Cornell University

Department of Physics

Phys 480/680, Astro 690 February 16, 2000

Computational Physics, Spring 2000

Homework Assignment # 3

(Due Thursday, February 24 at 8:40 am in Lab.)

Agenda and readings for the week of February 1:

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

- Lec7, 02/22: Schrödinger's equation for higher angular momenta $\ell > 0$; Matching method for eigenvalue problems, Bracketing of roots
  Reading: 9.0-9.2, 17.0-17.2
- Lab3, 02/24: Spectrum of hydrogen, convergence, stability of solutions, effectiveness of root finders

Contents

1 Getting organized **2**
1.1 Routine naming conventions . . . 2
1.2 p480.h . . . 2
1.3 Makefile . . . 3

2 "schint": outward Schrödinger integrator **3**
2.1 "derivs_Schrodinger" . . . 3
2.2 Integrating outward . . . 4
2.3 Trapping bugs . . . 4
2.4 Packaging a new subroutine . . . 4

3 Debugging "schint" **4**

4 Solving Schrödinger's equation **5**
4.1 "func_Schrodinger" . . . 5
4.2 "rtbisp480" . . . 5
4.3 "main" . . . 5
4.4 "zriddrp480" . . . 6

5 Inward Schrödinger integrator **6**
5.1 Modifying "schint" . . . 6
5.2 Modifying "func_Schrodinger" . . . 7

1 Getting organized

By the end of this problem set, we will have developed a fair number of routines toward our goal of density functional calculations for atoms. Now that we have some idea about the basic structure which our routines assume, this is the time to focus on organization.

To help with organization, you will have to make some small changes to your previous software, as the items" below summarize.

1.1 Routine naming conventions

By this point, it should be clear that there are four types of routine which we develop:

1. modified, double precision versions of the *Numerical Recipes* routines ("rk4" and, below, "rtbis" and "zriddr");
2. our own specialized versions of the subroutines called by the *Numerical Recipes* routines ("derivs" and, below, "func");
3. our own routines which call upon the *Numerical Recipes* routines;
4. our own, stand-alone routines ("trapint" and "simpint").

For the modified *Numerical Recipes* routines (1), it would be more systematic to name them all with the standard *Numerical Recipes* name, but with the extension "p480". Thus, for example, "rk4ind" now becomes "rk4p480". We shall use this convention from now on.

> **Action item:** Rename "rk4ind" to "rk4p480" and make appropriate changes to your Poisson solver.

As we shall have several different versions of the utility routines which the *Numerical Recipes* routine call (2) for solving different equations, such as "derivs" for the Poisson or the Schrödinger equation, we shall name them with an extension corresponding to their use. Thus, below, we introduce "derivs_Schrodinger."

> **Action item:** Rename your versions of "dervis" for the Poisson solver as "derivs_Poisson", and make appropriate changes to the call to "rk4p480" and to your Poisson solver program.

To keep things organized, for our routines which call upon *Numerical Recipes* (3) which then call upon specialized subroutines, we should keep both our routine and the specialized subroutine together in the same file. For instance, ""schint," the Schrödinger integrator which calls "rk4p480," should be kept in the same file right next to "derivs_Schrodinger."

Finally, our completely stand-alone routines (4), such as "trapint" and "simpint" are fine. It would be useful to keep them grouped together in files with appropriate names, such as integ.c for these integrators.

1.2 p480.h

This and the next item do not require any work on your part. These are useful files which you can cut and paste (from the web version of the problem set) for your own use for those programming in C.

If you are programming in C, it will be useful to have a header file "p480.h" will all of the prototypes for our routines gathered together, so that you can include them quickly with an include statement:

```
#include "p480.h"
```

At present, my "p480.h" file, including the prototypes for the new routines from this homework, appears as

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

```
double trapint(double f[],double x[],double dx[],int N);
double simpint(double f[],double x[],double dx[],int N);

void rk4p480(double y[], double dydx[], int n, double x, double h,
     double yout[],
     void (*derivs)(double, double [], double [],
    int, double [], double [], double [], int),
     int k, int dk, double f[], double r[], double dr[], int N);

double rtbisp480(double (*func)(double,int,double [],double [],double[],int),
double x1, double x2, double xacc,
int kmatch, double Veff[], double r[], double dr[], int N);

double zriddrp480(double
      (*func)(double,int,double[],double[],double[],int),
      double x1, double x2, double xacc,
      int kmatch, double Veff[], double r[], double dr[], int N);

#endif
```

1.3 Makefile

Also, this is about the time (if you are working with Unix) to make a good makefile. (The Microsoft compiler takes care of these things for you.) For your reference, my "makefile" at present appears as

```
OPTS      = -g
LIBRARIES = -lm
OBJECTS   = integ.o rk4p480.o rtbisp480.o zriddrp480.o nrutil.o

.c.o:
        cc ${OPTS} -c $<

a.out: main.o $(OBJECTS)
        cc ${OPTS} main.o $(OBJECTS) $(LIBRARIES)
```

2 "schint": outward Schrödinger integrator

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

The first step in making a Schrödinger solver is to integrate the Schrödinger equation with a given energy E. Following your Poisson solver as a guide, design a routine "schint" with the above prototype which (1) takes Psi and Psip as the values of Ψ and Ψ', respectively, at the point r[k1], (2) integrates the Schrödinger equation to the point r[k2] using Runge-Kutta, and (3) returns in Psi and Psip the final values of Ψ and Ψ'.

As you design this routine, you will encounter several issues for which you will find help and suggestions below.

2.1 "derivs_Schrodinger"

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

In standard form with change of variables, the equation we are solving is

$$\frac{d}{dx}\begin{pmatrix}\Psi\\ \Psi'\end{pmatrix} = \begin{pmatrix}\Psi'(r(x))\frac{dr}{dx}(x)\\ -2(E-V_{eff}(r(x)))\Psi(r(x))\frac{dr}{dx}(x)\end{pmatrix}.$$

Note that this equation requires $E - V_{eff}(r)$, but, so that we can use the same "rk4p480" we made for the Poisson solver, we only have room in the prototype for one double precision array T[] in addition to the change of variables information (r[], dr[], N). You thus should define an array T[k]=E−Veff[k] in "schint", and pass it into "rk4p480" and "derivs_Schrodinger" in the place which we used for rho[] in the Poisson solver.

2.2 Integrating outward

The primary difference between this and your previous Poisson routine is that are going to be integrating outward from $r = 0$ so that k2>k1. Thus, be sure to define h and dk appropriately and to make sure that your loop functions in increasing order.

2.3 Trapping bugs

The Runge-Kutta algorithm always advances the solution two points at a time (dk=2). Thus, k2−k1 should always be an *even* number. Also, we are designing "schint" to integrate outward so that k2−k1 should also always be positive. It is clear that we are making these assumptions now. However, in a few weeks "schint" will be buried deep in the heart of our code, and we are likely to forget about these restrictions and produce code with bugs which are quite difficult to track.

To avoid such problem well designed routine 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 ("stop" in Fortran) the program.

In this case, you should test than k2-k1 is a positive, even number, and, if not, print very explicit messages such as "Error in schint: k2-k1 must be even in routine schint, but k1=999 and k2=0."

2.4 Packaging a new subroutine

Finally, 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, you may which 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.

3 Debugging "schint"

To debug your Schrödinger integrator routine, write a main program which calls "schint" with k1=0, Psi=0, Psip=1, $V_{eff} = -1/r$ and $E = -0.5$. To check that your routine is working have "schint" print out its solution for each k along with r[k]*exp(-r[k]), which is the exact 1s wave function. With N=100 points, your results should agree to about 4-5 decimal places (except for k>80).

Finally, once you are sure that schint it working, turn off the print statements!

Hint: Because $V_{eff}(r) = -1/r$ and r[0]=0, you may have numerical trouble at the origin. However, if you use the map $r(x) = x^2/(1-x)$, then near the origin $V_{eff}(r) \sim 1/x^2$, $\Psi(r) \sim x^2$ 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. You can get the same effect in your code, but without the division by zero and the NaN's, by simply setting Veff[0]=0 when you initialize the array Veff[].

4 Solving Schrödinger's equation

4.1 "func_Schrodinger"

```
double func_Schrodinger(double E,
 int kmatch, double Veff[], double r[], double dr[], int N)
```

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.

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.

Note that we do not make use of the argument kmatch but will use it in the next homework.

4.2 "rtbisp480"

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

Modify the *Numerical Recipes* bisection routine "rtbis" with the above prototype so that it (1) accepts and then (2) passes to all calls to (*func) the additional arguments kmatch, Veff[], r[], dr[] and N which "func_Schrodinger" requires. Also, make sure to remember to convert "rtbis" to double precision, for which you should also double the value of JMAX.

Finally, 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);
```

4.3 "main"

```
            .
            .
            .
read N from keyboard
            .
            .
            .
/* Set up change of variables grid */
for (k=0; k<=N; k++) {
  u=((double) k)/((double) N);
  r[k]=1./(1.-u)-1.-u-u*u;
  dr[k]=1./(1.-u)/(1.-u)-1.-2*u;
  Veff[k]=-1./r[k];
}
Veff[0]=0.; dr[N]=0; /* Prevents NaN errors at origin and infinity */

kmatch=0; /* kmatch is just a dummy argument for now */
E1=-1.; E2=-0.25; /* 1s state is in between here */
```

```
/* Call solver */
E=rtbisp480(func_Schrodinger,E1,E2,1e-15,
    kmatch,Veff,r,dr,N);

printf("N=%6d E=%20.15f\n",N,E); /* Give result to all digits */
            .
            .
            .
```

Write a main program along the lines above which solves and prints out to 15 digits the 1s energy of the hydrogen atom ($V_{eff} = -1/r$).

Note: Be certain to accept N as input from the keyboard so that we can study the convergence properties of the solution in lab.

4.4 "zriddrp480"

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

Once you have "rtbisp480" working, make the same modifications to "zriddr", making certain to printout the final number of iterations needed and final function value.

Also, change your main program to print out the results for E both from "rtbisp480" and "zriddrp480" as well...

```
            .
            .
            .
E=rtbisp480(func_Schrodinger,E1,E2,1e-15,
    kmatch,Veff,r,dr,N);
printf("N=%6d E=%20.15f\n",N,E);

E=zriddrp480(func_Schrodinger,E1,E2,1e-15,
    kmatch,Veff,r,dr,N);
printf("N=%6d E=%20.15f\n",N,E);
            .
            .
            .
```

5 Inward Schrödinger integrator

Our future also requires us to integrate Schrödinger's equation in the inward direction. A hall mark of well organized software is that adding new capabilities like this be straightforward.

5.1 Modifying "schint"

Generalize "schint" so that it tests whether k1<k2 (outward integration) or k1>k2 (inward integration) and then acts appropriately: changes the increment in the "for" loop and assigns positive values to h and dk. If you are clever, you can redesign the new code so that the same "for" loop handles both cases. Avoiding such replication of code (where you have nearly two identical "for" loops) also helps to reduce bugs. Often, when you make some change, you will forget to change the replicants.

5.2 Modifying "func_Schrodinger"

With "schint" generalized, you can switch quickly from inward to outward integration by changing a single line in "func_Schrodinger"

```
k1=0; k2=N; /* Outward integration */

k1=N; k2=0; /* Inward integration */
```

Have both of these lines (one commented out) ready to run when you come to lab.