Steven Wray

Physics 4620

6/4/13

**The Lorenz Model**

***Introduction***

This project uses an adaptive Runga-Kutte routine to analyze the behavior of the Lorenz model, the Lotka-Volterra model, and the Hopf model. Particular attention is given to the chaotic behavior that arises from the Lorenz equations.

***Activities***

2.15:

2.16:

2.17:

2.23: The Hopf model is given by the ODEs:

1. Transform the equations into polar coordinates. Show that for the trajectories will spiral towards the origin and that for they spiral towards a circle of radius .

Beginning from

and taking the time derivative of both sides gives

Substituting these values into the model

Multiply the first equation by and the second by .

Add the two equations.

or

This is a separable differential equation, and can be rewritten

The left-hand side can be decomposed into partial fractions

So

Allowing to be either positive or negative lets us remove the absolute value

For , and so . Therefore the trajectory will spiral in towards the origin.

For , and so . So the trajectory will spiral towards a circle of radius .

1. Use an adaptive Runge-Kutta routine to plot the trajectories of the Hopf model and confirm the result in part (a).

**Conclusion**

**APPENDIX:**

**Source Code**

# pendulum.py

# Python 2.7.2

# Steven Wray

# Physics 4620

# Project 1: Simple pendulum

# 2.16: Use Euler, Euler-Cromer, leap-frog, midpoint, and Verlet's methods

# to approximate the motion of a simple pendulum.

import math

import numpy as np

import matplotlib.pyplot as plt

EULER = 1

VERLET = 2

EULER\_CROMER = 3

MIDPOINT = 4

LEAPFROG = 5

def main():

# Get parameters from user

print("Numerical methods")

print("\t1. Euler")

print("\t2. Verlet")

print("\t3. Euler-Cromer")

print("\t4. Midpoint")

print("\t5. Leapfrog")

s = input("Enter a numerical method: ")

numerical\_method = int(s)

s = input("Initial angle [degrees]: ")

theta0 = float(s)

s = input("Enter time step [seconds]: ")

tau = float(s)

s = input("Enter number of time steps: ")

nstep = int(s)

# Set physical constants and variables

G\_OVER\_L = 1

time = 0 # Initial time

reversals = 0 # Count reversals of pendulum

# Set initial position of pendulum

theta = theta0 \* math.pi / 180 # radians

omega = 0 # angular velocity

# Take a backward step to start Verlet

# and leapfrog methods

accel = - G\_OVER\_L \* math.sin(theta)

if numerical\_method == VERLET:

theta\_old = theta - omega \* tau + 0.5 \* tau\*\*2 \* accel

if numerical\_method == LEAPFROG:

omega = omega - tau \* accel

# Lists to store data for plotting

theta\_list = [theta]

time\_list = [0]

time = time + tau

for i in range(nstep):

# Compute new position and velocity

accel = - G\_OVER\_L \* math.sin(theta)

if numerical\_method == EULER:

theta\_old = theta

theta = theta + tau \* omega

omega = omega + tau \* accel

elif numerical\_method == VERLET:

theta\_new = 2 \* theta - theta\_old + tau\*\*2 \* accel

theta\_old = theta

theta = theta\_new

elif numerical\_method == EULER\_CROMER:

omega = omega + tau \* accel

theta\_old = theta

theta = theta + tau \* omega

elif numerical\_method == MIDPOINT:

omega\_new = omega + tau \*accel

theta\_old = theta

theta = theta + tau \* (omega\_new + omega) / 2

omega = omega\_new

else: # Leapfrog

omega = omega + 2 \* tau \* accel

time = time + tau

theta\_old = theta

theta = theta + 2 \* tau \* omega

# Record angle and time for plotting

theta\_list.append(theta \* 180 / math.pi) # in degrees

time\_list.append(time)

# See if the pendulum has passed through theta = 0

# and, if so, estimate period

if theta \* theta\_old < 0:

print "Turning point at time t = ", round(time, 4), " [s]."

if reversals == 0:

time\_old = time

elif reversals == 1:

period\_list = [2\*(time - time\_old)]

time\_old = time

else:

period\_list.append(2\*(time - time\_old))

time\_old = time

reversals = reversals + 1

time = time + tau

# Estimate period of oscillation with error

if reversals > 1:

average\_period = np.average(period\_list)

error = np.std(period\_list)

print "Average period ", round(average\_period, 3), " +/- ", \

round(error, 3), " [s]"

# Plot the data

line1 = plt.plot(time\_list, theta\_list, 'b')

plt.ylabel('Angle [degrees]')

plt.xlabel('Time [s]')

plt.show()

main()

# pendulum.py

# Python 2.7.2

# Steven Wray

# Physics 4620

# Project 1: Simple pendulum

# 2.17: Use Verlet's method to find the period of a pendulum for a range

# of initial angles. Compare this to the period given by different

# approximations.

import math

import numpy as np

import matplotlib.pyplot as plt

def main():

# Set physical constants and parameters

G\_OVER\_L = 1

nstep = 300 # Number of timestep for Verlet method

tau = 0.05 # Timestep

# Lists to store data for plotting

theta\_list = [0]

average\_period\_list = [0]

for theta0 in range(2, 170):

# Set initial conditions for pendulum

time = 0 # initial time

reversals = 0 # count number of reversals

theta = theta0 \* math.pi / 180 # radians

omega = 0 # angular velocity

# Take a backward step to start Verlet method

accel = - G\_OVER\_L \* math.sin(theta)

theta\_old = theta - omega \* tau + 0.5 \* tau\*\*2 \* accel

for i in range(nstep):

# Update position with Verlet method

accel = - G\_OVER\_L \* math.sin(theta)

theta\_new = 2 \* theta - theta\_old + tau\*\*2 \* accel

theta\_old = theta

theta = theta\_new

# See if the pendulum has passed through theta = 0

# and estimate period

if theta \* theta\_old < 0:

if reversals == 0:

time\_old = time

elif reversals == 1:

period\_list = [2\*(time - time\_old)]

time\_old = time

else:

period\_list.append(2\*(time - time\_old))

time\_old = time

reversals = reversals + 1

time = time + tau

# Estimate period of oscillation with error

if reversals > 1:

average\_period = np.average(period\_list)

error = np.std(period\_list)

if error / average\_period > 0.01:

print ("WARNING: Error in period is greater than 1%")

print ("\tfor initial angle", theta0)

# Record data for plotting

if theta\_list[-1] == 0:

theta\_list = [theta0]

average\_period\_list = [average\_period]

else:

theta\_list.append(theta0)

average\_period\_list.append(average\_period)

# Record dataset for plotting small angle approximation of period

average\_period = 2 \* math.pi \* math.sqrt(1/G\_OVER\_L)

small\_angle\_period = [average\_period]

for theta0 in range(3, 170):

small\_angle\_period.append(average\_period)

# Find theta where error in small angle approximation exceeds 10%

for i in range(len(theta\_list)):

error = math.fabs(average\_period\_list[i] - small\_angle\_period[i]) \

/ average\_period\_list[i]

if error > 0.1:

theta = theta\_list[i]

s = 'Error in small angle approximation exceeds 10% for theta = ' \

+ str(theta) + ' degrees.'

print(s)

break

# Record dataset for plotting quadratic approximation of period

elliptic\_period\_list = [0]

for theta0 in range (2,170):

theta = math.pi \* theta0 / 180

period = 2 \* math.pi \* math.sqrt(1/G\_OVER\_L) \* (1 + theta\*\*2 / 16)

if elliptic\_period\_list[-1] == 0:

elliptic\_period\_list = [period]

else:

elliptic\_period\_list.append(period)

# Find theta where error in quadratic approximation exceeds 10%

for i in range(len(theta\_list)):

error = math.fabs(average\_period\_list[i] - elliptic\_period\_list[i]) \

/ average\_period\_list[i]

if error > 0.1:

theta = theta\_list[i]

s = 'Error in elliptic integral approximation exceeds 10% for theta = ' \

+ str(theta) + ' degrees.'

print(s)

break

# Plot the datasets

fig = plt.figure()

ax = fig.add\_subplot(111)

ax.plot(theta\_list, average\_period\_list, 'b', \

theta\_list, small\_angle\_period, 'b--', \

theta\_list, elliptic\_period\_list, 'b:')

leg = ax.legend(("Verlet method", "Small angle (constant) approximation", \

"Quadratic approximation"), 'upper left', shadow=True)

ax.set\_ylim([0,16])

ax.grid(False)

ax.set\_xlabel('Initial Angle [degrees]')

ax.set\_ylabel('Period [s]')

plt.show()

main()

# pendulum.py

# Python 2.7.2

# Steven Wray

# Physics 4620

# Project 1: Simple pendulum

# 2.18: Use a numerical method to create a phase space plot

# of the pendulum's motion.

import math

import numpy as np

import matplotlib.pyplot as plt

EULER = 1

VERLET = 2

EULER\_CROMER = 3

MIDPOINT = 4

LEAPFROG = 5

def main():

# Get parameters from user

print("Numerical methods")

print("\t1. Euler")

print("\t2. Verlet")

print("\t3. Euler-Cromer")

print("\t4. Midpoint")

print("\t5. Leapfrog")

s = input("Enter a numerical method: ")

numerical\_method = int(s)

s = input("Initial angle [degrees]: ")

theta0 = float(s)

s = input("Enter time step [seconds]: ")

tau = float(s)

# Set physical constants and variables

G\_OVER\_L = 1

time = 0 # Initial time

reversals = 0 # Count reversals of pendulum

max\_steps = 10000 # Maximum timesteps

# Set initial position of pendulum

theta = theta0 \* math.pi / 180 # radians

omega = 0 # angular velocity

# Take a backward step to start Verlet

# and leapfrog methods

accel = - G\_OVER\_L \* math.sin(theta)

if numerical\_method == VERLET:

theta\_old = theta - omega \* tau + 0.5 \* tau\*\*2 \* accel

if numerical\_method == LEAPFROG:

omega = omega - tau \* accel

# Lists to store data for plotting

theta\_list = [theta0]

omega\_list = [omega]

time = time + tau

i = 0

while i < max\_steps:

# Compute new position and velocity

accel = - G\_OVER\_L \* math.sin(theta)

if numerical\_method == EULER:

theta\_old = theta

theta = theta + tau \* omega

omega = omega + tau \* accel

elif numerical\_method == VERLET:

theta\_new = 2 \* theta - theta\_old + tau\*\*2 \* accel

omega = (theta\_new - theta\_old) / (2 \* tau)

theta\_old = theta

theta = theta\_new

elif numerical\_method == EULER\_CROMER:

omega = omega + tau \* accel

theta\_old = theta

theta = theta + tau \* omega

elif numerical\_method == MIDPOINT:

omega\_new = omega + tau \*accel

theta\_old = theta

theta = theta + tau \* (omega\_new + omega) / 2

omega = omega\_new

else: # Leapfrog

omega = omega + 2 \* tau \* accel

time = time + tau

theta\_old = theta

theta = theta + 2 \* tau \* omega

# Record angle and time for plotting

theta\_list.append(theta \* 180 / math.pi) # in degrees

omega\_list.append(omega \* 180 / math.pi)

# See if the pendulum has passed through theta = 0

# and estimate period. Exit the loop if the pendulum

# has finished one complete period.

if theta \* theta\_old < 0:

print "Turning point at time t =", round(time, 4), " [s]."

if reversals == 0:

time\_old = time

elif reversals == 1:

period\_list = [2\*(time - time\_old)]

time\_old = time

else:

period\_list.append(2\*(time - time\_old))

time\_old = time

break

reversals = reversals + 1

time = time + tau

i = i + 1

# If we are using the Verlet method we need one additional value

# of omega to make our lists match up.

if numerical\_method == VERLET:

theta\_new = 2 \* theta - theta\_old + tau\*\*2 \* accel

omega = (theta\_new - theta\_old) / (2 \* tau)

omega\_list.pop(0)

omega\_list.append(omega \* 180 / math.pi)

# Estimate period of oscillation with error

if reversals > 1: # Make sure there is at least one data point

period\_sum = 0

average\_period = np.average(period\_list)

error = np.std(period\_list)

print "Average period ", round(average\_period, 5), " +/- ", \

round(error, 5), " [s]"

# Plot the data

# Plot the datasets

fig = plt.figure()

ax = fig.add\_subplot(111)

ax.plot(theta\_list, omega\_list, 'b')

ax.set\_ylim([-theta0 - 5, theta0 + 5])

ax.set\_xlim([-theta0 - 5, theta0 + 5])

ax.grid(False)

ax.set\_xlabel('Angle [degrees]')

ax.set\_ylabel('Angular velocity [deg/s]')

plt.show()

main()

# pendulum.py

# Python 2.7.2

# Steven Wray

# Physics 4620

# Project 1: Simple pendulum

# 2.20: Improve the estimate of the pendulum's period by using

# interpolation.

import math

import numpy as np

import matplotlib.pyplot as plt

# Function to interpolate between three data points

# using a quadratic Lagrange polynomial.

def interp(xi, x, y):

# x and y are arrays that contain the coordinates for the three

# data points

# xi: interpolate the function at this value

yi = (xi-x[1]) \* (xi-x[2]) / ((x[0]-x[1]) \* (x[0]-x[2])) \* y[0] \

+ (xi-x[0]) \* (xi-x[2]) / ((x[1]-x[0]) \* (x[1]-x[2])) \* y[1] \

+ (xi-x[0]) \* (xi-x[1]) / ((x[2]-x[0]) \* (x[2]-x[1])) \* y[2]

return yi

EULER = 1

VERLET = 2

EULER\_CROMER = 3

MIDPOINT = 4

LEAPFROG = 5

def main():

print("Numerical methods")

print("\t1. Euler")

print("\t2. Verlet")

print("\t3. Euler-Cromer")

print("\t4. Midpoint")

print("\t5. Leapfrog")

s = input("Enter a numerical method: ")

numerical\_method = int(s)

s = input("Initial angle [degrees]: ")

theta0 = float(s)

s = input("Enter time step [seconds]: ")

tau = float(s)

s = input("Enter number of time steps: ")

nstep = int(s)

# Set physical constants and variables

G\_OVER\_L = 1

time = 0 # Initial time

reversals = 0 # Count reversals of pendulum

# Set initial position of pendulum

theta = theta0 \* math.pi / 180 # radians

omega = 0 # angular velocity

# Take a backward step to start Verlet method

accel = - G\_OVER\_L \* math.sin(theta)

theta\_old = theta - omega \* tau + 0.5 \* tau\*\*2 \* accel

# Lists to store data for plotting

theta\_list = [theta]

time\_list = [0]

time = time + tau

for i in range(nstep):

# Record angle and time for plotting

# Compute new position

accel = - G\_OVER\_L \* math.sin(theta)

if numerical\_method == EULER:

theta\_old = theta

theta = theta + tau \* omega

omega = omega + tau \* accel

elif numerical\_method == VERLET:

theta\_new = 2 \* theta - theta\_old + tau\*\*2 \* accel

theta\_old = theta

theta = theta\_new

elif numerical\_method == EULER\_CROMER:

omega = omega + tau \* accel

theta\_old = theta

theta = theta + tau \* omega

elif numerical\_method == MIDPOINT:

omega\_new = omega + tau \*accel

theta\_old = theta

theta = theta + tau \* (omega\_new + omega) / 2

omega = omega\_new

else: # Leapfrog

omega = omega + 2 \* tau \* accel

time = time + tau

theta\_old = theta

theta = theta + 2 \* tau \* omega

# Record angle and time for plotting

theta\_list.append(theta \* 180 / math.pi) # in degrees

time\_list.append(time)

# See if the pendulum has passed through theta = 0

# and estimate period

if theta \* theta\_old < 0:

# Make sure we have at least three data points for interpolation

# Otherwise just use the current time as turning point

if (len(theta\_list) >= 3):

# Estimate turning point using interpolation

turning\_point = interp(0.0, \

[theta\_list[-3], theta\_list[-2], theta\_list[-1]],

[time\_list[-3], time\_list[-2], time\_list[-1]])

else:

turning\_point = time

print "Turning point at time t =", round(turning\_point, 4), " [s]."

if reversals == 0:

turning\_point\_old = turning\_point

elif reversals == 1:

period\_list = [2\*(turning\_point - turning\_point\_old)]

turning\_point\_old = turning\_point

else:

period\_list.append(2\*(turning\_point - turning\_point\_old))

turning\_point\_old = turning\_point

reversals = reversals + 1

time = time + tau

# Estimate period of oscillation with error

if reversals > 1:

period\_sum = 0

average\_period = np.average(period\_list)

error = np.std(period\_list)

print "Average period ", round(average\_period, 5), " +/- ", \

round(error, 7), " [s]"

# Plot the data

line1 = plt.plot(time\_list, theta\_list, 'b')

plt.ylabel('Angle [degrees]')

plt.xlabel('Time [s]')

plt.show()

main()