Cornell University

Department of Physics

Phys 480/680, Astro 690 March 3, 2000

Computational Physics, Spring 2000

Homework Assignment # 5

(Due Thursday, March 9 at 8:40 am in Lab.)

Agenda and readings for the week of February 1:

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

- Lec9, 03/07: Solution of many non-linear equations in many unknowns (Broyden approaches) **Reading: 9.6–9.7; and Sections "Intoduction," "Proceedure" (you can skip the relativistic part), "Results," and "The LDA approximation," from http://math.nist.gov/DFTdata/atomdata/narrative.html**
- Lab4, 03/09: Self-consistent solutions for atoms

Goal: This homework develops and packages into subroutines the functionality needed to perform complete calculations for atoms.
Important note: This problem set comes with four complete main programs (A-D below) which call your subroutines to test them. Have these programs ready to run when you come to lab!

Contents

1 Odds and ends **3**
1.1 Avoiding overflow in "schint" . . . 3
1.2 Excess precision . . . 3
1.3 Memory "leaks" . . . 3

2 Complete and normalized wave functions **4**
2.1 Modifying "schint" . . . 4
2.2 "interpolate" . . . 5
2.3 "getPsi" . . . 5
2.4 Debugging . . . 6

3 Packaging **6**
3.1 "d3tensor" and "free_d3tensor" . . . 6
3.2 "getallPsis" . . . 7
3.2.1 Debugging . . . 7
3.3 "getRho" . . . 8
3.3.1 Debugging . . . 8
3.4 "getphi" . . . 8
3.4.1 Debugging . . . 8

A "test_getPsi.c" **9**

B "test_getallPsis" **11**

C "test_getRho.c" **13**

D "test_getphi.c" **15**

1 Odds and ends

As we proceed, we bury your "getallEs" routine deeper and deeper into the code, and it becomes critical that the routine be very robust and dependable. The items below help to ensure this.

1.1 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". In particular, you should add the "if" statement in the code fragment below to your "schint" routine.

```
#define SMALL (1.e-300)
#define SQRTSMALL (1.e-150)
#define BIG (1.e300)
#define SQRTBIG (1.e150)
#define TOL (1.e-12)

int schint(double *Psi, double *Psip, double Psiout[],
   int k1, int k2, double Veff[], double E,
   double r[], double dr[], int N)
{
                                .
                                .
    rk4p480(y,dydx,2,x,h,y,derivs_Schrodinger,k,dk,T,r,dr,N);
    if (fabs(*Psi)>SQRTBIG) {
      y[1]=SMALL*y[1];
      y[2]=SMALL*y[2];
    }
                                .
                                .
```

Note that above, we "#define" the various constants ("parameter" in **Fortran**). To avoid debugging problems, such constants should never appear explicitly in your code. Rather, as above, they should be defined in a way which is easily seen and easily modified.

1.2 Excess precision

Specifying too much precision for the eigenvalues in "getEs" can cause the program to get stuck in a long loop. Rather, you should use a tolerance of about 1e-12, which is both sufficiently accurate and stable. You should specify this in "getEs" with a statement like

```
                                .
                                .
    E[k]=zriddrp480(func_Schrodinger,E1,E2,TOL,0,Veff,r,dr,N);
                                .
                                .
```

with TOL defined in an appropriate #define statement.

1.3 Memory "leaks"

A "memory leak" is when you allocate space for an object in a subroutine but then never free the space when the routine is done. Your program then quickly begins to use up memory as it calls the routine over and over. This not only causes crashes, but also can slow your program dramatically. Take the time now to check your that subroutines have a call to "free_dvector" for every call to "dvector", and similarly for "free_dmatrix".

2 Complete and normalized wave functions

The routine "getallEs" 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".

2.1 Modifying "schint"

```
int schint(double *Psi, double *Psip, double Psiout[],
           int k1, int k2, double Veff[], double E,
           double r[], double dr[], int N);
```

Return value:

- number of nodes in solution between r[k1] and r[k2]

Output:

- Psi, Psip (passed by reference): values of $\Psi(r[k2])$ and $\Psi'(r[k2])$
- Psiout[]: values of $\Psi(r[k])$ generated by "rk4p480"

Input:

- Psi, Psip (passed by reference): values of $\Psi(r[k1])$ and $\Psi'(r[k1])$
- k1, k2: indices of initial and final integration points (r[k1] $\rightarrow$ r[k2])
- Veff[]: effective potential on grid
- E: proposed electron energy
- r[], dr[], N: standard grid information

Modify your "schint" routine to return the solution for $\Psi(r[k])$ for all points k=k1..k2. This involves two steps.

- First, if Psiout is not "NULL" we should record the values of the solution in Psiout:

```
                              .
                              .
for (k=k1; k!=k2; k+=dk) {
  x=0.5*k*fabs(h);
  if (Psiout!=NULL) Psiout[k]=*Psi;
  derivs_Schrodinger(x,y,dydx,k,T,r,dr,N);
  rk4p480(y,dydx,2,x,h,y,derivs_Schrodinger,k,dk,T,r,dr,N);
                              .
                              .
}
if (Psiout!=NULL) Psiout[k]=*Psi;
                              .
                              .
```

  Note that because we record the initial value of Psi *before* "rk4p480" makes each step, we require one final assignment after the completion of the loop.

- Second, each time we scale the solution to keep Psi within 1e-150 to 1e150 (See Section 1.), we must be sure to scale *all* of the results stored in Psiout. But, to avoid losses in efficiency, this should only be done if we are returning Psioutp[]. A suitable code fragment, which works for both k1<k2 and k1>k2, is

```
      if (fabs(*Psi)>SQRTBIG) {
        y[1]=SMALL*y[1];
        y[2]=SMALL*y[2];
        if (Psiout!=NULL) { /* If filling array, need to repair */
  for (kp=k; kp!=k1; kp-=dk)
    Psiout[kp]=SMALL*Psiout[kp];
  Psiout[kp]=SMALL*Psiout[kp];
        }
      }
```

2.2 "interpolate"

```
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

Write a routine "interpolate", of the above prototype, which interpolates 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 $f_{-1} = 0$ and $f_{N+1} = 0$, respectively.

2.3 "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 points

Write a routine "getPsi", of the above prototype, 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. 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, 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", and 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 your "simpint" routine.

2.4 Debugging

Cut and paste Appendix A 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 relative to the analytic solution as a function of the number of intervals N.

3 Packaging

You now have built all of the capabilities needed for atomic calculations. It is helpful at this point to organize these capabilities by packaging them into convenient routines.

3.1 "d3tensor" and "free_d3tensor"

```
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)

To store the values for each wave function $\Psi_{n\ell}(r[k])$, we need a three-dimensional, double-precision array (with indices n, ℓ, k). Numerical recipes has routines to handle such objects for normal floating point, but not double precision. 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 above.

3.2 "getallPsis"

```
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

Construct a routine of the above prototype 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.)

If you are working in **Fortran**, then you should index your arrays in the reverse order, Psi(k,n,l), so that then your code fragment will appear as

```
                    .
                    .
do l=0,lmax
  do n=0,nmax(l)
    call getPsi(Psi(1,n,l),E(n,l),l,V,r,dr,N)
                    .
                    .
```

3.2.1 Debugging

Cut and paste Appendix B from the web version of the problem set and compile and run it with your code. 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.

3.3 "getRho"

```
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 ($F_{n\ell} \equiv (2\ell+1) f_{n\ell}$)
- 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 grid points

Produce a function of the above prototype which computes the quantity

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

3.3.1 Debugging

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

3.4 "getphi"

```
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

Package your previous Poisson solver program into a routine of the above prototype. 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]. 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" 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];
            .
            .
```

3.4.1 Debugging

Test your code by compiling and running it with the main program in Appendix D. 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 "test_getPsi.c"

```
#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.;
  lmax=0;

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

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

  /* Allocate remaining vectors */

  E=dmatrix(0,lmax,0,nmaxmax); /* E's */
  psi=dvector(0,Nmx); /* psi */

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

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

  /* Convenient loop over N */
  for (N=40; N<=Nmx; N*=10) {

    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];
    }
    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_dvector(r,0,Nmx);
  free_dvector(dr,0,Nmx);
  free_dvector(V,0,Nmx);

  free_dmatrix(E,0,lmax,0,nmaxmax);
  free_dvector(psi,0,Nmx);
  free_dvector(integrand,0,Nmx);
}
```

B "test_getallPsis"

```
#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);
}
```

C "test_getRho.c"

```
#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);
}
```

D "test_getphi.c"

```
#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.;
  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);
  phi=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-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);
  }

  /* 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(phi,0,Nmx);
}
```