(Classical) Simple Pendulum

Theoretical and numerical analysis

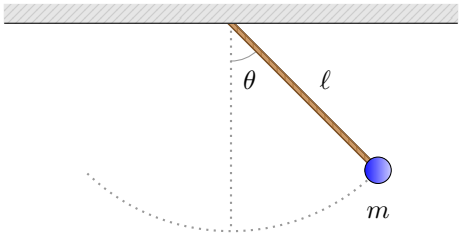
Edy Alberto **Flores Leal**
Ernesto **Guzmán Saleh**

B.S. in Engineering Physics
Instituto Tecnológico y de Estudios Superiores de Monterrey

1 Theoretical description

1.1 Review of the classical version of the simple pendulum

One of the most common systems studied in Mechanics is the *pendulum*. A (simple) pendulum consists of a mass attached to a rod from a pivot, oscillating under the influence of gravity. To illustrate this, see the following figure:



For this system, we could use Newton's or Lagrange's formalisms to obtain the equations of motion. Taking advantage of the latter, we get the kinetic and potential energies of the pendulum. First, the (x, y) coordinates are given by

$$\begin{aligned} x &= \ell \sin(\theta), \\ y &= \ell - \ell \cos(\theta). \end{aligned} \tag{1}$$

Therefore, the first time-derivative of x and y yields

$$\begin{aligned} \dot{x} &= \ell\dot{\theta} \cos(\theta), \\ \dot{y} &= \ell\dot{\theta} \sin(\theta). \end{aligned} \tag{2}$$

The kinetic energy is thus expressed as

$$T = \frac{1}{2}(\dot{x}^2 + \dot{y}^2) = \frac{1}{2}\ell^2\dot{\theta}. \tag{3}$$

On the other hand, the potential energy is only determined by the y coordinate,

$$U = mgy = mg\ell\left[1 - \cos(\theta)\right]. \tag{4}$$

Thus, the *Lagrangian* of the system is given by

$$\mathcal{L} = T - U = \frac{1}{2}\ell^2\dot{\theta}^2 - mg\ell\left[1 - \cos(\theta)\right]. \tag{5}$$

As we already know, knowing the Lagrangian, we could use the *Euler-Lagrange* equations:

$$\frac{\partial \mathcal{L}}{\partial q} - \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}}\right) = 0. \tag{6}$$

This situation is fairly easy because there is only one variable in the Lagrangian. Now, we have that

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta} &= -mg\ell \sin(\theta), \\ \frac{\partial \mathcal{L}}{\partial \dot{\theta}} &= \ell^2 \dot{\theta} \Rightarrow \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \dot{\theta}}\right) = \ell^2 \ddot{\theta}. \end{aligned} \tag{7}$$

Putting everything together, the Euler-Lagrange equation takes the following form:

$$\begin{aligned} -mg\ell \sin(\theta) - \ell^2 \ddot{\theta} &= 0 \\ \ddot{\theta} + \frac{mg}{\ell} \sin(\theta) &= 0. \end{aligned} \tag{8}$$

This is a *Nonlinear Second-Order Ordinary Differential Equation* (ODE), so we will require to exploit the benefits of numerical methods.

2 Numerical approach

2.1 Fourth-Order Runge-Kutta method

Because the equation of motion of the pendulum is a Nonlinear Second-Order ODE, it is convenient to think about the *Runge-Kutta method* (Tenenbaum and Pollard, 1985). In particular, we use the Fourth-Order Runge-Kutta method, which establishes that the approximation of $y(x_1)$ is

$$y(x_0 + h) = y(x_0) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{9}$$

where

$$k_1 = hf(x_0, y_0), \tag{10}$$

$$k_2 = hf\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_1\right), \tag{11}$$

$$k_3 = hf\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_2\right), \tag{12}$$

$$k_4 = hf(x_0 + h, y_0 + k_3). \tag{13}$$

This process yields an approximation of $y(x_{n+1})$ using $y(x_n)$ as input value. Now, recall equation (8). Let's introduce a new variable, let's say ω, which represents the *angular velocity*,

$$\dot{\theta} = \omega, \tag{14}$$

$$\dot{\omega} = \ddot{\theta} = -\frac{mg}{\ell} \sin(\theta). \tag{15}$$

We use the new variable because the Runge-Kutta method is valid for First-Order ODE, so we changed a Second-Order ODE into two First-Order ODE. Conversely, we will need to solve both equations. Let's say that equation (14) is $f(t, \theta, \omega) = \omega$ and equation (15) is $g(t, \theta, \omega) = -(mg/\ell)\sin(\theta)$. Hence, we need to

evaluate the following expressions:

$$
\begin{aligned}
k_{1,\theta} &= \Delta t f(t_0, \theta_0, \omega_0), & k_{1,\omega} &= \Delta t g(t_0, \theta_0, \omega_0),\\
k_{2,\theta} &= \Delta t f\left(t_0 + \frac{1}{2}\Delta t, \theta_0 + \frac{1}{2}k_{1,\theta}, \omega_0 + \frac{1}{2}k_{1,\omega}\right), & k_{2,\omega} &= \Delta t g\left(t_0 + \frac{1}{2}\Delta t, \theta_0 + \frac{1}{2}k_{1,\theta}, \omega_0 + \frac{1}{2}k_{1,\omega}\right),\\
k_{3,\theta} &= \Delta t f\left(t_0 + \frac{1}{2}\Delta t, \theta_0 + \frac{1}{2}k_{2,\theta}, \omega_0 + \frac{1}{2}k_{2,\omega}\right), & k_{3,\omega} &= \Delta t g\left(t_0 + \frac{1}{2}\Delta t, \theta_0 + \frac{1}{2}k_{2,\theta}, \omega_0 + \frac{1}{2}k_{2,\omega}\right),\\
k_{4,\theta} &= \Delta t f(t_0 + \Delta t, \theta_0 + k_{3,\theta}, \omega_0 + k_{3,\omega}), & k_{4,\omega} &= \Delta t g(t_0 + \Delta t, \theta_0 + k_{3,\theta}, \omega_0 + k_{3,\omega}).
\end{aligned}
$$

More explicitly, we need to evaluate these expressions:

$$
\begin{aligned}
k_{1,\theta} &= \Delta t \omega_0, & k_{1,\omega} &= -\frac{mg\Delta t}{\ell}\sin(\theta_0), && (16)\\
k_{2,\theta} &= \Delta t\left(\omega_0 + \frac{1}{2}k_{1,\omega}\omega_0\right), & k_{2,\omega} &= -\frac{mg\Delta t}{\ell}\sin\left(\theta_0 + \frac{1}{2}k_1\theta_0\right), && (17)\\
k_{3,\theta} &= \Delta t\left(\omega_0 + \frac{1}{2}k_{2,\omega}\omega_0\right), & k_{3,\omega} &= -\frac{mg\Delta t}{\ell}\sin\left(\theta_0 + \frac{1}{2}k_2\theta_0\right), && (18)\\
k_{4,\theta} &= \Delta t(\omega_0 + k_{3,\omega}) & k_{4,\omega} &= -\frac{mg\Delta t}{\ell}\sin\left(\theta_0 + k_{3,\theta}\right). && (19)
\end{aligned}
$$

Remember that we use these parameters to compute these approximations:

$$
\begin{aligned}
\theta_{n+1} &= \theta_n + \frac{1}{6}(k_{1,\theta} + 2k_{2,\theta} + 2k_{3,\theta} + k_{4,\theta}),\\
\omega_{n+1} &= \omega_n + \frac{1}{6}(k_{1,\omega} + 2k_{2,\omega} + 2k_{3,\omega} + k_{4,\omega})
\end{aligned}
\qquad (20)
$$

To solve this ODE, we require initial conditions. It is possible to explore the behavior of this system under different sets of these initial conditions. We start considering an initial angle and initial angular velocity of

$$
\theta(0) = \frac{\pi}{4}\,\text{rad}, \;\; \omega(0) = 1\,\frac{\text{rad}}{\text{s}}. \qquad (21)
$$

At this point, we have all the information needed to solve the problem. To do so, we will provide a detailed approach using different programming languages scripts to solve numerically the ODE.

2.2 Python's code

In this section, we provide a detailed description of the code used to simulate the (classical) simple pendulum. Even though it is possible to use solvers of differential equations, we write the whole algorithm as a matter of practice and illustrate its functioning. We begin by defining the parameters and initial conditions.

```
# Libraries
import numpy as np
import matplotlib.pyplot as plt

# Physical parameters
m = 1    # mass    (kg)
g = 9.81 # gravity (kg/m^2)
l = 1    # length  (m)

# Initial conditions
θ0 = np.pi/4 # initial angle            (rad)
ω0 = 3.0     # initial angular velocity (rad/s)
```

We will solve the ODE in a period from $t_0 = 0\,\text{s}$ to $t_\text{f} = 10\,\text{s}$ using a step size of $\Delta t = 0.01\,\text{s}$. We declare this as follows:

```
# Time parameters
t_0 = 0                         # initial time (s)
t_f = 10                        # final time   (s)
Δt = 0.01                       # step size    (s)
n = int((t_f - t_0) / Δt) + 1 # iterations
t = np.linspace(t_0, t_f, n)  # time array  (s)
```

It is relevant to mention that we define the time parameter as a `np.linspace` array. However, it can also be declared as an empty vector that gets updated in every iteration. In the end, this choice is a matter of preference. We now define the Fourth-Order Runge-Kutta method:

```
# Fourth-Order Runge-Kutta function
def RK4(f, x0, y0, z0, h):
    k1y = h * f(x0, y0, z0)[0]
    k1z = h * f(x0, y0, z0)[1]
    #
    k2y = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[0]
    k2z = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[1]
    #
    k3y = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[0]
    k3z = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[1]
    #
    k4y = h * f(x0 + h, y0 + k3y, z0 + k3z)[0]
    k4z = h * f(x0 + h, y0 + k3y, z0 + k3z)[1]

    # Approximation
    y1 = float(y0 + (k1y + 2 * k2y + 2 * k3y + k4y) / 6)
    z1 = float(z0 + (k1z + 2 * k2z + 2 * k3z + k4z) / 6)
    return y1, z1
```

This function takes the following inputs:

- `f`: functions to evaluate.
- `x0`: initial condition (independent variable).
- `y0`: initial condition (dependent variable).
- `z0`: initial condition (dependent variable).
- `h`: step size.

The next step is to define the functions.

```
# Differential equations
def f(t, θ, ω):
    return ω, -(m * g / l) * np.sin(θ)
```

That is, `f(t, θ, ω)` defines the equations

$$\dot{\theta} = \omega,$$
$$\dot{\omega} = -\frac{mg}{\ell}\sin(\theta).$$

We initialize our storage arrays:

```
# Initial arrays
θ, θ[0] = np.zeros(n), θ0
ω, ω[0] = np.zeros(n), ω0
```

In the previous code, we create an empty array of n elements and store the initial conditions as the first value. Finally, we perform the `for` cycle:

```
# RK4 method evaluation
for i in range(n - 1):
    θ[i + 1], ω[i + 1] = RK4(f, t[i], θ[i], ω[i], Δt)
```

If we plot t versus θ, we obtain the next plot:

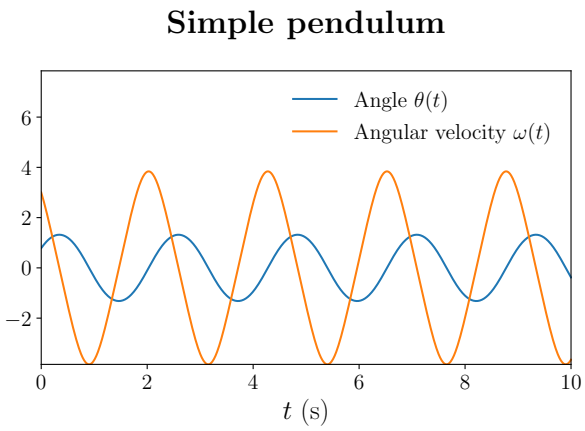


Figure 1: Evolution of the angle through time.

The full code is shown here:

```
# Libraries
import numpy as np
import matplotlib.pyplot as plt

# Physical parameters
m = 1    # mass    (kg)
g = 9.81 # gravity (kg/m^2)
l = 1    # length  (m)

# Initial conditions
θ0 = np.pi/4 # initial angle            (rad)
ω0 = 3.0     # initial angular velocity (rad/s)

# Time parameters
t_0 = 0                       # initial time (s)
t_f = 10                      # final time   (s)
Δt = 0.01                     # step size    (s)
n = int((t_f - t_0) / Δt) + 1 # iterations
```

```
t = np.linspace(t_0, t_f, n)  # time array   (s)

# Fourth-Order Runge-Kutta function
def RK4(f, x0, y0, z0, h):
    k1y = h * f(x0, y0, z0)[0]
    k1z = h * f(x0, y0, z0)[1]
    #
    k2y = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[0]
    k2z = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[1]
    #
    k3y = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[0]
    k3z = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[1]
    #
    k4y = h * f(x0 + h, y0 + k3y, z0 + k3z)[0]
    k4z = h * f(x0 + h, y0 + k3y, z0 + k3z)[1]

    # Approximation
    y1 = float(y0 + (k1y + 2 * k2y + 2 * k3y + k4y) / 6)
    z1 = float(z0 + (k1z + 2 * k2z + 2 * k3z + k4z) / 6)
    return y1, z1

# Differential equations
def f(t, θ, ω):
    return ω, -(m * g / l) * np.sin(θ)

# Initial arrays
θ, θ[0] = np.zeros(n), θ0
ω, ω[0] = np.zeros(n), ω0

# RK4 method evaluation
for i in range(n - 1):
    θ[i + 1], ω[i + 1] = RK4(f, t[i], θ[i], ω[i], Δt)

# Plot parameters
import matplotlib.pyplot as plt
plt.rcParams['text.usetex'] = True
plt.rcParams['font.family'] = 'serif'
plt.rcParams.update({'font.size': 16})
# Plot
plt.plot(t, θ, label = r'$\theta(t)$')
plt.title(r'Time evolution of $\theta$', y = 1.02, fontsize = 20)
plt.xlabel(r'$t$ (s)', fontsize = 20)
plt.ylabel(r'$\theta$ (rad)', fontsize = 20)
plt.legend(loc = 'upper right', frameon = False)
plt.xlim((min(t), max(t)))
plt.ylim((min(θ), max(θ) + 1))
plt.tight_layout()
plt.gcf()
plt.savefig("pyplot.pdf")
```

2.3 Julia's code

In this section, we provide a detailed description of the code to simulate the (classical) simple pendulum, but this time, using `Julia`. We expect to do the same for other programming languages. Once again, we write the whole Runge-Kutta method instead of using the ODE solvers available because we want to do it as a matter of practice. We establish the initial conditions:

```julia
# Libraries
using LinearAlgebra
using Plots, ColorSchemes, LaTeXStrings

# Physical parameters
m = 1    # mass    (kg)
g = 9.81 # gravity (kg/m^2)
l = 1    # length  (m)

# Initial conditions
θ0 = π/4
ω0 = 3.0 # initial angular
```

We use the same time consideration as in the `Python`'s script.

```julia
# Time parameters
t0 = 0                          # initial time (s)
t1 = 10                         # final time   (s)
Δt = 0.01                       # step size    (s)
n = Int64((t1 - t0) / Δt) + 1 # iterations
t = t0:Δt:(t1 - Δt)             # time array (s)
```

Next, we define the Runge-Kutta method using the previous equations.

```julia
# Fourth-Order Runge-Kutta function
function RK4(f, x0, y0, z0, h)
    k1y = h * f(x0, y0, z0)[1]
    k1z = h * f(x0, y0, z0)[2]
    #
    k2y = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[1]
    k2z = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[2]
    #
    k3y = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[1]
    k3z = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[2]
    #
    k4y = h * f(x0 + h, y0 + k3y, z0 + k3z)[1]
    k4z = h * f(x0 + h, y0 + k3y, z0 + k3z)[2]

    # Approximation
    y1 = Float64(y0 + (k1y + 2 * k2y + 2 * k3y + k4y) / 6)
    z1 = Float64(z0 + (k1z + 2 * k2z + 2 * k3z + k4z) / 6)
    return y1, z1
end
```

Once we declare the Runge-Kutta expressions, we can proceed to declare the function that contains the equations to solve:

```julia
# Differential equations
function f(t, θ, ω)
    return ω, -(m * g / l) * sin(θ)
end
```

We initialize the storage arrays:

```julia
# Initial arrays
θ, θ[1] = zeros(n), θ0
ω, ω[1] = zeros(n), ω0
```

Finally, we evaluate the Runge-Kutta method with the `for` cycle:

```julia
# RK4 method evaluation
for i in 1:(n - 1)
    θ[i + 1], ω[i + 1] = RK4(f, t[i], θ[i], ω[i], Δt)
end
```

Plotting the time t versus the angle $\theta(t)$ and angular velocity $\omega(t)$, we get the following plot:

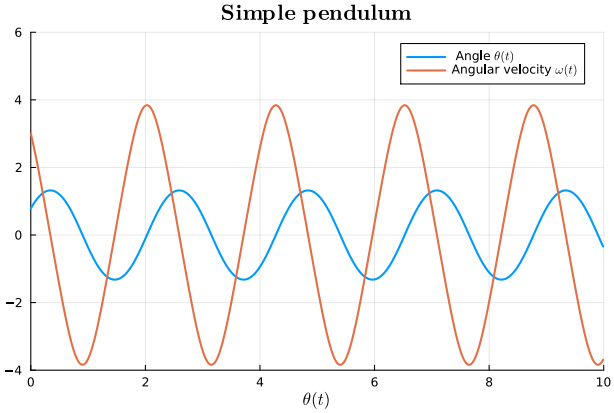


Figure 2: Evolution of the angle through time.

References

M Tenenbaum and H Pollard. *Ordinary Differential Equations*. Dover Books on Mathematics. Dover Publications, Mineola, NY, October 1985.