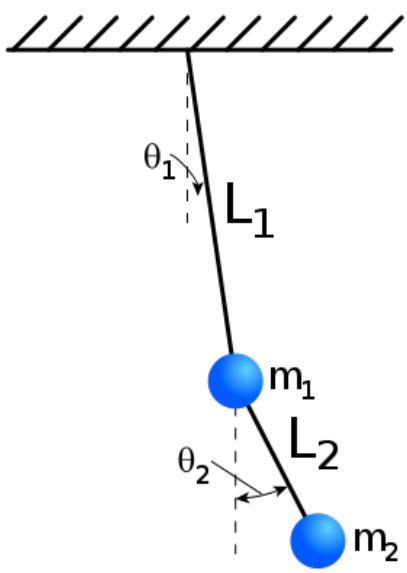
Plotting the Chaotic Motion of a Double Pendulum in Python
By Catherine Terrell
Shumway - PHY 121

Introduction



Physicists run computer simulations to predict the motion of complicated mechanical systems. In this project, we will plot the motion of a double pendulum, an extremely variable system depending on its starting conditions (also known as a chaotic system). Before plotting the double pendulum, we will program the path of a simple trajectory (resembling a thrown ball), taking into account air resistance. We will find the distance traveled by the trajectory, and the angle at which to throw the object that yields the greatest possible distance. We will then program the motion for a simple pendulum with one mass, followed by the motion of a double pendulum.

We will be using the Python programming language, and the Verlet algorithm to calculate the trajectories.

Fig. 1 A double pendulum
(courtesy of JabberWok, under Wikipedia Commons)

Part I: The Verlet Algorithm

To graph an object's motion, we need coordinates dependent on time. Using python, we specify how long we want the simulation to run, and how frequently coordinates are calculated within that time period (e.g. run a simulation for 10 seconds, and calculate the position every 0.01 seconds).

To do this, we create an array for each coordinate, or a string of numbers. Originally, we begin with a string of zeros. We specify the first number in the string (the starting position or starting velocity), and we give the computer an equation to calculate the following velocity based on the first two velocities, and another velocity from previous two, and so on. The method of calculating coordinates based on the last two previous coordinates is called the Verlet Algorithm.

Part II: Projectile Motion with Friction

A. The Force Function

The position of a number in an array is specified by the number i (ex. for the third number, i =3). We have already created arrays of zeros for x, y, vx, and vy. After setting the initial conditions (x[0], y[0], vx[0], and vy[0]), we need to define the force. We will use the 'force' function to fill in the arrays of zeros for x and y. For a simple projectile (where alpha is air resistance [constant]):

```
def force(x,y,vx,vy):
  return -alpha*vx, -alpha*vy -mass*g

for i in range(nstep):
  x[i+1] = x[i] + dt * vx[i]
  y[i+1] = y[i] + dt * vy[i]

  fx,fy = force(x[i],y[i],vx[i],vy[i])

  vx[i+1] = vx[i] + dt * fx/mass
  vy[i+1] = vy[i] + dt * fy/mass
```

We use the function 'force' to find fx and fy, which are used to find vx and vy, which are then used to find the x and y coordinates at i.

Note Fig. 2 and Fig. 3. With air resistance, the path of the trajectory is no longer symmetrical, changing the optimal angle used to obtain the maximum horizontal distance.

Fig. 2 *Without air resistance*

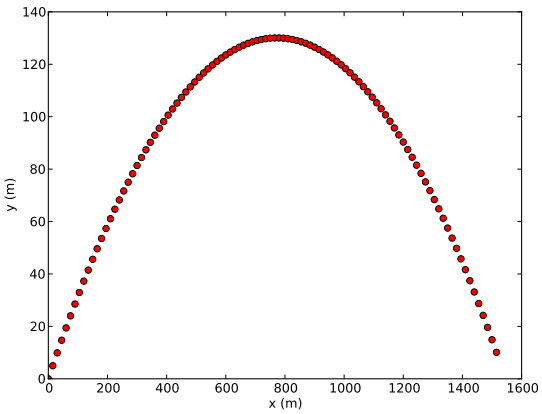
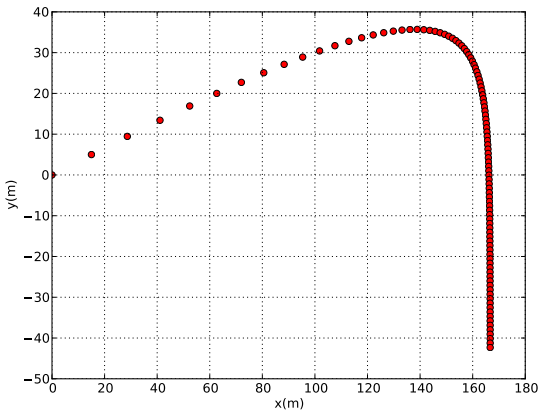


Fig. 3 *alpha = 0.9*



B. The Shoot Function

To calculate the distance an object travels, we create a 'shoot' function. This function will find the x-coordinate when y=0.

Essentially, we tell the computer to find the point with the last positive y-coordinate (imax) and the point with the first negative y-coordinate (imax+1). We draw a line from each point to the x-axis, creating two similar triangles. We use the ratio of these triangles to find the x-coordinate at which the object hits the ground (groundx).

With similar triangles (Fig. 4), we find the ratio of the sides to be:

$$\frac{y[imax]}{groundx - x[imax]} = \frac{y[imax+1]}{x[imax+1] - groundx}$$

and solve for groundx.

To find the time (groundt) at which the object hits the ground, we add t[imax] to the time interval between x[imax] and groundx. This time interval is simply the ratio of the x-distance from x[imax] to groundx over the total x-distance, multiplied by dt.

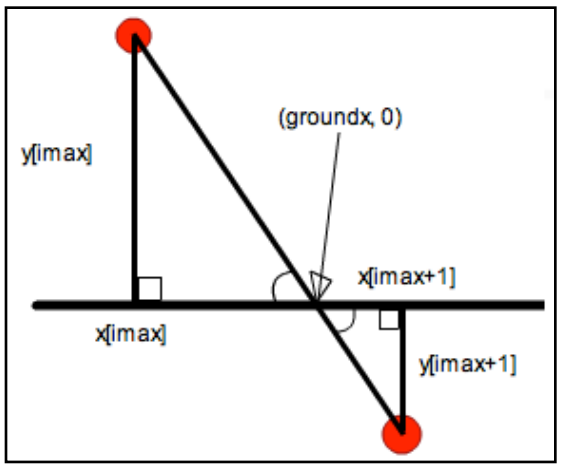


Fig. 4 Using like triangles to find groundx

The 'shoot' function:

```
imax = numpy.where (y>0)[0][-1]
groundx = (y[imax] * x[imax+1] - x[imax] * y[imax+1])/(y[imax] - y[imax+1])
groundt = t[imax] + (groundx - x[imax])/(x[imax + 1] - groundx) * dt
def shoot(tfinal,dt,force):
   return groundx, groundt
```

(Note this is only an approximation of groundx, as the path of the trajectory follows a curve, not a line.)

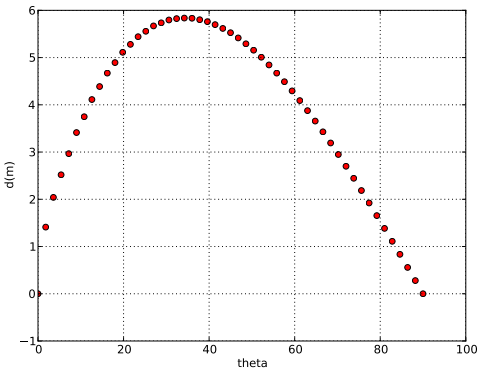
C. Calculating the Optimal Angle

Without air resistance, the angle at which a thrown object travels farthest is 45º. We will now calculate the optimal angle of a trajectory with air resistance.

First, we alter our initial conditions vx and vy so they are dependent on the initial velocity (10) and theta:

```
vx[0] = 10*math.cos(theta)
vy[0] = 10*math.sin(theta)
```

Since we have already know groundx, we can plot (theta, groundx). (See Fig. 5)



In Fig. 5, the angle that yields the greatest distance is the maximum of the graph. We see that the optimal angle is around 35º.

We will use the golden ratio to find the exact maximum of the graph.

Fig. 5 Graph of (theta, groundx) given alpha = 0.9

We take three points on Fig. 5: (a, fa), (b, fb), and (c, fc) where f is the 'shoot' function. At first, a = 0º, c = 90º, and b = (1-GOLD)a + (GOLD)c (where GOLD is the ratio (3-sqrt[5])/2). Think of the graph as divided into two sections – the first is from a to b, and the second from b to c.

We compare the two sides. Where is the maximum located? We tell the computer that if fb is higher than the midpoint between a and c, to redefine b as a. If fb is lower than the midpoint, to redefine b as c. Our new b is described by the same equation. The process continues until b equals the midpoint between and c, which is the maximum.

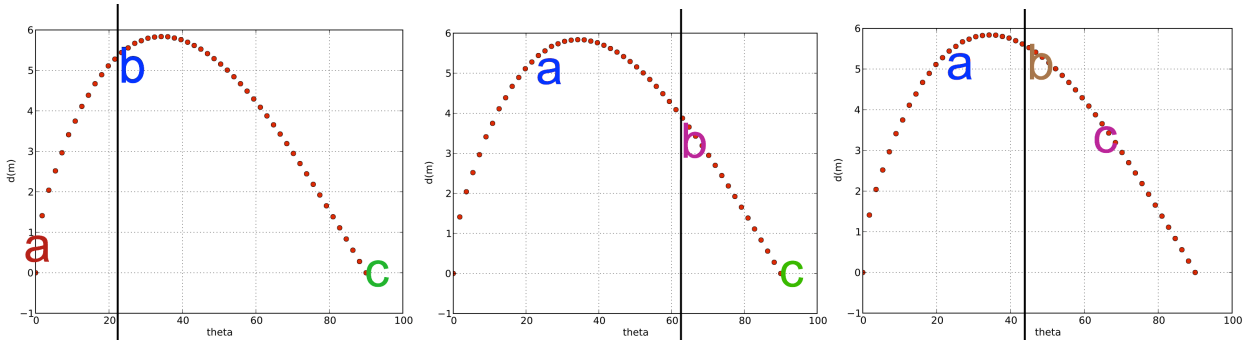


Fig. 6 Narrowing down the maximum. The optimal angle is the last value found for b.

```
GOLD = 0.5*(3.-math.sqrt(5.))
a = 0.
c = 0.5*math.pi
b = (1.-GOLD)*a + GOLD*c

fa = shoot(a)[0]
fb = shoot(b)[0]
fc = shoot(c)[0]

while (fb - 0.5*(fa+fc) > 1e-6):
  if ( (b-a) > (c-b) ):
     d = (1.-GOLD)*b + GOLD*a
     fd = shoot(d)[0]
     if (fd < fb):
        a = d; fa = fd
     else:
        c = b; fc = fb
        b = d; fb = fd
  else:
     d = (1.-GOLD)*b + GOLD*c
     fd = shoot(d)[0]
     if (fd < fb):
        c = d; fc = fd
     else:
        a = b; fa = fb
        b = d; fb = fd

groundx, groundt = shoot(b)
```

Note: d is used as a placeholder for the new b, as two different b's cannot be defined at the same time.

Part III: Plotting the Motion of Pendulums

A. The Simple Pendulum

For a straightforward pendulum, we will use theta, omega, and alpha in place of x and y, vx and vy, and force, respectively. Instead of the 'force' function, we will use 'a' (alpha), where 'a' is derived from $\tau=I\alpha$, and I is the inertia of a hollow sphere:

```
def a(theta):
  return -3*g/(2*L)*numpy.sin(theta)
```

We also need to define the initial theta using

$$\theta = \theta_0 + \omega_0 t + 0.5*\alpha t^2$$

```
theta[0] = theta[1] - dt * omega[1] + dt * dt * a(theta[1])/2
```

Using Verlet method, we now define theta, omega, and alpha:

```
for i in range(1,nstep):
    theta[i+1] = 2 * theta[i] - theta[i-1] + dt * dt * alpha[i]
    omega[i] = (theta[i+1] - theta[i-1])/(2 * dt)

    alpha[i+1] = a(theta[i+1])
```

However, this method does not provide for omega[0] and the final omega[nstep], which we need to define ourselves:

```
omega[0] = (4*theta[1]-theta[2]-3*theta[0])/(2 * dt)
omega[nstep] = (4*theta[nstep-1]-theta[nstep-2]-3*theta[nstep])/(2 * dt)
```

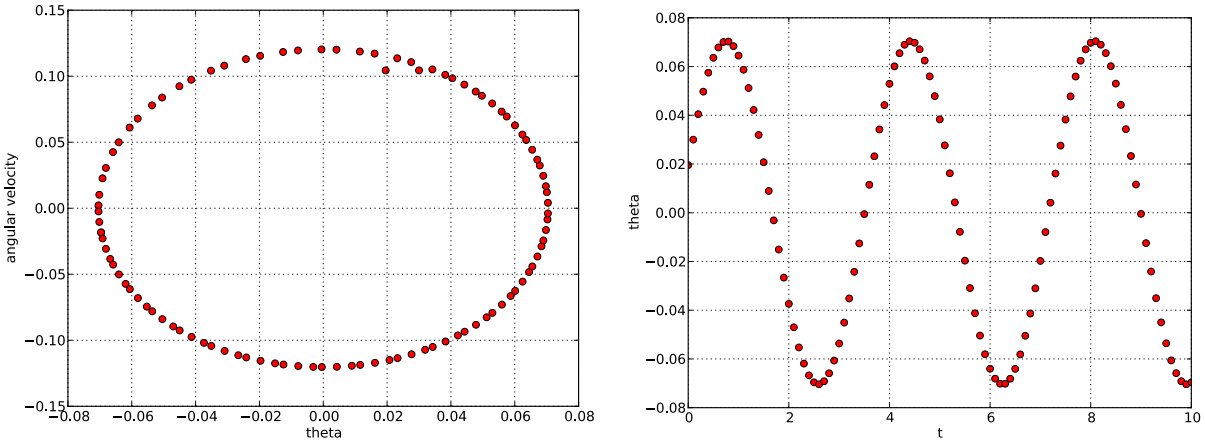


Fig. 7-8 Plotting the simple pendulum. On the left, theta vs angular velocity. On the right, time vs theta.

B. The Double Pendulum

For a double pendulum, we now have two sets of theta, omega, and alpha (see Fig 1). (We must also define two masses and two lengths.) First we must derive 'a1' and 'a2' using sum of the forces:

$$m_1 a_1 = -T_1 \sin\theta_1 + T_2 \sin\theta_2$$
$$m_1 a_1 = T_1 \cos\theta_1 - T_2 \cos\theta_2 - m_1 g$$
$$m_2 a_2 = -T_2 \sin\theta_2$$
$$m_2 a_2 = T_2 \cos\theta_2 - m_2 g$$

Solving for a_1 and a_2, we find:

```
def a1(theta1, theta2, omega1, omega2):
    return (-g*(2*m1+m2)*sin(theta1) - m2*g*sin(theta1-2*theta2) - 2*sin(theta1-theta2)
*m2*(omega1**2*L2+omega1**2*L1*cos(theta1-theta2)))/(L1*(2*m1+m2-m2*cos
(2*theta1-2*theta2)))

def a2(theta1, theta2, omega1, omega2):
    return (2*sin(theta1-theta2)*(omega1**2*L1*(m1+m2)+g*(m1+m2)*cos(theta1) +
omega2**2*L2*m2*cos(theta1-theta2)))/(L2*(2*m1+m2-m2*cos(2*theta1-2*theta2)))
```

a1 and a2 can also be found using sum of the torques. We also need to define the initial thetas:

```
theta1[0] = theta1[1] - dt * omega1[1] + dt * dt * a1(theta1[1], theta2[1], omega1[1],
omega2[1])/2
theta2[0] = theta2[1] - dt * omega2[1] + dt * dt * a2(theta1[1], theta2[1], omega1[1],
omega2[1])/2
```

Now we program theta1, theta2, omega1, omega2, alpha1, and alpha2 similarly to how we programed theta, omega, and alpha in the simple pendulum program:

```
for i in range(1,nstep):
    theta1[i+1] = 2 * theta1[i] - theta1[i-1] + dt * dt * alpha1[i]
    omega1[i] = (theta1[i+1] - theta1[i-1])/(2 * dt)

    alpha1[i+1] = a1(theta1[i+1], theta2[i+1], omega1[i+1], omega2[i+1])

    theta2[i+1] = 2 * theta2[i] - theta2[i-1] + dt * dt * alpha2[i]
    omega2[i] = (theta2[i+1] - theta2[i-1])/(2 * dt)

    alpha2[i+1] = a2(theta1[i+1], theta2[i+1], omega1[i+1], omega2[i+1])
```

Finally, the first and last values of omega1 and omega2 are not defined above, so:

```
omega1[0] = (4*theta1[1]-theta1[2]-3*theta1[0])/(2 * dt)
omega1[nstep] = (4*theta1[nstep-1]-theta1[nstep-2]-3*theta1[nstep])/(2 * dt)

omega2[0] = (4*theta2[1]-theta2[2]-3*theta2[0])/(2 * dt)
omega2[nstep] = (4*theta2[nstep-1]-theta2[nstep-2]-3*theta2[nstep])/(2 * dt)
```

When plotting the double pendulum, say we want to plot the motion of m1 and m2. We would first need to convert theta1 and theta2 into Cartesian coordinates:

```
def cartesian(theta1,theta2):
    return L1*numpy.sin(theta1),\
          -L1*numpy.cos(theta1),\
           L1*numpy.sin(theta1)+L2*numpy.sin(theta2),\
          -L1*numpy.cos(theta2)-L2*numpy.cos(theta2)
```

We would be plotting the paths of (x1, y1) and (x2, y2) on the same graph:

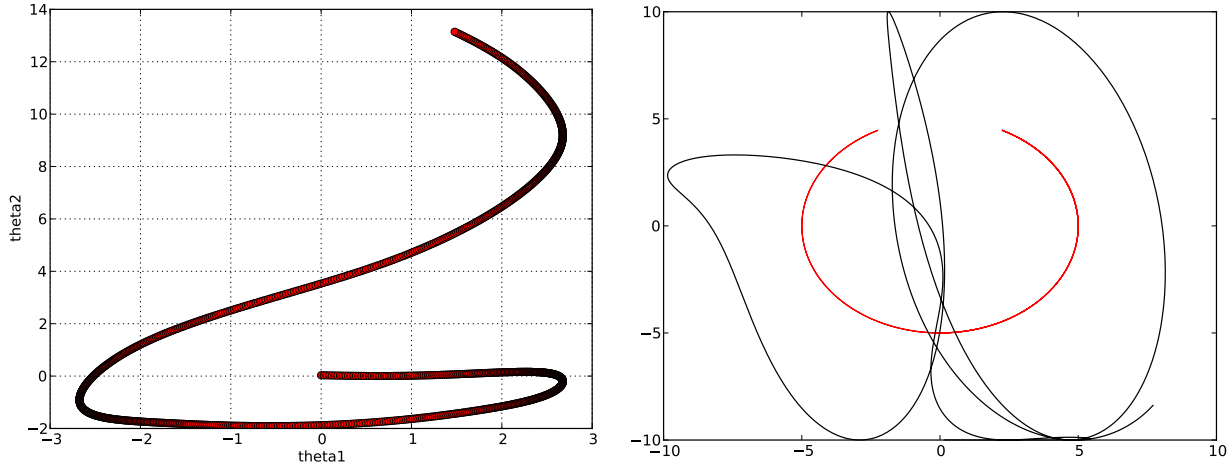


Fig. 9-10 Plotting the double pendulum. On the left, theta1 vs theta2. On the right, the paths of m1 (red) and m2 (black).

Conclusion

Using only Newton's second law, we were able to predict the movement of a chaotic system. Real-world application of this concept allows us to plot the motion of any particle if the initial conditions are known – some applications include testing the movement of atoms in collisions and under stress.