Cornell University

Department of Physics

Phys 480/680, Astro 690 February 11, 2002

Computational Physics, Spring 2002

Homework Assignment # 2

(Due Tuesday, February 19 at 8:40am in Computer Lab.)

Agenda and readings for the week of February 5:

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.

- Lec1 01/22 (Tue): Course overview; Introduction to density functional theory.
  Reading: LN "Overview of quantum mechanics . . . ": 1, 2, 3.1-3.2, 4
- Lec2 01/29 (Tue): Atomic Units; Formal Density Functional Theory — Hohenberg-Kohn Theorem, Kohn-Sham Theorem.
  Reading: LN "Overview of quantum mechanics . . . ": 5
- Lec3, 01/31 (Thu): Overview of numerical integration; regular interval methods; midpoint rule, error analysis
  Reading: LN: "Numerical Integration . . . " 1-3; NR 4.0-4.2.
- Lec4, 02/05 (Tue): Trapezoidal rule; Richardson Extrapolation; Simpson's rule; Romberg integration; singular integrals and infinite domains via change of variables
  Reading: LN: "Numerical Integration . . . " 4-5; NR 4.3-4.4.
- Lec5, 02/07 (Thu): Hartree potential in atoms; Poisson's equation; numerical solution of ordinary differential equations (ODEs) — standard form, change of variables, first and second order methods
  Reading: NR 16.0-16.1.

- Lec6, 02/12 (Tue: mini-lecture, 25mins) Runge-Kutta; practical issues; 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; IEEE arithmetic; care with change of variables at limiting cases.

- Lec7, 02/14 (Thu): Schrödinger's equation for atoms (spherically symmetric solutions); boundary value problems; standard form for root finding; bisection; secant method; convergence rates
  Reading: NR 17.0-17.1, 9.0-9.1
- Lec8, 02/19 (Tue: mini-lecture, 25mins) Overview of root finding; Bracketing of roots for Schrödinger's equation.
- Lab2, 02/19 (Tue): Convergence of solution of ODE's; numerical solutions of Schrödinger's equation.

Note: For the programs in this problem set we face a small notational difficulty. We have been using the variable $n(r)$ to represent the number of electrons per unit volume throughout space. It would make sense then to define an array $n[]$ to hold the values of the electron density. However, n is so commonly used as an indexing variable (particularly in *Numerical Recipes*) that this is likely to cause difficulties. Thus, we shall use the array $rho[]$ to represent the electron density $n(r)$.

Contents

1 Modifying *Numerical Recipes* ODE routines **2**
- 1.1 "rk4p480()" . . . 2
- 1.2 "derivs_int()" . . . 3
- 1.3 File organization . . . 3
  - 1.3.1 "nrutil.c" . . . 3
  - 1.3.2 "nrutil.h" . . . 3
  - 1.3.3 "p480.c" . . . 4
  - 1.3.4 "p480.h" . . . 4
  - 1.3.5 "makefile" . . . 4
- 1.4 Debugging . . . 4

2 Solving Poisson's equation **4**
- 2.1 "poisson.c" . . . 4
- 2.2 Kernel for Poisson's equation . . . 5
- 2.3 Poisson's equation solver . . . 5
- 2.4 Running and Debugging . . . 5

3 Schrödinger's equation **5**
- 3.1 Running and debugging . . . 5

A "rk4debug.c" **6**

B "poisson.c" **8**

C "quicksch.c" **10**

1 Modifying *Numerical Recipes* ODE routines

This is the first time that we shall use routines from *Numerical Recipes*. Often these routines meet our needs directly. However, in this case, the *Numerical Recipes* routines lack a number of important capabilities. In particular, we must modify these routines to allow us to (a) work in double precision, (b) effect a change of variables, (c) solve equations which involve an function whose values are stored in an array.

1.1 "rk4p480()"

Copy (carefully) the numerical recipes routine "rk4" from NR 16.1 to a file named "p480.c" *and modify* it to produce a function of the following form:

```
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)
```

Output:

- yout[]: Vector solution of ODE advanced one step according to Runge-Kutta.

Input:

- y[]: Value of initial vector.
- dydx[]: space to store evaluation of derivative vector returned by routine "derivs()".
- n: dimensionality of the ODE.
- x: initial value of independent variable in ODE.

- h: interval width.
- derivs: pointer to routine which evaluates derivative vector.
- k: index of arrays f[], r[], dr[] which corresponds to x ($k = Nx$).
- dk: change in array index corresponding to h ($dk = Nh$).
- f[]: array of function values which may enter ODE.
- r[], dr[], N: change of variables information as in Problem Set #1.

Notes:

Be certain to modify "rk4" to function *entirely* in double precision. For this, you must not only declare all variables double precision, but also must replace the calls to "vector()" and "free_vector()" with calls to to "dvector()" and "free_dvector()", repectively.

To be able to access the input arrays f[], r[], dr[] properly, treat the indices k and dk as integer versions of x and h: (1) when "rk4p480()" defines $xh = x + hh$ with $hh = h/2$, define $kh = k + dk/2$; (2) whenever "rk4p480()" calls "derivs()", add five new appropriate arguments to the call (k,f,r,dr,N); (3) make sure that the appropriate value for k (either k, kh or $k + dk$) is passed to each call to "derivs()", paralleling how rk4p480 treats x.

1.2 "derivs_int()"

Write a derivs routine of the following form appropriate for solution of the equation $\frac{dy}{dx} = f(x)$, a simple *one*-dimensional ODE. Note that because the ODE is one-dimensional, there is only one component, $\frac{dy}{dx}$=dydx[1].

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

Output:

- dydx[1]: derivative $\frac{dy}{dx}(x) = f(x)$

Input:

- x: point where $\frac{dy}{dx}(x)$ is evaluated.
- y[1]: value of $y(x)$.
- k: array index corresponding to x, so that x=r[k].
- f[]: array of function values so that f(r[k])=f[k].
- r[], dr[], N: information describing change of variables.

1.3 File organization

Now that our software is becoming more complex, we will need a number of different files. *Organize* your programs into files as described below.

1.3.1 "nrutil.c"

Copy the *entire* file "nrutil.c" from http://www.nr.com/public-domain.html, and place it in your code directory. This file contains the *Numerical Recipes* utility functions "dvector()" and "free_dvector()", which rk4p480() requires.

1.3.2 "nrutil.h"

Copy the file "nrutil.h" from http://www.nr.com/public-domain.html, and place it in your code directory. This file contains the function prototypes for the utility routines so that the compiler can call them properly.

1.3.3 "p480.c"

This file will contain all of our own subroutines. At the moment, it contains rk4p480(). You should now add to this file your functions trapint() and simpint(). Because the programs in this file will call upon routines from the math library and certain *Numerical Recipes* utilities, you should place

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

at the top of the file.

1.3.4 "p480.h"

Programs which we write to call upon our routines will need to know the function prototypes to call them properly. To ensure this, you should create a file "p480.h" which contains just the function prototypes for all of the routines in "p480.c". *Each time we add a new routine to "p480.c" you should include it's prototype here!* At present, this file should read just

```
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);
```

1.3.5 "makefile"

In Unix, a file named "makefile" traditionally holds the instructions for combing all of the files to make the final program. A simple makefile will do for the present purposes. It should read

```
rk4debug: rk4debug.c p480.o nrutil.o p480.h nrutil.h
        cc -o rk4debug rk4debug.c p480.o nrutil.o -lm
```

This specifies that the program rkrdebug depends on the C-code in "rk4debug.c" as well as the routines in p480.o and nrutil.o and the specifications in p480.h and nrutil.h. Typing "make rk4debug" at the Unix prompt should suffice to compile your code completely. You may then type "rk4debug" to run it. Feel free to modify your makefile as you wish.

Note: For the makefile to function properly, the space before the "cc" should be a "tab" character.

1.4 Debugging

As we discussed in class, using fourth-order Runge-Kutta to solve $F'(x) = g(x)$ is equivalent to integrating $\int g(x)\,dx$ with Simpson's rule. Debug your ODE programs using the program "rk4debug.c" provided in Appendix A to compare the results of Runge-Kutta and Simpson's rule for this integration. Be sure to place your "derivs_int" at the top of the file in the indicated location.

2 Solving Poisson's equation

2.1 "poisson.c"

Cut and paste the main program provided in Appendix B into the file "poisson.c".

2.2 Kernel for Poisson's equation

Write a "derivs_Poisson()" routine at the location indicated near the top of "poisson.c". This routine should implement the first-order version of Poisson's equation as discussed in class,

$$\frac{d}{du}\begin{pmatrix} \Phi \\ \Phi' \end{pmatrix} = \begin{pmatrix} \Phi' \\ -4\pi n(r) r \end{pmatrix} \cdot \frac{dr}{du},$$

where $\phi(r) = \Phi(r)/r$.

2.3 Poisson's equation solver

Fill in code at the remaining locations indicated "..." in "poisson.c" to complete your solution of Poisson's equation.

Notes: Be sure to integrate backwards from $r = \infty$ (u=1) to $r = 0$ (u=0): you'll have to change the order of the loop and the values of *both* h and dk! Also, remember the boundary conditions:
$\Phi(\infty) = \Phi(r(u = 1)) = Q$ and $\Phi'(\infty) = \Phi'(r(u = 1)) = 0$.

2.4 Running and Debugging

Run your code "poisson.c" and verify that you recover the analytic solution with the expected scaling in the error.

Note: You may wish to add another entry to your makefile:

```
poisson: poisson.c p480.o nrutil.o p480.h nrutil.h
        cc -o poisson poisson.c p480.o nrutil.o -lm
```

Then, be sure to compile and run your code with the commands "make poisson" and "poisson".

3 Schrödinger's equation

Complete the code in Appendix C to take the parameter E as input from the keyboard and solve the equation

$$\frac{d}{du}\begin{pmatrix} \Psi \\ \Psi' \end{pmatrix} = \begin{pmatrix} \Psi' \\ -2(E + 1/r)\Psi \end{pmatrix} \cdot \frac{dr}{du},$$

by integrating inwards from infinity with boundary conditions ($\Psi(u = 1) = 0, \Psi'(u = 1) = 1e - 20$) using N=200 intervals.

3.1 Running and debugging

Your program should produce small values for inputs near E=-0.5 and large values for E=-0.4 and E=-0.6. We will explore this behavior in more depth in lab.

Note: You may wish again to add a new entry to your makefile.

A "rk4debug.c"

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

/* Place derivs_int() here ... */


#define Nmx 200000
main()
{
  /* Grid vectors */
  double *r,*dr,*f;

  /* Working variables */
  int i,N;
  double u;
  double *y,*dydx; /* NR vector for diff eq's */
  double x,h;
  int k,dk;

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

  /* Allocate NR vectors for maximum size used */
  r=dvector(0,Nmx);
  dr=dvector(0,Nmx);
  f=dvector(0,Nmx);
  y=dvector(1,1);
  dydx=dvector(1,1);

  /* Loop over different integration mesh sizes */
  for (N=2; N<=Nmx; N*=10) {

    /* Construct change of variables information and
       integrand function for computing V_el-nuc */
    for (i=0; i<=N; i++) {
      u=((double) i)/((double) N);
      r[i]=1./(1.-u)-1.;
      dr[i]=1./(1.-u)/(1.-u);
      f[i]=(-1/r[i])*(exp(-2*r[i])/pi)*4.*pi*r[i]*r[i];
    }
    /* Special conditions for handling end-points of integration */
    dr[0]=f[0]=0.;
    dr[N]=f[N]=0.;


    /* Runga-Kutta solution using rk4p480(). */
    /* Note step size (h,dk) is 2 because RK algorithm produces
       results only at the end points. */
```

```
    h=2./N;
    dk=2;
    y[1]=0.;
    for (k=0; k<=N-2; k+=dk) {
      x=k*h;
      derivs_int(x,y,dydx,k,f,r,dr,N);
      rk4p480(y,dydx,1,x,h,y,derivs_int,k,dk,f,r,dr,N);
    }

    printf("T,S,RK: %6d %18.14f,%18.14f,%18.14f\n",N,trapint(f,r,dr,N),simpint(f,r,dr,N),y[1]);
  }

  /* Deallocate NR vectors: Should always clean up space at the end. */
  free_dvector(r,0,Nmx);
  free_dvector(dr,0,Nmx);
  free_dvector(f,0,Nmx);
  free_dvector(y,1,1);
  free_dvector(dydx,1,1);
}
```

B "poisson.c"

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

/* Place derivs_Poisson() here ... */

#define Nmx 200000
main()
{
  /* Grid vectors */
  double *r,*dr,*rho;

  /* Working variables */
  int i,N;
  double u;
  double *y,*dydx; /* NR vector for diff eq's */
  double x,h;
  int k,dk;

  double rms;

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

  /* Allocate NR vectors for maximum size used */
  r=dvector(0,Nmx);
  dr=dvector(0,Nmx);
  rho=dvector(0,Nmx);
  y=dvector(1,2);
  dydx=dvector(1,2);

  /* Loop over different integration mesh sizes */
  for (N=2; N<=Nmx; N*=10) {

    /* Construct change of variables information and
       electron density "rho" for solving for phi */
    for (i=0; i<=N; i++) {
      u=((double) i)/((double) N);
      r[i]=1./(1.-u)-1.;
      dr[i]=1./(1.-u)/(1.-u);
      rho[i]=exp(-2*r[i])/pi;
    }

    /* Runga-Kutta solution using rk4p480(). */
    /* Set up initial step sizes (h,dk) and initial conditions ... */


    rms=0.;
    for (k=N; k>=2; k+=dk) {
```

```
      /* Perform rk4 step here ... */


      /* This compares your solution with the analytic solution.  This
         comparison must be made *after* you make the step with rk4p480.

         Replace the ...'s with the appropriate analytic expression
         from Problem Set 1 in terms of r[k]!  Remember that, here, we
         solve for Phi(r)=r*phi(r)
      */
      rms=rms+pow(y[1]-( ... ), 2.);
    }
    rms=sqrt(rms/(N/2));
    printf("N, phi[0], rms error: %6d %20.16f %20.16f\n",N,y[1],rms);
  }

  /* Deallocate NR vectors: Should always clean up space at the end. */
  free_dvector(r,0,Nmx);
  free_dvector(dr,0,Nmx);
  free_dvector(rho,0,Nmx);
  free_dvector(y,1,1);
  free_dvector(dydx,1,1);
}
```

C "quicksch.c"

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

/* Place derivs_Schrodinger() here ... */

#define Nmx 200
main()
{
  /* Grid vectors */
  double *r,*dr,*T;

  /* Working variables */
  int i,N;
  double u;
  double *y,*dydx; /* NR vector for diff eq's */
  double x,h;
  int k,dk;

  double E;

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

  /* Allocate NR vectors for maximum size used */
  r=dvector(0,Nmx);
  dr=dvector(0,Nmx);
  T=dvector(0,Nmx);
  y=dvector(1,2);
  dydx=dvector(1,2);

  /* Loop over different integration mesh sizes */
  N=Nmx;

  while (1) {
    printf("E: "); scanf("%lf",&E);

    /* Construct change of variables information and kinitic energy
       for solving for phi */
    for (i=0; i<=N; i++) {
      u=((double) i)/((double) N);
      r[i]=1./(1.-u)-1.;
      dr[i]=1./(1.-u)/(1.-u);
      T[i]=(E+1./r[i]);
    }

    /* Runga-Kutta solution using rk4p480(). */
    /* Set up step sizes (h,dk) and initial conditions here ... */
```

```
    for (k=N; k>=2; k+=dk) {
      x=k*h;
      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);
    }
    printf("N=%d; E, Psi[0]: %20.16f %e\n",N,E,y[1]);
  }


  /* Deallocate NR vectors: Should always clean up space at the end. */
  free_dvector(r,0,Nmx);
  free_dvector(dr,0,Nmx);
  free_dvector(T,0,Nmx);
  free_dvector(y,1,1);
  free_dvector(dydx,1,1);
}
```