A Simple Chaotic Toy

Grant Mathews: gmmathews@ucdavis.edu

Michael Woods: mwoods@ucdavis.edu

University of California at Davis Physics Department

June 8, 2007

Abstract:

The dynamic system to be studied is a simple chaotic toy consisting of two rotors, each with three arms with a magnet at the end of each arm. The two rotors are let to spin with low friction about their center of mass while being close enough for the magnets from each rotor to influence each other. The magnetic forces cause chaotic motion in the dynamic system. The motion of the arms is simulated and visualized before being analyzed. The dynamic system is analyzed with and without friction showing that the stable points found on the physical model are only obtainable with the use of friction. Study of the Lyapunov exponents illustrate the chaos in the system.

Introduction:

The chaotic toy is superficially simple. With only two moving parts and a base, it appears to be a straightforward mechanics problem where one should be able to calculate quite easily the future path of the rotors. What we find is that the simple toy exhibits extremely complex motion. One goal is to investigate the erratic movement to determine whether the system is in fact chaotic and whether it can be successfully modeled.

The original viewing of the toy offers only a glimpse of the behaviors possible. There are very apparent fixed points that occur when friction has diminished the kinetic energy. This exhibits itself as a near motionless bond between arms near the origin. The simple nature of the toy begs a simulation engine to represent it and to be able to model the fixed points accurately. The fact that the toy is such a tangible object is intriguing. An engine should very accurately describe the toy if it so simple.

Background:

Very little background information is needed to understand the project. One point worth discussing is the formulation of our potential calculations. The Lagrangian method is used which should be discussed in a slight amount of depth. Firstly, our potential energy and kinetic energy are given as follows:

$$T = k_T\left(\dot{\theta}_1^2 + \dot{\theta}_2^2\right) \qquad\qquad U = k_U f(\theta_1, \theta_2)$$

The Lagrangian equations (one for each independent variable $\dot{\theta}_1$and $\dot{\theta}_2$) are then formulated as follows using $L = T - U$:

$$\frac{\partial L}{\partial \dot{\theta}} = \frac{d}{dt}\frac{\partial L}{\partial \ddot{\theta}}$$

Solving this system returns the equations used for calculating the behavior of the rotors. The use of a generic $f(\theta_1, \theta_2)$ function allows for switching potentials very easily. Both magnetic monopole and dipole potentials were used.

Dynamical System:

Again, the system chosen to study is a simple magnetic toy. Two rotors exist on a wooden base. At the end of each arm is a small magnet. Each of the three magnets of one motor will interact with the three magnets on the other and vice versa. The placement of the rotors allows for very close proximity of one arm to another when the distance is minimized. This minimized distance is thus the region of largest interaction and strongest forces.

The system does not have any motion until set into motion by hand. One or both rotors are to be given an original angular velocity. As soon as motion begins, the behavior to be studied occurs. The magnets interact and their angular velocities are constantly altered (hence, dynamic). This leads to the erratic behavior of the rotors that is the concentration of this study.

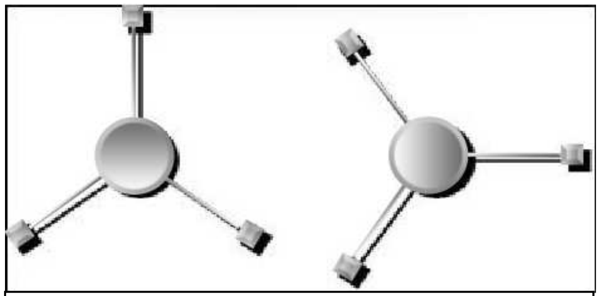
Figure 1. Diagram of a Simple Chaotic Toy. Magnet at each lever arm and freedom to rotate.

The first equation of motion is based off of the kinetic energy

$$T = k_T\left(\dot{\theta}_1^2 + \dot{\theta}_2^2\right)$$

that describes the combined kinetic energy of our two rotors. The k_T term is an easy way to control all of the coefficient terms that occur in the kinetic energy term (like rotational inertia).The potential energy used was originally a magnetic monopole (or electric monopole) because it was much easier to program, was less CPU intensive, and was able to be solved explicitly so as to check the RK4 process we used. The magnetic dipole form was of the form

$$U \propto \frac{k_U}{r^3}$$

The value for k_U is, like above, the single coefficient we use so allow easier alterations to the strength of the potential.

Using these two potentials, the appropriate accelerations are calculable and can be used with a given interval of time to calculate the new positions of the arms on the rotors as well as their angular velocity.

Methods:

The project began with the assembly of our simulation engine in Python. The magnetic monopole potential was used to simplify computation and to lessen the wait of trials. The engine itself was fairly easy to construct after the vector analysis of the distances between the magnets on each arm was built.

An initial visualization system was used from the beginning to gauge the accuracy of the simulation. Using Tk, a very crude live simulation was produced that allowed the initial critique of the system.

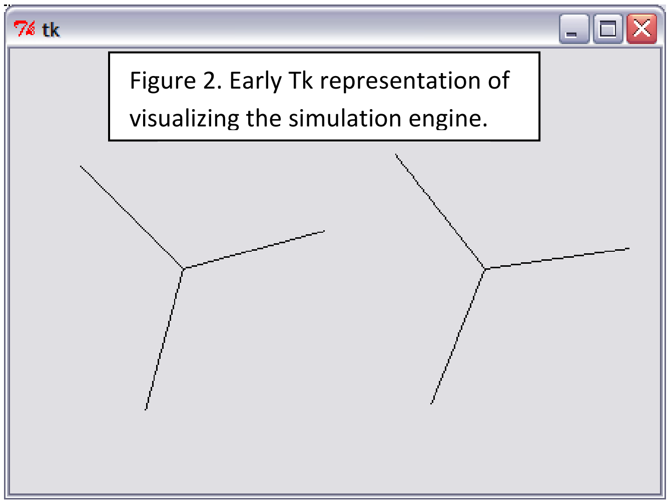

Figure 2. Early Tk representation of visualizing the simulation engine.

Once the engine was up and working with the monopole simulation, the dipole potential was used. This led to a greater number of calculations and slowed down the simulation so much that new implementations for calculation and display were developed because real time generation of data was not possible. The simulation code was rewritten in C so to implement available optimization that reduced the time needed to calculate thirty seconds of data from 15 minutes down to 36 seconds.

Since a data file can be read into a visualization system quickly when it is not calculated on the go, a new display system was created from Visual Python. This added three dimensions with easy implementation for creating multiple rotor systems to study phase deconstruction or destabilization.

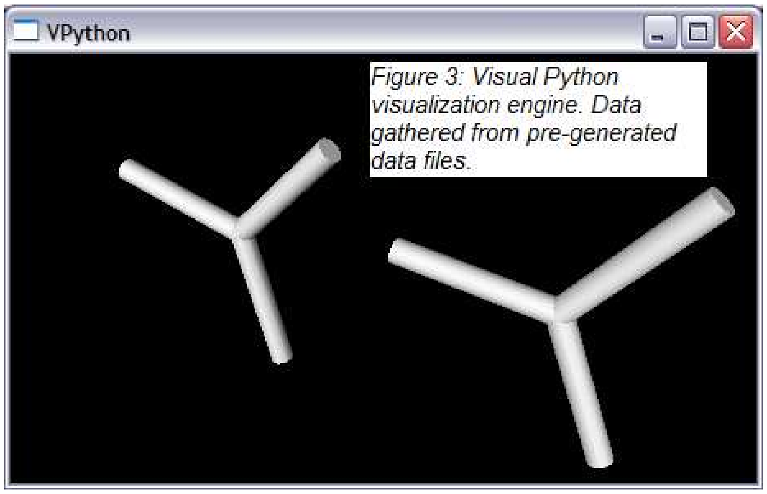

Figure 3: Visual Python visualization engine. Data gathered from pre-generated data files.

The most difficult part of the programming to this point was obtaining a conservation of energy. When the rotors met near their minimum distance, there was a net decrease in energy due to the extreme increase in their accelerations. Increasing the accuracy of certain variables (double to long double) and decreasing the time step helped while using the optimized calls of trigonometric functions from the Math library (versus the slower Numeric functions) to keep data generation at a moderate pace.

Results:

The first result we found was the difference in several (near identical) versions of our code. Below is a graph showing the exact same code implementation for four slight variants. We switch between two C compilations, one with double and one with long double, and two Python versions, one with a main function call inline versus explicit. For

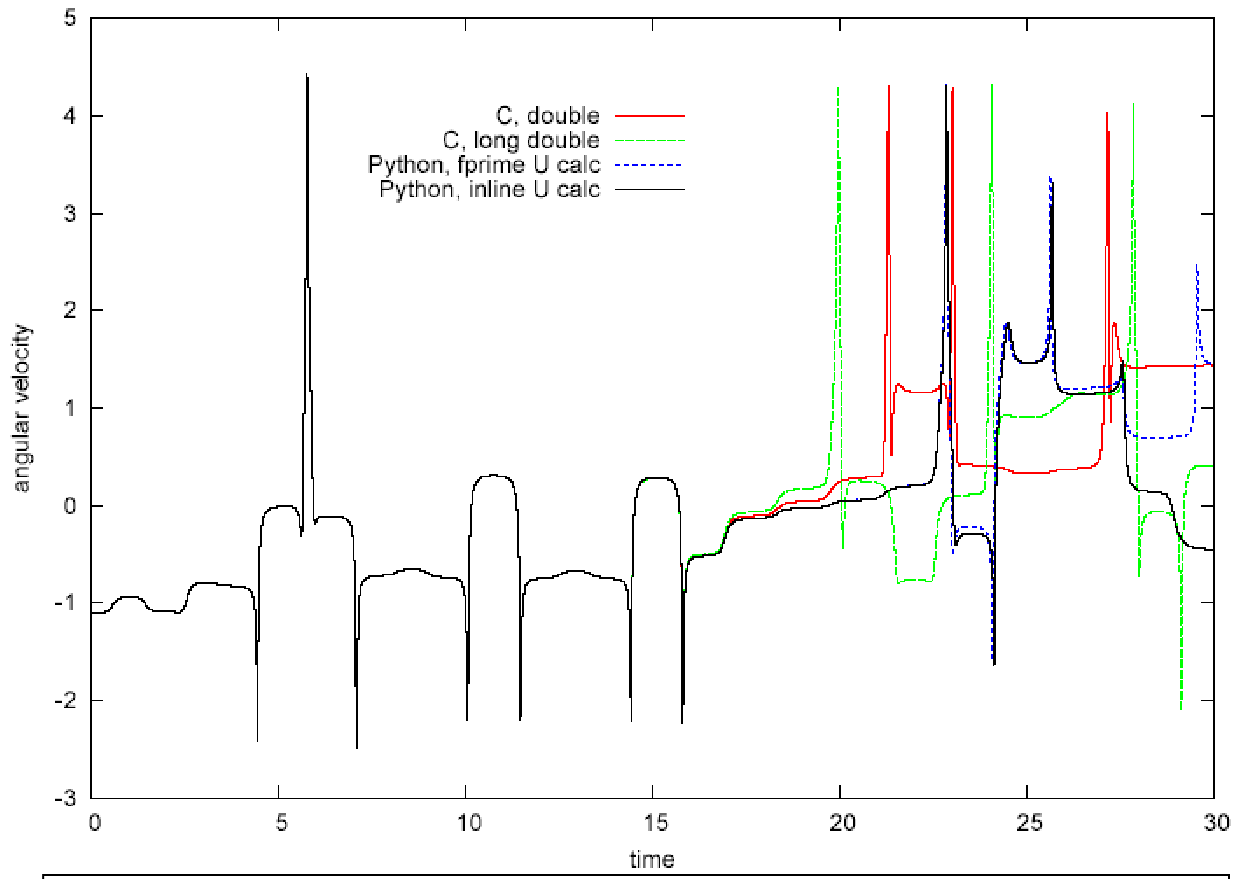

Figure 4. Plot of Time (sec) versus ω_1 of the right rotor. After 15 seconds, each diverges suggesting that even writing one function inline can alter the final state of the system.

the first 15 seconds of simulation, they match perfectly with no deviation. However, the error in the data is exponentially magnified until they split off into their own chaotic behavior. The difference between to two C codes is understandable given the numerical accuracy change leading to the exponentially increasing difference, but the Python version represents two runs that have the exact same code implementation, just written in a difference place.

To confirm the chaotic behavior of the system, a plot of the Lyapunov exponent was created as shown below using the equation

$$\lambda = \lim_{x\to 0} \lim_{t\to\infty} \frac{1}{t} \log_2 \frac{|\delta x(t)|}{|\delta x_0|}$$

The state of the system, after a brief period where λ is negative indicating non-chaotic behavior quickly turns positive and chaotic. The dips that go to zero represent when the rotors momentarily pass by the same theta displacement. We see that the system never returns to a period of non-chaos. When this was repeated after including a velocity-

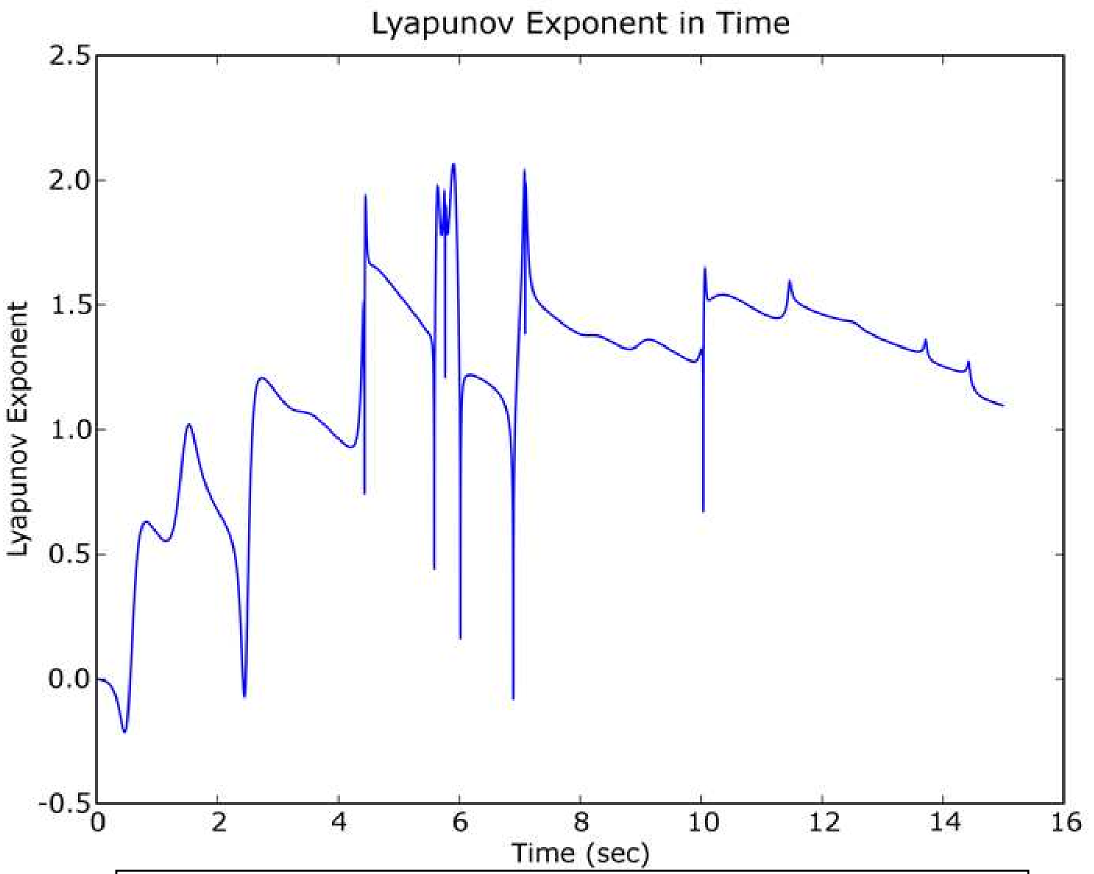

Figure 5. Lyapunov Exponent. The positive value for λ indicates an always constant chaotic behavior for a frictionless system.

dependent friction term, the Lyapunov exponent tends toward zero. The time factor in the Lyapunov equation above tends toward zero as time increases and the two rotors settle into an oscillatory state near the origin. Thus we find that the exponent gets smaller and smaller as $\frac{|\delta x(t)|}{|\delta x_0|}$ stops changing while the $1/t$ factor goes to zero. In this case, the system is always chaotic except in the limit of $t \to \infty$.

The following graph illustrates the chaotic differences between several runs in a system with friction. The initial angular velocity of one of the rotors is altered by the values shown in the key. Each one differs by one in one-hundred thousandth from the one before it. The time series begins after thirty seconds of non-divergence due to fast moving rotors.

It is clearly visible that once the chaotic movement sets in that the trials separate and end in a certain fixed points. There seem to be five such points but these each correlate

Time versus θ_1 for Different Initial Conditions

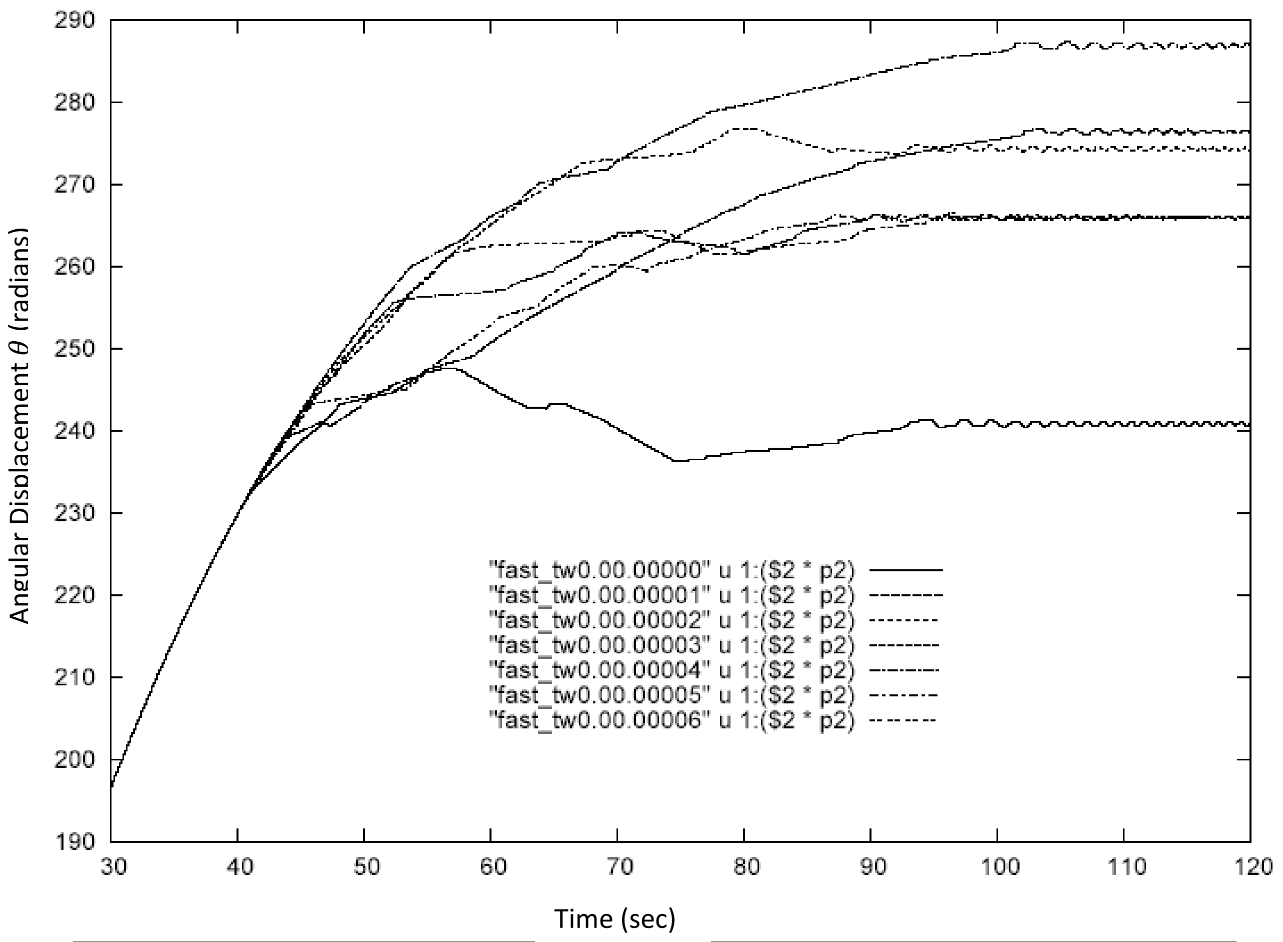

Figure 6. Time versus angular displacement for varying initial conditions (on the order of 1/100,000 in angular velocity) in a system with dissipative friction. Neighboring initial conditions do not necessarily end in neighboring fixed points.

to only one of three different fixed points where two arms hover near each other oscillating slightly. If the displacement were subject to modulus three, we would see these three fixed points explicitly: one for theta equals $0, 2\pi/3$ and $, 4\pi/3$.

Neighboring initial conditions do not have a correlation to their final state. The lowest final state correlates to trial 00.0000 and the others show no pattern for filling the varied fixed points. The fixed points do not even occur in the frictionless system. Without dissipation, the system has an energy which is much too high and overcomes the bounding energy between these stable states. This pattern of the spreading of initial conditions occurs no matter what the initial distance between conditions is. A trial where initial conditions were separated by one-ten-millionth exhibited identical patterns of congruity, chaos, and independent final states.

Conclusion:

The Simple Chaotic Toy illustrated a “simple” chaotic system containing several chaotic behaviors illustrating interesting points in its chaotic study and the computation of the data from the simulation engine. Only the model that included friction could be considered to accurately describe the system. The original goal of the simulation was to create an engine that properly illustrated the given fixed points the physical model exhibits. In the system without friction, there are **no** fixed points. Even the case were $\omega_1 \gg \omega_2$and the slow moving rotor oscillates slightly about the origin was not possible without friction. The dissipative model could accurately model all physical fixed points and is thus deemed a good physical model. The Lyapunov exponent of the frictionless model illustrates clearly the chaotic nature of the toy, confirming the visual inclination one has about the behavior, but in order to determine the level of chaos for the model with friction we examine that system’s λ plot. Due to the friction, the mathematical term $1/t$ causes the exponent to tend toward zero but since it is only a limit it can be said that it will always be chaotic. This again suggests congruence to the observation of the real world system.

The computational differences in variances of code suggest that the accuracy of any method of any simulation will be scrutinized for accuracy after only a short amount of time (thirty seconds at most for moderate time steps). Using certain Python packages that offer the ability to control the size of the time step where accuracy is much more important – like when the rotors are interacting the strongest – would help to increase accuracy by decreasing step size while not slowing the simulation to a halt. This could be used in the future for longer term data runs.

Bibliography:

None.

Code:

The code in the on the following pages represents the following

If any information is missing or desired (e.g. a sample data run) please contact the authors.

```
# This is the MEAT of the program. This portion is the data
# creation or the simulation engine.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FPRIME_H (1.0e-7)
#define Q 0.08
//#define Q 0.15
#define L 0.04
#define D 0.085
#define KT (0.025*L)
#define KU (1.e-7)
#define A 2.0943951023931954923084289221863
#define CA -0.5
#define SA 0.86602540378443864676372317075294
#define PKU (KU*Q*Q) //(Cruft below)
//double PKU; //  = KU*Q*Q;

#define KF 0.03125

double theta1, theta2, omega1, omega2, t=0.0, dt=0.001;

typedef struct {
        double a, b;
} ddouble;

inline double fprime(double (*f)(double), double x){
        return (f(x + FPRIME_H) - f(x - FPRIME_H)) / (2.*FPRIME_H);
}

inline double fpartial1(double (*f)(double, double), double x, double y){
        return (f(x + FPRIME_H, y) - f(x - FPRIME_H, y)) / (2.*FPRIME_H);
}

inline double fpartial2(double (*f)(double, double), double x, double y){
        return (f(x, y + FPRIME_H) - f(x, y - FPRIME_H)) / (2.*FPRIME_H);
}

double rk4(double (*f)(double, double), double x, double y, double h)
{
        double k1, k2, k3, k4;
        k1 = f(x,y);
        k2 = f(x + 0.5*h, y + 0.5*h*k1);
        k3 = f(x + 0.5*h, y + 0.5*h*k2);
        k4 = f(x + h, y + h*k3);
        return y + (1./6.)*h*(k1 + 2*k2 + 2*k3 + k4);
}

ddouble rk42(double (*f)(double, double, double), \
                double (*g)(double, double, double), \
                double x, double y1, double y2, double h)
{
        double k11, k12, k21, k22, k31, k32, k41, k42;
        ddouble r;

        k11 = f(x, y1, y2);
        k12 = g(x, y1, y2);

        k21 = f(x + 0.5*h, y1 + 0.5*h*k11, y2 + 0.5*h*k12);
        k22 = g(x + 0.5*h, y1 + 0.5*h*k11, y2 + 0.5*h*k12);

        k31 = f(x + 0.5*h, y1 + 0.5*h*k21, y2 + 0.5*h*k22);
        k32 = g(x + 0.5*h, y1 + 0.5*h*k21, y2 + 0.5*h*k22);

        k41 = f(x + h, y1 + h*k31, y2 + h*k32);
        k42 = g(x + h, y1 + h*k31, y2 + h*k32);

        r.a = y1 + (1./6.)*h*(k11 + 2.*k21 + 2.*k31 + k41);
```

```
		r.b = y2 + (1./6.)*h*(k12 + 2.*k22 + 2.*k32 + k42);
		return r;
}

double Utotdip(double t1, double t2)
{
		double i, j, Utot=0., ct1, st1, ct2, st2, rx, ry, ri;
		for(i=0.; i<(3.*A); i+=A)
		{
				ct1 = cos(t1+i);
				st1 = sin(t1+i);
				for(j=0.; j<(3.*A); j+=A)
				{
						ct2 = cos(t2+j);
						st2 = sin(t2+j);
						rx = D - L*(ct1 + ct2);
						ry = L*(st2 - st1);
						ri = 1.0 / sqrt(rx*rx + ry*ry);
						Utot += ((st1*st2-ct1*ct2) - \
								3.*ri*(ct1*rx+st1*ry)*(st2*ry-ct2*rx))*ri*ri*ri;
				}
		}
		return PKU*Utot;
}

double Ttot() {
		return KT*(omega1*omega1 + omega2*omega2);
}

double Utot() {
		return Utotdip(theta1, theta2);
}

double Etot() {
		return Utot() + Ttot();
}

double g1(double t, double w1, double w2) {
		double t1, t2;
		t1 = theta1 + w1*t;
		t2 = theta2 + w2*t;
		return (-0.25/(KT*FPRIME_H)) * \
				(Utotdip(t1 + FPRIME_H, t2) - Utotdip(t1 - FPRIME_H, t2)) \
				- KF*w1;
}

double g2(double t, double w1, double w2) {
		double t1, t2;
		t1 = theta1 + w1*t;
		t2 = theta2 + w2*t;
		return (-0.25/(KT*FPRIME_H)) * \
				(Utotdip(t1, t2 + FPRIME_H) - Utotdip(t1, t2 - FPRIME_H)) \
				- KF*w2;
}

void step(void)
{
		ddouble w;
		double wo1, wo2;
		wo1 = omega1;
		wo2 = omega2;
		w = rk42(g1, g2, 0.0, omega1, omega2, dt);
		omega1 = w.a;
		omega2 = w.b;
		w.a = wo1 + 0.5*dt*g1(0.0, wo1, wo2);
		w.b = wo2 + 0.5*dt*g2(0.0, wo1, wo2);
		theta1 += dt*w.a;
		theta2 += dt*w.b;
		t += dt;
}
```

```
void stepn(int n)
{
	int i;
	ddouble w;
	double wo1, wo2;
	for(i=0; i<n; i++)
	{
		wo1 = omega1;
		wo2 = omega2;
		w = rk42(g1, g2, 0.0, omega1, omega2, dt);
		omega1 = w.a;
		omega2 = w.b;
		w.a = wo1 + 0.5*dt*g1(0.0, wo1, wo2);
		w.b = wo2 + 0.5*dt*g2(0.0, wo1, wo2);
		theta1 += dt*w.a;
		theta2 += dt*w.b;
		t += dt;
	}
}

int main(int argc, char **argv)
{
	int i;

	if(argc != 5)
	{
		printf("Err1\n");
		exit(-1); // Err!
	}

	/*
	theta1 = 1.0;
	theta2 = 0.0;
	omega1 = 1.0;
	omega2 = -1.1;
	*/
	dt = 50e-6;

	theta1 = atof(argv[1]);
	theta2 = atof(argv[2]);
	omega1 = atof(argv[3]);
	omega2 = atof(argv[4]);

	for(i=0; i<30000; i++)
	{
		stepn(20);
		printf("%.6e %.11e %.11e %.11e %.11e\n", \
			t, theta1, theta2, omega1, omega2);
	}


	//printf("%f, %f\n", Utot(), Ttot());
	return 0;
}
```

```
# This replay file will replay data from TWO files
# to illustrate the phasing seen between initial
# conditions that vary slightly from eachother.

#from numpy import *
from visual import *
import fileinput
import magtoy

#import psyco
#psyco.full()


# begin _other_ graphics cruft

scene.autocenter = true

a = 2.*pi/3.
l = 0.04
D = 0.085
kT = 0.025*l
kU = 1.e-7
q = 0.15
kUqq = kU*q*q

# ===========FIRST TWO ROTORS===========
f1 = frame(pos=(0,0,0))
f1c1 = cylinder(frame=f1, axis=(-l,0,0), pos=(l,0,0), radius=0.08*l)
f1c2 = cylinder(frame=f1, axis=(-l*cos(a),-l*sin(a),0), pos=(l*cos(a),l*sin(a),0), radius=0.08*l)
f1c3 = cylinder(frame=f1, axis=(-l*cos(a+a),-l*sin(a+a),0), pos=(l*cos(a+a),l*sin(a+a),0), radius=0.08*l)

f2 = frame(pos=(D,0,0))
f2c1 = cylinder(frame=f2, pos=(l,0,0), axis=(-l,0,0), radius=0.08*l)
f2c2 = cylinder(frame=f2, pos=(l*cos(a),l*sin(a),0), axis=(-l*cos(a),-l*sin(a),0), radius=0.08*l)
f2c3 = cylinder(frame=f2, pos=(l*cos(a+a),l*sin(a+a),0), axis=(-l*cos(a+a),-l*sin(a+a),0), radius=0.08*l)

f1.axis = (-1,0,0)
f2.axis = (1,0,0)

# ===========SECOND TWO ROTORS===========
f3 = frame(pos=(0,0,0))
f3c1 = cylinder(frame=f3, axis=(-l,0,0), pos=(l,0,0), radius=0.08*l)
f3c2 = cylinder(frame=f3, axis=(-l*cos(a),-l*sin(a),0), pos=(l*cos(a),l*sin(a),0), radius=0.08*l)
f3c3 = cylinder(frame=f3, axis=(-l*cos(a+a),-l*sin(a+a),0), pos=(l*cos(a+a),l*sin(a+a),0), radius=0.08*l)
# f3c1.color,f3c2.color,f3c3.color = color.red,color.red,color.red

f4 = frame(pos=(D,0,0))
f4c1 = cylinder(frame=f4, pos=(l,0,0), axis=(-l,0,0), radius=0.08*l)
f4c2 = cylinder(frame=f4, pos=(l*cos(a),l*sin(a),0), axis=(-l*cos(a),-l*sin(a),0), radius=0.08*l)
f4c3 = cylinder(frame=f4, pos=(l*cos(a+a),l*sin(a+a),0), axis=(-l*cos(a+a),-l*sin(a+a),0), radius=0.08*l)
# f4c1.color,f4c2.color,f4c3.color = color.red,color.red,color.red

f3.axis = (-1,0,0)
f4.axis = (1,0,0)

scene.autocenter = false
scene.autoscale = false


from itertools import izip

i = 0
U1, T1, E1 = [], [], []
U2, T2, E2 = [], [], []

# ========== NAME THE DATA FILES HERE ============
file1 = fileinput.input("thirtyseconds09dip")
file2 = fileinput.input("thirtyseconds08dip")
```

```
for (il1,il2) in izip(file1,file2):
        sl1 = il1.split()
        sl2 = il2.split()

        if sl1[0] == '#':
                continue
        if sl2[0] == '#':
                continue
        t1 = float(sl1[1])
        t2 = float(sl1[2])
        t3 = float(sl2[1])
        t4 = float(sl2[2])
        #w1 = float(sl[3])
        #w2 = float(sl[4])
        #E1.append(float(sl1[5]) + float(sl1[6]))
        #E2.append(float(sl2[5]) + float(sl2[6]))
        #print max(E1) - min(E1),  '\t\t'  , max(E2) - min(E2)
        rate(1000)
        f1.axis = (l*math.cos(t1), l*math.sin(t1), 0.)
        f2.axis = (-l*math.cos(t2), l*math.sin(t2), 0.)
        f3.axis = (l*math.cos(t3), l*math.sin(t3), 0.)
        f4.axis = (-l*math.cos(t4), l*math.sin(t4), 0.)
        #cwin.interact()
```

```
# This program is designed to produce a graph of
# the Lyapunov exponent of our Simple Chaotic Toy
# in ipython. It is not set up to create the graph
# automatically. In ipython, plot(x,y) is needed to
# generate the graph, as possibly well adding one more
# item to y to make len(x) = len(y) due to erroring on
# a divide by zero.

from itertools import izip
from math import *
from pylab import *

x = []
y = []


#file1 = open('lypnvw2-1.1000000000001')
#file1 = open('lypnvw2-1.10001')
#file2 = open('lypnvw2-1.1')
file1 = open('thirtyseconds08dip')
file2 = open('thirtyseconds09dip')

#for line1,line2 in izip(file1,file2):
#           line1 = line1.split()
#           line2 = line2.split()
#           if line1[0] == '#':
#                       continue
#           print line1
#           delta_x0 = abs(abs(float(line1[3])) - abs(float(line2[3])))
#           print delta_x0
#           print line1
#           break


for line1,line2 in izip(file1,file2):
            line1 = line1.split()
            line2 = line2.split()
            #print line1, '\n'

            if line1[0] == '#' or float(line1[0]) == 0:
                        continue

            if len(x) == 0:
                        delta_x0 = abs(abs(float(line1[4])) - abs(float(line2[4])))

            x.append(float(line1[0]))
            y.append( (1 / float(line1[0])) * log( abs(float(line1[4]) - float(line2[4])) / abs(delta_x0)) / log(2))

            # t theta1 theta2 omega1 omega2 U T
#0.0 1.0 0.0 1.0 1.1 -2.34122905708e-005 0.00221

#lambda = (1 / TOTAL_TIME) * log(||delta_x(t)|| / ||delta_x(0)||)    /   log(2)
```