hw3_ex12

February 20, 2020

[1]:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb  6 13:33:08 2020

@author: aswart
"""

# python 3 version 2/15
import numpy as np
import matplotlib.pyplot as plt

def rk4(x,t,tau,derivsRK):
#%  Runge-Kutta integrator (4th order)
#% Input arguments -
#%   x = current value of dependent variable
#%   t = independent variable (usually time)
#%   tau = step size (usually timestep)
#%   derivsRK = right hand side of the ODE; derivsRK is the
#%             name of the function which returns dx/dt
#%             Calling format derivsRK(x,t).
#% Output arguments -
#%   xout = new value of x after a step of size tau
    half_tau = 0.5*tau
    F1 = derivsRK(x,t)
    t_half = t + half_tau
    xtemp = x + half_tau*F1
    F2 = derivsRK(xtemp,t_half)
    xtemp = x + half_tau*F2
    F3 = derivsRK(xtemp,t_half)
    t_full = t + tau
    xtemp = x + tau*F3
    F4 = derivsRK(xtemp,t_full)
    xout = x + tau/6.*(F1 + F4 + 2.*(F2+F3))
    return xout
def rka(x,t,tau,err,derivsRK):
```

```python
#% Adaptive Runge-Kutta routine
#% Inputs
#%   x          Current value of the dependent variable
#%   t          Independent variable (usually time)
#%   tau        Step size (usually time step)
#%   err        Desired fractional local truncation error
#%   derivsRK   Right hand side of the ODE; derivsRK is the
#%              name of the function which returns dx/dt
#%              Calling format derivsRK(x,t).
#% Outputs
#%   xSmall     New value of the dependent variable
#%   t          New value of the independent variable
#%   tau        Suggested step size for next call to rka

#%* Set initial variables
    tSave = t;  xSave = x    # Save initial values
    safe1 = .9;  safe2 = 4.  # Safety factors
    eps = np.spacing(1) # smallest value

#%* Loop over maximum number of attempts to satisfy error bound
    maxTry = 100

    for iTry in range(1,maxTry):

#%* Take the two small time steps
        half_tau = 0.5 * tau
        xTemp = rk4(xSave,tSave,half_tau,derivsRK)
        t = tSave + half_tau
        xSmall = rk4(xTemp,t,half_tau,derivsRK)

  #%* Take the single big time step
        t = tSave + tau
        xBig = rk4(xSave,tSave,tau,derivsRK)

  #%* Compute the estimated truncation error
        scale = err * (np.abs(xSmall) + np.abs(xBig))/2.
        xDiff = xSmall - xBig
        errorRatio = np.max( [np.abs(xDiff)/(scale + eps)] )

        #print safe1,tau,errorRatio

  #%* Estimate news tau value (including safety factors)
        tau_old = tau

        tau = safe1*tau_old*errorRatio**(-0.20)
        tau = np.max([tau,tau_old/safe2])
        tau = np.min([tau,safe2*tau_old])
```

```
  #%* If error is acceptable, return computed values
        if errorRatio < 1 :
          # xSmall = xSmall #% +  (xDiff)/15
          #   xSmall = (16.*xSmall - xBig)/15. # correction
            return xSmall, t, tau

#%* Issue error message if error bound never satisfied
    print ('ERROR: Adaptive Runge-Kutta routine failed')
    return
def gravrk(s,t):
#%  Returns right-hand side of Kepler ODE; used by Runge-Kutta routines
#%  Inputs
#%    s      State vector [r(1) r(2) v(1) v(2)]
#%    t      Time (not used)
#%  Output
#%    deriv  Derivatives [dr(1)/dt dr(2)/dt dv(1)/dt dv(2)/dt]

    GM = 4*np.pi**2

#%* Compute acceleration
    r = np.array([s[0], s[1]])  # Unravel the vector s into position and␣
 ↪velocity
    v = np.array([s[2] ,s[3]])
    accel = -GM*r/np.linalg.norm(r)**3  # Gravitational acceleration

#%* Return derivatives [dr(1)/dt dr(2)/dt dv(1)/dt dv(2)/dt]
    derivs = np.array([v[0], v[1], accel[0], accel[1]])
    return derivs

def gravrk_ex12(s,t):
#%  Returns right-hand side of Kepler ODE; used by Runge-Kutta routines
#%  Inputs
#%    s      State vector [r(1) r(2) v(1) v(2)]
#%    t      Time (not used)
#%  Output
#%    deriv  Derivatives [dr(1)/dt dr(2)/dt dv(1)/dt dv(2)/dt]

    GM = 4*np.pi**2
    # Hardcoded in vars since
    # Im not sure on an elegant way to add them to function
    r0 = np.array([1.0,0])
    init_Grav_accel = .01*-GM*r0/np.linalg.norm(r0)**3

#%* Compute acceleration
    r = np.array([s[0], s[1]])  # Unravel the vector s into position and␣
 ↪velocity
```

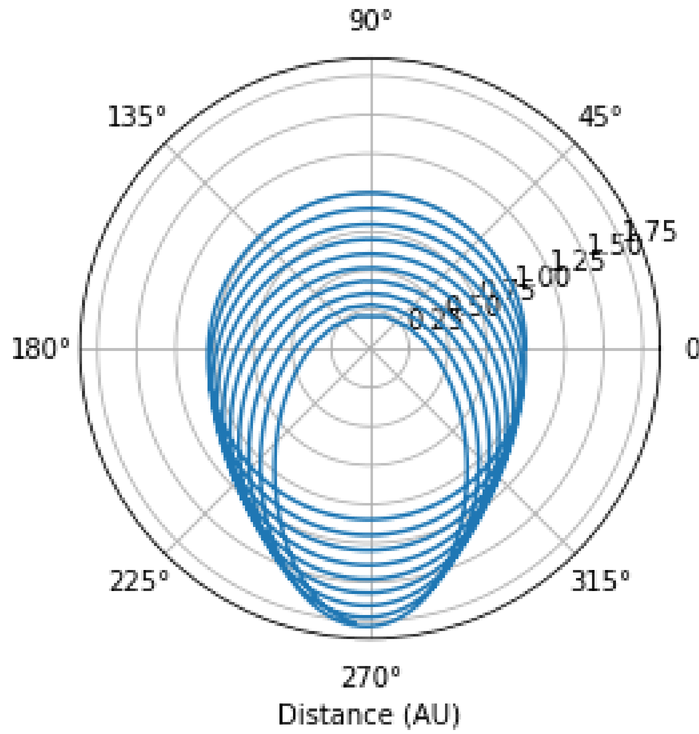
```
    v = np.array([s[2] ,s[3]])
    # Gravitational acceleration
    accel = -GM*r/np.linalg.norm(r)**3 + init_Grav_accel

#%* Return derivatives [dr(1)/dt dr(2)/dt dv(1)/dt dv(2)/dt]
    derivs = np.array([v[0], v[1], accel[0], accel[1]])
    return derivs


# orbit - Program to compute the orbit of a comet.
#clear all;  help orbit;  % Clear memory and print header

def orbit(input_dict = {},calc_info = False, plot_momentum = False,
          inter_input = False, plot_traj = True, plot_energy = False):
    if inter_input:
        # Set initial position and velocity of the comet.
        r0 = float(input("Enter initial radial distance (AU): "))
        # v0 = float(input("Enter initial tangential velocity (AU/yr): "))
        vstr = str(input("Enter initial tangential velocity (AU/yr) as a number␣
 ↪or multiple of Pi (e.g.,2*pi): "))

        # modify input to allow 'pi'
        vinp = vstr.split('*')
        if (vinp[-1].lower() == 'pi'):
          v0 = float(vinp[0])*np.pi
        else:
          v0 = float(vinp[0])

        nStep = int(input("Enter number of steps: "))
        tau =  float(input("Enter time step (yr): "))
        NumericalMethod=0
        while(NumericalMethod not in np.array([1,2,3,4,5,6,7])):
            NumericalMethod = int(input("Choose a number for a numerical method:
 ↪\n\
         1-Euler, 2-Euler-Cromer, 3-Runge-Kutta 4-Adaptive R-K: "))
    elif input_dict:
        r0 = input_dict['r0']
        v0 = input_dict['v0']
        nStep = input_dict['nStep']
        tau = input_dict['tau']
        NumericalMethod = input_dict['NumericalMethod']
    else:
        r0 = 1
        v0 = 2*np.pi
        nStep = 1000
        tau = .01
        NumericalMethod = 3
```

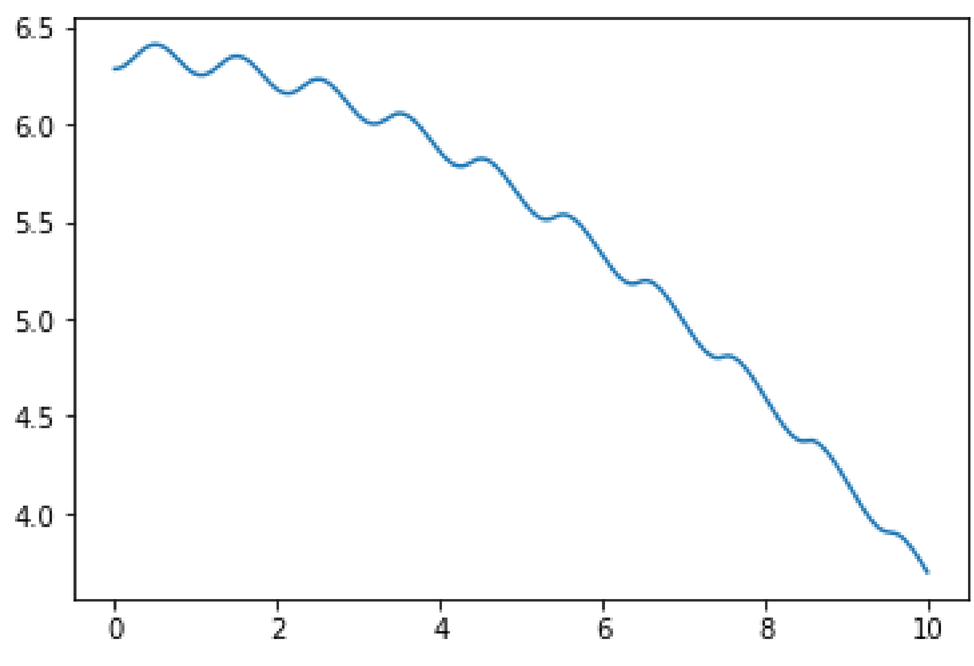
```
r = np.array([r0, 0.])
v = np.array([0., v0])
state = np.array([ r[0], r[1], v[0], v[1] ])   # Used by R-K routines

#Set physical parameters (mass, G*M)
GM = 4*np.pi**2      # Grav. const. * Mass of Sun (au^3/yr^2)
mass = 1.        # Mass of comet
adaptErr = 1.e-4 # Error parameter used by adaptive Runge-Kutta
time = 0.0

#%* Loop over desired number of steps using specified
#%  numerical method.
for istep in range(0,nStep):

  #%* Record position and energy for plotting.
  # Initially set the arrays for the first step
  if istep == 0:
      rplot = np.linalg.norm(r)
      thplot = np.arctan2(r[1],r[0])
      tplot = time
      kinetic = .5*mass*np.linalg.norm(v)**2
      potential= - GM*mass/np.linalg.norm(r)
      momentum = [np.linalg.norm(np.cross(r,mass*v))]
  else:
      rplot = np.append(rplot,np.linalg.norm(r))           #Record position for polar plot
      thplot = np.append(thplot,np.arctan2(r[1],r[0]))
      tplot = np.append(tplot,time)
      kinetic = np.append(kinetic,0.5*mass*np.linalg.norm(v)**2)   # Record energies
      potential= np.append(potential,- GM*mass/np.linalg.norm(r))
      momentum.append(np.linalg.norm(np.cross(r, mass*v)))

  #%* Calculate new position and velocity using desired method.
  if NumericalMethod == 1 :
    accel = -GM*r/np.linalg.norm(r)**3
    r = r + tau*v             # Euler step
    v = v + tau*accel
    time = time + tau
  elif NumericalMethod == 2:
    accel = -GM*r/np.linalg.norm(r)**3
    v = v + tau*accel
    r = r + tau*v             # Euler-Cromer step
    time = time + tau
  elif NumericalMethod == 3:
    state = rk4(state,time,tau,gravrk_ex12)
```

```
        r = np.array([state[0], state[1]])   # 4th order Runge-Kutta
        v = np.array([state[2], state[3]])
        time = time + tau
      else:
        [state, time, tau] = rka(state,time,tau,adaptErr,gravrk)
        r = np.array([state[0], state[1]])   # Adaptive Runge-Kutta
        v = np.array([state[2], state[3]])
      # If sometime after first step and theta goes from neg to pos, then we␣
↪know we've completed an orbit
#     if istep != 0 and (thplot[-2:]*[-1,1] > 0).all():
#         break

    #print(thplot[-5:])

    if plot_traj:
        #%* Graph the trajectory of the comet.
        plt.figure(1); plt.clf()  #Clear figure 1 window and bring forward
        plt.polar(thplot,rplot,'-')  # Use polar plot for graphing orbit
        plt.xlabel('Distance (AU)')
        plt.grid(True)

    if plot_energy:
        #%* Graph the energy of the comet versus time.
        plt.figure(2); plt.clf()   # Clear figure 2 window and bring forward
        totalE = kinetic + potential   # Total energy
        plt.plot(tplot,kinetic,'-.',tplot,potential,'--',tplot,totalE,'-')
        #plt.legend('Kinetic','Potential','Total')
        plt.xlabel('Time (yr)'); plt.ylabel('Energy (M AU^2/yr^2)')
        plt.grid(True)
        plt.show()

    if plot_momentum:
        # Plots angular momentum as a function of time
        plt.figure(3)
        plt.plot(tplot,momentum)

    return rplot, thplot
```

[2]:

```
    # non-elliptical
    input_dict = {
        'r0': 1,
        'v0': 2*np.pi,
        'nStep': 1000,
        'tau': .01,
        'NumericalMethod': 3
        }
    rplot, thplot  = orbit(input_dict,plot_momentum = True)
```


90°
135°
45°
180°
0°
225°
315°
270°
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Distance (AU)


6.5
6.0
5.5
5.0
4.5
4.0
0
2
4
6
8
10

[ ]:

```
[ ]:
```