

(Classical) Simple Pendulum

Theoretical and numerical analysis

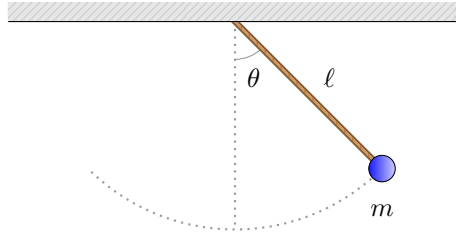
Authors: Edy Alberto Flores Leal
Ernesto Guzmán Saleh

[†] *Instituto Tecnológico y de Estudios Superiores de Monterrey*
Av. Eugenio Garza Sada 2501 Sur, Tecnológico, 64849 Monterrey, N.L.

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 (with negligible mass) from a pivot, oscillating under the influence of gravity (Yépez Mulia and Yépez Martínez, 2012). 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

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

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

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

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

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

Knowing the Lagrangian, we could use the *Euler-Lagrange* equations:

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

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

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

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

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

This is a *Nonlinear Second-Order Ordinary Differential Equation* (ODE), so we will require to exploit the benefits of numerical methods. However, if we consider a small angle, we see that

$$\sin(\theta) = \theta + O(\theta^3), \quad (9)$$

so the differential equation reduces to

$$\ddot{\theta} + \frac{g}{\ell} \theta = 0.$$

This ODE could be solved analytically. We will focus on the non-linear situation.

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

$$y(x_0 + h) = y(x_0) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad (10)$$

where

$$k_1 = hf(x_0, y_0), \quad (11)$$

$$k_2 = hf\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_1\right), \quad (12)$$

$$k_3 = hf\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_2\right), \quad (13)$$

$$k_4 = hf(x_0 + h, y_0 + k_3), \quad (14)$$

h being the step taken. 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*,

$$\dot{\theta} = \omega, \quad (15)$$

$$\dot{\omega} = \ddot{\theta} = -\frac{g}{\ell} \sin(\theta). \quad (16)$$

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 (15) is $f(t, \theta, \omega) = \omega$ and equation (16) is $g(t, \theta, \omega) = -(g/\ell) \sin(\theta)$. 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:

$$k_{1,\theta} = \Delta t \omega_0, \quad k_{1,\omega} = -\frac{g\Delta t}{\ell} \sin(\theta_0), \quad (17)$$

$$k_{2,\theta} = \Delta t \left(\omega_0 + \frac{1}{2}k_{1,\omega} \right), \quad k_{2,\omega} = -\frac{g\Delta t}{\ell} \sin\left(\theta_0 + \frac{1}{2}k_{1,\theta}\right), \quad (18)$$

$$k_{3,\theta} = \Delta t \left(\omega_0 + \frac{1}{2}k_{2,\omega} \right), \quad k_{3,\omega} = -\frac{g\Delta t}{\ell} \sin\left(\theta_0 + \frac{1}{2}k_{2,\theta}\right), \quad (19)$$

$$k_{4,\theta} = \Delta t(\omega_0 + k_{3,\omega}) \quad k_{4,\omega} = -\frac{g\Delta t}{\ell} \sin(\theta_0 + k_{3,\theta}). \quad (20)$$

Remember that we use these parameters to compute these approximations:

$$\begin{aligned} \theta_{n+1} &= \theta_n + \frac{1}{6}(k_{1,\theta} + 2k_{2,\theta} + 2k_{3,\theta} + k_{4,\theta}), \\ \omega_{n+1} &= \omega_n + \frac{1}{6}(k_{1,\omega} + 2k_{2,\omega} + 2k_{3,\omega} + k_{4,\omega}) \end{aligned} \quad (21)$$

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, but we will start considering an initial angle and initial angular velocity of

$$\theta(0) = \frac{\pi}{4} \text{ rad}, \quad \omega(0) = 3 \frac{\text{rad}}{\text{s}}. \quad (22)$$

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 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.

```

1  # Libraries
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Plot parameters
6  plt.rcParams.update({
7      'lines.linewidth': 2,      # linewidth
8      'text.usetex': True,      # LaTeX font

```

```

9     'font.family': 'serif',      # Serif family
10    'font.size': 16,             # font size
11    'axes.titlesize': 20,        # title size
12    'axes.grid': True,           # grid
13    'grid.linestyle': "-.",      # grid style
14 })
15
16 # Physical parameters
17 g = 9.81 # gravity (kg/m^2)
18 l = 1    # length (m)
19
20 # Initial conditions
21  $\theta_0 = \text{np.pi}/4$  # initial angle (rad)
22  $\omega_0 = 3.0$          # initial angular velocity (rad/s)

```

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

```

1 # Time parameters
2 t_0 = 0                # initial time (s)
3 t_f = 10               # final time (s)
4  $\Delta t = 0.01$          # step size (s)
5 n = int((t_f - t_0) /  $\Delta t$ ) + 1 # iterations
6 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:

```

1 # Fourth-Order Runge-Kutta function
2 def RK4(f, x0, y0, z0, h):
3     k1y = h * f(x0, y0, z0)[0]
4     k1z = h * f(x0, y0, z0)[1]
5     #
6     k2y = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[0]
7     k2z = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[1]
8     #
9     k3y = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[0]
10    k3z = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[1]
11    #
12    k4y = h * f(x0 + h, y0 + k3y, z0 + k3z)[0]
13    k4z = h * f(x0 + h, y0 + k3y, z0 + k3z)[1]
14
15    # Approximation
16    y1 = float(y0 + (k1y + 2 * k2y + 2 * k3y + k4y) / 6)
17    z1 = float(z0 + (k1z + 2 * k2z + 2 * k3z + k4z) / 6)
18    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.

```
1  # Differential Equations
2  def f(t, theta, omega):
3      return omega, -(g / l) * np.sin(theta)
```

That is, `f(t, theta, omega)` defines the equations

$$\begin{aligned}\dot{\theta} &= \omega, \\ \dot{\omega} &= -\frac{g}{\ell} \sin(\theta).\end{aligned}$$

We initialize our storage arrays:

```
1  # Initial arrays
2  theta, theta[0] = np.zeros(n), theta0
3  omega, omega[0] = np.zeros(n), omega0
```

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:

```
1  # RK4 method evaluation
2  for i in range(n - 1):
3      theta[i + 1], omega[i + 1] = RK4(f, t[i], theta[i], omega[i], dt)
```

If we plot t versus θ , and t versus ω , we obtain the next plots:

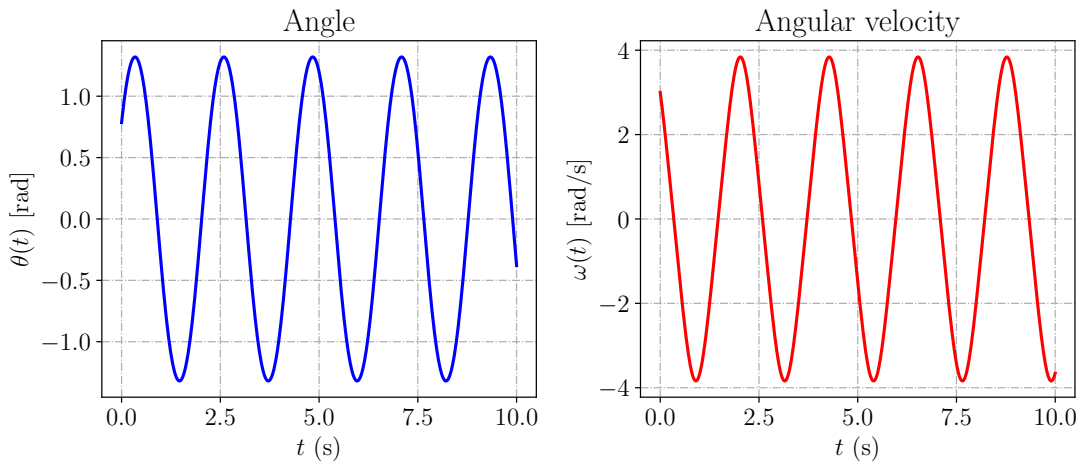


Figure 1: Evolution of the angle through time.

The full code is shown here:

```
1  # Libraries
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Plot parameters
6  plt.rcParams.update({
7      'lines.linewidth': 2,      # linewidth
8      'text.usetex': True,      # LaTeX font
9      'font.family': 'serif',   # Serif family
10     'font.size': 16,           # font size
11     'axes.titlesize': 20,      # title size
12     'axes.grid': True,         # grid
13     'grid.linestyle': "-.",    # grid style
14 })
15
16 # Physical parameters
17 g = 9.81 # gravity (kg/m^2)
18 l = 1    # length (m)
19
20 # Initial conditions
21 θ0 = np.pi/4 # initial angle (rad)
22 ω0 = 3.0      # initial angular velocity (rad/s)
23
24 # Time parameters
25 t_0 = 0        # initial time (s)
26 t_f = 10       # final time (s)
27 Δt = 0.01      # step size (s)
28 n = int((t_f - t_0) / Δt) + 1 # iterations
29 t = np.linspace(t_0, t_f, n) # time array (s)
30
31 # Fourth-Order Runge-Kutta function
32 def RK4(f, x0, y0, z0, h):
33     k1y = h * f(x0, y0, z0)[0]
34     k1z = h * f(x0, y0, z0)[1]
35     #
36     k2y = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[0]
37     k2z = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[1]
38     #
39     k3y = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[0]
40     k3z = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[1]
41     #
42     k4y = h * f(x0 + h, y0 + k3y, z0 + k3z)[0]
43     k4z = h * f(x0 + h, y0 + k3y, z0 + k3z)[1]
44
45     # Approximation
46     y1 = float(y0 + (k1y + 2 * k2y + 2 * k3y + k4y) / 6)
47     z1 = float(z0 + (k1z + 2 * k2z + 2 * k3z + k4z) / 6)
48     return y1, z1
49
50 # Differential Equations
51 def f(t, θ, ω):
52     return ω, -(g / l) * np.sin(θ)
```

```

53
54 # Initial arrays
55  $\theta$ ,  $\theta[0]$  = np.zeros(n),  $\theta_0$ 
56  $\omega$ ,  $\omega[0]$  = np.zeros(n),  $\omega_0$ 
57
58 # RK4 method evaluation
59 for i in range(n - 1):
60      $\theta[i + 1]$ ,  $\omega[i + 1]$  = RK4(f, t[i],  $\theta[i]$ ,  $\omega[i]$ ,  $\Delta t$ )
61
62 plt.figure(figsize = (10, 4.5))
63
64 # Plot
65 plt.subplot(1, 2, 1)
66 plt.plot(t,  $\theta$ , 'b-')
67 plt.xlabel(r'$t$ (s)')
68 plt.ylabel(r'$\theta(t)$ [rad]')
69 plt.title(r'Angle')
70
71 plt.subplot(1, 2, 2)
72 plt.plot(t,  $\omega$ , 'r-')
73 plt.xlabel(r'$t$ (s)')
74 plt.ylabel(r'$\omega(t)$ [rad/s]')
75 plt.title(r'Angular velocity')
76 plt.tight_layout()
77 plt.gcf()
78 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:

```

1 # Libraries
2 using LinearAlgebra
3 using Plots, ColorSchemes, LaTeXStrings
4
5 # Physical parameters
6 g = 9.81 # gravity (kg/m^2)
7 l = 1    # length (m)
8
9 # Initial conditions
10  $\theta_0$  =  $\pi/4$  # initial angle (rad)
11  $\omega_0$  = 3.0 # initial angular (rad/s)

```

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

```

1 # Time parameters
2 t0 = 0           # initial time (s)
3 t1 = 10          # final time (s)

```

```

4  Δt = 0.01                                # step size (s)
5  n = Int64((t1 - t0) / Δt)                # iterations
6  t = t0:Δt:(t1 - Δt)                     # time array (s)

```

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

```

1  # Fourth-Order Runge-Kutta function
2  function RK4(f, x0, y0, z0, h)
3      k1y = h * f(x0, y0, z0)[1]
4      k1z = h * f(x0, y0, z0)[2]
5      #
6      k2y = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[1]
7      k2z = h * f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[2]
8      #
9      k3y = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[1]
10     k3z = h * f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[2]
11     #
12     k4y = h * f(x0 + h, y0 + k3y, z0 + k3z)[1]
13     k4z = h * f(x0 + h, y0 + k3y, z0 + k3z)[2]
14
15     # Approximation
16     y1 = Float64(y0 + (k1y + 2 * k2y + 2 * k3y + k4y) / 6)
17     z1 = Float64(z0 + (k1z + 2 * k2z + 2 * k3z + k4z) / 6)
18     return y1, z1
19 end

```

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

```

1  # Differential Equations
2  function f(t, θ, ω)
3      return ω, -(g / l) * sin(θ)
4  end

```

We initialize the storage arrays:

```

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

```

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

```

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

```

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

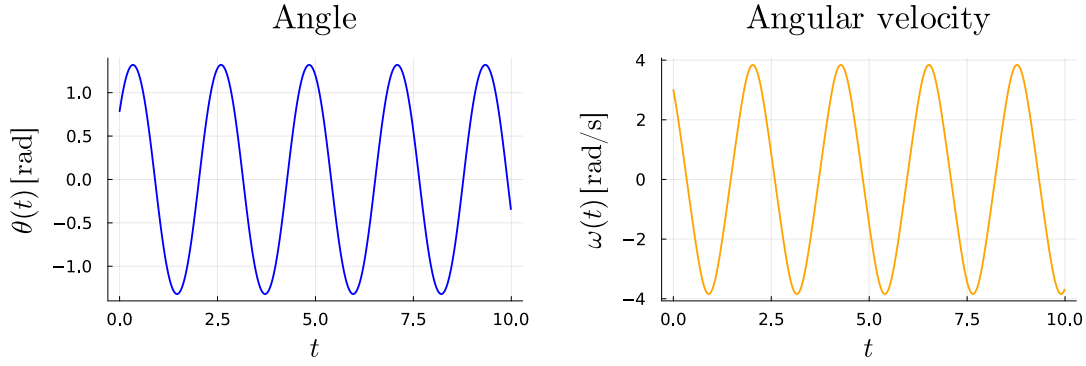


Figure 2: Evolution of the angle through time.

Here is the full code in Julia:

```

1  # Libraries
2  using LinearAlgebra
3  using Plots, ColorSchemes, LaTeXStrings
4
5  # Physical parameters
6  g = 9.81 # gravity (kg/m^2)
7  l = 1    # length (m)
8
9  # Initial conditions
10 θ0 = π/4 # initial angle (rad)
11 ω0 = 3.0 # initial angular (rad/s)
12
13 # Time parameters
14 t0 = 0                                # initial time (s)
15 t1 = 10                              # final time (s)
16 Δt = 0.01                            # step size (s)
17 n = Int64((t1 - t0) / Δt)            # iterations
18 t = t0:Δt:(t1 - Δt)                  # time array (s)
19
20 # Fourth-Order Runge-Kutta function
21 function RK4(f, x0, y0, z0, h)
22     k1y = h .* f(x0, y0, z0)[1]
23     k1z = h .* f(x0, y0, z0)[2]
24     #
25     k2y = h .* f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[1]
26     k2z = h .* f(x0 + h / 2, y0 + k1y / 2, z0 + k1z / 2)[2]
27     #
28     k3y = h .* f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[1]
29     k3z = h .* f(x0 + h / 2, y0 + k2y / 2, z0 + k2z / 2)[2]
30     #
31     k4y = h .* f(x0 + h, y0 + k3y, z0 + k3z)[1]
32     k4z = h .* f(x0 + h, y0 + k3y, z0 + k3z)[2]
33
34     # Approximation
35     y1 = Float64(y0 + (k1y + 2 * k2y + 2 * k3y + k4y) / 6)
36     z1 = Float64(z0 + (k1z + 2 * k2z + 2 * k3z + k4z) / 6)
37     return y1, z1

```

```

38 end
39
40 # Differential equations
41 function f(t, θ, ω)
42     return ω, -(g / l) .* sin.(θ)
43 end
44
45 # Initial arrays
46 θ, θ[1] = zeros(n), θ0
47 ω, ω[1] = zeros(n), ω0
48
49 # RK4 method evaluation
50 for i in 1:(n - 1)
51     θ[i + 1], ω[i + 1] = RK4(f, t[i], θ[i], ω[i], Δt)
52 end
53
54 # Plot
55 default(fontfamily = "Computer modern", legend = false, tickfont = font(12), guidefont =
    ↪ font(20), titlefont = font(24), linewidth = 2)
56 p1 = plot(t, θ, title = L"\mathrm{Angle}", xlabel = L"$t$", ylabel =
    ↪ L"$\theta(t)\colon [\mathrm{rad}]$", color = "blue")
57 p2 = plot(t, ω, title = L"\mathrm{Angular}\colon \mathrm{velocity}", xlabel = L"$t$",
    ↪ ylabel = L"$\omega(t)\colon [\mathrm{rad}/\mathrm{s}]$", color = "orange")
58 plot(p1, p2, layout = (1, 2), size = (1200, 400), margin = 10Plots.mm)
59 savefig("juliaplot2.pdf")

```

References

- M. Tenenbaum and H. Pollard. *Ordinary Differential Equations*. Dover Books on Mathematics. Dover Publications, Mineola, New York, October 1985.
- E. Yépez Mulia and Y. Yépez Martínez. *Mecánica analítica*. Universidad Nacional Autónoma de México, Facultad de Ciencias, Mexico City, Mexico, 2012.