Numerical Analysis of Chaos in Lagrangian Systems

Jessie Wu
PH235: Physics Simulations
Prof. Anita Raja
May 7, 2017

Table of Contents

Introduction

Background

Chaos theory is a branch of physics and mathematics that focuses on the behavior of dynamical systems that are highly sensitive to initial conditions. An infinitesimally small change in initial condition would lead to a large change in state in a small amount of time; the errors in state grow exponentially over time. The systems are deterministic, which means that they are—rather than random—fully governed by a system of equations.

Double pendulums are a classic example of chaos, as they have been characterized, analyzed, and simulated by a number of physicists.[1] The motion of double pendulums are dependent on two second-order differential equations, but have seemingly-random motion based on initial condition. Reproduced in Figure 1, the two double pendulums start with a 0.1 degree difference in initial angle. At first, the two pendulums seem to follow the same path; however, within a few seconds, the pendulums have completely different angles and angular velocities.

Figure 1: Two double pendulums with a 0.1 degree difference in initial condition diverge in a short period of time. (Note: all animations are available on the project page.)

Problem Statement

This project focuses on the analysis and simulation of a triple pendulum, in a specific configuration shown in Figure 2. The specific configuration consists of three rods; the first rod rotates freely around its center, while the other two rods are attached to the end of the first rod. Friction and drag forces are not considered. The physical constants of the rods are shown in Table 1 and the energies of each rod is shown in Equation 1, 2, and 3.

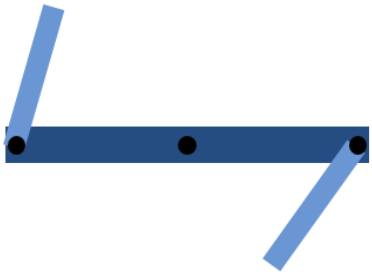

Figure 2: A specific configuration of a triple that is analyzed in this project.

Gravity, m/s^2	9.8
Length, m	0.4
Mass, kg	1.0

Table 1: Physical constants of the rods of a triple pendulum

$$TKE_i = \frac{1}{2} m_i v_i^2 \quad (1)$$

$$RKE_i = \frac{1}{2} I_i \omega_i^2 \quad (2)$$

$$GPE_i = m_i g h_i \quad (3)$$

Lagrangian Method

The Lagrangian Method is used to determine the equations of motion for the triple pendulum system. The Lagrangian Method is a powerful method to define the system without knowing all of the forces acting upon the system.[2] The solution process involves defining the Lagrangian and taking a series of derivatives, as shown in Equations 4 and 5.

$$L \equiv T - V \quad (4)$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) = \frac{\partial L}{\partial x} \quad (5)$$

Case Study: Pendulum Spring

In order to show the correctness and usefulness of the Lagrangian Method, a pendulum spring will be analyzed as a case study. The kinetic and potential energies are shown in Equations 6 and 7 respectively.

$$T = \frac{1}{2}m\left(\dot{x}^2 + (l+x)^2\dot{\theta}^2\right) \quad (6)$$

$$V = -mg(l+x)\cos\theta + \frac{1}{2}kx^2 \quad (7)$$

The kinetic energy contains the tangential and radial motion, while the potential energy contains the energy from gravity and the spring. After taking the appropriate derivatives according to the Lagrangian Method, there are two second-order differential equations shown in Equations 8 and 9.

$$m\ddot{x} = m(l+x)\dot{\theta}^2 + mgcos\theta - kx \quad (8)$$

$$m(l+x)\ddot{\theta} + 2m\dot{x}\dot{\theta} = -mgsin\theta \quad (9)$$

Equation 8 is the radial $F = ma$ equation, with gravity, a spring force, and centripetal acceleration. Equation 9 is the tangential form of $F = ma$, with gravity and the Coriolis effect. As demonstrated, the Lagrangian Method is useful because it would determine the correct equations of motions without knowing all the forces acting on the system.

Solution Process

The below section walks through the solution process of the code and how PH235 topics apply. The entire code with comments is in the Appendix. Animations, data sets, and in-line comments can be found on the project page.

From the constants and energy input, the Lagrangian Method is implemented. In the code, the derivatives are taken by using the SymPy library, a symbolic mathematics library. The output is three second-order differential equation. These equations can be re-written into six simultaneous first-order differential equations. The six equations are solved using the Runge-Kutta 4th Order algorithm, based on initial conditions and a given step size. The numerical solution of the three angles are converted into coordinates, which were plotted and animated using the Matplotlib library.

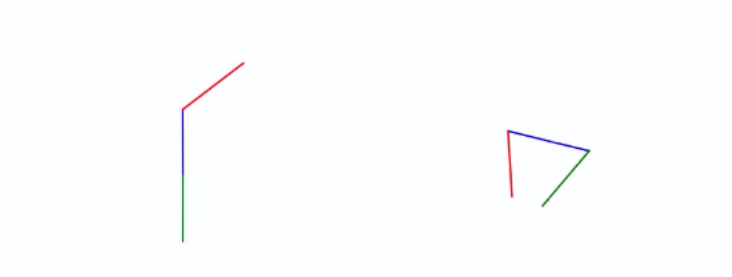

Figure 3: Animation of the triple pendulum. Full-animations can be found on the project page.

The solution is show in Figure 3. The middle pendulum rotates about its center, and has two pendulums attached to its ends. In order to demonstrate chaos, Figure 4 shows two sets of triple pendulums are animated in the same frame, with a 0.1 difference in initial condition. Within in a few seconds, the angles of the pendulums are completely different.

Figure 4: The two pendulums start at nearly similar positions, but have different states within a few seconds.

Correctness of RK4

To check for correctness of the RK4 algorithm, two methods are used: a crude energy conservation check and an error analysis. The energy is plotted against time, as shown in Figure 5. The maximum difference in energy is 3.5e-05. This number is small, showing that energy is conserved in the system.

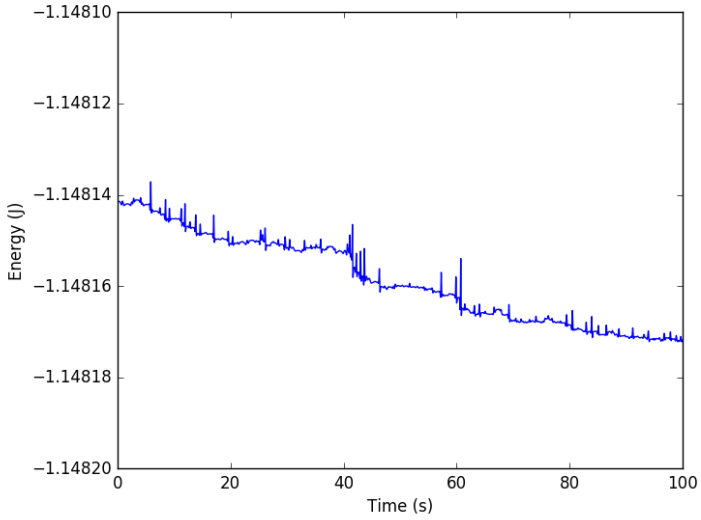

Figure 5: Energy of the system over time. Maximum difference in energy is 3.5e-05.

A more accurate method to check for correctness would be to compare the error as the step size varies.[3] The RK4 method has a fourth-order error, as shown in Equation 10. This means that if the step-size were halved, the error would be smaller by a factor of 16.

$$O(h^4) \approx Ch^4 \quad (10)$$

	Step Size, h	Approx. to y(0.3), ym	F.G.E. Error at t = 0.3, y(0.3)-ym	Error Ratio
0.1000	1/10	1.633115656339	0.000000940651	
0.0500	1/20	1.633116473076	0.000000123914	**8**
0.0250	1/40	1.633116587352	0.000000009638	**13**
0.0125	1/80	1.633116596333	0.000000000657	**15**
0.0001	0.0001	1.633116596990		

Table 2: RK4 approximation at different step sizes. The error ratios are approximately 16 as each the step size is halved

Table 2 compares the RK4 approximation at different step sizes. The numerical solution with a small step size, 0.0001, is assumed to be the exact solution. The final global error is calculated by comparing the approximations at the same time value. While fluctuations exist due to the assumption of the exact solution, the error ratios are within an order of magnitude, showing that the RK4 method is correctly implemented.

Extension to Other Systems

The above solution process can be expanded to other Lagrangian systems besides the specific pendulum. There are five steps in the solution process: (1) define constants and energies, (2) take derivatives to compute equations of motion, (3) run RK4 to determine numerical solutions, (4) convert from angles to coordinates (5) plot and animate results. Of the above steps, only two steps are required to be done manually: (1) defining constants and (4) converting from angles and coordinates. (The derivatives for the Lagrangian Method are taken using the SymPy symbolic math library.) Because of the simplicity of these two steps, the solution process can easily and quickly be extended to other systems.

Figure 6: Animation of a pendulum spring

Figure 6 shown above is the pendulum spring described in the above case study for the Lagrangian Method. Figure 7 shown below is another configuration of the triple pendulum, where one end of the pendulum is linked to another. These animations are all done by adapting the main code of the original solution process. As long as the energies and the coordinates of objects to be plotted can be quickly calculated, any dynamical system can be plotted using an adapted version of the code.

Figure 7: Animation of a linked triple pendulum

Further Work

Initially, the proposed project was to analyze, simulate, and animate the motion of the specific pendulum. The most challenging part was finding the correct equations of motion, as the derivatives were long and tedious. However, the difficulty was decreased after finding a symbolic math library that could perform derivatives. After discovering the library, the project was expanded to easily analyze, simulate, and animate any Lagrangian system, as shown with the pendulum spring and the linked triple pendulum.

Moving forward, there are additional features that can be implemented if more time was available. Currently, the program prompts the user for a file name to write the data to. A further improvement would be to prompt the user for a requested step size and initial condition, as these parameters typically change with each iteration of the code. Additionally, an adaptive step size can be implemented. Although the code would be slightly more complex, a new step size would not need to be calculated with each new Lagrangian system.

If this was a two-person project, a major improvement would be to include a GUI. Ideally, the user would be able to draw their own Lagrangian system, specify attachment points and spring properties, and then animate the results. Essentially, this would further automate the steps of determining constants and converting from angle to coordinates. Nonetheless, this project met and exceeded its initial proposal and successfully implemented the topics covered in PH235.

References

[1] Tony Shinbrot, Celso Grebogi, Jack Wisdom, James York. Chaos in a double pendulum. American Journal of Physics, American Association of Physics Teachers, 1992, 60, pp.491-491.

[2] David Morin. The Lagrangian Method. Introduction to Classical Mechanics. Cambridge University Press, 2008.

[3] John Mathews and Kurtis Fink. Runge-Kutta Methods. Numerical Methods Using Matlab, 4th Edision. Prentice-Hall Inc., 2004.

Appendix

There are two files of code used to solve, plot, and animate the system. The first, Code_Main.py, calculates the numerical solution and outputs it into a text file. The second, Animation_Main.py, reads the text file and animates the results. For further detail, data sets, animations, and examples of alternative set-ups, please visit the project page.

Code_Main.py

```
from __future__ import division,print_function
from numpy import array,arange,pi,zeros,savetxt,loadtxt
from pylab import plot,show
from sympy import cos,sin
from sympy.physics.mechanics import *
from time import time

# Jessie Wu, PH235 Final Project, Spring 2017
# The program writes the data for the motion of a triple pendulum given initial conditions
# Inputs are system dimensions and a Lagrange equation
# Sympy library computes derivatives and outputs EOM as 6 simaltaneous 1st-order ODE
# Use RK4 to numerically solve and output

## TOGGLES: h step, r initial condition
##

## INITIALIZE and find EQUATION OF MOTION

# Declare Dynamic Variables
u1, u2, u3 = dynamicsymbols('u1 u2 u3')
u1d, u2d, u3d = dynamicsymbols('u1 u2 u3', 1)

# Set Constants and Coefficients
m = 1.0   # kg
g = 9.8   # m/s
l = 0.40  # m
I = 0.4   # Moment of Intertia CHANGE FOR I1
I1 = I / 4

c1 = m*l**2/2 + m*l**2/2 + I1/2
c2 = m*l**2/2 + I/2
c3 = m*l**2/2 + I/2
c5 = l*m*g
c6 = l*m*g
c7 = -l**2*m
c8 =l**2*m

# Calculate Kinetic, Potential, and Lagrangian
T = c1*u1d**2 + c2*u2d**2 + c3*u3d**2 + c7*u1d*u2d*cos(u2-u1) + c8*u1d*u3d*cos(u3-u1)
V = c5*sin(u2) + c6*sin(u3)

L = T - V

# Form EOM
LM = LagrangesMethod(L, [u1, u2, u3])
mechanics_printing(pretty_print=False)
LM.form_lagranges_equations()
EOM = LM.rhs()
r_new = zeros(6)

print("Finished calculating EOM")
raw_input("Ready to run RK4 algorithm? : ")
print("Calculating...")

def f(r):
    # EOM is of the form [u1 u2 u3 u1d u2d u3s]
    # Substitute r values into EOM using subs and msubs
    subs = {u1:r[0],u2:r[1],u3:r[2],u1.diff():r[3],u2.diff():r[4],u3.diff():r[5]}
```

```
    dummy = msubs(EOM,subs)
    # Convert from Matrix into Array, and output array
    for i in range(len(EOM)):
        r_new[i] = dummy[i]
    return r_new

## Time Step and Arrays
a = 0       # Initial time
b = 0.31     # Final time
h = 0.0001 # Size of Runge-Kutta steps
tpoints = arange(a,b,h)

# TEST 0.1, 0.05, 0.025, 0.0125

theta1_pts = []
theta2_pts = []
theta3_pts = []
E_pts = []
i = 1
print("Total Points:")
print(len(tpoints))
start = time()

r = array([pi/2,-pi/2,pi/4,0,0,0],float) # SET INITIAL CONDITIONS [u1 u2 u3 ud1 ud2 ud3]

## Run through RK4 for time steps
for t in tpoints:
    theta1_pts.append(r[0])
    theta2_pts.append(r[1])
    theta3_pts.append(r[2])

    # Energy calculation
    T = c1*r[3]**2 + c2*r[4]**2 + c3*r[5]**2 + c7*r[3]*r[4]*cos(r[1]-r[0]) +
c8*r[3]*r[5]*cos(r[2]-r[0])
    V = c5*sin(r[1]) + c6*sin(r[2])
    E = T + V
    E_pts.append(E)

    # Algorithm for a 4th Order Runge-Kutta
    k1 = h*f(r)
    k2 = h*f(r+0.5*k1)
    k3 = h*f(r+0.5*k2)
    k4 = h*f(r+k3)
    r += (k1+2*k2+2*k3+k4)/6

    # Print Status and Time
    end = time() - start
    if i%200 == 0:
        print(i)
        print(end/60)
    i += 1

print("Done calculating, writing output")
fname = raw_input("What to name output file? : ")
savetxt(fname, (theta1_pts,theta2_pts,theta3_pts,tpoints,E_pts))
```

Animation_Main.py

```
from __future__ import division,print_function
from numpy import array,arange,cos,sin,pi,savetxt,loadtxt
from pylab import plot,show
from matplotlib import pyplot as plt
from matplotlib import animation

fname = raw_input("What file to read from? ")
data = loadtxt(fname)
[theta1_pts, theta2_pts, theta3_pts,tpoints,E_pts] = data

l1 = 0.4 # in
l2 = 0.4 # in
l3 = 0.4 # in

# Calculate position from theta
# Pendulum 1 has endpoints (x0,y0) and (x1,y1)
# Pendulum 2 has endpoints (x0,y0) and (x2,y2)
# Pendulum 3 has endpoints (x1,y1) and (x3,y3)
x0 = -l1/2*cos(theta1_pts)
y0 = -l1/2*sin(theta1_pts)
x1 = l1/2*cos(theta1_pts)
y1 = l1/2*sin(theta1_pts)
x2 = -l1/2*cos(theta1_pts) + l2*cos(theta2_pts)
y2 = -l1/2*sin(theta1_pts) + l2*sin(theta2_pts)
x3 = l1/2*cos(theta1_pts) + l3*cos(theta3_pts)
y3 = l1/2*sin(theta1_pts) + l3*sin(theta3_pts)

# Animation
fig = plt.figure()
ax = plt.axes(xlim=(-1, 1), ylim=(-1, 1))
line1, = ax.plot([], [], lw=2)
line2, = ax.plot([], [], lw=2)
line3, = ax.plot([], [], lw=2)

def init():
    line1.set_data([], [])
    line2.set_data([], [])
    line3.set_data([], [])
    return line1,line2,line3

def animate(i):
    i *= 1
    line1_x = array([x0[i],x1[i]])
    line1_y = array([y0[i],y1[i]])
    line2_x = array([x0[i],x2[i]])
    line2_y = array([y0[i],y2[i]])
    line3_x = array([x1[i],x3[i]])
    line3_y = array([y1[i],y3[i]])
    line1.set_data(line1_x, line1_y)
    line2.set_data(line2_x, line2_y)
    line3.set_data(line3_x, line3_y)

    return line1,line2,line3

anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=25000000, interval = 0.040, blit=True)

## Plots
#plot(tpoints,E_pts)
#show()
```