Cornell University

Department of Physics

Phys 480/680, Astro 690 February 27, 2004

Computational Physics, Spring 2002

Homework Assignment # 4

(Due Thu, March 11)

Agenda and readings for the week of February 26:

Goal: This homework develops and packages into subroutines the functionality needed to perform complete calculations for atoms.

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:

- Lec9, 02/24 (Tue) Secant method; Ridders method; bracketing of roots for Schrdinger's equation **Reading: NR 9.2**
- Lec10, 02/26 (Thu): Shooting to a fitting point; normalization conventions for atoms; Non-spherical states (higher angular momenta)
- Lab3, 02/27 (Fri): Stability of solutions, effectiveness of root finders; convergence of eigenvalues with N.

Preview of coming week:

- 03/02 NO LECTURE
- 03/04 Numerical interpolation; exchange and correlation energy functionals; solution of many non-linear equations in many unknowns (Broyden approaches); our need for a "Modified" Broyden's Method **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 Relations among self-consistent iteration, charge mixing, and Broyden's method; line optimization, full, modified Broyden's method
- 03/11 Introduction to Numerical linear algebra; benefits of LAPACK/ATLAS; Analysis of convergence of charge mixing; condition numbers

Contents

1 Automated Schrödinger energy solver **3**
- 1.1 func_SchrodingerNodes() . . . 3
- 1.2 getEs() . . . 3
- 1.3 Debugging . . . 4

2 Finishing touches for schint() **4**
- 2.1 First and last generated elements of Psiout[] . . . 4
- 2.2 Avoiding overflow in "schint" . . . 5
- 2.3 schint(): Psiout==NULL case . . . 5
- 2.4 Updating func_Schrodinger() and func_SchrodingerNodes() . . . 6
- 2.5 Debugging . . . 6

3 getallEs() **6**
- 3.1 Debugging . . . 8

4 Complete and normalized wave functions **8**
- 4.1 interpolate() . . . 8
- 4.2 getPsi() . . . 9
- 4.3 Debugging . . . 9

5 Getting all of the wave functions **10**
- 5.1 d3tensor() and free_d3tensor() . . . 10
- 5.2 getallPsis() . . . 10
- 5.3 Debugging . . . 11

6 Final components of a full atomic solver **11**
- 6.1 "getRho" . . . 11
  - 6.1.1 Debugging . . . 11
- 6.2 "getphi" . . . 12
  - 6.2.1 Debugging . . . 12

A "getEdebug.c" **13**

B "getallEsdebug.c" **15**

C "getPsidebug.c" **16**

D "getallPsisdebug.c" **19**

E "getRhodebug.c" **21**

F "getphidebug.c" **23**

1 Automated Schrödinger energy solver

The solvers in the previous problem set require your input to provide the initial bounds E1 and E2 for each state. This is impractical when we wish to find all of the states for many different atoms. This process, fortunately, can be automated by exploiting a mathematical fact about the number of nodes in the solutions.

The lowest energy state is always guaranteed to have zero nodes (never cross the x axis), and each subsequent state is guaranteed to have one additional node (cross the x axis one more time than the previous solution). This implies that if the energy E1 gives n nodes and energy E2 gives $n+1$ nodes, then only one state, the one with n nodes, lies between E1 and E2. A call to zriddrp480() then is guaranteed to find the state with n nodes.

Thus, if we can find a method to always find states with a given number of nodes, we can use this to systematically generate all of the solutions. Our strategy for doing this is to first define a function which returns the *difference* between number of nodes at energy E and the desired number of nodes. Then, we simply find the value of E which makes this function zero by calling rtbisp480()!

1.1 func_SchrodingerNodes()

```
double func_SchrodingerNodes(double E,
                             int match, double V[],
                             double r[], double dr[], int N)
```

Return value:

- (number of nodes at energy E) – (match)

Input:

- E: proposed electron energy
- match: desired number of nodes in solution
- V[]: potential on grid
- r[], dr[], N: standard grid information

Write a routine func_SchrodingerNodes() in "physics.c" (don't forget to put the prototype in "physics.h" – **last reminder!**), which is very much like func_Schrodinger(), *except* that it returns the number of nodes reported by schint *minus* the value of the argument "match". Your return statement, for instance, will appear something like:

```
return ((double) (schint(&Psi,&Psip,Psiout,k1=0,k2=N,V,E,r,dr,N)-match));
```

Note: Although the return value is always an integer, we return it as a double so that we can use rtbisp480() routine without modification.

1.2 getEs()

```
void getEs(double E[],int nmax,double Elower,double V[],
           double r[],double dr[],int N)
```

Output:

- E[n]: linear *array* of energies of states with n=0 nodes through n=nmax nodes (inclusive!)

Input:

- nmax: maximum number of nodes sought
- Elower: energy guaranteed to be lower than any valid solutions

- V[]: effective potential on grid
- r[], dr[], N: standard grid information

Write a routine of the above prototype in "physics.c" to return the energies of the electronic states with zero through nmax nodes in the potential V[] to a precision of TOL, which you should set at the top of "physics.c" with a statement like

```
#define TOL (1.e-12)
```

The purpose of doing this is that you should never bury "magic constants" in your code, but rather define them in a single, centralized location where they are easily identified and changed.

Your routine should use rtbisp480() in conjunction with func_SchrodingerNodes() to first locate appropriate bounding energies, and then use zriddrp480 to find the solution to high precision.

A workable code fragment (but not the most efficient – you should try to do better) is

```
/* Loop to get states */
for (n=0; n<=nmax; n++) {
  /* Get E1 as an energy with n nodes */
  E1=rtbisp480(func_SchrodingerNodes,Elower,0.,TOL,n,V,r,dr,N);

  /* Get E2 as an energy with n+1 nodes */
  E2=rtbisp480(func_SchrodingerNodes,Elower,0.,TOL,n+1,V,r,dr,N);

  /* Now, get the solution, which is in between! */
  E[n]=zriddrp480(func_Schrodinger,E1,E2,TOL,0,V,r,dr,N);
}
```

Note: All solutions will have E<0, so using [Elower,0.] as the bounds for the rtbisp480() searches is guaranteed to work!

1.3 Debugging

Compile and run your routines with the main program "getEdebug.c" in Appendix A. This program accepts as input the number of integration intervals N and the number of states you want, and then returns all of the energies. At N=1000 intervals, you should be able to recover the first five states to within 5 decimal places. You should also be able to verify the scaling of your error in going from N=1000 to N=10000.

2 Finishing touches for schint()

Your routine schint() is almost complete. It requires only three finishing touches, that you double-check one part of its functionality, that you improve its reliability and that you improve its efficiency.

In terms of functionality, as parts of this problem set begin to use Psiout[], it is important to be sure that *all* values are entered into the array. (See Problem 2.1.)

As discussed in lab, the final value of Psi which schint() outputs sometimes overflows. Problem 2.2 deals with this issue of reliability.

Most of the calls which your program will make to schint() are to find the energies. These calls use only the final returned value *Psi and not the full array of values Psiout[]. Problem 2.3 deals with this inefficiency using "NULL" pointers, an elegant solution common in C.

2.1 First and last generated elements of Psiout[]

Depending on how you structured your loop for calling rk4p480(), you generally will need an extra assignment statement to make sure that either the first, Psiout[k1], or last, Psiout[k2], required elements of Psiout[] are actually stored in the array. Double check that your schint() routine fills in these values!

2.2 Avoiding overflow in "schint"

As we discussed in lab, we can exploit the linearity of the Schrödinger equation to avoid overflow of Psi and Psip in schint() by renormalizing the solution by multiplying it through by a small constant each time that the magnitude of the solution exceeds a critical value. The IEEE range of double precision values is somewhat beyond 10^{-300} to 10^{300}. To ensure enough room to avoid overflows, it is safest to keep the magnitude of the solution between 10^{-150} and 10^{150}. You may do this by scaling the solution by 10^{-300} every time its absolute value exceeds 10^{150}. Once again, it is best to collect such special numerical constants in '#define' statements near to top of your file. You should thus add the following lines to the top of "physics.c" next to the '#define TOL (1.e-12)' statement,

```
#define SMALL (1.e-300)
#define SQRTSMALL (1.e-150)
#define BIG (1.e300)
#define SQRTBIG (1.e150)
```

Then, you should modify your main loop in schint(), so that whenever the *absolute value* of Psi (probably called y[1]) exceeds "SQRTBIG", your entire solution is multiplied by "SMALL".

Hint: When you rescale your solution, you must scale both components of $\vec{y}$ in your differential equation (i.e., both y[1] and y[2]), and you must also rescale all of the values *previously* generated and stored in Psiout[]. The rescaling of Psiout[] will require writing another loop and some care to be certain that all of the appropriate elements of Psiout[] are rescaled, for both dk>0 and dk<0.

2.3 schint(): Psiout==NULL case

Often times, when schint() is called, the values placed in Psiout[] are ignored. A common device in C for communicating to the subroutine that a set of values need not be computed is to pass the value "NULL" to the subroutine. The subroutine may then check whether it needs to return the value and may then skip the associated calculations if they are not needed. Another significant advantage is that the calling function no longer needed to provide space for the result!

For instance, at the moment, your func_Schrodinger() routine in "physics.c" should look something like the function below:

```
double func_Schrodinger(double E,
int match, double V[], double r[], double dr[], int N)
{
  double Psi=0.,Psip=1.;
  int k1,k2;

  /* Allocation of space that is NOT actually NEEDED */
  double *Psiout;
  Psiout=dvector(0,N);

  /* Outward integration */
  schint(&Psi,&Psip,Psiout,k1=0,k2=N,V,E,r,dr,N);

  /* Free the space that was NOT actually NEEDED */
  free_dvector(Psiout,0,N);

  return Psi;
}
```

If, on the other hand, we rewrite schint() to not compute Psiout if it is passed as "NULL", then we can write simply

```
double func_Schrodinger(double E,
int match, double V[], double r[], double dr[], int N)
{
  double Psi=0.,Psip=1.;
  int k1,k2;

  /* Outward integration */
  schint(&Psi,&Psip,NULL,k1=0,k2=N,V,E,r,dr,N);

  return Psi;
}
```

and never need to allocate (or have schint() compute) Psiout[]!

You should thus **modify** schint() so that, before each operation involving Psiout, you check whether the operation actually needs to be performed by using a statement like "if (Psiout!=NULL) ...".

Hints:
(1) Note that when you have to rescale an entire range of values in Psiout[] (Problem 2.2) it is much more efficient to place the "if (Psiout!=NULL) ..." statement *before* the "for (...)" loop, rather than after it.

(2) You may have to rearrange your program a little to use new variables if (when counting nodes, for instance) you were using values stored in Psiout[].

2.4 Updating func_Schrodinger() and func_SchrodingerNodes()

Update your func_Schrodinger() and func_SchrodingerNodes() programs to pass NULL into schint() and to stop allocating space for Psiout[]!

2.5 Debugging

First, check the value for N (probably 10000) for which your *old* routines generated NaN's when running "getEdebug.c". Then, compile and run your *new*, modified routines with "getEdebug.c" for the same value of N and check that you do not get NaN's and that you see the expected scaling of your results with N.

3 getallEs()

The routine getEs() from the previous problem set finds all wave functions for a given one-dimensional potential. As discussed in class, we can generate all of the solutions (spherically symmetric or not) to a given radial potential by solving for the one-dimensional states in the effective potential

$$V_{eff}(r) \equiv V(r) + \frac{\ell(\ell+1)}{2r^2},$$

where ℓ (ℓ=0, 1, 2, ...) is the *angular momentum* of the given state. For atoms, we know how many states we need for each angular momentum. Thus to find all the states, all we need do is to call getEs() once for each value of ℓ. For convenience, we package this capability into a new routine, which you should place in "physics.c" along with our other physics subroutines:

```
void getallEs(double **E,
              int lmax, int nmax[],
              double Z, double V[],
              double r[], double dr[], int N)
```

Output:

- E[l][n]: *matrix* of energies of states of angular momentum l and n=0 nodes through n=nmax[l] nodes (inclusive!)

Input:

- lmax: maximum angular momentum to consider (l=0 through l=lmax)
- nmax[l]: maximum number of nodes sought for angular momentum l
- Z: charge on the nucleus
- V[]: "self-consistent potential" (V[k]=-Z/r[k], for now ...)
- r[], dr[], N: standard grid information

Write a routine of the above prototype which returns the energies of all requested electronic states in an atom with radial potential V[]. The routine should return the results in the form of a double precision two dimensional matrix of values E[l][n]. It should determine the following table of values,

l=0	E[0][0]	E[0][1]	...	E[0][nmax[0]]
l=1	E[1][0]	E[1][1]	...	E[1][nmax[1]]
⋮			⋮	
l=lmax	E[lmax][0]	E[lmax][1]	...	E[0][nmax[lmax]]

If you are new to C, you should note here how two- (and higher-) dimensional arrays are accessed.

A suitable, but not the most efficient, code fragment is

```
double *Veff;
                        .
                        .
Veff=dvector(0,N); /* Must declare space for Veff */
                        .
                        .
  for (k=0; k<=N; k++) { /* Compute Veff */
    Veff[k]=V[k]+l*(l+1)/(2*r[k]*r[k]);
  }
  Veff[0]=0.; /* Mathematically proper for origin */
                        .
                        .
  getEs(E[l],nmax[l],-Z*Z,Veff,r,dr,N); /* Solve for states of ang mom l */
  /* Note: Above, passing in E[l] sends the pointer to the appropriate row
           of E[][], which getEs() just fills in appropriately as a
           one-dimensional array */
                        .
                        .
free_dvector(Veff,0,N); /* Must free Veff at end of routine! */
```

Note that, here, as discussed in lecture, we use -Z*Z as a guaranteed lower bound. Note the convenient use of the *Numerical Recipes* dvector() routine to declare space for Veff.

Hints on working with two-dimensional matrices:

Note that the above code fragment will only work if E is properly passed in as an array of pointers, which can get quite tricky unless you have a lot of experience with dealing with pointers in **C**. Fortunately, the *Numerical Recipes* dmatrix() function can take care of all of the needed set-up for you. A typical use of dmatrix() in a program would be

```
{
  double **E; /* In the declarations section */
                    .
                    .
  E=dmatrix(0,lmax,0,nmaxmax); /* Among the first executables -
                    .              lmax is the maximum l value we expect;
                    .              nmaxmax is the maximum expected of all nmax[l]'s */
    getallEs(E,lmax,nmax,Z,V,r,dr,N); /* Example call */
    printf("E[l=0][n=0]=%f, E[l=1][n=0]=%f\n",E[0][0],E[1][0]); /* Use E[][] like this */
                    .
                    .
  free_dmatrix(E,0,lmax,0,nmaxmax); /* At end of routine or main, must call free
                                       for each dvector and dmatrix before exiting! */
}
```

3.1 Debugging

Compile and run your routine with the main program "getallEsdebug.c" in Appendix B. The program calls your routines to get and print out the energies of the filled states for the carbon atom. Carbon has electrons in three states: "1s" (l=0 with zero nodes), "2s" (l=0 with one node), and "2p" (l=1 with zero nodes). Note that, for now, we ignore electron-electron interactions and are taking the potential for carbon to be just V[k]=-Z/r[k], where Z=6. Verify that your results approach the correct analytic results with the expected error scaling with N.

4 Complete and normalized wave functions

The routine getallEs() from the previous problem finds for us the unknown energies for the Schrödinger equation. Given these energies, the Schrödinger equation becomes an ordinary differential equation (ODE) which we may solve with rk4p480().

Three things must be done to the raw output of rk4p480 before we can return the final wave functions. First, as discussed in lecture, numerical stability demands that we integrate Schrödinger's equation both inward and outward and match the solutions at a given matching point. Second, rk4p480() provides values only on every other data-point k and so we must interpolate to find values for the remaining points. Finally, we must normalize the wave functions.

4.1 interpolate()

To provide the ability to interpolate between the values produced by rk4p480(), write a function of the following form in "p480.c":

`void interpolate(double f[],int N)`

Output:

- f[k]: values of f[] on *all* points

Input:

- f[k]: values of f[] on *even* points *only*
- N: total number of points

Your function should interpolate the odd values of f[] from the even values according to the formula

$$f_k = 0.0625(-f_{k-3} + 9 * (f_{k-1} + f_{k+1}) - f_{k+3}).$$

For the special endpoint cases, k=1 and k=N-1, you may take $f_{-1} \equiv 0$ and $f_{N+1} \equiv 0$, respectively.

4.2 getPsi()

```
void getPsi(double Psi[], double E, int l,
            double V[], double r[], double dr[], int N)
```

Output:

- Psi[k]: values of *normalized* $\Psi(r)$ on *all* grid points r[k]

Input:

- E: electron energy (found from "getallEs")
- l: angular momentum of Ψ
- V[]: potential on grid points
- r[], dr[], N: standard grid information

Write a routine getPsi() of the above prototype in "physics.c", which produces the values on the grid r[] of the wave function with energy E and angular momentum l, properly normalized so that $\int_0^\infty |\Psi(r)|^2\, dr = 1$. Your routine will consist of several steps:

First, you should construct the appropriate effective potential,

$$V_{eff}(r) = V(r) + \frac{\ell(\ell+1)}{2r^2}.$$

Next, you need to determine an appropriate point, kmatch, for matching the inward and outward solutions. An appropriate value is the boundary between the exponential and oscillatory solutions, the so-called "classical turning point" where $V(r) = E$. To find the outermost point where $V(r) < E$, you may use the following fragment:

```
                        .
                        .
kmatch=0;
for (k=0; k<=N; k++)
  if (Veff[k]<E) kmatch=k;
if (kmatch==0) {
  printf("Veff never below E=%f in getPsi.\n",E);
  exit(1);
}
                        .
                        .
```

(This is the so-called "classical turning point".) Then, you should perform the inward (k=0..kmatch) and outward (k=N..kmatch) integrations using schint(), scaling both solutions to match so that $\Psi(r[\text{kmatch}]) = 1$.

Next, interpolate your result to include values at the odd points using your new routine interpolate().

Finally, normalize the result using the formula

$$\Psi(r) \leftarrow \frac{\Psi(r)}{\sqrt{\int_0^\infty |\Psi(r)|^2\, dr}},$$

where you perform the integral using simpint().

4.3 Debugging

Cut and paste the routine "getPsidebug.c" from Appendix C from the web version of the problem set and compile and run it with your code. This program will compute the root mean square (rms) error of your calculation of the 1s state of hydrogen relative to the analytic solution as a function of the number of intervals N.

5 Getting all of the wave functions

Just as we packaged getEs()'s ability to find the energies for all of the states in getallEs(), we now package getPsi()'s capabilities into getallPsis().

5.1 d3tensor() and free_d3tensor()

The output of getallPsis() will be a three-dimensional array, one full wave function for each value of (n,l): Psi[l][n][0..N]. For some reason, *Numerical Recipes* provides routines for three-dimensional arrays in single precision, but not double precision. Accordingly, generalize the *Numerical Recipes* utility routines f3tensor() and free_f3tensor() to double precision by replacing all occurrences of **float** with *double*, thereby producing routines with the two prototypes below. You should place these routines at the top of "p480.c" (and their prototypes in "p480.h"). Your new routines will have the following prototypes:

```
double ***d3tensor(long nrl,long nrh,long ncl,long nch,long ndl,long ndh)
```

Return value:

- pointer to three dimensional array of doubles

Input:

- nrl, nrh: lowest and highest values of indices for first index (rows)
- ncl, nch: lowest and highest values of indices for second index (columns)
- ndl, ndh: lowest and highest values of indices for third index (depth)

```
void free_d3tensor(double ***t,
                   long nrl,long nrh,long ncl,long nch,long ndl,long ndh)
```

Input:

- nrl, nrh: lowest and highest values of indices for first index (rows)
- ncl, nch: lowest and highest values of indices for second index (columns)
- ndl, ndh: lowest and highest values of indices for third index (depth)

5.2 getallPsis()

Write a routine of the below prototype in "physics.c" which calls getPsi() to generate all of the wave functions $\Psi_{n\ell}$ for l=0..lmax and, for each l, n=0..nmax[l]. A suitable code fragment is

```
                    .
                    .
for (l=0; l<=lmax; l++)
  for (n=0; n<=nmax[l]; n++)
    getPsi(Psi[l][n],E[l][n],l,V,r,dr,N);
                    .
                    .
```

Note that the object Psi[l][n] is, as getPsi expects, actually a pointer to the first storage location among the elements Psi[l][n][0..N]. (Note also that your "getallPsis" routine will be rather short.)

```
void getallPsis(double ***Psi,double **E,
int lmax, int *nmax,
double *V,
double r[], double dr[], int N)
```

Output:

- Psi[l][n][k]: values of $\Psi_{n\ell}(r[k])$ (properly normalized) for wave functions with angular momentum ℓ and with n nodes

Input:

- E[l][n]: *matrix* of energies of states of angular momentum l and with n=0..nmax[l] nodes
- lmax: maximum angular momentum to consider (l=0..lmax)
- nmax[l]: maximum number of nodes sought for angular momentum l
- V[]: potential
- r[], dr[], N: standard grid information

5.3 Debugging

Cut and paste the program "getallPsisdebug.c" from Appendix D and compile and run it with your new routines. This program will use your code to compute the 1s, 2s and 2p states of hydrogen at N=400 points, and will print out a comparison of your and the analytic results.

6 Final components of a full atomic solver

We now complete the only two remaining components needed to solve the self-consistent equations for atoms: adding up the total electron density from all of the wave functions and solving Poisson's equation for this density.

6.1 "getRho"

Write a function of the below prototype in "physics.c" which computes the quantity

$$\mathrm{Rho}(r) = \sum_{n\ell} F_{n\ell} \left|\Psi_{n\ell}(r)\right|^2 ,$$

where $F_{n\ell}$ represents the filling of each of the states.

```
void getRho(double *Rho,
       double ***Psi, double **F, int lmax, int *nmax, int N)
```

Output:

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

Input:

- Psi[l][n][k]: values of normalized $\Psi_{n\ell}(r[k])$
- F[l][n]: shell occupancies
- lmax: maximum angular momentum to consider (l=0 through l=lmax)
- nmax[l]: maximum number of nodes sought for angular momentum l
- N: number of intervals for the grid

6.1.1 Debugging

Test your routine by integrating $\int_0^\infty \mathrm{Rho}(r)\, dr$ to get the total number of electrons in the carbon atom by compiling and running with the main program provided in Appendix E.

6.2 "getphi"

Package your previous Poisson solver program into a routine of the prototype below in "physics.c". Note that the return value is the actual electrostatic potential, phi(r)≡Phi(r)/r, whereas the input is Rho(r)≡$4\pi r^2 n(r)$. Be sure to use "interpolate" to find the values on the odd grid points *after* you have done the Runge-Kutta integration, but *before* dividing Phi[k] by r[k] to get phi[k].

```
void getphi(double phi[], double Rho[], double r[], double dr[], int N)
```

Output:

- phi[k]: electrostatic potential $\phi(r)$ on the grid

Input:

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

Finally, because your input is now Rho[k]≡4*pi*r[k]*r[k]*rho[k] instead of just rho[k], you must modify derivs_Poisson() (be sure to include this in "physics.c" before getphi()!) so that it now contains a code fragment of the form

```
            .
            .
dydx[1]=y[2]*dr[k];
dydx[2]=-Rho[k]/r[k]*dr[k];
            .
            .
```

6.2.1 Debugging

Test your code by compiling and running it with the main program in Appendix F. This program uses your subroutines to compute the Hartree energy for the 1s state of hydrogen as a function of the number of intervals N. (Recall that your previous answer for this energy was 5/16.) Don't worry if you see only third-order convergence; we will discuss this in lab.

Finally, for a thorough test, replace the "Specs for hydrogen" with the appropriate specifications for Carbon. The Hartree energy of the (non-interacting) carbon atom is 26.239197530865 hartree.

A "getEdebug.c"

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

main()
{
  /* Grid information */
  double *r,*dr;
  int N;

  /* Physics vectors */
  double *V;

  /* Working variables */
  int k,Nstates;
  double x,Elower;
  double *E;

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

  /* Loop over different input N and numbers of states */
  while (1) {
    printf("N="); scanf("%d",&N);
    printf("Nstates="); scanf("%d",&Nstates);

    /* Allocate NR vectors */
    r=dvector(0,N);
    dr=dvector(0,N);
    V=dvector(0,N);

    E=dvector(0,Nstates-1);

    /* Construct change of variables information and
       integrand function for computing V_el-nuc */
    for (k=0; k<=N; k++) {
      x=((double) k)/((double) N);
      r[k]=x*x/(1.-x);
      dr[k]=1./(1.-x)/(1.-x)-1.;
      V[k]=-1./r[k];
    }
    /* Special conditions for handling end-points of integration */
    V[0]=0.;
    dr[N]=0.;


    getEs(E,Nstates-1,Elower=-1.,V,r,dr,N);
```

```
    for (k=0; k<Nstates; k++)
      printf("State with %2d nodes, E=%16.12f\n",k,E[k]);

    /* Clean up vectors before next set is allocated! */
    free_dvector(r,0,N);
    free_dvector(dr,0,N);
    free_dvector(V,0,N);
    free_dvector(E,0,Nstates-1);
  }

}
```

B "getallEsdebug.c"

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

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

  /* Physics variables */
  double *V;
  double Z;
  int lmax;
  int nmaxmax,*nmax;
  double **E,*psi;

  int n,l;

  /* Working variables */
  double x,*integrand;
  int k;

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

  /* String containing angular momentum codes */
  char names[5]="spdf";

  /* Specs for carbon */
  Z=6.; /* Charge on nucleus */
  lmax=1; /* Only up to 'p' states (l=0) are filled in H */

  nmax=ivector(0,lmax); /* Declare space for max nodes for each l */
  nmax[0]=1; /* Specify that only up to the first two (zero and one nodes)
                's' states are filled in C */
  nmax[1]=0; /* Specify that only the first (zero node) 'p' state is
                filled in C. */

  /* General loop for finding nmaxmax (needed to define arrays)...
     Note: This is a little silly for H, but is handy when you have
     some random element. */
  nmaxmax=0;
  for (l=0; l<=lmax; l++)
    if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

  /* Allocate space for table for E's */
  E=dmatrix(0,lmax,0,nmaxmax);
```

```
  /* Convenient loop over N */
  for (N=40; N<=Nmx; N*=10) {
    /* Allocate space for objects with size which depends on N */
    psi=dvector(0,N); /* psi */
    r=dvector(0,N); /* Grid vectors */
    dr=dvector(0,N);
    V=dvector(0,N);

    integrand=dvector(0,N); /* Working grid for integrating things */

    /* Set up grid and -Z/r potential */
    for (k=0; k<=N; k++) {
      x=((double) k)/((double) N);
      r[k]=1./(1.-x)-1.-x-x*x-x*x*x;
      dr[k]=1./(1.-x)/(1.-x)-1.-2*x-3*x*x;
      V[k]=-Z/r[k];
    }
    /* Below gives proper limits at end points */
    V[0]=0.; dr[N]=0.;

    getallEs(E,lmax,nmax,Z,V,r,dr,N); /* Get E */

    for (l=0; l<=lmax; l++)
      for (n=0; n<=nmax[l]; n++)
        printf("E_%d%c=%16.12f; ",n+l+1,names[l],E[l][n]);
    printf("\n");

    /* Free up space for N dependent objects before going to next N*/
    free_dvector(r,0,N);
    free_dvector(dr,0,N);
    free_dvector(V,0,N);

    free_dvector(psi,0,N);
    free_dvector(integrand,0,N);
  }

  /* Free up all remaining objects */
  free_dmatrix(E,0,lmax,0,nmaxmax);
  free_ivector(nmax,0,lmax);
}
```

C "getPsidebug.c"

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

#define Nmx 40000
main()
```

```
{
  /* Change of variables info */
  double *r,*dr;
  int N;

  /* Physics variables */
  double *V;
  double Z;
  int lmax;
  int nmaxmax,*nmax;
  double **E,*psi;

  int n,l;

  /* Working variables */
  double x,*integrand;
  int k;

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

  /* Specs for hydrogen */
  Z=1.; /* Charge on nucleus */
  lmax=0; /* Only up to 's' states (l=0) are filled in H */

  nmax=ivector(0,lmax); /* Declare space for max nodes for each l */
  nmax[0]=0; /* Specify that only up to the first (zero nodes) s state
                is filled in H */

  /* General loop for finding nmaxmax (needed to define arrays)...
     Note: This is a little silly for H, but is handy when you have
     some random element. */
  nmaxmax=0;
  for (l=0; l<=lmax; l++)
    if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

  /* Allocate space for table for E's */
  E=dmatrix(0,lmax,0,nmaxmax);

  /* Convenient loop over N */
  for (N=40; N<=Nmx; N*=10) {
    /* Allocate space for objects with size which depends on N */
    psi=dvector(0,N); /* psi */
    r=dvector(0,N); /* Grid vectors */
    dr=dvector(0,N);
    V=dvector(0,N);

    integrand=dvector(0,N); /* Working grid for integrating things */

    /* Set up grid and -Z/r potential */
    for (k=0; k<=N; k++) {
```

```
      x=((double) k)/((double) N);
      r[k]=1./(1.-x)-1.-x-x*x-x*x*x;
      dr[k]=1./(1.-x)/(1.-x)-1.-2*x-3*x*x;
      V[k]=-Z/r[k];
    }
    /* Below gives proper limits at end points */
    V[0]=0.; dr[N]=0.;

    getallEs(E,lmax,nmax,Z,V,r,dr,N); /* Get E */
    getPsi(psi,E[0][0],0,V,r,dr,N); /* Get Psi for E[l=0][n=0] */

    for (k=0; k<=N-1; k++) /* Get integrand for root mean square (rms) error */
      integrand[k]=(psi[k]-2*r[k]*exp(-r[k]))*(psi[k]-2*r[k]*exp(-r[k]));

    printf("%6d %20.16f\n",N,sqrt(simpint(integrand,r,dr,N)));

    /* Free up space for N dependent objects before going to next N*/
    free_dvector(r,0,N);
    free_dvector(dr,0,N);
    free_dvector(V,0,N);

    free_dvector(psi,0,N);
    free_dvector(integrand,0,N);
  }

  /* Free up all remaining objects */
  free_dmatrix(E,0,lmax,0,nmaxmax);
  free_ivector(nmax,0,lmax);
}
```

D "getallPsisdebug.c"

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

#define Nmx 400
main()
{
  /* Change of variables info */
  double *r,*dr;
  int N;

  /* Physics variables */
  double *V,Z;
  int lmax,*nmax,nmaxmax;
  double **E,***Psi;

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

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

  /* Specifications for hydrogen */
  Z=1.;
  lmax=1;

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

  nmaxmax=0; /* Find max of all nmax's */
  for (l=0; l<=lmax; l++)
    if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

  /* 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);

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

  /* Set up grid */
  N=400;
  for (k=0; k<=N; k++) {
    x=((double) k)/((double) N);
    r[k]=1./(1.-x)-1.-x-x*x;
```

```
    dr[k]=1./(1.-x)/(1.-x)-1.-2*x;
    V[k]=-Z/r[k];
  }
  V[0]=0.; dr[N]=0.;

  /* Test section */
  getallEs(E,lmax,nmax,Z,V,r,dr,N);
  getallPsis(Psi,E,lmax,nmax,V,r,dr,N);

  for (k=0; k<=N; k++)
    printf("1s: %20.12f %15.12f %15.12f\n",r[k],Psi[0][0][k],
           2*r[k]*exp(-r[k]));

  for (k=0; k<=N; k++)
    printf("2s: %20.12f %15.12f %15.12f\n",r[k],Psi[0][1][k],
           r[k]*(r[k]-2.)*exp(-r[k]/2)/sqrt(8.));


  for (k=0; k<=N; k++)
    printf("2p: %20.12f %15.12f %15.12f\n",r[k],Psi[1][0][k],
           r[k]*r[k]*exp(-r[k]/2)/sqrt(24.));

  /* 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);
}
```

E "getRhodebug.c"

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

#define Nmx 400
main()
{
  /* Change of variables info */
  double *r,*dr;
  int N;

  /* Physics variables */
  double *V,*Rho,Z;
  int lmax,*nmax,nmaxmax;
  double **E,***Psi,**F;

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

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

  /* Specifications for carbon */
  Z=6.;
  lmax=1;

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

  nmaxmax=0; /* Find max of all nmax's */
  for (l=0; l<=lmax; l++)
    if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

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

  /* 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);

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

```
  /* Set up grid */
  N=400;
  for (k=0; k<=N; k++) {
    x=((double) k)/((double) N);
    r[k]=1./(1.-x)-1.-x-x*x;
    dr[k]=1./(1.-x)/(1.-x)-1.-2*x;
    V[k]=-Z/r[k];
  }
  V[0]=0.; dr[N]=0.;

  /* Test section */
  getallEs(E,lmax,nmax,Z,V,r,dr,N);
  getallPsis(Psi,E,lmax,nmax,V,r,dr,N);
  getRho(Rho,Psi,F,lmax,nmax,N);
  printf("Total charge is: %20.15f\n",simpint(Rho,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_dmatrix(F,0,lmax,0,nmaxmax);
  free_d3tensor(Psi,0,lmax,0,nmaxmax,0,Nmx);
  free_dvector(Rho,0,Nmx);
}
```

F "getphidebug.c"

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

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

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

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

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

  /* Specs for hydrogen */
  Z=1.; /* Charge on nucleus */
  lmax=0; /* Only up to 's' states (l=0) are filled in H */

  nmax=ivector(0,lmax); /* Declare space for max nodes for each l */
  nmax[0]=0; /* Specify that only up to the first (zero node)
                's' states are filled in H */

  /* General loop for finding nmaxmax (needed to define arrays)...
     Note: This is a little silly for H, but is handy when you have
     some random element. */
  nmaxmax=0;
  for (l=0; l<=lmax; l++)
    if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

  F=dmatrix(0,lmax,0,nmaxmax); /* Allocate space for filling data */
  F[0][0]=1.; /* For H, there is 1 electron in the '1s' (l=0,n=0) state */

  /* Allocate space for table for E's */
  E=dmatrix(0,lmax,0,nmaxmax);

  for (N=40; N<=Nmx; N*=10) {
    /* Allocate space for N dependent physics arrays */
    Psi=d3tensor(0,lmax,0,nmaxmax,0,N);
    Rho=dvector(0,N);
    phi=dvector(0,N);
```

```
    integrand=dvector(0,N);

    /* Space for grid vectors */
    r=dvector(0,N);
    dr=dvector(0,N);
    V=dvector(0,N);

    /* Set up grid */
    for (k=0; k<=N; k++) {
      x=((double) k)/((double) N);
      r[k]=1./(1.-x)-1.-x-x*x;
      dr[k]=1./(1.-x)/(1.-x)-1.-2*x;
      V[k]=-Z/r[k];
    }
    V[0]=0.; dr[N]=0.;

    /* Test section */
    getallEs(E,lmax,nmax,Z,V,r,dr,N);
    getallPsis(Psi,E,lmax,nmax,V,r,dr,N);
    getRho(Rho,Psi,F,lmax,nmax,N);
    getphi(phi,Rho,r,dr,N);

    for (k=0; k<=N; k++)
      integrand[k]=phi[k]*Rho[k];
    printf("%6d %20.12f (Hartree Energy)\n",N,simpint(integrand,r,dr,N)/2);

    /* Clean up N-dependent vectors BEFORE next value of N! */
    free_dvector(r,0,Nmx);
    free_dvector(dr,0,Nmx);
    free_dvector(V,0,Nmx);

    free_d3tensor(Psi,0,lmax,0,nmaxmax,0,Nmx);
    free_dvector(Rho,0,Nmx);
    free_dvector(phi,0,Nmx);
    free_dvector(integrand,0,N);
  }

  /* Free up all remaining objects */
  free_dmatrix(E,0,lmax,0,nmaxmax);
  free_ivector(nmax,0,lmax);
}
```