PHY407
Lab #7: Ordinary Differential Equations, Pt. 2

Q1, Q2: Nicholas Pavanel, Q3: Hayley Agler

October 2020

Q1

a)

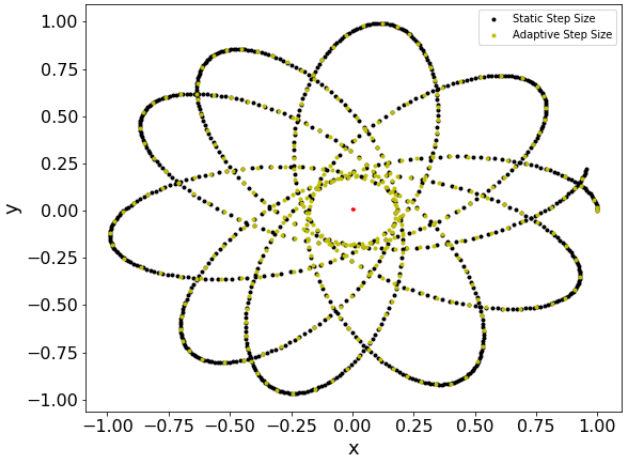

Figure 1: This plot shows a comparison between two methods of integrating the orbit of a spherical ball bearing around a steel rod (represented by the red dot) in zero gravity. The rod is assumed to have mass large enough such that the bearing's influence on it is negligible. The orbit of the bearing is around the rod's mid-point, in a plane that is perpendicular with the rod. The orbit of the ball bearing was computed with: 1) a fixed step size 4^{th}-order Runge-Kutta (RK4) method (in black) and 2) an adaptive step size RK4 method (in yellow).

See Figure 1. The orbit shown in Figure 1 was computed with two different 4^{th}-order Runge-Kutta (RK4) methods for solving ODEs: 1) a fixed step size method and 2) an adaptive step size method. These methods were used to solve the 2^{nd} order ODEs that are the bearing's equations of motion, Equations 3 and 4.

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^2\sqrt{r^2+\frac{L^2}{4}}} \tag{1}$$

$$\frac{d^2y}{dt^2} = -GM\frac{y}{r^2\sqrt{r^2+\frac{L^2}{4}}} \tag{2}$$

The adaptive step size method can most noticeably be seen during times at which the ball bearing is moving slowly, at the bearing's apoapsides. At the bearing's nine completed apoapsides the adaptive step size method

takes larger step sizes because the bearing's position is changing slowly. This is seen clearly in Figure 1 at the bearing's apoapsides: there are few yellow dots compared to many black dots.

b)

Subsection implies that the adaptive RK4 method should compute the bearing's orbit quicker than the static RK4 method, as it uses less points in its computation of the orbit. This is shown to be true by the printed outputs of the time taken to write each method shown below:

```
Time taken for static RK4 10,000 steps: 0.6977870464324951  seconds
```

```
Time taken for adaptive RK4 $\delta = 10^{-6}$ steps: 0.11997747421264648  seconds
```

c)

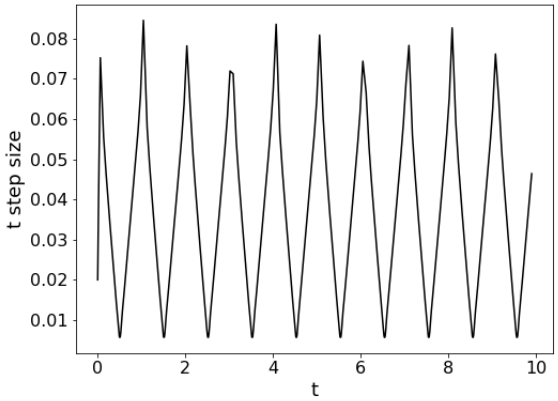

Figure 2: This plot shows the orbit of a piece of space junk; a spherical ball bearing (represented by the black line) orbits around a host steel rod (represented by the red dot) in zero gravity. The rod is assumed to have mass large enough such that the bearing's influence on it is negligible. Note that the orbit of the bearing is around the rod's mid-point, in a plane that is perpendicular with the rod.

See Figure 2. From Figure 2 we can identify ten peaks and ten troughs. These peaks and troughs represent times when the adaptive RK4 method is taking respectively its largest and smallest steps. To make sense of how the step size changes, we can identify ten (nine complete and one incomplete) apoapsides from Figure 1. Peaks (the largest step sizes) occur in Figure 2 when the bearing is moving the slowest, around its apoapsides. Troughs (the smallest step sizes) occur in Figure 2 when the bearing is moving the fastest, around its periapsides. The adaptive RK4 method used in this section to compute the orbit can be found in Section .

Q2

a)

See Figure 3. The orbit shown in Figure 3 is the orbit of the Earth around the Sun. It was computed with a Bulirsch-Stoer method methods for solving ODEs with a step size of one week. The Bulirsch-Stoer method was used to solve the 2^{nd} order ODEs that are the Earth's equations of motion, Equations 3 and 4.

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^3} \tag{3}$$

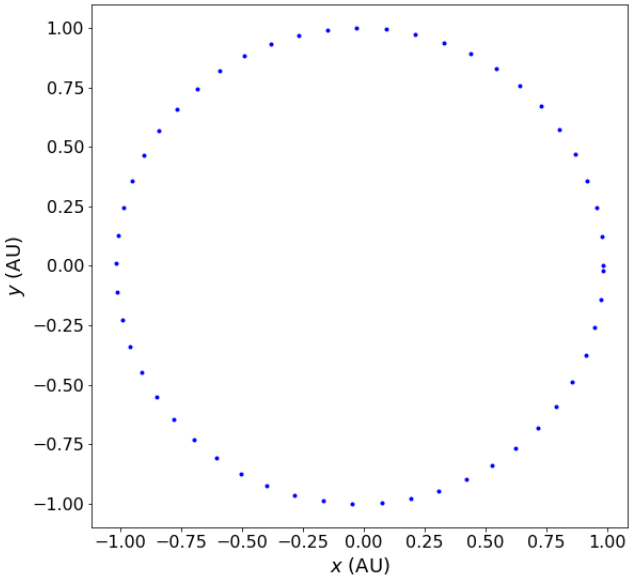

Figure 3: This plot shows the orbit of a piece of space junk; a spherical ball bearing (represented by the black line) orbits around a host steel rod (represented by the red dot) in zero gravity. The rod is assumed to have mass large enough such that the bearing's influence on it is negligible. Note that the orbit of the bearing is around the rod's mid-point, in a plane that is perpendicular with the rod.

$$\frac{d^2y}{dt^2} = -GM\frac{y}{r^3} \tag{4}$$

As Figure 3 shows, Earth's orbit is quite circular.

b)

See Figure 4. The orbit shown in Figure 4 is the orbit of the Pluto around the Sun. It was computed with a Bulirsch-Stoer method for solving ODEs with a step size of 248 weeks. This step size was chosen to match the fractional step size of Section as Pluto's orbital period is 248 years. The Bulirsch-Stoer method was used to solve the 2^{nd} order ODEs that are the Pluto's equations of motion, Equations 3 and 4. As Figure 4 shows, Pluto's orbit is much more elliptical than Earth's orbit. Pluto's elliptical orbit is one of the pieces of evidence that implies that it should not be considered a planet of our solar system as all other planets have largely circular orbits. The Bulirsch-Stoer method used in this section to compute the orbit can be found in Section .

Q3

a)

Nothing to submit.

In this question, we use shooting and RK4 to find the bound states of hydrogen for several cases. The second order ODE in the lab handout:

$$\frac{\mathrm{d}}{\mathrm{d}r}\left(r^2\frac{\mathrm{d}R}{\mathrm{d}r}\right) - \frac{2mr^2}{\hbar^2}[V(r) - E]R = \ell(\ell+1)R \tag{5}$$

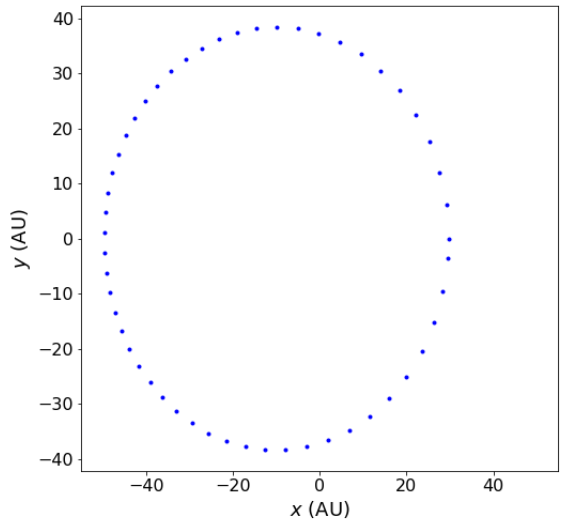

Figure 4: This plot shows the orbit of a piece of space junk; a spherical ball bearing (represented by the black line) orbits around a host steel rod (represented by the red dot) in zero gravity. The rod is assumed to have mass large enough such that the bearing's influence on it is negligible. Note that the orbit of the bearing is around the rod's mid-point, in a plane that is perpendicular with the rod.

ℓ	n	E (eV)	E actual (eV)
0	1	-13.60346	-13.6
0	2	-3.44099	-3.4
1	2	-3.43993	-3.4

Table 1: A table displaying the energies for different values of n and ℓ.

where $V(r) = \frac{e^2}{4\pi\epsilon_0 r}$ was written as a pair of coupled first order ODEs:

$$R(r) = \frac{S}{r^2} \tag{6}$$

$$S(r) = \frac{2mr^2}{\hbar^2}(V(r) - E)R + \ell(\ell+1)R \tag{7}$$

By referencing the code squarewell.py and with reference to Section 8.6.3, Example 8.9, this pair of equations is implemented to find various energies of the bound states of hydrogen in the question below.

b)

The first few energies of hydrogen were found as outlined above. Parameters h and the target energy convergence were adjusted to return the result closest to the known energies. The value of h was adjusted to $h = 0.003 * a$, and the target energy convergence was made $target = e/10$. The maximum value of r was kept at $r_\infty = 20 * a$. The energies of the first few states of hydrogen are found in 1

c)

Nothing to submit. The eigenfunctions R(r) were normalized using the trapezoidal rule to compute the integral

$$\int |R(r)|^2 dr$$

over the range of r. The eigenfunctions were scaled to bohr radii. These eigenfunctions were plotted in part d) below.

d)

The explicit solutions for R(r) were found, from the link in the lab handout, to be:

$$\frac{2}{a_0^{3/2}} e^{-r/a_0} \tag{8}$$

$$\frac{1}{2\sqrt{2}a_0^{3/2}} \left[2 - \frac{r}{a_0}\right] e^{-r/2a_0} \tag{9}$$

$$\frac{1}{2\sqrt{6}a_0^{3/2}} \frac{r}{a_0} e^{-r/2a_0} \tag{10}$$

The solutions were normalized as described in c) and converted to unit of bohr radii to match the numerically calculated solutions. Both the numerical and explicit solutions were plotted in figures 5, 6, and 7. For the l=0, n=1 case and the l=0, n=2 case, the explicit solution follows the same shape almost perfectly except for the beginning, where the numerical solution has a maximum and the explicit solution matches only the decreasing half of this maximum peak. For the l=1, n=2 case, the explicit solution follows the same shape as the numerical solution. In all three cases, there is a difference between the numerical and explicit solutions at large Bohr radii as the numerical solution begins to dip below zero. Furthermore, for the l=0, n=1 case, the values plotted were limited to the first third of r values because at high r values the wave function begins to attain very large magnitudes (near -400). This is likely because the energy value is not exact and the wave function is very sensitive to the energy.

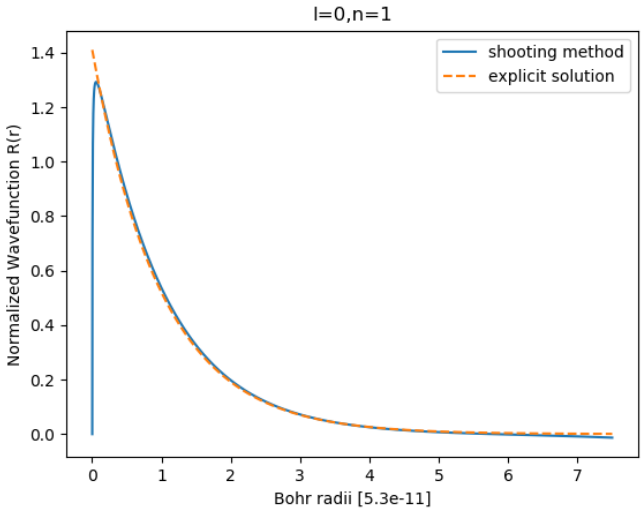

Figure 5

Code

Q1

```
#global variables
G = 1
M = 10
```

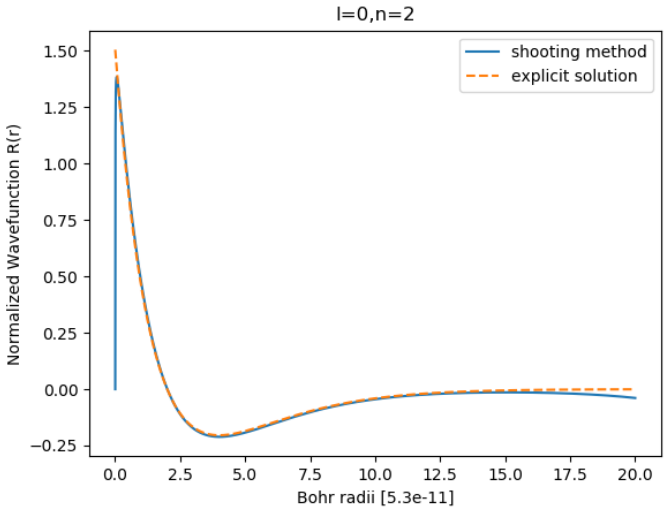

Figure 6: Caption

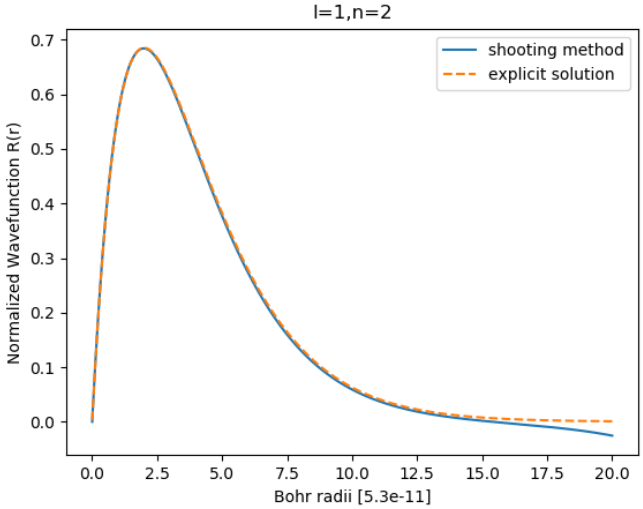

Figure 7: Caption

```
L = 2
#define time start and time stop, number of points in time domain, steps, and time domain
t1= 0
t2=10
N = 10000
h = (t2 - t1) / N
t = np.arange(t1,t2,h)
#define lists for output info
xs = []
ys = []
#define inital conditions [x,vx,y,vy]
r = [1., 0, 0, 1.]
```

```
#define RHS eqn
def RHS(G,M,L,x_y,r):
    return - G * M * (x_y)/(r**2 * np.sqrt(r**2 + (L**2/4)))

#define equation to use in RK4
def f(ics,t):
    x = ics[0]
    vx = ics[1]
    y = ics[2]
    vy = ics[3]
    r = np.sqrt(x**2 + y**2)
    RHS_x = RHS(G,M,L,x,r)
    RHS_y = RHS(G,M,L,y,r)
    return np.array([vx, RHS_x, vy, RHS_y],float)

# define a function to perform a step of RK4
# adaptive RK4 is the same method with additional constraints
def RK4_step(r, t, h):
    k1 = h * f(r, t)
    k2 = h * f(r + 0.5 * k1, t + 0.5 * h)
    k3 = h * f(r + 0.5 * k2, t + 0.5 * h)
    k4 = h * f(r + k3, t + h)
    return (k1 + 2 * k2 + 2 * k3 + k4) / 6

# define a function to perform RK4 with adaptive step size
def RK4_adaptive(r, t, h):

    # perform 2 RK4 steps of size h
    hstep_1 = RK4_step(r, t, h)
    hstep_2 = RK4_step(r + hstep_1, t + h, h)
    # add the steps of size h for one step of size 2h
    double_2hstep = hstep_1 + hstep_2

    # perform 1 RK step with size 2h
    single_2hstep = RK4_step(r, t, 2 * h)

    # compute error estimates
    e_x = (1/30) * (double_2hstep[0] - single_2hstep[0])
    e_y = (1/30) * (double_2hstep[2] - single_2hstep[2])

    # calculate rho
    rho = h * delta / np.sqrt(e_x**2 + e_y**2)

    # choose the next step size h
    # if accurate enough, move on
    if  rho >= 1:
        t = t + 2 * h

        # ensure h does not get too large
        if rho**(1/4) > 2:
            h *= 2
        else:
            h *= rho**(1/4)
```

```
        return double_2hstep, h, t
    # if not accurate enough, do again with smaller step
    else:
        return RK4_adaptive(r, t, rho**(1/4) * h)

# set accuracy, arrays for output, initial values
delta = 10**(-6)
tpoints = []
xpoints2 = []
ypoints2 = []
t = t1
r = [1., 0, 0, 1.]

# perform adaptive RK4
t0 = time()
while(t < t2):
    tpoints.append(t)
    xpoints2.append(r[0])
    ypoints2.append(r[2])
    delta_r, h, t = RK4_adaptive(r, t, h)
    r += delta_r
t_total_adaptive = time() - t0

print('Time taken for adaptive RK4 $\delta = 10^{-6}$ steps:',t_total_adaptive,' seconds')

# plot adaptive RK4 and regular RK4
rod = plt.Circle((0.512,0.5),radius=0.0025,color='red')
axs = plt.figure(figsize=[9,7])
axs.add_artist(rod)
plt.plot(xs,ys,'k.',label='Static Step Size')
plt.plot(xpoints2,ypoints2,'y.',label='Adaptive Step Size')
plt.legend()
plt.xticks(size=16)
plt.yticks(size=16)
plt.xlabel('x',size=18)
plt.ylabel('y',size=18)
plt.savefig('Q1a')
plt.show()
```

Q2

```
# i modify the code from the text in example 8.7

from math import sin,pi
from numpy import empty,array,arange,sqrt
import matplotlib.pyplot as plt

# define constants in SI units - kg, s, m
G = 6.6738 * 10**(-11)
M = 1.9891 * 10**(30)
Peri = 1.4710 * 10**(11)
Vperi = 3.0287 * 10**(4)
delta = 1000     # Required position accuracy per unit time - 1km in m
```

```
#define time domain
H = 604800      # Size of "big steps" - 1 week in s
N = 5           # do 5 revolutions around the sun
yr = 365.25*24*3600.   # seconds in a yr
T = N * yr         # time domain in s
Nsteps = int(T/H)

# define a function to compute equations of motion
def x_eom(x,y):
    return - G * M * x / (sqrt(x**2 + y**2))**3
def y_eom(x,y):
    return - G * M * y / (sqrt(x**2 + y**2))**3

def f(ics):
    x = ics[0]
    vx = ics[1]
    y = ics[2]
    vy = ics[3]
    r = sqrt(x**2 + y**2)
    return array([vx,x_eom(x,y),vy,y_eom(x,y)],float)

tpoints = arange(0,T,H)
xpoints = []
ypoints = []
r = array([Peri,0,0,Vperi],float)  #ics in [x,vx,y,vy]

# Do the "big steps" of size H
for t in tpoints:

    xpoints.append(r[0])
    ypoints.append(r[2])

    # Do one modified midpoint step to get things started
    n = 1
    r1 = r + 0.5*H*f(r)
    r2 = r + H*f(r1)

    # The array R1 stores the first row of the
    # extrapolation table, which contains only the single
    # modified midpoint estimate of the solution at the
    # end of the interval
    R1 = empty([1,4],float)
    R1[0] = 0.5*(r1 + r2 + 0.5*H*f(r2))

    # Now increase n until the required accuracy is reached
    error = 2*H*delta
    while error>H*delta:

        n += 1
        h = H/n

        # Modified midpoint method
        r1 = r + 0.5*h*f(r)
        r2 = r + h*f(r1)
```

```
        for i in range(n-1):
            r1 += h*f(r2)
            r2 += h*f(r1)

        # Calculate extrapolation estimates.  Arrays R1 and R2
        # hold the two most recent lines of the table
        R2 = R1
        R1 = empty([n,4],float)
        R1[0] = 0.5*(r1 + r2 + 0.5*h*f(r2))
        for m in range(1,n):
            epsilon = (R1[m-1]-R2[m-1])/((n/(n-1))**(2*m)-1)
            R1[m] = R1[m-1] + epsilon
        error = abs(epsilon[0])

    # Set r equal to the most accurate estimate we have,
    # before moving on to the next big step
    r = R1[n-1]

# change xpoints and ypoints to AU
x_au = array(xpoints[:53])*6.68459e-12
y_au = array(ypoints[:53])*6.68459e-12
# Plot the results
plt.figure(figsize=[9,9])
plt.plot(x_au,y_au,'b.')
plt.xticks(size=16)
plt.yticks(size=16)
plt.xlabel('$x$ (AU)',size=18)
plt.ylabel('$y$ (AU)',size=18)
plt.savefig('Q2a')
plt.show()

# for Pluto change the ICS
#define ics for pluto
Peri = 4.4368 * 10**(12)
Vperi = 6.1218 * 10**(3)
delta = 1000     # Required position accuracy per unit time - 1km in m

#define time domain for pluto
H = 604800 * 248      # Size of "big steps" - 1 week in s - Pluto's year is 248 times larger than Earth
N = 2           # do 5 revolutions around the sun
yr = 365.25*24*3600. * 248   # seconds in a yr - again with Pluto's year
T = N * yr         # time domain in s
Nsteps = int(T/H)
```

Q3

```
import numpy as np
import matplotlib.pyplot as plt


# Constants
m = 9.1094e-31     # Mass of electron
hbar = 1.0546e-34  # Planck's constant over 2*pi
```

```
e = 1.6022e-19     # Electron charge
N = 1000
a=5.3e-11 # Bohr radius
h = 0.003*a #r_f-r_0/N
r_0=h #R=0 here
r_f=20*a  #R=0 here
ep_not=8.8e-12


# Potential function
def V(r):
    return -(e**2)/(4*np.pi*ep_not*r)

#separated solutions to the ODE
def f(p,r,E,l):
    R = p[0] #psi
    S = p[1] #phi
    fR = S/(r**2)
    fS = ((2*(r**2)*m)/(hbar**2))*(V(r)-E)*R+l*(l+1)*R
    return np.array([fR,fS])

# Calculate the wavefunction for a particular energy
def solve(E,l):
    R = 0.0
    S = 1.0
    p = np.array([R,S],float)
    wavefn=[]
    for r in np.arange(r_0,r_f, h):
        wavefn.append(p[0])
        #print(p, r, E)
        k1 = h*f(p,r,E,l)
        #print(k1, p+0.5*k1, r+0.5*h)
        k2 = h*f(p+0.5*k1,r+0.5*h,E,l)
        k3 = h*f(p+0.5*k2,r+0.5*h,E,l)
        k4 = h*f(p+k3,r+h,E,l)
        p += (k1+2*k2+2*k3+k4)/6
        #print(p+0.5*k1,k2,k3,k4)

    return p[0], wavefn #returns wavefunctions R


# Main program to find the energy using the secant method
E11 = -15*e/1**2 #l=0,n=1
E21 = -13*e/1**2
R21 = solve(E11,0)[0]

target = e/10
while abs(E11-E21)>target:
    R11,R21 = R21, solve(E21,0)[0]
    E11,E21 = E21,E21-R21*(E21-E11)/(R21-R11)
    wavefn1=solve(E21,0)[1]
print("For l=0,n=1, E =",E21/e,"eV")

#l=0,n=2
```

```
E12 = -15*e/2**2 #l=0,n=2
E22 = -13*e/2**2
R22 = solve(E12,0)[0]

target = e/10
while abs(E12-E22)>target:
    R12,R22 = R22, solve(E22,0)[0]
    E12,E22 = E22,E22-R22*(E22-E12)/(R22-R12)
    wavefn2=solve(E22,0)[1]
print("For l=0,n=2, E =",E22/e,"eV")


#l=1,n=2
E13 = -15*e/2**2 #l=1,n=2
E23 = -13*e/2**2
R23 = solve(E13,1)[0]

target = e/10
while abs(E13-E23)>target:
    R13,R23 = R23, solve(E23,1)[0]
    E13,E23 = E23,E23-R23*(E23-E13)/(R23-R13)
    wavefn3=solve(E23,1)[1]
print("For l=1,n=2, E =",E23/e,"eV")




#normalize R(r) using trapezoidal rule
#modified trapezoidal for an array
def trapz_int(N, wavefn):
    f=np.array(wavefn)**2
    # 2.: the beginning and end
    result = 0.5*(f[0] + f[N-1])

    # 3. Then, loop over interior points
    for k in range(1, N):
        result += f[0+k]

    return h/a*result

#%%
#define explicit solutions
def explicit01(r): #l=0,n=1
    return 2*np.exp(-r)

#define the wavefunction squared for the normalization
def explicit01_2(r): #l=0,n=1
    return (2*np.exp(-r))**2

def explicit02(r): #l=0,n=2
    return 1/(2*np.sqrt(2))*(2-r)*np.exp(-r/(2))

def explicit02_2(r): #l=0,n=2
    return (1/(2*np.sqrt(2))*(2-r)*np.exp(-r/(2)))**2
```

```
def explicit12(r):
    return 1/(2*np.sqrt(6))*r*np.exp(-r/2)

def explicit12_2(r):
    return (1/(2*np.sqrt(6))*r*np.exp(-r/2))**2



#define different trapezoidal integration for a function not an array
def trapz(z_0,z_f, N, f):

    # 1.: from a, b and N, compute h
    h = 0.003

    # 2.: the beginning and end
    result = 0.5*(f(z_0)+ f(z_f))

    # 3. Then, loop over interior points
    for k in range(1, N):
        result += f(z_0 + k*h)

    return h*result  #

rvals=np.arange(r_0,r_f, h)/a

#normalize the explicit solutions
explicit01_norm=explicit01(rvals)/np.sqrt(trapz(r_0,r_f,1000,explicit01_2))
explicit02_norm=explicit02(rvals)/np.sqrt(trapz(r_0,r_f,1000,explicit02_2))
explicit12_norm=explicit12(rvals)/np.sqrt(trapz(r_0,r_f,1000,explicit12_2))

#normalize the wave functions and convert to units of bohr radii
wavefn_norm1=np.array(wavefn1)*a**1.5/np.sqrt(trapz_int(N, np.array(wavefn1)*a**1.5)) #l=0,n=1
wavefn_norm2=np.array(wavefn2)*a**1.5/ np.sqrt(trapz_int(N, np.array(wavefn2)*a**1.5)) #l=0,n=2
wavefn_norm3=np.array(wavefn3)*a**1.5/ np.sqrt(trapz_int(N, np.array(wavefn3)*a**1.5)) #l=1,n=2
rvals=np.arange(r_0,r_f, h)/a

#plot the wavefunctions
plt.figure(1)
plt.title("l=0,n=1")
plt.plot(rvals[0:2500], wavefn_norm1[0:2500], label="shooting method")
plt.plot(rvals[0:2500], explicit01_norm[0:2500], label="explicit solution", linestyle="dashed")
plt.xlabel("Bohr radii [5.3e-11]")
plt.ylabel("Normalized Wavefunction R(r)")
plt.legend()


plt.figure(2)
plt.title("l=0,n=2")
plt.plot(rvals, wavefn_norm2, label="shooting method")
plt.plot(rvals, explicit02_norm, label="explicit solution", linestyle="dashed")
plt.xlabel("Bohr radii [5.3e-11]")
plt.ylabel("Normalized Wavefunction R(r)")
plt.legend()
```

```
plt.figure(3)
plt.title("l=1,n=2")
plt.plot(rvals, wavefn_norm3, label="shooting method")
plt.plot(rvals, explicit12_norm, label="explicit solution", linestyle="dashed")
plt.xlabel("Bohr radii [5.3e-11]")
plt.ylabel("Normalized Wavefunction R(r)")
plt.legend()
```