Cornell University

Department of Physics

Phys 480/680, Astro 690 February 18, 2002

Computational Physics, Spring 2002

Homework Assignment # 3

(Due Tuesday, February 26 at 8:40 am in computer lab.)

Agenda and readings for the week of February 12:

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.

Goal: This problem set develops a fully automated solver which finds all of the energy states for any specified external potential "V[]" to high precision.

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:

- Lec6, 02/12 (Tue: mini-lecture, 25mins) Runge-Kutta; Overview of ODE solvers.
  Reading: NR 16.0-16.1.
- Lab1, 02/12 (Tue): Change of variables; Scaling of errors for $\int \sqrt{(}x)\,dx$, and for periodic functions
- Lec7, 02/14 (Thu): Derivation of Kohn-Sham equations, Schrödinger's equation for atoms (spherically symmetric solutions); boundary value problems; standard form for root finding; bisection
  Reading: NR 17.0-17.1, 9.0-9.1

Preview of coming week:

- Lec8, 02/19 (Tue: mini-lecture, 25mins) Overview of root finding; secant method; convergence rates
  Reading: NR 9.2
- Lab2, 02/19 (Tue): Convergence of solution of ODE's; numerical solutions of Schrödinger's equation.
- Lec9, 02/21 (Thu): Ridder's method for roots; bracketing of roots for Schrödinger's equation; numerical interpolation
  Reading: NR 9.2, 3.0-3.2
- Lec10, 02/26 (Tue: mini-lecture, 25mins): Numerical stability of integrating ODEs
  Reading: NR 17.2
- Lab3, 02/23: Spectrum of hydrogen, convergence, stability of solutions, effectiveness of root finders

Contents

1 Unix tips and suggestions **2**
1.1 "p480.h" 2
1.2 "makefile" 2

2 Schrödinger integrator: schint() 3
2.1 "physics.c" and "physics.h" . . . 3
2.2 Software development hint: packaging main programs as subroutines . . . 3
2.3 schint() . . . 3
2.4 derivs_Schrodinger() . . . 5
2.5 Good programming practice: bug trapping . . . 5
2.6 Debugging . . . 5
2.6.1 Endpoint values . . . 5
2.6.2 Output wave functions . . . 6
2.6.3 Number of nodes . . . 6

3 Schrödinger energy solver 6
3.1 rtbisp480() . . . 6
3.2 func_Schrodinger() . . . 7
3.3 Debugging . . . 7
3.4 zriddrp480() . . . 7

4 Automated Schrödinger energy solver 8
4.1 func_SchrodingerNodes() . . . 8
4.2 getEs() . . . 9
4.3 Debugging . . . 10

A "schdebug.c" 11

B "Edebug.c" 14

C "getEdebug.c" 16

1 Unix tips and suggestions

1.1 "p480.h"

Sometimes header files (like "p480.h") end up included twice, once in the main program and once in another header file, which can cause difficulties at compile time. A standard trick to avoid this is to have the header define a variable for the C preprocessor and also to check whether it is defined. To do this simply add the following two lines to the very top of your "p480.h" file,

```
#ifndef _P480_H_
#define _P480_H_
```

and the following line to the very bottom,

```
#endif
```

Each ".h" file which you make should have its own distinct marker variable like this to avoid multiple inclusions.

1.2 "makefile"

Unix makefiles have many useful features. For instance, you can define variables so that you do not have to retype the same information over and over again. More importantly, this makes it easy to include new changes to every entry without having to go through each one. You only need add the new entries to the line defining the appropriate variable.

Below is an example makefile which should work with last week's assignment. You should use this as an example to build upon as the starting point for this week. (Remember that the spaces before the cc's are always tab characters!)

```
LIBRARIES = -lm
OBJECTS   = p480.o nrutil.o
HEADERS   = p480.h nrutil.h

rk4debug: rk4debug.c $(OBJECTS) $(HEADERS)
        cc -o rk4debug rk4debug.c $(OBJECTS) $(LIBRARIES)

poisson: poisson.c $(OBJECTS) $(HEADERS)
        cc -o poisson poisson.c $(OBJECTS) $(LIBRARIES)

quicksch: quicksch.c $(OBJECTS) $(HEADERS)
        cc -o quicksch quicksch.c $(OBJECTS) $(LIBRARIES)
```

2 Schrödinger integrator: schint()

This routine will ultimately have two uses: (1) to compute the solution $\Psi(r)$ to Schrödinger's equation for a given energy E, (2) to return information to allow us to determine the energies E. In this problem set, we focus on (2), determining E, and thus one of the arguments of the function (Psiout[]) will not be used until later.

2.1 "physics.c" and "physics.h"

The Schrödinger is a new type of subroutine for us. It isn't a basic mathematical routine like simpint() or rk4p480(), but a physics subroutine. We will have a number of these, and thus it is useful to keep them in their own file, which we shall call "physics.c". Separating these routines means that we will need yet another header file "physics.h" to keep the function prototypes. If you are working in Unix, this also means that you will have to add "physics.o" and "physics.h" to the definition lines for OBJECTS and HEADERS in your makefile.

As you know, rk4p480() will want to call a function specifying the differential equation, derivs_Schrodinger() in this case. As this function is only called through schint(), it is not necessary or appropriate to include the prototype in "physics.h". Be sure to include derivs_Schrodinger() in "physics.c" *before* schint() so that the compiler will have the prototype for derivs_Schrodinger() while it is working on schint().

2.2 Software development hint: packaging main programs as subroutines

Bugs in new subroutines frequently have nothing to do with the routine itself, but with the calling of the routine through the subroutine interface. To reduce the complexity of such debugging, often it is useful to first write the subroutine as a main program to get it up and running and debugged. Then, once you *know* for sure that the code works, cut the relevant parts from the main program and paste them to the subroutine and make sure that you produce exactly the same results. In the first phase, you can be sure that there are no subroutine interface bugs and, in the second phase, you can be sure that there are no actual bugs in the routine. Reducing the possible phase space of bugs always speeds the debugging process, often exponentially.

This is precisely what we have done with "quicksch.c" in the last assignment. This way, most of this problem involves just repackaging "quicksch.c" as schint().

2.3 schint()

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

Return value:

- number of "nodes" (zero crossings) in solution between r[k1] and r[k2]

Output:

- Psi, Psip (passed by reference): values of $\Psi(r[k2])$ and $\Psi'(r[k2])$
- Psiout[]: Psi(r[k]) along the entire integration

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])
- V[]: potential on corresponding grid points, r[]
- E: electron energy
- r[], dr[], N: standard grid information

As we learned in Lab2, solving for the energies can be accomplished by setting the boundary conditions at one end point of the problem (e.g., $r(x = 0) = 0$), and investigating the resulting behavior at the other end point (e.g., $r(x = 1) = \infty$). As we will see later in this problem set, it is also very useful to count the number of times that the solution crosses zero (number of "nodes"). Accordingly, write a function of the above form in "physics.c" and include its prototype in "physics.h".

Note that to write this routine, you should be able to take the appropriate parts from your "quicksch.c" program and repackage them as a subroutine. There are a number of details to consider. Among these are

1. the integration range is no longer k=N to k=0, but k=k1 to k=k2;
2. there are two cases k1>k2 and k1<k2;
3. you are to store in Psiout[k] all the values for $\Psi(r[k])$ as they are generated;
4. you must set up a counter for the number of times that $\Psi(r)$ (i.e., y[1]) changes sign;
5. you must construct T[]=E-V[] on your own.

The last of these considerations is a bit tricky because T[] is not passed in as a variable, and the routine thus must allocate the space for the array it without "knowing" how much space is needed until the actual call is made. One solution would be to declare a fixed, large amount of space, but this is wasteful and can cause bugs if the size is exceeded. The best way to handle this kind of situation is "dynamic memory allocation," where the space for T[] is allocated while the programming is running. Fortunately, nrutil.c includes functions for doing just this. To accomplish this, you should declare the variable T (technically, a pointer) in the declarations section of schint() using

```
double *T;
```

and, then, allocate the space for T[] as one of the first statements in the program using

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

This latter statement sets up space for an array T[] which holds values T[0], T[1], ..., T[N]. The last, but most important, detail in dynamic memory allocation is to be sure to free up the space. Otherwise, each time the routine is called, it will eat up more program space, and eventually your code will slow down and crash (and possibly even crash the machine...)!

To avoid this, you should clean up the space when you are done with it, by including the following statement at the very end of schint(),

```
free_dvector(T,0,N);
```

2.4 derivs_Schrodinger()

```
void derivs_Schrodinger(double x, double y[], double dydx[],
                        int k, double T[], double r[], double dr[], int N)
```

Output:

- dydx[]: value of $\frac{d}{dx}\vec{y}$, defined as below

Input:

- x: value of independent variable (not used)
- y[]: y[1]=$\Psi(r)$, y[2]=$\Psi'(r)$
- k: integer index of arrays corresponding to x (x=k/N), so that r(x)=r[k], etc.
- T[]: kinetic energy, T[k]=T(r[k])
- r[], dr[], N: standard grid information

Include a function of the form above in the file "physics.c" *before* the code for schint(), where

$$\frac{d}{dx}\vec{y} = \begin{pmatrix} \Psi \\ \Psi' \end{pmatrix} = \begin{pmatrix} \Psi'(r(x))\frac{dr}{dx}(x) \\ -2T(r(x))\Psi(r(x))\frac{dr}{dx}(x) \end{pmatrix}.$$

2.5 Good programming practice: bug trapping

The Runge-Kutta algorithm always advances the solution two points at a time (dk=2). Thus, k2−k1 should always be an *even* number. This is quite clear to us now as we are designing the routine. However, in a few weeks schint() will be buried deep in the heart of our code, and we are likely to forget about this restriction and produce code with bugs which are quite difficult to track.

To avoid such problems, well designed routines will, under all possible circumstances, either work appropriately *or* print a message as to the nature of the problem (including the values of the offending variables so that you'll know what the problem is) and then exit the program.

Thus, you should make one of the first things that schint() does to check that k2-k1 is even number and, if not, to print a very explicit message such as "Error in schint: k2-k1 must be even in routine schint, but k1=999 and k2=0." and exit the program with the statement

```
exit(1);
```

The argument "1" is a standard Unix argument to signal to the operating system that the program did not end happily.

2.6 Debugging

Compile and run the main program "schdebug.c" in Appendix A with your schint() routine. This program inputs E and then calls your schint() routine with N=100, k1=0, k2=N, Psi=0, Psip=1, and $V = -1/r$. The program then prints out both your solution and the analytic solution at $E = -0.5$, which is r[k]*exp(-r[k]), and the ratio between these solutions. Finally, the program integrates backwards, calling your schint() routine with N=100, k1=N, k2=0, Psi=0, Psip=1, and $V = -1/r$, and printing a summary of the results.

Thorough debugging requires that you verify *all* of the outputs of the routine. In this case, schint() returns three separate quantities, which you should verify as below.

2.6.1 Endpoint values

Run the program with E=-0.5, and verify that the "Return value for Psi" (listed near the top of each test) matches the corresponding value for Psiout[] from the detailed printout.

2.6.2 Output wave functions

Again, run the program with E=-0.5. Your results for the outward integration should agree with the analytic solution to about 4-5 decimal places (except for k>80). Because the Schrödinger equation determines the wave function only up to a constant normalization factor, the results for the inward integration will not match the analytic solution, but the ratio should be constant to within about 4-5 decimal places, except for numerical errors near the ends of the integration interval.

Note: The facts that $V(r) = -1/r$ and r[0]=0 cause numerical difficulties at the origin. To avoid this, "schdebug.c" uses the change of variables map $r(x) = x^2/(1-x)$. Under this map, near the origin $r \approx x^2$, $V_{eff}(r) \sim 1/x^2$, $\Psi(r) \sim x^2$ and $\frac{dr}{dx} \sim 2x$. Thus, it is mathematically correct to regard the offending term $V_{eff}\Psi\frac{dr}{dx} \sim x \to 0$ as zero at the origin. The program "schdebug.c" achieves the same effect, but without the division by zero and the NaN's, by simply setting V[0]=0 when initializing the array V[].

2.6.3 Number of nodes

Verify that the number of nodes which your code outputs (printed near the top of each test) is zero for E less than (about) -0.5, 1 for E between (approximately) -0.5 and -0.125, 2 for E between (approximately) -0.125 and -0.0556, and more than 2 for higher values of E.

3 Schrödinger energy solver

As discussed in lecture, our basic strategy to find the energies of the atom is to set $\Psi = 0$ and $\Psi' = 1$ at one end of our solution ($r = 0$ or $r \to \infty$), define a function $f(E)$ as the value of Ψ at the other end, and then seek the roots of $f(E)$.

As discussed in class, there are a variety of methods for *root finding*, solving one equation in one unknown of the form $f(x) = 0$. Computationally, these methods depend upon calling a function subroutine which returns the value of the function $f(x)$ for any input value x. In this problem, we will implement versions of two of the *Numerical Recipes* subroutines for doing this, rtbisp480() and zriddrp480(). As with rk4p480(), we will also have to write a helper routine, which in this case we call func_Schrodinger().

3.1 rtbisp480()

```
double rtbisp480(double (*func)(double,int,double [],double [],double[],int),
                 double x1, double x2, double xacc,
                 int match, double V[], double r[], double dr[], int N)
```

Return value:

- solution x to equation $f(x)$=0

Input:

- func: subroutine which returns the values $f(x)$
- x1, x2: bracket on desired solution, x1<x<x2
- xacc: accuracy of desired solution
- match: dummy variable (needed for Problem 4)
- V[]: potential on corresponding grid points, r[]
- r[], dr[], N: standard grid information

Modify the *Numerical Recipes* bisection routine rtbis() (NR 9.1) to produce a routine of the above prototype in "p480.c". (Don't forget to add the prototype to "p480.h"!) You should modify rtbis so that it (1) accepts and then (2) passes to all calls to *func (which will actually be func_Schrodinger in our case) the additional arguments match, V[], r[], dr[] and N (to define the physical potential and grid information

). Also, make sure to remember to convert rtbis() to double precision, for which you should also double the value of JMAX.

So that we can study the working of our code, modify the routine so that it prints out the final number of iterations and the final function value just before it returns rtb, with a print statement of the form

```
printf("   (%d iterations in rtbisp480, func=%e)\n",j,fmid);
```

Finally, the numerical recipes routine rtbis() has a small bug: If one of your two initial guesses x1 or x2 for the solution gives f(x)=0, the routine prints an error message and halts the program. Modify rtbisp480() to test for these conditions and to return the appropriate value, x1 or x2, as the solution (as well as to print out that zero iterations were required to find the solution).

3.2 func_Schrodinger()

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

Return value:

- Value of Psi returned by schint() at end of integration interval

Input:

- E: energy variable in Schrödinger equation used in schint().
- match: dummy variable (needed for Problem 4)
- V[]: potential energy on the grid
- r[], dr[], N: standard grid information

Design a routine func_Schrodinger() with the above prototype which calls upon schint() to integrate Schrödinger's equation from k1=0 to k2=N using Psi=0 and Psip=1 as initial conditions, and then returns the final value of Psi. This routine is most appropriate for the file "physics.c". (Don't forget to put the prototype in "physics.h"!)

Hint: You will have to define an internal variable array Psiout[] to send into schint(). You should accomplish this dynamically in func_Schrodinger() just as you did for T[] in schint().

3.3 Debugging

To test your routines, run and compile them with the main program "Edebug.c" in Appendix B. This program takes as inputs E1 and E2 as initial lower and upper bounds on the energy for which you wish to solve, and then asks for different values of N, the number of integration intervals. Running the code with E1=-1 and E2=-0.3, you should be able to solve for the ground state and verify that it approaches the analytic result (E=-0.5) with the expected behavior for N=10,100,1000,10000. Similarly, you should be able to find all of the excited states, which have analytic solutions $E = -1/(2n^2)$, for $n = 1, 2, \ldots$.

3.4 zriddrp480()

```
double zriddrp480(double (*func)(double,int,double[],double[],double[],int),
                  double x1, double x2, double xacc,
                  int match, double V[], double r[], double dr[], int N)
```

Return value:

- solution x to equation $f(x)$=0

Input:

- func: subroutine which returns the values $f(x)$

- x1, x2: bracket on desired solution, x1<x<x2
- xacc: accuracy of desired solution
- match: dummy variable (needed for Problem 4)
- V[]: potential on corresponding grid points, r[]
- r[], dr[], N: standard grid information

A much more powerful algorithm for finding zeros of a function is Ridder's method. This method works with the same basic information as the bisection method and so can be called in exactly the same way, but requires far fewer function evaluations to reach the same accuracy.

Make the same modifications to zriddr() that you made to rtbis() to produce a function of the above prototype in "p480.c" and "p480.h". Be sure print out the final number of iterations needed and final function value.

Finally, the numerical recipes routine zriddr() also has a small bug: for wildly varying functions such as func_Schrodinger(), zriddr() sometimes returns too early. To fix this, replace the code fragment

```
if (fabs(xnew-ans) <= xacc) ...
```

in zriddrp480() with

```
if (fabs(xl-xh) <= xacc) ... .
```

To test your function, uncomment out the lines labeled "Test of zriddrp480()" in "Edebug.c", and verify that zriddrp480() returns the same results as rtbisp480(), but with far fewer function calls.

4 Automated Schrödinger energy solver

The solvers in Problem 3 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()!

4.1 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.

4.2 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!

4.3 Debugging

Compile and run your routines with the main program "getEdebug.c" in Appendix C. 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.

A "schdebug.c"

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

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

  /* Physics vectors */
  double *V,*Psianal,*Psiout;

  /* Working variables */
  int k;
  double x;
  int k1,k2,nodes;
  double Psi,Psip,E;

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

  /* Loop over different input energies for fixed N=100 */
  N=100;
  while (1) {
    printf("E=");
    scanf("%lf",&E);

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

    /* 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];
      Psianal[k]=r[k]*exp(-r[k]);
    }
    /* Special conditions for handling end-points of integration */
    V[0]=0.;
    dr[N]=0.;

    /* Outward integration test... */
```

```
Psi=0.;
Psip=1.;
/* Because Psi and Psip contain both input and output values, they
   must be passed as pointers (hence the '&') */
/* Psiout, as an array, is already a pointer, so no '&' is needed. */
/* We could just use "0,N," instead of "k1=0,k2=N,", but this notation
   makes the code easier to read, i.e., debug. */
nodes=schint(&Psi,&Psip,Psiout,k1=0,k2=N,V,E,r,dr,N);

/* Only compare on even points, as these are the only places where
   RK generates values */

printf("============================================================\n");
printf("Outward integration test:\n");
printf("-------------------------\n");
printf("   %d nodes\n",nodes);
printf("   Return value of Psi: %16.12f\n\n",Psi);

printf("   k           r             Psianal          Psiout       Psiout/Psianal\n");
for (k=0; k<=N; k+=2) {
  printf("%5d %16.12f %16.12f %16.12f %16.12f\n",
 k,r[k],Psianal[k],Psiout[k],Psiout[k]/Psianal[k]);
}

/* Inward integration test... */
Psi=0.;
Psip=-1.;
/* Because Psi and Psip contain both input and output values, they
   must be passed as pointers (hence the '&') */
/* Psiout, as an array, is already a pointer, so no '&' is needed. */
/* We could just use "0,N," instead of "k1=0,k2=N,", but this notation
   makes the code easier to read, i.e., debug. */
nodes=schint(&Psi,&Psip,Psiout,k1=N,k2=0,V,E,r,dr,N);

/* Only compare on even points, as these are the only places where
   RK generates values */

printf("============================================================\n");
printf("Inward integration test: %d nodes\n",nodes);
printf("-------------------------\n");

printf("Outward integration test:\n");
printf("-------------------------\n");
printf("   %d nodes\n",nodes);
printf("   Return value of Psi: %16.12e\n\n",Psi);

printf("   k           r             Psianal          Psiout       Psiout/Psianal\n");
for (k=0; k<=N; k+=2) {
  printf("%5d %16.12f %16.12f %16.12e %16.12e\n",
 k,r[k],Psianal[k],Psiout[k],Psiout[k]/Psianal[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(Psianal,0,N);
  }

}
```

B "Edebug.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,match;
  double x;
  double E,E1,E2,tol;

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

  /* Input energy bounds */
  printf("E1="); scanf("%lf",&E1);
  printf("E2="); scanf("%lf",&E2);
  /* Loop over different input N */
  while (1) {
    printf("Search for energy in range (%20.16f,%20.16f):\n",E1,E2);
    printf("N=");
    scanf("%d",&N);

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

    /* 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.;

    E=rtbisp480(func_Schrodinger,E1,E2,tol=1e-15,
```

```
match,V,r,dr,N);
    printf("Solution from rtbisp480: E=%20.16f\n",E);

    /* Test of zriddrp480 */
    /* Uncomment lines below when zriddrp480 is ready */
    /* E=zriddrp480(func_Schrodinger,E1,E2,tol=1e-15, */
    /*              match,V,r,dr,N); */
    /* printf("Solution from zriddrp480: E=%20.16f\n",E); */

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

}
```

C "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);
  }

}
```