A Numerical Approach to the Dynamics of a Harmonic Rotating System

Siddharth Yajaman Soumil Roychowdhary

July 12, 2020

1 Introduction

In this paper, we will be exploring the motion of a system wherein two particles joined together by a spring are rotating about their center of mass. This system is then launched as a projectile. We do not think about the motion that led up to the throw, or how one might achieve such a motion. Rather, we start our analysis from the moment where the system has started moving as a projectile. We also make the assumption that the motion begins with the two particles having no velocity about the length of the spring[1].

Since the spring force is an internal force for the system, we expect the center of mass of the system to follow a parabolic trajectory. The particles should revolve about the center of mass during this motion. We therefore expect the resulting motion of each particle to somewhat resemble a disfigured parabola.

We first take a theoretical approach in solving this problem, by constructing Newton's equations for the system, and arriving at the ODEs for the resulting motion. After that, we take a numerical approach, and try to model the trajectory of the particles computationally.

[1]This means that at the start of the motion the system is at one of the two possible extreme states.

2 Theoretical formulation of the problem

In this section we shall take a look at how to formulate the relevant differential equations of motion for the system in question using Newton's laws of motion.

We consider a system of two particles, of masses m_1 and m_2, connected by a spring of spring constant k and natural length l_N. The formulation of the ODEs will be done in a Cartesian coordinate system.

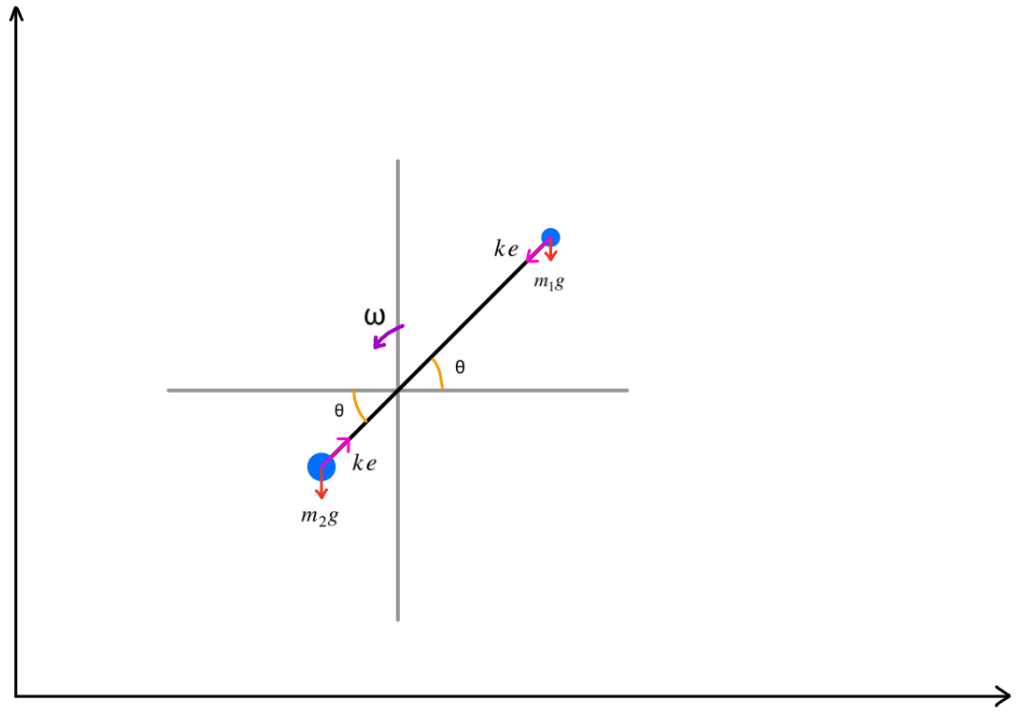

Figure 1: Free Body Diagram of the System

If (x_1, y_1) and (x_2, y_2) be the coordinates of the two masses at some time t, then the extension e of the spring is given by,

$$e = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - l_N \tag{1}$$

Since the distance of the particles from the center of mass changes with time, and there is no net external torque on the system, we must take care to conserve the angular momentum of the system.

We first note that the position of the center of mass is given by

$$(x_c, y_c) = (\frac{m_1 x_1 + m_2 x_2}{m_1 + m_2}, \frac{m_1 y_1 + m_2 y_2}{m_1 + m_2}),$$

Then the distances of the particles from the center of mass are,

$$r_1 = \frac{m_2}{m_1 + m_2}\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

$$r_2 = \frac{m_1}{m_1 + m_2}\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Keep in mind that at $t = 0$, $\theta = 0$, and angular velocity is given by ω_0, while at $t = t$, $\theta = \theta$ and angular velocity is given by ω_t.

At $t = 0$, let the moment of inertia of the system be I_0 and the length of the spring be l_i. At any later time t, let it be I_t such that,

$$I_0 = \frac{m_1 m_2}{m_1 + m_2} l_i^2$$

$$I_t = \frac{m_1 m_2}{m_1 + m_2}((x_1 - x_2)^2 + (y_1 - y_2)^2)$$

Since angular momentum is conserved,

$$I_0 \omega_0 = I_t \omega_t$$

$$\omega_t = \frac{\omega_0 l_i^2}{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2}$$

Now we see that there are only two external forces on each particle, i.e., the gravitational force[2] acting vertically downwards, and the spring force, acting along the length of the spring. So, their motion results from the vector sum of these forces[3].

Taking the components of the forces on particle 1 along the X direction:

$$m_1 \ddot{x}_1 = -ke\cos\theta$$

Using the same logic for both particles along the X and Y directions, we have the four equations:

$$m_1 \ddot{x}_1 = -ke\cos\theta \tag{3}$$
$$m_1 \ddot{y}_1 = -ke\sin\theta - m_1 g \tag{4}$$
$$m_2 \ddot{x}_2 = ke\cos\theta \tag{5}$$
$$m_2 \ddot{y}_2 = ke\sin\theta - m_2 g \tag{6}$$

Thus, it should be possible to completely describe the motion of our particles using the equations (1) to (6),

$$\begin{aligned} e &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - l_N \\ m_1 \ddot{x}_1 &= -ke\cos\theta \\ m_1 \ddot{y}_1 &= -ke\sin\theta - m_1 g \\ m_2 \ddot{x}_2 &= ke\cos\theta \\ m_2 \ddot{y} &= ke\sin\theta - m_2 g \\ \dot{\theta} = \omega_t &= \frac{\omega_0 l_i^2}{(x_1 - x_2)^2 + (y_1 - y_2)^2} \end{aligned}$$

However, in practice it is very difficult to find the solutions to the above system of equations analytically. We therefore look to solve this problem numerically in the following sections.

[2]We assume g is constant throughout.

[3]The initial conditions of the system play a big role as well.

3 Reformulation of the problem

In this section, we shall see how we can simplify the results obtained in the previous section in order to make them easier to solve numerically. As in the previous section we assume that g is constant throughout. Taking a look at our system, we note that there is no external torque[4]. The absence of an external torque turns out to be extremely helpful. What it means is that the rotation of the system is invariant under translation in both space and time. This allows us, in essence, to separate the overall motion into the non-inertial rotational motion of the system and the inertial projectile motion of the centre of mass and obtain two sets of differential equations which can be solved independently.

We first turn our attention to the rotation of the particles. Since there is no linear motion, a very natural choice of coordinates is a polar coordinate system. We approach this part of the problem using the reduced-mass of the system. The reduced-mass approach allows us to treat the system as one particle rotating about the other rather than the entire system rotating about the centre of mass. This will later on allow us to deal with fewer variables and differential equations.

The reduced-mass of the system is given by:

$$\mu = \frac{m_1 m_2}{m_1 + m_2}$$

We can treat the reduced-mass the way we would treat any regular mass. We let the position of the reduced-mass be given by (r, θ).

We shall use the following modified[5] result from our previous section:

$$\omega_t = \frac{l_i^2}{r^2}\omega_0$$

We also redefine the extension of the string (equivalently) as:

$$e = r - l_N$$

[4]The gravitational acceleration is the same for both particles.

[5]We have converted it from Cartesian to polar coordinates. One can check that the result is dimensionally correct.

We proceed to write out the set of differential equations[6] that define the rotational motion:

$$\ddot{r} = r\omega_t^2 - \frac{k}{\mu}(r - l_N) \tag{7}$$

$$\dot{\theta} = \omega_t \tag{8}$$

To deal with the projectile motion, we would like to have our rotational motion in the centre-of-mass frame of reference. We do this by a simple set of transformations with the positions of our two masses given by (r_1, θ_1) and (r_2, θ_2), respectively.

$$r_1 = \frac{\mu}{m_1} r$$

$$r_2 = \frac{\mu}{m_2} r$$

$$\theta_1 = \theta$$

$$\theta_2 = \pi + \theta$$

Our next task is to write out the differential equations for the projectile motion. These are textbook equations and they will not be derived here. Assuming an initial velocity v and an angle of projection ϕ, our equations are as follows:

$$\dot{x}_c = v\cos\phi \tag{9}$$

$$\dot{y}_c = v\sin\phi - gt \tag{10}$$

Whilst r_1, θ_1, r_2 and θ_2 are unknown, it does not stop us from seeing that at any time t,

$$x_1 = x_c + r_1\cos\theta_1 \tag{11}$$

$$y_1 = y_c + r_1\sin\theta_1 \tag{12}$$

$$x_2 = x_c + r_2\cos\theta_2 \tag{13}$$

$$y_2 = y_c + r_2\sin\theta_2 \tag{14}$$

Over here, (x_1, y_1) and (x_2, y_2) are the same Cartesian coordinates that were specified in the previous section. In the next section, we proceed to actually solve the two systems of differential equations we obtained in this section.

[6]One shall note that there is an additional centrifugal term in the equations. This arises from being in a non-inertial reference frame.

4 The Numerical Solution

With our differential equations set up, we are now in a position to solve them numerically. Let us start with the rotational motion of our system. We shall rewrite the relevant equations from previous sections:

$$\omega_t = \frac{l_i^2}{r^2}\omega_0$$

$$\ddot{r} = r\omega_t^2 - \frac{k}{\mu}(r - l_N)$$

$$\dot{\theta} = \omega_t$$

Our next step is to rewrite the second-order ODEs as a set of two first-order ODEs. We do this by letting $\dot{r} = u$. Then, in matrix form our system becomes:

$$\frac{d}{dt}\begin{bmatrix} u \\ r \\ \theta \end{bmatrix} = \begin{bmatrix} r\omega_t^2 - \frac{k}{\mu}(r - l_N) \\ u \\ \omega_t \end{bmatrix}$$

With initial conditions,
$u(0) = 0$, i.e., no initial radial motion along the spring.

$r(0) = l_i$

$\theta(0) = 0$

To solve this system, we construct a fourth-order Runge-Kutta method[7] in Python.

[7]Runge-Kutta methods are a popular family of methods to solve initial-value ODEs. This method has been chosen over a method like the Euler method as it has a faster convergence. You can read more in a book or on the web.

Over here, we will enumerate a few test cases and see what kinds of plots they generate.
Let us consider a system with the following properties[8]:
$l_N = 2$
$l_i = 2$
$m_1 = 1$
$m_2 = 1$
$k = 5$
$\omega_0 = 2\pi$
$T = 5$, where T is the duration of the plot.
$dt = 10^{-5}$, where dt is the step size of our RK method.

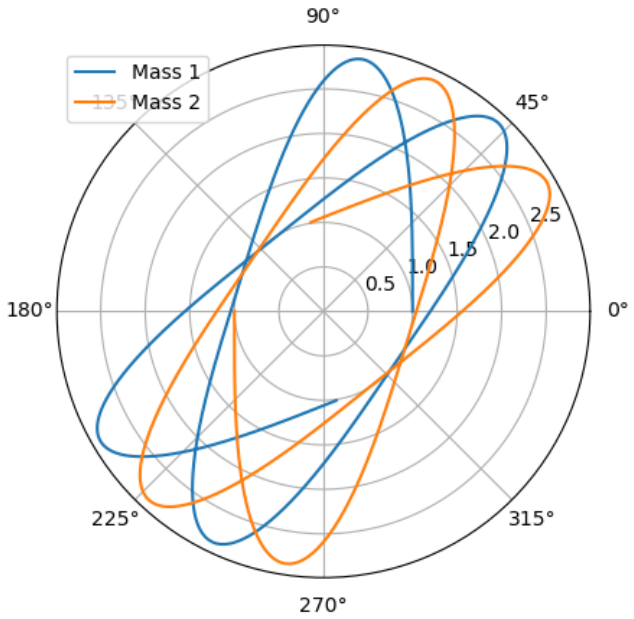

Figure 2: Plot for the given data

From the above plot (refer to Figure 2) itself, it is hard to make out the exact path that the two particles take in time[9]. However, there are a few things we can infer from the plot. Firstly, given that the masses of the two particles

[8]Units can be assumed to be SI.
[9]For completeness, the code will be included at the end. The simulation can be run in a software like Jupyter.

are equal, we would expect their motion to be identical albeit diametrically opposite. This is something which is visible in the plot. Secondly, we would expect that when the particles are further away from the centre, they would move slower due to the conservation of angular momentum. This manifests itself in our plot in terms of the smaller displacements of the particles at farther radial distances.
Let us next consider a system in which the two masses are unequal and the natural spring length is zero but the rest of the properties are identical to that of the previous case. Here, $m_1 = 1$, $m_2 = 5$ and $l_N = 0$.

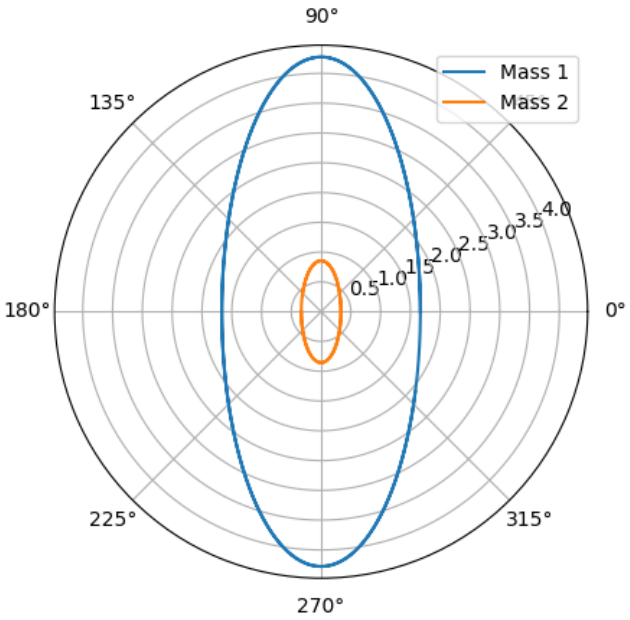

Figure 3: Plot for the updated data

As with the previous plot, the best we can do is to see if it makes physical sense. There are, however, a few stark differences. Let us try to analyse that aspect of it (refer to Figure 3). When we have a system of two particles with one much more massive than the other, it often appears that the lighter particle "orbits" the heavier one, when in reality they both orbit a

common point. An example would be the Sun-Jupiter[10] system or Bohr's model of the atom. One may observe in the plot that it is almost like the blue particle is "orbiting" the orange one. This agrees with what one sees in many classical systems. The more noteworthy aspect would be the fact that both particles have a perfectly elliptical trajectory with no orbital precession unlike the previous case. This is not a result of the change in mass. Rather, it is a result of the natural spring length being zero. This is a well known result and gives us a good indication that our model is correct.

Now that we have analysed the purely rotational aspect of the motion, we can look to extend it to a projectile path. The first step is to solve equations (11) and (12) from the previous section. These equations have analytical solutions which are given by:

$$x_c = v\cos\phi t$$

$$y_c = v\sin\phi t - \frac{1}{2}gt^2$$

We also know that the time of flight of a projectile is given by $T = \dfrac{2v\sin\phi}{g}$. We round this down to the nearest integer.

We generate our list of x_c and y_c by evaluating each function on a list of time values ranging from 0 to T at intervals of dt so that it corresponds with the numerical solution of the harmonic-rotatory motion[11]. Using equations (13) through (16) we get the desired values for x_1, y_1, x_2 and y_2.

[10] They orbit a point just outside the surface of the sun.

[11] Note that we do not individually check whether the particles have reached the ground. We use the time of flight of the centre of mass as an approximate time.

Let us try to understand the motion better by plotting graphs corresponding to our previous two test cases. We shall take $v = 40$ and $\phi = 60°$. We also take $l_N = l_i = 5$ so that the plot is more clear. We first consider the case when the two masses are equal (refer to Figure 4).

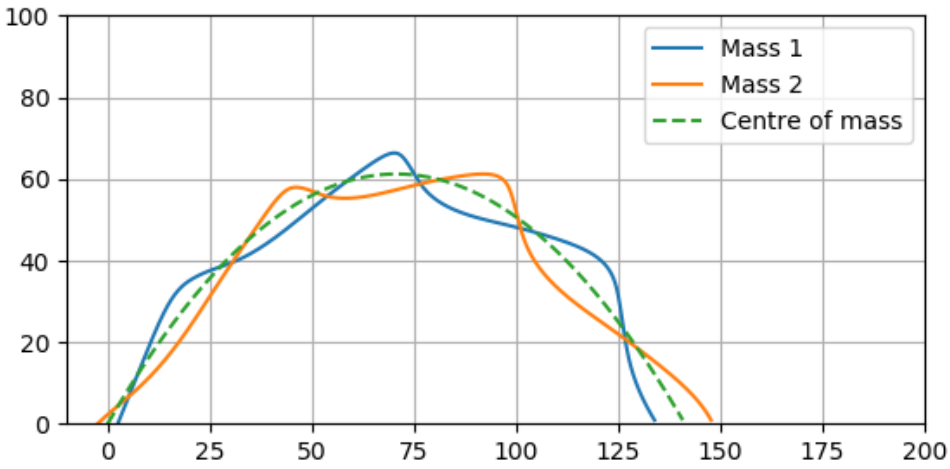

Figure 4: Plot for a projectile with equal masses

At the start of the paper, we had predicted that the path of the particles would be in the form of a disfigured parabola. And indeed, that is what we see in the plot. In addition, since the two particles are of equal mass, we expect each path to have the same amount of disfigurement which is apparent in the figure. We also see the paths of the two masses criss-cross which indicates some kind of oscillatory motion. This is to be expected.

Lastly, we consider the case when the two masses are unequal and the natural length is zero. There are a few things we expect to differ from the previous case. Firstly, we expect the heavier particle to trace a path which is much closer to a parabola than in the previous case. This is because the centre of mass lies much closer to the heavier particle. Secondly, we expect the lighter particle's oscillations about the heavier particle to be more pronounced.

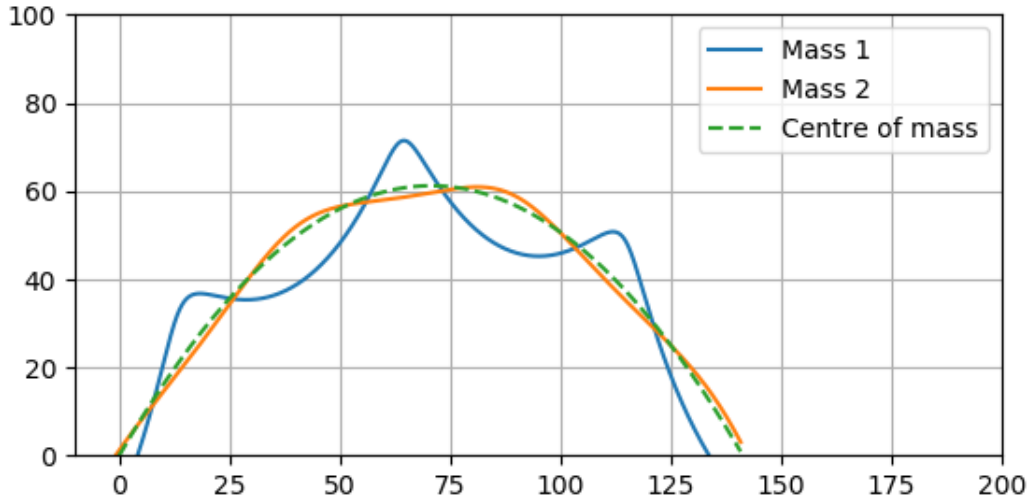

Figure 5: Plot for a projectile with unequal masses

On inspection of the plot (refer to Figure 5) this is exactly what we see.

To recap, we started by formulating the differential equations with our predictions of what the trajectory might look like. We approached the problem analytically. However, our analytical approach proved insufficient. We then broke our problem into parts and solved the problem numerically. Finally, we checked our results to see if they were close to what we expected and also whether they were realistic or not. On the face of it, the problem seemed highly complex. Using the appropriate approach, in this case numerical methods, we were able to model the trajectory without doing any kind of experiment. Given that we built our approach starting with basic principles, we also expect that our model is an accurate one.

A Code for Elastic Circular Motion

Make sure you have the numpy and matplotlib packages installed on your Python interpreter.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim


def ot(il, r, omega):
    return il**2/r**2*omega


def dudt(r, mu, k, nl, o):
    return r*o**2-k/mu*(r - nl)


def drdt(u):
    return u


def solve_ODE(omega, k, mu, nl, il, T, dt):
    n = int(T/dt)
    t = np.linspace(0, T, n+1)
    r = np.empty(n + 1)
    u = np.empty(n + 1)
    theta = np.empty(n + 1)
    o = np.empty(n + 1)
    KE = np.empty(n + 1)
    PE = np.empty(n + 1)
    r[0] = il
    u[0] = 0
    theta[0] = 0
    o[0] = omega
    for i in range(n):
        o1 = ot(il, r[i], omega)*dt
        u1 = dudt(r[i], mu, k, nl, o[i])*dt
```

```
        r1 = drdt(u[i])*dt
        o2 = ot(il, r[i] + r1*0.5, omega)*dt
        u2 = dudt(r[i] + r1*0.5, mu, k, nl, o[i] + o1*0.5)*dt
        r2 = drdt(u[i] + u1*0.5)*dt
        o3 = ot(il, r[i] + r2 * 0.5, omega) * dt
        u3 = dudt(r[i] + r2 * 0.5, mu, k, nl, o[i] + o2 * 0.5) * dt
        r3 = drdt(u[i] + u2 * 0.5) * dt
        o4 = ot(il, r[i] + r3, omega) * dt
        u4 = dudt(r[i] + r3, mu, k, nl, o[i] + o3) * dt
        r4 = drdt(u[i] + u3) * dt
        u[i + 1] = u[i] + 1.0/6.0 * (u1 + 2 * u2 + 2 * u3 + u4)
        r[i + 1] = r[i] + 1.0/6.0 * (r1 + 2 * r2 + 2 * r3 + r4)
        theta[i + 1] = theta[i] + 1.0/6.0 * (o1 + 2 * o2 + 2 * o3 + o4)
        o[i + 1] = 1.0/6.0 * (o1 + 2 * o2 + 2 * o3 + o4) / dt
    return r, theta


v = 40
phi = np.pi/3
omega = 2*np.pi
k = 5.0
m1 = 1.0
m2 = 5.0
nl = 0
il = 5.0
dt = 1e-4
T=5
mu = m1*m2/(m1+m2)
r, theta1 = solve_ODE(omega, k, mu, nl, il, T, dt)
theta2 = theta1 + np.pi
r1 = mu/m1 * r
r2 = mu/m2 * r
x1 = r1 * np.cos(theta1)
y1 = r1 * np.sin(theta1)
x2 = r2 * np.cos(theta2)
y2 = r2 * np.sin(theta2)
plt.style.use('dark_background')
fig = plt.figure()
```

```
ax = fig.add_subplot(111, autoscale_on=False, xlim=[-20, 20], ylim=[-20, 20])
ax.set_aspect('equal')
ax.grid()
line1, = ax.plot([], [], 'o-', lw=1)
line2, = ax.plot([], [], 'o-', lw=1)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9,' ', transform=ax.transAxes)


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


def animate(i):
    line1.set_data([x1[i], x2[i]], [y1[i], y2[i]])
    time_text.set_text(time_template % (i * dt))
    return line1, time_text


ani = anim.FuncAnimation(fig, animate, range(0, len(x1), 100),interval=10, blit=Tr
plt.show()
```

B Code for Projectile Elastic Circular Motion

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.animation as anim


def dudt(r1, r2, m, k, l, o):
    return (r1 * o ** 2) - k / m * (r1 + r2 - l)


def drdt(u):
    return u


def solve_ODE_RK4(omega, k, m1, m2, l, T, dt):
    n = int(T/dt)
    t = np.linspace(0, T, n+1)
    r1 = np.empty(n + 1)
    r2 = np.empty(n + 1)
    u1 = np.empty(n + 1)
    u2 = np.empty(n + 1)
    theta = np.empty(n + 1)
    r1[0] = 2 * m2 * 1 / (m1 + m2)
    u1[0] = 0
    r2[0] = 2 * m1 * 1 / (m1 + m2)
    u2[0] = 0
    for i in range(n):
        o = m1 * m2 / (m1 + m2) * ((r1[0] + r2[0]) ** 2) / (m1 * (r1[i] ** 2) + m2
        u1k1 = dudt(r1[i], r2[i], m1, k, l, o) * dt
        u2k1 = dudt(r2[i], r1[i], m2, k, l, o) * dt
        r1k1 = drdt(u1[i]) * dt
        r2k1 = drdt(u2[i]) * dt
        u1k2 = dudt(r1[i] + r1k1 * 0.5, r2[i] + r2k1 * 0.5, m1, k, l, o) * dt
        u2k2 = dudt(r2[i] + r2k1 * 0.5, r1[i] + r1k1 * 0.5, m2, k, l, o) * dt
```

```
        r1k2 = drdt(u1[i] + u1k1 * 0.5) * dt
        r2k2 = drdt(u2[i] + u2k1 * 0.5) * dt
        u1k3 = dudt(r1[i] + r1k2 * 0.5, r2[i] + r2k2 * 0.5, m1, k, l, o) * dt
        u2k3 = dudt(r2[i] + r2k2 * 0.5, r1[i] + r1k2 * 0.5, m2, k, l, o) * dt
        r1k3 = drdt(u1[i] + u1k2 * 0.5) * dt
        r2k3 = drdt(u2[i] + u2k2 * 0.5) * dt
        u1k4 = dudt(r1[i] + r1k3, r2[i] + r2k3, m1, k, l, o) * dt
        u2k4 = dudt(r2[i] + r2k3, r1[i] + r1k3, m2, k, l, o) * dt
        r1k4 = drdt(u1[i] + u1k3) * dt
        r2k4 = drdt(u2[i] + u2k3) * dt
        u1[i + 1] = u1[i] + 1.0 / 6.0 * (u1k1 + 2 * u1k2 + 2 * u1k3 + u1k4)
        u2[i + 1] = u2[i] + 1.0 / 6.0 * (u2k1 + 2 * u2k2 + 2 * u2k3 + u2k4)
        r1[i + 1] = r1[i] + 1.0 / 6.0 * (r1k1 + 2 * r1k2 + 2 * r1k3 + r1k4)
        r2[i + 1] = r2[i] + 1.0 / 6.0 * (r2k1 + 2 * r2k2 + 2 * r2k3 + r2k4)
        theta[i + 1] = theta[i] + o * dt
        if theta[i + 1] >= 2 * np.pi:
            theta[i + 1] -= 2 * np.pi
    return r1, r2, theta


def solve_COM(v, phi, T, dt, g=9.8):
    num_steps = int(T /dt)
    x = np.empty(num_steps+1)
    y = np.empty(num_steps+1)
    vy = np.empty(num_steps+1)
    x[0] = 0
    y[0] = 0
    vy[0] = v*np.sin(phi)
    for i in range(num_steps):
        x[i+1] = x[i] + v*np.cos(phi)*dt
        vyh = vy[i] - g*dt/2
        vy[i+1] = vy[i] - g*dt
        y[i+1] = y[i] + vyh*dt
    return x, y


omega = 2 * np.pi
k = 5
```

```
m1 = 1
m2 = 5
l = 0
dt = 1e-4
v = 40
phi = np.pi/3
T = int(2*v*np.sin(phi)/9.8)
r1, r2, theta1 = solve_ODE_RK4(omega, k, m1, m2, l, T, dt)
x, y = solve_COM(v, phi, T, dt)
theta2 = theta1 + np.pi
x1 = x + r1 * np.cos(theta1)
y1 = y + r1 * np.sin(theta1)
x2 = x + r2 * np.cos(theta2)
y2 = y + r2 * np.sin(theta2)


fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=[-10, 200], ylim=[0, 100])
ax.set_aspect('equal')
ax.grid()


line1, = ax.plot([], [], 'o-', lw=1)
line2, = ax.plot([], [], '-', lw=1, label="Mass 1")
line3, = ax.plot([], [], '-', lw=1, label="Mass 2")
line4, = ax.plot([], [], '--', lw=1, label="Centre of mass")
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, ' ', transform=ax.transAxes)


def init():
    line1.set_data([], [])
    line2.set_data([], [])
    line3.set_data([], [])
    line4.set_data([], [])
    return line1, line2, line3, line4
```

```
def animate(i):
    line1.set_data([x1[i], x2[i]], [y1[i], y2[i]])
    line2.set_data(x1[:i], y1[:i])
    line3.set_data(x2[:i], y2[:i])
    line4.set_data(x[:i], y[:i])
    time_text.set_text(time_template % (i*dt))
    return line1, line2, line3, line4, time_text


ani = anim.FuncAnimation(fig, animate, range(0, len(x1), 100), interval=10, blit=T
ax.legend()
plt.show()
```