) + \epsilon_{xc}(n(\vec{x}))) \\ &= -\epsilon'_{xc}(n(\vec{x}))n(\vec{x}) \end{aligned} \tag{2}$$

1.2 "excPZ" and "excpPZ"

`double excPZ(double rs)`

`double excpPZ(double rs)`

Return values:

- exc: value of Perdew-Zunger form for $\epsilon_{xc}(r_s)$
- excp: value of Perdew-Zunger form for $\epsilon'_{xc}(r_s) \equiv d\epsilon_{xc}/dr_s$

Input:

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

As described in lecture, the literature specifies the exchange-correlation energy *per electron* $\epsilon_{xc}\left(r_s(n)\right)$ as a function of the average distance between electrons $r_s(n) \equiv \left(\frac{3}{4\pi}\right)^{1/3} n^{-1/3}$. Perdew and Zunger (*Phys. Rev.* **B 23**, 5048 (1981).) gave one of the earlier, simplest forms for ϵ_{xc},

$$\epsilon_{xc} = -\frac{\alpha}{r_s} + \begin{cases} a\ln r_s + b + cr_s \ln r_s + dr_s & r_s < 1 \\ \frac{A}{1+B\sqrt{r_s}+Cr_s} & r_s \geq 1 \end{cases}, \tag{3}$$

where Table 1 gives the values of the constant coefficients.

Write routines of the two above two prototypes which compute $\epsilon_{xc}(r_s)$ and $\epsilon'_{xc}(r_s) \equiv d\epsilon_x c/dr_s$ for the Perdew-Zunger form.

1.2.1 Debugging

Even a tiny analytic error in taking the derivative $\epsilon'_{xc}(r_s)$ will result in completely incorrect results for your calculations. (There is no "partial" credit for analytic errors when doing numerical calculations!)

To make certain that you have taken the derivative correctly *and coded it properly*, run your routines exc() and excp() with the main program in Appendix A. When your derivative is right, your results should approach the value one with a very regular digit-shifting pattern!

1.3 "getVxc" and "getDepsxc"

`getVxc(double Vxc[], double Rho[], double r[], double dr[], int N)`

`getDepsxc(double Depsxc[], double Rho[], double r[], double dr[], int N)`

Output:

- Vxc[k]: $V_{xc}(r[k])$ for all grid points r[k]
- Depsxc[k]: $\Delta\epsilon_{xc}(n(r[k])) \equiv -\epsilon'_{xc}(n(\vec{x}))n(\vec{x})$ for all grid points r[k]

Input:

- Rho[k]: $4\pi r^2 n(r[k])$ for all grid points r[k]
- r, dr, N: standard grid information

Write routines of the above prototypes which (a) loop over all grid points k, (b) convert our variable Rho[] to the true value of the density, (c) evaluate rs, (d) call upon excPZ and excpPZ, and (e) evaluate V_{xc} and $\Delta\epsilon_{xc}$ according to Eqs. (1,2). There are two special cases to consider to avoid NaNs corrupting your data. At the origin (k=0), n=Rho[0]/r[0] gives NaN, and, at infinity, (k=N), Rho=0. Thus, if either k=0 *or* Rho<SMALL, your routine should set Vxc=0 and Depsxc=0. (Recall that we #define SMALL 1e-300.)

Hint: In Eqs. (1,2) $\epsilon'_{xc}(n)$ refers to $d\epsilon_{xc}/dn \equiv (d\epsilon_{xc}/dr_s)(dr_s/dn)$, whereas "excpPZ" returns $\epsilon'_{xc}(r_s) \equiv d\epsilon_{xc}/dr_s$. You therefore must multiply the return value of "excpPZ" by $(dr_s/dn) = (-r_s/(3n))$ to evaluate $\epsilon'_{xc}(n)$ in Eqs. (1,2).

1.3.1 Debugging

Run your routines with the main program in Appendix B. This program works with hydrogen. It first makes phi and Vxc for a zero charge density (testing that you have handled Rho[]=0 properly) and creates -1/r+phi+Vxc, which *should* be just -1/r because Rho=0. It then solves for the 1s state. From the density of the 1s state, it then computes Vxc and Depsxc, and finally $\int V_{xc} n\, dV$ and $\Delta E_{xc} = \int \Delta\epsilon_{xc} n\, dV$. You should find that your results converge to -0.3316096642 and 0.0772489584, respectively, for these quantities.

1.4 Vosko, Wilk, and Nusair form for exchange-correlation

The US National Institute of Standards and Technology (NIST) maintains a web site with density functional calculations of all atoms in the periodic table. So that we may compare our results directly with theirs, we must use exactly the same form for exchange-correlation. NIST employs the form suggested by Vosko, Wilk, and Nusair (VWN). Appendix C provides the routines exc() and excp() for this form. You should use these in your code, replacing all calls to excPZ() and excpPZ() with calls to exc(), and excp() respectively.

1.4.1 Debugging

Running "test_getxc" with the VWN form, your results should now approach the slightly different values of -0.3315027563 and 0.0773497767, respectively.

2 Putting it all together

Finally, you are ready to perform density functional calculations of atoms! For this problem set, we will use one of the simplest (but least efficient) approaches to attain self-consistency, "charge mixing".

2.1 "test_atoms.c"

Write a main program which attains self-consistency with the algorithm,

- Initialize:
  1. Rho_0: Rho[k]=0 for all k
- Iterate:
  1. get phi and Vxc from Rho_n
  2. compute V[k]=-1/r[k]+phi[k]+Vxc[k] for all k
  3. get E's from V
  4. get Psi's from E's and V
  5. get Rhonew from Psi's
  6. Rho_{n+1}[k]=$(1-\alpha)$ Rho_n[k]+ α Rhonew[k] for all k

where α is any value between 0 and 1.

Note that if $\alpha = 1$, Rho_{n+1}=Rhonew, and we are just putting the output charge density back in as the input, and iterating. This process, however, often is unstable because the electron density Rho changes too quickly. Taking $\alpha < 1$ helps calm this instability by changing Rho more slowly. For instance, for $\alpha = 0.25$ (which seems to work in most cases), we are "mixing" 75% of the previous estimate of Rho with 25% of the new estimate.

Finally, to monitor the progress of your program, you should have it print out the total energy at each iteration. A suitable code fragment for computing this is

```
                    .
                    .
/* Compute and output total energy */

/* Get correction to sum of electron energies */
getDepsxc(Depsxc,Rho,r,dr,N);
for (k=0; k<=N; k++)
  integrand[k]=(-0.5*phi[k]+Depsxc[k])*Rho[k];
Etot=simpint(integrand,r,dr,N);

/* Add on the sum of the electron energies times the occupancies */
for (l=0; l<=lmax; l++)
  for (n=0; n<=nmax[l]; n++)
    Etot+=F[l][n]*E[l][n];

printf("Etot: %20.15f\n",Etot);
                    .
                    .
```

Note that, because, your solution is converging (hopefully!) and all the quantities eventually stop changing, it doesn't really matter if you use Rho[] or Rhonew[] or if you compute phi[] from Rho[] or Rhonew[] in the above code fragment.

2.1.1 Debugging

Run your test_atoms.c for hydrogen (which has 1 1s electron) with (a) the VWN form for the exchange-correlation functional, (b) N=4000 intervals, (c) the change of variables $r(x) = 1/(1-x)-1-x-x^2-x^3$, and (d) $\alpha = 0.25$. After about about 50 iterations, you should converge to a value of about -0.445671. Compare your result with the "official" result at

```
http://math.nist.gov/DFTdata/atomdata/tables/01H.html
```

(Note that we are using the *local density approximation* (LDA) to density functional theory.)

Hint: The following code fragment gives an example of a suitable initialization of our arrays for hydrogen:

```
Z=1.; /* Charge on nucleus */
lmax=0; /* Only s states are filled in H */

nmax=ivector(0,lmax); /* Make space for nmax[] */
nmax[0]=0; /* For s states, we only need the first, 1s, with zero nodes */

/* Find nmaxmax=max{nmax} */
nmaxmax=0;
for (l=0; l<=lmax; l++)
  if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

F=dmatrix(0,lmax,0,nmaxmax);
F[0][0]=1.; /* There is only one electron in the 1s for H */
```

2.2 "test_atomsC.c": Carbon

Run your code with the same α, number of intervals N, change of variables $r(x)$ and form for exchange and correlation as in Problem 2.1.1, but now compute the total energy of Carbon. A suitable initialization for Carbon, which has the electronic structure $1s^2 2s^2 2p^2$, is

```
/* Specs for C */
Z=6.;
lmax=1;

nmax=ivector(0,lmax);
nmax[0]=1;
nmax[1]=0;

nmaxmax=0;
for (l=0; l<=lmax; l++)
  if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

F=dmatrix(0,lmax,0,nmaxmax);
F[0][0]=2.; /* 1s */
F[0][1]=2.; /* 2s */
F[1][0]=2.; /* 2p */
```

Again, compare your results with the official results

```
http://math.nist.gov/DFTdata/atomdata/tables
```

2.3 "test_atomsX.c": an atom of your very own

Locate the element on the periodic table whose number (Z value) is equal to the last two digits of your social security number or Cornell ID. (If this makes Z>92, pick your favorite element.) Compute the total energy of your element and compare against the official results.

Hint: You may wish to consult a periodic table to first determine the electronic configuration ($1s^2 2s^2 3p^6 \ldots$) for your element.

3 Packaging

3.1 "getg"

```
double getg(double g[], double Rho[],
 double Z, int lmax, int nmax[], int nmaxmax, double **F,
 double r[], double dr[], int N)
```

Return value:

- Etot: total energy of atom

Output:

- g[k]: $\vec{f}(\vec{n}) - \vec{n}$ at the grid points r[k]

Input:

- Rho[k]: $4\pi r^2 n(r[k])$ for all grid points r[k]
- Z: nuclear charge
- lmax: maximum angular momentum to consider (l=0..lmax)
- nmax[l]: maximum number of nodes sought for angular momentum l
- nmaxmax: maximum of all nmax[l]
- F[l][n]: shell occupancies ($F_{n\ell} \equiv (2\ell+1) f_{n\ell}$)
- r[], dr[], N: standard grid information

Using the pieces of your main routine from the Homework #6, construct a subroutine of the above prototype, which takes Rho[] as input, and then computes from it the total potential V=Vnuc+phi+Vxc, the wave functions Psi[][][], and the density corresponding to those wave functions, Rhonew[]. Finally, the routine should put the difference Rhonew[]-Rho[] in g[] and return the value Etot.

Hint: Note that getg() will have to allocate space for many variables such as Vxc[], phi[], Psi[][][]. You will find the *Numerical Recipes* utilities dvector, dmatrix, d3tensor very useful for this. Now that you are allocating this space in a subroutine, it is *essential* that you *deallocate* everything before you exit the routine.

3.1.1 Debugging

The main program in Appendix D solves for the total energy of the *oxygen* atom using your getg() routine. It initializes with zero density, Rho[]=0, and then iterates Rho_{i+1}[]=Rho[]+alpha*g[], as discussed in class, with alpha=0.25. This program should converge to a value of -74.473076 hartree in about 30 iterations.

3.2 "ludcmpp480" and "lubksbp480"

```
void ludcmpp480(double **a, int n, int *indx, double *d);

void lubksbp480(double **a, int n, int *indx, double b[]);
```

Output:

- a[n][m]: LU decomposition of A ("permuted")
- n: size of matrix
- index[n]: permutation of LU decomposition
- d: sign of determinant of A

Input:

- a[n][m]: matrix elements, A_{nm}
- n: size of matrix

As discussed in lecture, our new algorithm for solving large systems of non-linear equations requires us to solve small systems of linear equations of the form $\mathbf{G}\vec{b} = \vec{u}$. *Numerical Recipes* provides two routines, "ludcmp" and "lubksb", which together solve such equations. Generalize them to double precision by making the replacements "float"→"double" and "vector"→"dvector", and rename them with our standard p480 extension.

3.2.1 Debugging

Run your "ludcmpp480" and "lubksbp480" routines with the main program in Appendix E. This test builds a 6*times*6 linear system, uses the LU routines to solve it, and then tests the result. You may use this main program as a model of how to use the LU routines to solve the linear system $\mathbf{G}\vec{b} = \vec{u}$ in 5.3.2.

4 Initial Performance

The focus of this problem set is enhancing the performance of your atomic solver. To get a sense of the impact of improvements which we make, please gather the following information.

4.1 Convergence Rate

As each iteration, "test_getg" prints both the total energy and the difference of this total from the final converged result. (If your code converges to a slightly different result, simply change the print statement to subtract your final value.)

So that we may assess the effectiveness of the algorithm, make plots to show the grader and to bring to lab of (a) the error as a function of iteration number and (b) the log (preferably, base 10) of the error as a function of iteration number.

4.2 Run time

Write down to show to the grader and at lab (a) how many iterations and (b) how many seconds "test_it.c" requires to reach convergence to all twelve digits printed to the output. (For our code and machine, this was 53 iterations and 26 seconds.)

5 Broyden Solver

You are now ready to use a better algorithm to speed your code by an order of magnitude! Implementing the new algorithm requires attention to detail, but is well worth it in the end.

Programming strategy: We are also now at a stage in our software project where our programming strategy must change. This is because our software (Appendix D) is now fully functional, albeit not optimized. Rather than building up the working pieces one at a time and carefully verifying each one, we now can make incremental changes and rerun the code with each change to make sure that it continues to function properly. For the exercises below, please begin with the main program in Appendix D and make the appropriate modifications to it, calling the new program "test_it".

5.1 Bookkeeping

Important note: The *Numerical Recipes* routines for linear systems are all "one based" (all array and matrix elements begin with one, not zero). The simplest way to work with this is to build all of our linear algebra objects (such as **G**) also as one based and to keep our physics grid objects (such as phi) as zero

based. This mixed strategy allows us to use the *Numerical Recipes* routines with minimal modification and requires no changes to our preexisting code.

The new algorithm requires that we keep track of the charge densities Rho_i and their residuals g_i from all previous iterations. Accordingly, we shall require two dimensional arrays (dmatrix's) for their storage in the main program.

To do this, modify the main program to store the Rho_i and g_i as it goes along, producing the new program "test_it". An appropriate code fragment is

```
#define Itmx 100
            .
            .
  Rho=dmatrix(1,Itmx+1,0,Nmx);
  g=dmatrix(1,Itmx+1,0,Nmx);
            .
            .
  for (it=1; it<=Itmx; it++) {
    Etot=getg(g[it],Rho[it],Z,lmax,nmax,nmaxmax,F,r,dr,N);
    printf("Iter:%4d  Etot:%20.12f  Error:%20.12f\n",
           it,Etot,fabs(Etot-(-74.473076803203738)));

    for (k=0; k<=N; k++)
      Rho[it+1][k]=Rho[it][k]+alpha*g[it][k];
  }
            .
            .
  free_dmatrix(Rho,1,Itmx+1,0,Nmx);
  free_dmatrix(g,1,Itmx+1,0,Nmx);
```

Note how we begin the matrices Rho and g with it=1 so that we can conform to the one-based routines from *Numerical Recipes*. Also, because the final iteration (Itmx) needs space for Rho[it+1], we declare space for 1..Itmx+1 in the dmatrix statements.

5.1.1 Debugging

Verify that "test_it" produces output identical to "test_getg".

5.2 Improved initial guess

The basis of the Broyden algorithm is linearization of the equations, which requires that we start not too far from the initial solution. Presently we begin with the very poor starting point Rho[k]=0. To improve upon this, we will start the Broyden solver with the density Rho after five iterations of our present charge-mixing algorithm. To accomplish this, *add* the following loop to "test_it" just before the main iteration loop (which starts with "for (it=1; it¡=Itmx; it++)"):

```
  for (it=1; it<=5; it++) {
    Etot=getg(g[1],Rho[1],Z,lmax,nmax,nmaxmax,F,r,dr,N);
    printf("Iter:%4d  Etot:%20.12f  Error:%20.12f\n",
           it,Etot,fabs(Etot-(-74.473076803203738)));

    for (k=0; k<=N; k++)
      Rho[1][k]=Rho[1][k]+alpha*g[1][k];
  }
```

Rerun your "test_it" to again verify that it produces the same output. At this stage, this is a trivial reorganization of the code, but it is important to ensure that everything is working before we implement the modified Broyden algorithm.

5.3 "broyden"

The modified Broyden algorithm discussed in class is:

1. Trial step —
   - Form trial step $\mathrm{Rho}_{it+1} = \mathrm{Rho}_{it} + \alpha \mathrm{g}_{it}$
2. Prediction step —
   - Form matrix

$$\mathbf{G} = \begin{pmatrix} \vec{g}_1 \cdot g_1 & \cdots & \vec{g}_1 \cdot \vec{g}_{it+1} & -\frac{1}{2} \\ \vec{g}_2 \cdot g_1 & \cdots & \vec{g}_2 \cdot \vec{g}_{it+1} & -\frac{1}{2} \\ & \vdots & & \vdots \\ \vec{g}_{it+1} \cdot g_1 & \cdots & \vec{g}_{it+1} \cdot \vec{g}_{it+1} & -\frac{1}{2} \\ 1 & \cdots & 1 & 0 \end{pmatrix}$$

   - Form vector

$$\vec{\mathrm{u}} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

   - Solve

$$\mathbf{G}\vec{\mathrm{b}} = \vec{\mathrm{u}}$$

   - Form final prediction

$$\mathrm{Rho}_{it+1} \longleftarrow \sum_{itp=1}^{it} \mathrm{b}_{itp}\mathrm{Rho}_{itp} + \mathrm{b}_{it+1}\mathrm{Rho}_{it+1} \tag{4}$$

Important note: This formulation of the algorithm uses Rho_{it+1} and g_{it+1} in two different ways: to store the trial step and residual (which we called Rho' and g' in lecture) *and* to store the final prediction step. This becomes important for coding only at the stage where we form the final Rho_{it+1}. At the final step, you should first multiply the contents of Rho_{it+1} by b_{it+1}, and then loop over all of the preceding Rho_{itp}, multiplying by b_{itp} and accumulating the final result into Rho_{it+1}.

Beginning with your code "test_it", make a new program "broyden" with the modifications below.

5.3.1 Matrix G and vector u

Declarations: In the above algorithm, **G** is an (it+2) × (it+2) matrix. Thus, at the top of your main program, you should declare space for **G** as a one-based, (Itmx+2) × (Itmx+2) dmatrix and also declare space for u as a one-based dvector of length Itmx+2. Finally, be good and free these variables at the end of main().

Residual vector $\vec{g}_{it+1}$: Once the trial Rho[it+1] is computed, you must also compute the corresponding g[it+1] with a call such as

```
getg(g[it+1],Rho[it+1],Z,lmax,nmax,nmaxmax,F,r,dr,N);
```

Formation: After forming Rho[it+1] and g[it+1] in the main iteration loop, add code to your main program to compute the matrix **G** and the vector u. A suitable code fragment (not the most efficient, but quite workable) for computing the $\vec{g}_{itp} \cdot \vec{g}_{itpp}$ components of **G** is

```
for (itp=1; itp<=it+1; itp++)
  for (itpp=1; itpp<=it+1; itpp++) {
    G[itp][itpp]=0.; /* Trust no one */
    for (k=0; k<=N; k++)
      G[itp][itpp]+=g[itp][k]*g[itpp][k];
  }
```

Note that you will also have to set the final column G[..][it+2]=-0.5, the final row G[it+2][..]=1, and the bottom corner element G[it+2][it+2]=0 to complete the matrix. Finally, be sure to properly set the values of u[..] as described in the algorithm above.

5.3.2 Solution of G b = u

Use your modified numerical recipes routines from 3.2.1 to solve the matrix equation G*b=u for b, taking the program in Appendix E as a model. Note that, for this, you will have to declare (and free!) a dmatrix LU with the same dimensions as G, a dvector b with the same dimensions as u, an ivector indx with the same dimensions as b, and a dummy double variable sgn. For debugging purposes, we highly recommend that you include the "Testing G*b=u?" part of the code from the appendix.

Note that, at this stage, your code should still output the same energies because we have not yet put the final prediction in Rho_{it+1}. Finally, once you are certain that the linear solver is functioning properly, comment out the test G*b=u part.

5.3.3 Formation of final Rho[it+1]

Use your elements of b[itp] from the lu solver to form the final value of Rho[it+1] according to the formula (4). (See the note in Section 5.3 for hints on how to do this.)

This completes the implementation of the modified Broyden algorithm! You now should be able to reach complete convergence within about ten Broyden steps!!!

Hint: For larger atoms, the modified Broyden approach sometimes induces NaN errors. This happens because some the coefficients b[itp] can become negative and the formation of Rho[it+1] sometimes results in very small, negative densities (on the order of -1e-12). To prevent these negative densities from causing NaNs in your exc() routines, it is best to simply take the absolute value after forming Rho[it+1] with a loop like

```
for (k=0; k<=N; k++)
  Rho[it+1][k]=fabs(Rho[it+1][k]);
```

5.4 Convergence rate

Make a new plot of the log of the error versus iteration number to show the TA and to bring to lab.

A "test_excpPZ.c"

```
#include <math.h>
#include <stdio.h>
double excPZ(double rs);

double excpPZ(double rs);

main()
{
  double rs,drs;

  rs=3.; /* Test rs > 1 "branch" */
  printf("Testing derivative for rs=%f ...\n",rs);
  for (drs=1; drs>1e-6; drs/=10)
    printf("    %20.16f %20.16f\n",drs,
           (excPZ(rs+drs)-excPZ(rs))/drs /* <- Finite difference slope */
           /                             /* Ratio should approach 1! */
           excpPZ(rs)                    /* <- Coded value of derivative */
           );

  printf("\n");

  rs=0.3; /* Test rs < 1 "branch" */
  printf("Testing derivative for rs=%f ...\n",rs);
  for (drs=1; drs>1e-6; drs/=10)
    printf("    %20.16f %20.16f\n",drs,
           (excPZ(rs+drs)-excPZ(rs))/drs /* <- Finite difference slope */
           /                             /* Ratio should approach 1! */
           excpPZ(rs)                    /* <- Coded value of derivative */
           );
}
```

B "test_getxc.c"

```
#include "p480.h"
#define Nmx 40000
main()
{
  /* Change of variables info */
  double *r,*dr;
  int N;

  /* Physics variables */
  double *V,*Rho,*Rhonew,*phi,*Vxc,*Depsxc,*integrand,Z;
  int lmax,*nmax,nmaxmax;
  double **E,***Psi,**F;
  double Etot;

  /* Working variables */
  int n,l,k;
  double x;

  /* Solver iteration varibles */
  int it;

  /* Value of pi */
  const double pi=4.*atan(1.);

  /* Specs for hydrogen */
  Z=1.;
  lmax=0;

  nmax=ivector(0,lmax);
  nmax[0]=0;

  nmaxmax=0;
  for (l=0; l<=lmax; l++)
    if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

  F=dmatrix(0,lmax,0,nmaxmax);
  F[0][0]=1.;

  /* The rest is now general for ANY case */
  E=dmatrix(0,lmax,0,nmaxmax); /* Make space for E's and Psi's */
  Psi=d3tensor(0,lmax,0,nmaxmax,0,Nmx);
  Rho=dvector(0,Nmx);
  Rhonew=dvector(0,Nmx);
  phi=dvector(0,Nmx);
  Vxc=dvector(0,Nmx);
  Depsxc=dvector(0,Nmx);

  integrand=dvector(0,Nmx);

  /* Grid vectors */
```

```
r=dvector(0,Nmx);
dr=dvector(0,Nmx);
V=dvector(0,Nmx);

for (N=40; N<=Nmx; N*=10) {

  /* Set up grid */
  for (k=0; k<=N; k++) {
    x=((double) k)/((double) N);
    r[k]=1/(1-x)-1-x;
    dr[k]=1/(1-x)/(1-x)-1;
  }
  dr[N]=0.;

  /* Initialize charge density */
  for (k=0; k<=N; k++)
    Rho[k]=0.;

  /* Test section */
  {
    /* Make potential from zero charge (debugs NaNs, etc.) */
    getphi(phi,Rho,r,dr,N);
    getVxc(Vxc,Rho,r,dr,N);
    for (k=0; k<=N; k++)
      V[k]=-Z/r[k]+phi[k]+Vxc[k];
    V[0]=0.;

    /* Get H 1s wave function, and DENSITY */
    getallEs(E,lmax,nmax,Z,V,r,dr,N);
    getallPsis(Psi,E,lmax,nmax,V,r,dr,N);
    getRho(Rhonew,Psi,F,lmax,nmax,N);

    /* Compute Exc energy correction */
    getDepsxc(Depsxc,Rhonew,r,dr,N);
    getVxc(Vxc,Rhonew,r,dr,N);

    for (k=0; k<=N; k++)
      integrand[k]=Vxc[k]*Rhonew[k];
    printf("N=%6d <Vxc>=%15.10f ",N,
           simpint(integrand,r,dr,N)
           );
    for (k=0; k<=N; k++)
      integrand[k]=(Depsxc[k])*Rhonew[k];
    printf("<DExc>=%15.10f\n",
           simpint(integrand,r,dr,N)
           );
  }
}

/* Be a good citizen and clean up... */
free_dvector(r,0,Nmx);
```

```
  free_dvector(dr,0,Nmx);
  free_dvector(V,0,Nmx);

  free_dmatrix(E,0,lmax,0,nmaxmax);
  free_d3tensor(Psi,0,lmax,0,nmaxmax,0,Nmx);
  free_dvector(Rho,0,Nmx);
  free_dvector(Rhonew,0,Nmx);
  free_dvector(phi,0,Nmx);
  free_dvector(Vxc,0,Nmx);
  free_dvector(Depsxc,0,Nmx);
}
```

C VMN forms for exchange correlation

```
double exc(double rs)
{
  /* constants */
  const double
    pi = 4.*atan(1.),
    X1 = 0.75*pow(3.0/(2.0*pi),2.0/3.0),  /* Exchange energy coeff */
    A  =  0.0310907,
    x0 = -0.10498,
    b  = 3.72744,
    c  = 12.9352,
    Q  = sqrt(4*c-b*b),
    X0 = x0*x0+b*x0+c;

  double x=sqrt(rs),X=x*x+b*x+c;

  return -X1/rs
    + A*(
        +log(x*x/X)+2*b/Q*atan(Q/(2*x+b))
        -(b*x0)/X0*(
                    log((x-x0)*(x-x0)/X)+2*(2*x0+b)/Q*atan(Q/(2*x+b))
                    )
        );
}

double excp(double rs)
{
  /* constants */
  const double
    pi = 4.*atan(1.),
    X1 = 0.75*pow(3.0/(2.0*pi),2.0/3.0),  /* Exchange energy coeff */
    A  =  0.0310907,
    x0 = -0.10498,
    b  = 3.72744,
    c  = 12.9352,
    Q  = sqrt(4*c-b*b),
    X0 = x0*x0+b*x0+c;

  double x=sqrt(rs),X=x*x+b*x+c;

  double dx=0.5/x; /* Chain rule needs dx/drho! */

  return dx*(
            2*X1/(rs*x)+A*(
                           2./x-(2*x+b)/X-4*b/(Q*Q+(2*x+b)*(2*x+b))
                           -(b*x0)/X0*(2/(x-x0)-(2*x+b)/X-4*(2*x0+b)/
                                                  (Q*Q+(2*x+b)*(2*x+b)) )
                           )
            );
}
```

D "test_getg.c"

```
#define Nmx 40000
#define Itmx 100
#define alpha 0.25
main()
{
  /* Change of variables info */
  double *r,*dr;
  int N;

  /* Physics variables */
  double *Rho,*g,Z;
  int lmax,*nmax,nmaxmax;
  double **F;
  double Etot;

  /* Working variables */
  int n,l,k;
  double x;

  /* Solver iteration variables */
  int it;
  /* Value of pi */
  const double pi=4.*atan(1.);

  /* Specs for O */
  Z=8.;
  lmax=1;

  nmax=ivector(0,lmax);
  nmax[0]=1;
  nmax[1]=0;

  nmaxmax=0;
  for (l=0; l<=lmax; l++)
    if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

  F=dmatrix(0,lmax,0,nmaxmax);
  F[0][0]=2.;
  F[0][1]=2.;
  F[1][0]=4.;

  /* The rest is now general for ANY case */
  Rho=dvector(0,Nmx);
  g=dvector(0,Nmx);

  /* Grid vectors */
  r=dvector(0,Nmx);
  dr=dvector(0,Nmx);

  N=4000;
```

```
  /* Set up grid */
  for (k=0; k<=N; k++) {
    x=((double) k)/((double) N);
    r[k]=1/(1-x)-1-x;
    dr[k]=1/(1-x)/(1-x)-1;
  }
  dr[N]=0.;

  /* Initialize charge density */
  for (k=0; k<=N; k++)
    Rho[k]=0.;

  /* Iteration loop */
  for (it=1; it<=Itmx; it++) {
    Etot=getg(g,Rho,Z,lmax,nmax,nmaxmax,F,r,dr,N);
    printf("Iter:%4d  Etot:%20.12f  Error:%20.12f\n",
           it,Etot,fabs(Etot-(-74.473076803203738)));

    for (k=0; k<=N; k++)
      Rho[k]=Rho[k]+alpha*g[k];
  }

  /* Be a good citizen and clean up... */
  free_dvector(r,0,Nmx);
  free_dvector(dr,0,Nmx);

  free_dmatrix(F,0,lmax,0,nmaxmax);
  free_dvector(Rho,0,Nmx);
  free_dvector(g,0,Nmx);
}
```

E "test_lu.c"

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#include "p480.h"

#define N 4
main()
{
  double **G,**LU,*b,*u;

  int *indx;
  double sgn,tmp;

  int i,j;

  /* Declare space */
  G=dmatrix(1,N,1,N); /* Matrix */
  LU=dmatrix(1,N,1,N); /* Space for LU decomposition of matrix */
  b=dvector(1,N); /* Space for answer to G*b=u */
  u=dvector(1,N); /* Space for right-hand side, u */
  indx=ivector(1,N); /* Space for integer vector needed by ludcmp */

  /* Make matrix Gij=1/(i+j) */
  for (i=1; i<=N; i++)
    for (j=1; j<=N; j++)
      G[i][j]=1./( (double) i+j );

  /* Make right-hand side u=(1 1 1 ...) for equation G*x=u */
  for (i=1; i<=N; i++) /* u=0 for all but the last component */
    if (i<N)
      u[i]=0.;
    else
      u[i]=1.;


  /* Print out the problem */
  printf("\nSolving...\n\n");
  for (i=1; i<=N; i++) {
    printf("| ");
    for (j=1; j<=N; j++)
      printf("%9.2e ",G[i][j]);
    printf("|   |b[%d]|  =  | %9.2e |\n",i,u[i]);
  }

  /* Solve G*b=u */
  for (i=1; i<=N; i++) /* Copy G into LU because ludcmp destroys its input */
    for (j=1; j<=N; j++)
      LU[i][j]=G[i][j];
  for (i=1; i<=N; i++) /* Copy u into b because lubksb destroys its input */
    b[i]=u[i];
```

```
  /* Actual solution of equations */
  ludcmpp480(LU,N,indx,&sgn); /* Get LU decomposition */
  lubksbp480(LU,N,indx,b); /* Use LU decomposition to solve equations */

  /* Does it work? */
  printf("\nTesting G*b=u?:\n");
  for (i=1; i<=N; i++) {
    tmp=0.;
    for (j=1; j<=N; j++)
      tmp+=G[i][j]*b[j];
    printf("%20.12f  =%20.12f  ?\n",tmp,u[i]);
  }
  printf("\n");

  /* Be good, and clean up! */
  free_dmatrix(G,1,N,1,N); /* Matrix */
  free_dmatrix(LU,1,N,1,N); /* Space for LU decomposition of matrix */
  free_dvector(b,1,N); /* Space for answer to G*b=u */
  free_dvector(u,1,N); /* Space for right-hand side, u */
  free_ivector(indx,1,N); /* Space for integer vector needed by ludcmp */
}
```