hw3b_ex25

March 6, 2020

[1]:

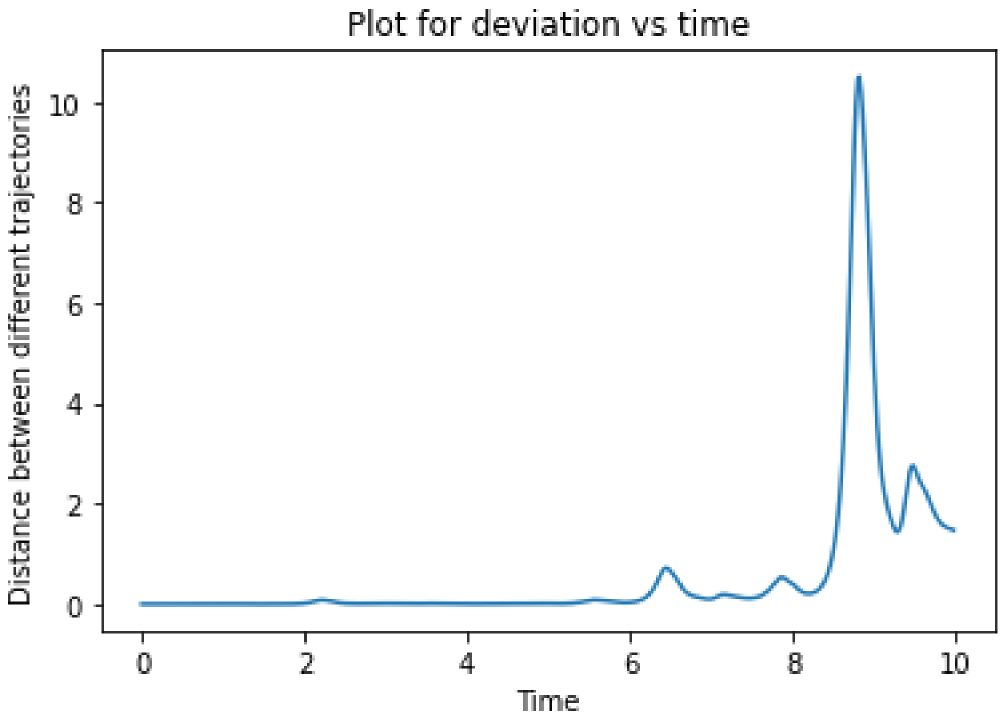
```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar  5 18:13:09 2020

@author: akswa

HW3b_ex25
"""

# python 3 version
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3

def rk4(x,t,tau,derivsRK,param):
    """
    ##  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,param)
    t_half = t + half_tau
    xtemp = x + half_tau*F1
    F2 = derivsRK(xtemp,t_half,param)
    xtemp = x + half_tau*F2
    F3 = derivsRK(xtemp,t_half,param)
    t_full = t + tau
    xtemp = x + tau*F3
    F4 = derivsRK(xtemp,t_full,param)
```

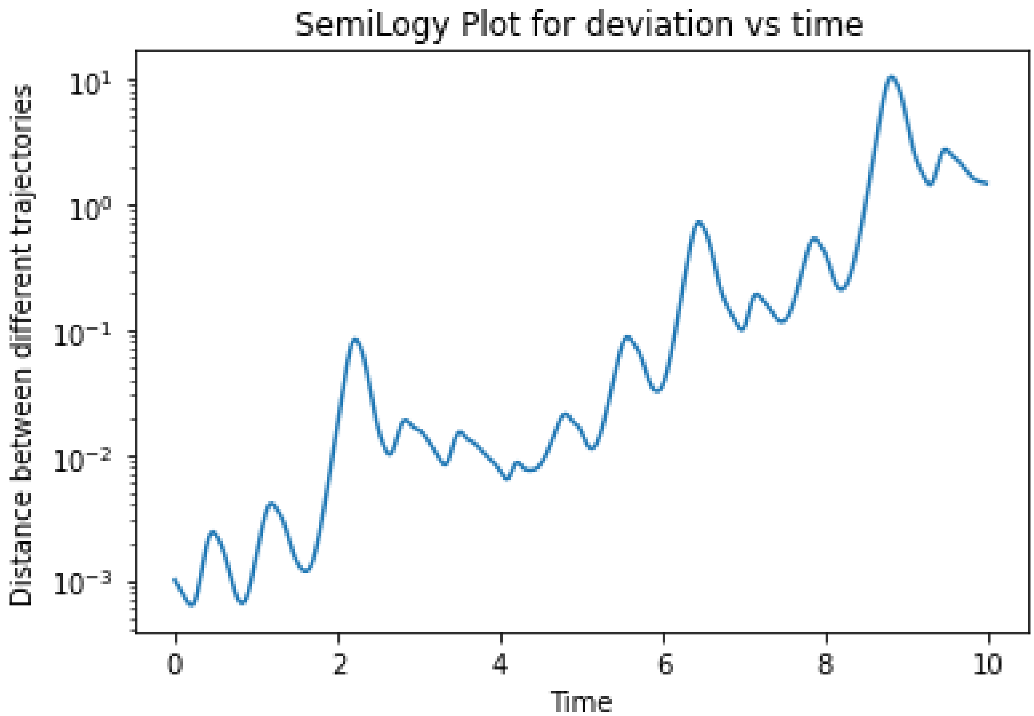
```python
    xout = x + tau/6.*(F1 + F4 + 2.*(F2+F3))
    return xout

def lorzrk(s,t,param):
    """
    #  Returns right-hand side of Lorenz model ODEs
    #  Inputs
    #    s      State vector [x y z]
    #    t      Time (not used)
    #    param  Parameters [r sigma b]
    #  Output
    #    deriv  Derivatives [dx/dt dy/dt dz/dt]
    """
    r = param[0]
    sigma = param[1]
    b = param[2]
    # For clarity, unravel input vectors
    x = s[0]; y = s[1]; z = s[2]
    # Return the derivatives [dx/dt dy/dt dz/dt]
    deriv = np.zeros(3)
    deriv[0] = sigma*(y-x)
    deriv[1] = r*x - y - x*z
    deriv[2] = x*y - b*z
    return deriv


def lorenz_data_gen(init_x,init_y,init_z,init_r):
    """
    Generates data needed to plot the results
    of lorentz.py using rk4 as in Ch3 ex 25

    Parameters
    ----------
    init_x : Float
        Inital x value.
    init_y : Float
        Initial y value.
    init_z : Float
        Inital z value.
    r : float
        Lorenz model parameter.


    Returns
    -------
    xplot : Numpy array
        Array of x-values used to plot.
```

```
    yplot : Numpy array
        Array of y-values used to plot.
    zplot : Numpy array
        Array of z-values used to plot.
    tplot : Numpy array
        Array of time-values used to plot.

    """
    # Set initial state x,y,z and parameters r,sigma,b
    sxin,syin,szin = init_x,init_y,init_z
    state = np.zeros(3)
    state[0] = float(sxin); state[1]  = float(syin); state[2]  = float(szin)


    r = init_r
    sigma = 10   # Parameter sigma
    b = 8./3.     # Parameter b
    param = np.array([r, sigma, b])  # Vector of parameters passed to rka
    tau = .02       # Timestep from lorenz with n=500
    #err = 1.e-3   # Error tolerance

    # Loop over the desired number of steps
    time = 0
    nstep = 500
    # initialize arrays
    tplot=np.array([]); tauplot=np.array([])
    xplot=np.array([]); yplot=np.array([]); zplot=np.array([])

    for istep in range(0,nstep):
        # Record values for plotting
        x = state[0]
        y = state[1]
        z = state[2]
        tplot = np.append(tplot,time)
        tauplot = np.append(tauplot,tau)
        xplot = np.append(xplot,x)
        yplot = np.append(yplot,y)
        zplot = np.append(zplot,z)
        #if( istep%50 ==0 ):
          #print('Finished %d steps out of %d '%(istep,nstep))
        # Find new state using Runge-Kutta4
        state = rk4(state,time,tau,lorzrk,param)
        time += tau
    """
    # Graph the time series x(t)
    plt.figure(1)
    plt.clf()  # Clear figure 1 window and bring forward
```

```
    plt.plot(tplot,xplot,'-')
    plt.xlabel('Time');  plt.ylabel('x(t)')
    plt.title('Lorenz model time series')
    # plt.show()

    # Graph the x,y,z phase space trajectory

    fig=plt.figure(2)
    ax=p3.Axes3D(fig)
    ax.plot3D(xplot,yplot,zplot)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.grid(True)
    # title('Lorenz model phase space')
    #ax.set_aspect('equal')
    plt.show()
    """
    return xplot,yplot,zplot,tplot

def lorenz_plot(initial_cond_list):

    ic_1 = initial_cond_list[0]
    ic_2 = initial_cond_list[1]

    xplot1,yplot1,zplot1,tplot1 =␣
↪lorenz_data_gen(ic_1[0],ic_1[1],ic_1[2],ic_1[3])
    xplot2,yplot2,zplot2,tplot2 =␣
↪lorenz_data_gen(ic_2[0],ic_2[1],ic_2[2],ic_2[3])

    # Calculate distance
    d = np.sqrt( (xplot1 - xplot2)**2 + (yplot1 - yplot2)**2 + (zplot1 -␣
↪zplot2)**2 )
    plt.figure(1)
    plt.plot(tplot1,d)
    plt.title("Plot for deviation vs time")
    plt.xlabel("Time")
    plt.ylabel("Distance between different trajectories")


    plt.figure(2)
    plt.semilogy(tplot1,d)
    plt.title("SemiLogy Plot for deviation vs time")
    plt.xlabel("Time")
    plt.ylabel("Distance between different trajectories")
```

[2]:
```
initial_cond_list = [(1,1,20,28),(1,1,20.001,28)]
lorenz_plot(initial_cond_list)
```




SemiLogy Plot for deviation vs time
Distance between different trajectories
Time
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
0
2
4
6
8
10

```
[ ]:
```