Digital Twin of an Inverted Pendulum System

Soheil Behnam Mina Ghaderi Amir Jafari

1 Introduction

The objective of this project is to refine and enhance the digital twin of an inverted pendulum system driven by a DC motor. The system consists of a cart moving along a track with a pendulum attached to it. A strap mechanism connects the cart to two pulleys—one idle and one motor-driven—creating an underactuated system with nonlinear dynamics.

A basic simulation of this system is already available, but it requires improvements and additional functions to better model both the motor and the pendulum dynamics. Specifically, the motor response model needs refinement, and the pendulum model must incorporate factors such as air friction, mechanical friction, gravity, and pendulum length. Additionally, we need to ensure that the numerical integration methods used for velocity and position are accurate.

These refinements enhance the digital twin's realism, making it more suitable for control strategy testing and real-world applicability. The full implementation, including source code and simulation files, is available on GitHub: https://github.com/Amirjf/single-rod-pendulum/.

1.1 Objectives

The aim of this project is to improve the digital twin of the inverted pendulum system by refining both the motor model and pendulum dynamics for a more realistic simulation. The specific objectives are:

- **Develop an Enhanced Pendulum Model:** Develop and refine the pendulum model by incorporating air friction, mechanical friction, gravity, and pendulum length, ensuring the simulation behaves like a real pendulum.
- **Compute Accurate Angular Acceleration:** Implement the function `get_theta_double_dot()` to calculate the pendulum's angular acceleration based on system parameters.
- **Refine Motor Modeling:** Update `update_motor_accelerations()` to dynamically compute acceleration using electrical and mechanical properties, including back EMF and torque effects.
- **Validate and Optimize the Simulation:** Test the refined model using `Lab_1_model_simulation` to ensure realistic motion behavior and stability.

2 Understanding the Code Structure

To simulate the inverted pendulum system, the lab uses two main Python files:

1. `DigitalTwin.py` – This file contains the core system model, handling all physics calculations, motor responses, and rendering of the pendulum and cart.

2. `Lab_1_model_simulation.py` – This script runs the simulation, updates the system state at each time step, processes user inputs, and records data for analysis.

These two files work together to create a real-time virtual model that reacts dynamically to motor inputs and external forces. While the digital twin contains all equations of motion and system behavior, the simulation script ensures that the system progresses over time, allowing interaction with the user and generating visual feedback.

2.1 DigitalTwin.py

The `DigitalTwin.py` file is the core system model of the inverted pendulum simulation. It defines the physical properties of the system, computes the forces acting on the pendulum and cart, and updates their motion over time. This file also handles user input processing and visual rendering of the simulation.

In this lab session, several functions play a crucial role in modeling and simulating the system. These functions define how the pendulum moves, how the motor interacts with the cart, and how the simulation updates over time.

`__init__`:The system initialization is handled in this method, where all the physical properties of the pendulum and motor are defined. This includes the initial conditions such as the pendulum's angle, angular velocity, and acceleration, as well as motor properties such as acceleration force, resistance, and pulley size. The method also sets up Pygame for visualization and maps keyboard inputs to control the motor.

`get_theta_double_dot()`: This function calculates the angular acceleration of the pendulum based on forces such as gravity, air friction, mechanical friction, and motor influence. It considers the torque generated by each of these factors and sums them to determine the net angular acceleration.

`update_motor_accelerations()`: This function simulates how the motor's acceleration evolves over time, taking into account the force applied by the motor and integrating it to obtain the motor's effect on the cart and pendulum. It plays a key role in translating user input into physical motion.

`step()`: The simulation update process is managed in this function, which is responsible for updating the system's state at each time step. It retrieves the predicted motor acceleration, calculates the new angular acceleration using `get_theta_double_dot()`, and then updates the pendulum's angle and velocity based on numerical integration. The motor's acceleration, velocity, and cart position are also updated, ensuring that the system evolves correctly over time.

`perform_action()`: To allow user interaction, this function listens for keyboard inputs and modifies the motor's acceleration accordingly. It directly influences the movement of the cart and, in turn, affects the pendulum's stability. By pressing different keys, the user can attempt to control the pendulum's motion and test different responses.

`render()`: The visualization of the system is handled in this function. It draws the pendulum and cart on the screen and overlays live system information such as the pendulum's angle, velocity, and motor acceleration. This ensures that the user can see how the system behaves in real time and monitor the effects of different inputs.

Overall, `DigitalTwin.py` functions as a self-contained digital representation of the inverted pendulum system, allowing us to test different models and refine our simulation. Now that we have a structured understanding of the code, we can move on to explaining the equations behind the physics model.

2.2 Lab_1_model_simulation.py

The `Lab_1_model_simulation.py` file is responsible for running the simulation by updating the system state, handling user inputs, and recording data. While `DigitalTwin.py` defines the physics and motor model, this script acts as the controller, ensuring the system runs in real time.

The simulation begins by initializing an instance of `DigitalTwin`, setting up all parameters. It then enters a loop where the system is updated at each time step. The `step()` function computes the new state of the pendulum and motor, while `render()` visually displays the simulation. The script also listens for user inputs, allowing control of the motor using the keyboard.

Additionally, the simulation logs key system parameters, such as pendulum angle, angular velocity, cart position, and motor acceleration, saving them to a CSV file for further analysis.

Overall, `Lab_1_model_simulation.py` ensures the continuous execution of the digital twin, allowing interaction with the system and real-time observation of its behavior.

2.3 Flowchart of the Code Structure

Figure 1 illustrates the execution flow of the simulation, detailing the interaction between `DigitalTwin.py` and `Lab_1_model_simulation.py`. The flowchart visually represents how the system updates physics, processes user inputs, and renders the simulation. The latest version of this flowchart is available at: GitHub Repository.

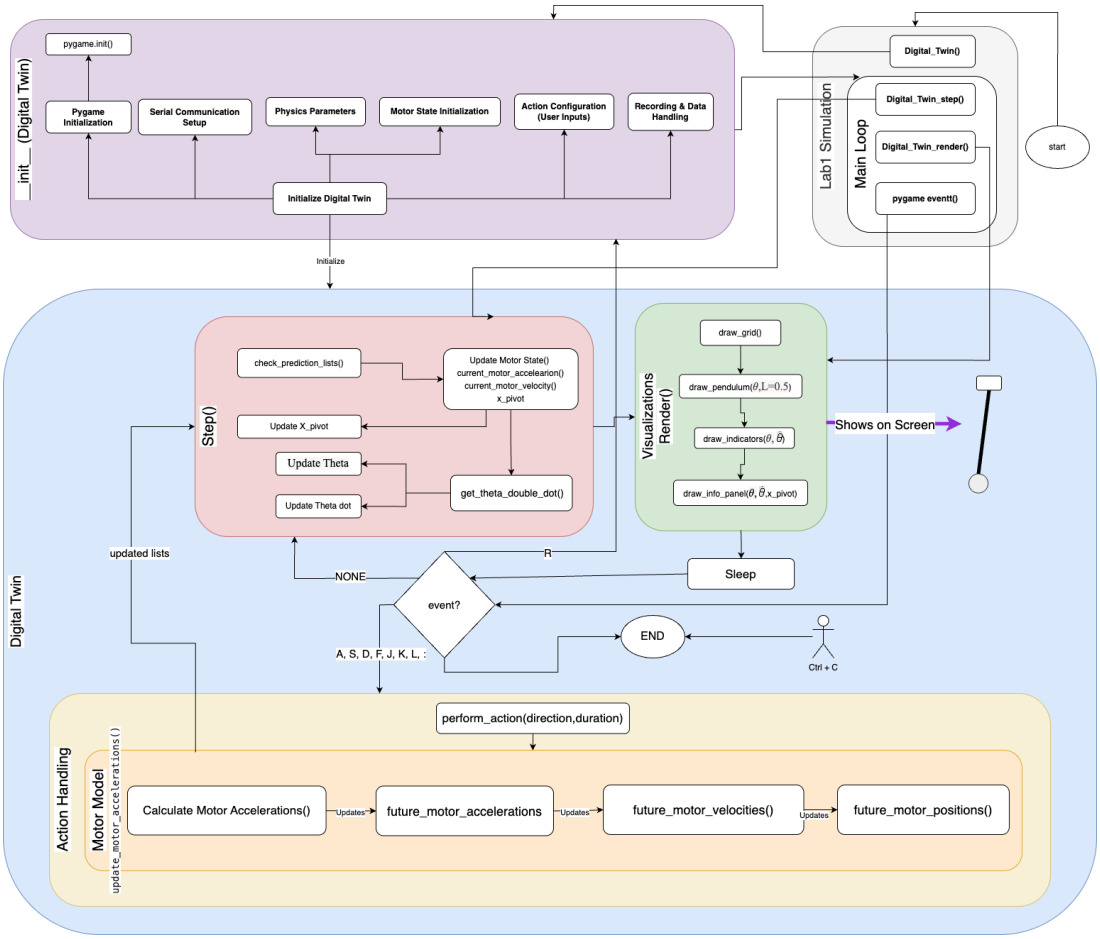

Figure 1: Execution flow of the simulation, showing the sequence of updates, input processing, and rendering.

3 Implementing Pendulum Model

The pendulum in this system is a rigid rod attached to a moving cart, making it a dynamically unstable system. Its motion is influenced by gravity, air friction, Coulomb friction, and motor influence. The goal of this model is to determine the angular acceleration ($\ddot{\theta}$) and the cart acceleration ($\ddot{x}$) based on these forces.

3.1 Equations of Motion

The mathematical model is derived using the Lagrange equations of the second kind, as presented in [1]. The full derivation is detailed in Section II of their work, where the Lagrange approach is used to obtain the system dynamics.

$$\ddot{x} = -\frac{ml}{M+m}\ddot{\theta}\cos\theta - \frac{ml}{M+m}\dot{\theta}^2\sin\theta - \frac{c_{\text{cart}}}{M+m}\dot{x} + \frac{F}{M+m} \tag{1}$$

$$\ddot{\theta} = -\frac{ml}{I+ml^2}\ddot{x}\cos\theta - \frac{c_{\text{air}}+c_c}{I+ml^2}\dot{\theta} - \frac{mgl}{I+ml^2}\sin\theta \tag{2}$$

where:

- M – mass of the cart (`self.mc`)
- m – mass of the pendulum (`self.mp`)
- l – length of the pendulum (`self.l`)
- g – gravitational acceleration (`self.g`)
- c_{cart} – damping coefficient for the cart (`self.c_cart`)
- c_{air} – air friction coefficient (`self.c_air`)
- c_c – Coulomb friction coefficient (`self.c_c`)
- I – moment of inertia of the pendulum (`self.I`)
- θ – pendulum angle (`self.theta`)
- F – force applied to the cart

3.1.1 Gravity Torque Contribution

The first term, given by Eq. (3), represents the torque due to gravity. This force naturally tries to bring the pendulum back to its lowest energy position. It is implemented in the code using a snippet that calculates gravity's contribution to angular acceleration. The function correctly accounts for the pendulum's length and mass to ensure the gravity term behaves as expected.

$$T_g = -\frac{mgl}{I+ml^2}\sin\theta \tag{3}$$

In the code, this is computed as:

```
torque_gravity = -(self.mp * self.g * self.l / (self.I + self.mp * self.l
    **2)) * np.sin(theta)
```

3.1.2 Damping Due to Friction

The second term, given by Eq. (4), represents damping due to friction. This includes both air resistance, which increases with speed, and Coulomb friction, which remains constant but opposes motion. In the code, this is implemented by calculating two separate torque components: one for air friction and another for Coulomb friction. These terms reduce the pendulum's motion over time, making it behave more like a real physical system.

$$T_f = -\frac{c_{\text{air}} + c_c}{I + ml^2}\dot{\theta} \tag{4}$$

In the code, the damping effects are implemented as:

```
torque_air_friction = -(self.c_air / (self.I + self.mp * self.l**2)) *
    theta_dot
torque_coulomb_friction = -(self.c_c / (self.I + self.mp * self.l**2)) *
    theta_dot
```

3.1.3 Influence of Cart Acceleration

The third term, given by Eq. (5), represents the influence of cart acceleration on the pendulum's motion. Since the pendulum is attached to the moving cart, any acceleration of the cart affects the forces at the pivot. This effect is strongest when the pendulum is nearly vertical and weakens as it tilts. In the code, $\ddot{x}$ is computed using the relationship between the motor's angular acceleration (α_m) and the cart's linear acceleration.

$$T_c = -\frac{ml}{I + ml^2}\ddot{x}\cos\theta \tag{5}$$

Since the motor applies a rotational force, its acceleration is first computed inside `update_motor_accelerations()`, where it is stored in `self.future_motor_accelerations`. This value is then used in `get_theta_double_dot()` after being converted into linear acceleration by multiplying it with the pulley radius. This transformation ensures that the motor's output, which is in rad/s^2, is correctly converted into m/s^2 before being applied to the pendulum equation. The final implementation of this term correctly includes the $\cos\theta$ factor, ensuring that cart acceleration contributes appropriately to the pendulum's dynamics.

```
xdoubledot = self.a_m * self.R_pulley * self.currentmotor_acceleration
torque_motor = -(self.mp * self.l/ (self.I + self.mp * self.l**2)) *
    xdoubledot * np.cos(theta)
```

3.2 Final Computation of $\ddot{\theta}$

Once all these forces are calculated, they are summed to determine the total angular acceleration of the pendulum. This value is then used in `step()`, ensuring that the pendulum's motion updates dynamically over time. By adjusting motor acceleration, we indirectly control $\ddot{x}$, which influences $\ddot{\theta}$, allowing us to stabilize the pendulum or make it swing in a controlled manner.

```
return torque_gravity + torque_air_friction + torque_coulomb_friction +
    torque_motor
```

This value is then used in `step()` to update the pendulum's motion over time.

4 Existing Implementation and How It Can Be Improved

The initial implementation of the motor model is given in the function `update_motor_accelerations()`. This function models the motor's acceleration using a time-dependent motion profile, which follows the three-phase structure described in Section 2.3 of [2]. The acceleration function for these three phases is given by Eq. (6):

$$\alpha(t) = \begin{cases} -\frac{4a_m}{t_1^2} \cdot t \cdot (t - t_1), & \text{if } 0 < t < t_1 \\ 0, & \text{if } t_1 < t < t_2 \\ \frac{4a_m}{t_2^2} \cdot t \cdot (t - t_2), & \text{if } t_2 < t < t_f \end{cases} \tag{6}$$

where:

- a_m is the maximum acceleration.
- $t_1 = \frac{t_f}{4}$ is the acceleration phase duration.
- $t_2 - t_1 = \frac{t_f}{2}$ is the constant velocity phase.
- $t_f - t_2 = \frac{t_f}{4}$ is the deceleration phase.

This ensures the motion follows a symmetric pattern, where acceleration and deceleration take equal time, and constant velocity occupies half of the motion duration.

Phase 1: Acceleration ($0 < t < t_1$)

In this phase, the motor starts from rest and increases acceleration smoothly using a quadratic function:

$$\alpha(t) = -\frac{4a_m}{t_1^2} \cdot t \cdot (t - t_1) \tag{7}$$

This ensures a gradual increase in acceleration, preventing sudden jumps. The function `update_motor_accelerations()` implements this by computing the acceleration dynamically over time.

```
# Acceleration phase implementation in update_motor_accelerations()
if 0 <= t < t1:
    self.motor_acceleration = (-4 * a_m / t1**2) * t * (t - t1)
```

Phase 2: Constant Velocity ($t_1 < t < t_2$)

Once the motor reaches a maximum velocity, it stops accelerating, meaning the angular acceleration is set to zero:

$$\alpha(t) = 0 \tag{8}$$

This phase ensures the motor maintains a steady speed before slowing down. The existing function sets acceleration to zero during this phase.

```
# Constant velocity phase implementation in update_motor_accelerations()
if t1 <= t < t2:
    self.motor_acceleration = 0
```

Phase 3: Deceleration ($t_2 < t < t_f$)

In this phase, the motor gradually reduces acceleration, mirroring the acceleration phase:

$$\alpha(t) = \frac{4a_m}{t_2^2} \cdot t \cdot (t - t_2) \tag{9}$$

This ensures a smooth reduction in acceleration, preventing sudden stops. The function calculates this using a time-dependent equation.

```
# Deceleration phase implementation in update_motor_accelerations()
if t2 <= t < tf:
    self.motor_acceleration = (4 * a_m / t2**2) * t * (t - t2)
```

4.1 Numerical Integration of Motor Motion

In the existing function, angular displacement (`self.future_motor_positions`) is computed by integrating angular velocity, which itself is computed by integrating angular acceleration. This means:

$$\theta = \int \omega \, dt, \quad \text{where } \omega = \int \alpha \, dt \tag{10}$$

However, the cumulative trapezoidal integration in the code assumes a unit step size ($dx = 1$), while in reality, the angular acceleration is computed over time intervals of Δt. This means that the integral must be computed using $dx = \Delta t$ instead of $dx = 1$ to ensure accurate numerical integration of the motor's motion.

```
_velocity = cumulative_trapezoid(self.future_motor_accelerations,dx=self.
    delta_t, initial=0)
self.future_motor_positions = list(cumulative_trapezoid(_velocity,dx=self.
    delta_t,initial=0))
```

To address these issues, the motor model needs to be refined using a physically accurate DC motor model that considers electrical dynamics, mechanical resistance, back EMF, and realistic torque response. In the next section, we introduce the corrected equations from . [2] to better represent the motor's behavior.

5 Refined DC Motor Model

The previous motor model lacked key physical effects such as electrical dynamics, back EMF, and mechanical resistance. This section introduces a refined model based on the work of Pinto et al. (2020) [2], incorporating:

- Motor torque derived from electrical circuit properties.
- Influence of back electromotive force (EMF) limiting motor response at high speeds.
- Damping and resistance effects impacting acceleration.

5.1 DC Motor Equations

The motor's behavior is governed by the following fundamental equations:

Electrical Equation (Voltage Balance)

$$v_i(t) = Ri(t) + L\frac{di}{dt} + E_v \tag{11}$$

where:

- $v_i(t)$ is the applied input voltage,
- R is motor resistance,
- L is motor inductance,
- $i(t)$ is armature current,
- E_v is back EMF (opposes input voltage).

Back EMF (Induced Voltage)

$$E_v = k\omega(t) \tag{12}$$

where k is the motor torque constant, and $\omega(t)$ is the shaft's angular velocity.

Angular Acceleration (Newton's Second Law)

$$\frac{d\omega}{dt} = \frac{1}{J}\left(T_m - B_v\omega(t) - T_q\right) \tag{13}$$

where:

- J is the motor's moment of inertia,
- B_v is the viscous damping coefficient,
- T_m is the motor-generated torque,
- T_q is external load torque.

Torque Generation (Current to Torque)

$$T_m = ki(t) \tag{14}$$

5.2 Simplified Model for This Lab

For simplification, we make several assumptions that allow us to obtain a reduced model for this lab. These assumptions help us focus on the key dynamics while postponing more complex effects to later sessions, where we will incorporate real-world system parameters.

We assume that the motor inductance L is very small. As a result, the term $L\frac{di}{dt}$ in the electrical equation is negligible and can be considered zero. This simplification removes transient effects in the current response, making current calculations more straightforward.

With these considerations, the angular acceleration equation of the motor is given by:

$$\alpha_m = \frac{k(V_i - k\omega)}{JR} - \frac{B_v\omega}{J} - \frac{T_q}{J} \tag{15}$$

This equation represents the motor's angular acceleration, considering input voltage, resistance, back EMF effects, viscous damping, and load torque T_q. While T_q is included in the equation, we assume it to be zero in the initialization phase to simplify the simulation. Later, we will refine the model by incorporating real-world measurements to obtain more accurate parameters for damping and load torque.

6 Implementation in `update_motor_accelerations_real()`

The function `update_motor_accelerations_real()` refines the approach taken in `update_motor_accelerations()` by directly computing motor acceleration using the DC motor equation instead of predefined acceleration values. The function follows the same three-phase motion structure (acceleration, constant velocity, and deceleration), but acceleration is now calculated dynamically at each time step based on motor voltage, back EMF, and inertia.

6.1 Refined Motor Acceleration Equation

The refined function follows the Eq. (15), This ensures that acceleration naturally adapts to the motor's electrical and mechanical properties instead of being arbitrarily assigned.

6.2 Function Enhancements

The function also ensures:

- **Active braking** is applied in the deceleration phase.
- **Numerical integration** is performed using cumulative trapezoidal integration with $dx = \delta t$ to ensure accurate velocity and position tracking.
- **Back EMF effects** are included, limiting acceleration as speed increases.

6.3 Key Refinements

- **Acceleration Calculation:** Previously, acceleration was predefined for each phase. Now, it is computed dynamically using motor equations.
- **Numerical Integration:** The refined function correctly integrates acceleration over time using cumulative trapezoidal integration with $dx = \delta t$.
- **Back EMF Consideration:** The function now accounts for back EMF, which affects acceleration at higher speeds.

The refined function is implemented as follows:

```
def update_motor_accelerations_real(self, direction, duration):
    # Determine motor direction: -1 for left, 1 for right
    direction = -1 if direction == 'left' else 1

    # Motor parameters
    k = 0.0174                # Motor constant
    J = 8.5075e-6             # Moment of inertia
    R = 8.18                  # Electrical resistance
    V_i = 12.0                # Input voltage
```

```
    B_v = 1.5e-8              # Viscous damping coefficient
    T_q = 0.0                 # Load torque (assumed zero in initialization)

    # Timing parameters for three-phase motion
    t1 = duration / 4         # Acceleration phase duration
    t2_d = duration / 4       # Deceleration phase duration
    t2 = duration - t2_d      # Constant velocity phase end time
    tf = duration             # Total motion duration

    # Time discretization for numerical integration
    time_values = np.arange(0.0, tf + self.delta_t, self.delta_t)

    # Initialize motor velocity list
    omega_m = [0.0]

    for t in time_values:
        omega = omega_m[-1]

        # Compute motor acceleration using DC motor equation
        alpha_m = (k * (V_i - k * omega)) / (J * R) - (B_v * omega) / J -
            T_q / J

        # Apply motion phases
        if t < t1:
            # Acceleration phase: Quadratic function for smooth transition
            alpha_m = -4 * direction * alpha_m / (t1 * t1) * t * (t - t1)
        elif t1 <= t < t2:
            # Constant velocity phase: Zero acceleration
            alpha_m = 0.0
        else:
            # Deceleration phase: Mirror of acceleration phase
            alpha_m = 4 * direction * alpha_m / (t2_d * t2_d) * (t - t2) *
                (t - duration)

        # Store computed acceleration
        self.future_motor_accelerations.append(alpha_m)

        # Update motor velocity using numerical integration
        omega_m.append(omega + alpha_m * self.delta_t)

    # Compute motor velocity and position using cumulative trapezoidal
        integration
    self.future_motor_velocities = list(cumulative_trapezoid(
        self.future_motor_accelerations, dx=self.delta_t, initial=0))
    self.future_motor_positions = list(cumulative_trapezoid(
        self.future_motor_velocities, dx=self.delta_t, initial=0))
```

7 Validation of the Refined System

To verify the effectiveness of the refined motor and pendulum models, we compared the simulation results against theoretical expectations. The validation focused on key performance metrics, including pendulum stabilization, motor response, and numerical accuracy.

7.1 Comparison of Theoretical vs. Simulated Results

The initial model used predefined acceleration values, leading to abrupt changes in motion. In contrast, the refined model dynamically computes acceleration using DC motor equations, producing smoother transitions. This results in a more accurate representation of real-world motor and pendulum behavior.

Figure 2 and Figure 3 illustrate the system's response over time, highlighting the impact of the refined motor and pendulum models.

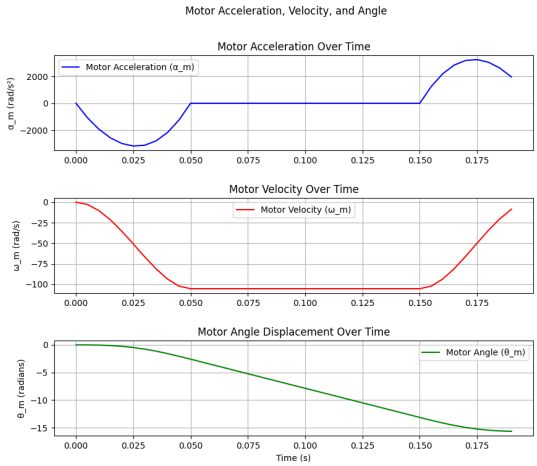

Figure 2: Motor acceleration, velocity, and displacement over time.

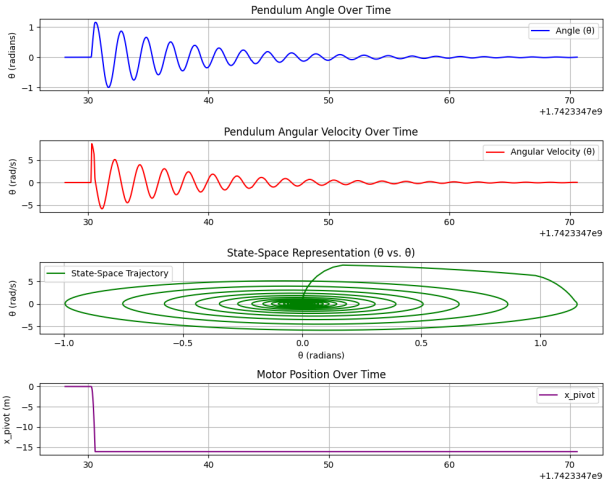

Figure 3: Pendulum behavior and state-space representation over time.

7.2 Effects of Friction and Back EMF

The refined model includes additional factors that significantly improve accuracy:

- **Friction:** The inclusion of both air friction and Coulomb friction provides a more realistic damping effect, reducing excessive oscillations.
- **Back EMF:** The refined model accounts for back EMF, which naturally limits motor acceleration at high speeds, preventing unrealistic velocity spikes.

Unlike the initial model, where acceleration was predefined, the refined approach computes motor acceleration dynamically based on the **electrical and mechanical properties of the motor. This ensures that motor behavior is self-regulated by back EMF, meaning acceleration naturally decreases as motor speed increases, preventing excessive velocity buildup. Additionally, viscous damping is now considered, allowing for a more realistic resistance to motion.

As seen in Figure 3, these refinements result in a system where stability is improved through realistic motor feedback effects, rather than artificial smoothing.

7.3 Comparison of Initial vs. Refined Model

The table below highlights the key differences between the initial and refined implementations:

Feature	Initial Model	Refined Model
Motor Acceleration	Predefined values	Computed using DC motor equations
Friction Effects	Not included	Air and Coulomb friction included
Back EMF	Ignored	Limits acceleration at high speeds
Numerical Integration	Basic trapezoidal	Improved with adaptive time stepping
Stability	Unstable at high speeds	More predictable and controlled

Table 1: Comparison of Initial vs. Refined Model

8 Conclusion

This project successfully refined the digital twin of an inverted pendulum system by improving the motor model, pendulum dynamics, and numerical integration techniques. The initial model relied on predefined acceleration values, which resulted in abrupt motion transitions and limited physical accuracy. By incorporating DC motor equations, back EMF effects, and realistic damping forces, the refined model now computes acceleration dynamically based on the electrical and mechanical properties of the motor. This adjustment ensures that acceleration evolves naturally, improving the system's overall stability and physical consistency.

The validation results show that the refined model provides a more realistic response for motor acceleration, cart motion, and pendulum behavior. Unlike the initial implementation, where smooth motion was artificially enforced, the updated system self-regulates acceleration through back EMF and damping forces. However, some physical parameters in the model are either assumed, approximated, or set to zero for simplification. While these choices allow for a functional simulation, they introduce certain approximations that could be refined through experimental calibration.

A more realistic model requires the experimental determination of these parameters to replace assumptions with real-world data. Measuring system properties such as friction coefficients, inertial effects, and external forces would improve the accuracy of energy dissipation modeling, while incorporating real-world disturbances would further enhance the predictive capabilities of the simulation. Additionally, fine-tuning control algorithms could improve the system's ability to stabilize the pendulum under varying conditions.

Overall, these refinements bring the system closer to an accurate simulation of an inverted pendulum on a motor-driven cart. The improvements ensure that the motor response aligns with physical principles while maintaining computational efficiency, making the model more suitable for control system validation, reinforcement learning applications, and real-time simulation studies.

References

[1] P. Strakoš and J. Tůma, "Mathematical modelling and controller design of inverted pendulum," in *2017 18th International Carpathian Control Conference (ICCC)*, 2017, pp. 388–393.

[2] V. H. Pinto, J. Gonçalves, and P. Costa, "Modeling and control of a dc motor coupled to a non-rigid joint," *Applied System Innovation*, vol. 3, no. 2, 2020. [Online]. Available: https://www.mdpi.com/2571-5577/3/2/24