Technical Report on the Inverted Cart and Pendulum System

Submitted by:

SCHOLAR ID:

2315049

2315064

2315020

2315051

2315021

Electronics And Instrumentation Engineering / NIT Silchar

Date: November 15, 2025

Contents

Chapter 1

Modeling

1.1 Non-linear Model using Newton's Laws

Using Newton's laws on the cart (mass M) and pendulum (mass m, length L), the full nonlinear equations are:

$$(M+m)\ddot{x} + ml\cos\theta\,\ddot{\theta} - ml\sin\theta\,\dot{\theta}^2 + b\dot{x} = u, \tag{1.1}$$

$$ml\cos\theta\,\ddot{x} + (ml^2 + J)\ddot{\theta} + mgl\sin\theta = 0. \tag{1.2}$$

1.2 Parameter Values

The parameters used (as in the sample) are:

Table 1.1: Parameter values

Parameter	Value
M (Mass of cart)	0.38 kg
m (Mass of pendulum)	0.4 kg
L (Pendulum length)	0.33 m
l (Centre of mass distance)	0.11 m
J (Moment of inertia)	0.0020736 kg·m^2
b (cart friction coef.)	0.1
g (gravity)	9.81 m/s^2

1.3 Linearization About Upright ($\theta = 0degree$)

Using $\sin\theta \approx \theta$ and $\cos\theta \approx 1$, the linearized equations become:

$$(M+m)\ddot{x} + ml\ddot{\theta} + b\dot{x} = u,$$

$$ml\ddot{x} + (ml^2 + J)\ddot{\theta} + mgl\theta = 0.$$

Define the state

$$x = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}.$$

The linear continuous-time state-space model is

$$\dot{x} = Ax + Bu,$$

with (numerical A and B taken from the sample):

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -0.208333 & 6.125000 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0.946970 & -72.3864 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 0 \\ 2.083333 \\ 0 \\ -9.469697 \end{bmatrix}.$$

Chapter 2

Discretization

2.1 Sampling Time (Ts)

A sampling time of $T_s = 0.120$ s (10 ms) is selected for discretization . This matches the justification given in the sample (fast unstable system, 120 Hz update, MCU feasibility) and is used throughout this report.

2.2 Discrete-Time Model

With zero-order hold (ZOH) the discrete model is

$$x_{k+1} = A_d x_k + B_d u_k,$$

$$A_d = e^{AT_s}, \qquad B_d = \int_0^{T_s} e^{A\tau} d\tau \, B.$$

The discrete matrices computed (via scipy `cont2discrete`) are:

$$A_d = \begin{bmatrix} 1.00000000 & 0.11856035 & 0.04006031 & 0.00166386 \\ 0 & 0.97687559 & 0.60574148 & 0.04006031 \\ 0 & 0.00619362 & 0.52403902 & 0.10025916 \\ 0 & 0.09365208 & -7.21946030 & 0.52403902 \end{bmatrix},$$

$$B_d = \begin{bmatrix} 0.01439657 \\ 0.23124445 \\ -0.06193616 \\ -0.93652052 \end{bmatrix}.$$

2.2.1 Python code used

```python
import numpy as np
from scipy.signal import cont2discrete

# Continuous-time matrices
A = np.array([
    [0.0,       1.0,        0.0,         0.0],
    [0.0, -0.208333,   6.125000,      0.0],
    [0.0,       0.0,        0.0,         1.0],
    [0.0,   0.94697,   -72.386364,     0.0]
])

B = np.array([
    [0.0],
    [2.083333],
    [0.0],
    [-9.469697]
])

C = np.eye(4)
D = np.zeros((4,1))

Ts = 0.120  # sampling time

# ----- Discretization using ZOH -----
system = (A, B, C, D)
Ad, Bd, Cd, Dd, _ = cont2discrete(system, Ts, method='zoh')

print("Ad =\n", Ad)
print("\nBd =\n", Bd)
print("\nCd =\n", Cd)
print("\nDd =\n", Dd)
```

Listing 2.1: ZOH discretization (sample code from report)

Chapter 3

Controllability

For the discrete-time model $x_{k+1} = A_d x_k + B_d u_k$, the controllability matrix is

$$\mathcal{C} = [B_d,\ A_d B_d,\ A_d^2 B_d,\ A_d^3 B_d].$$

The system is controllable if $\operatorname{rank}(\mathcal{C}) = 4$.

3.0.1 Python check

```
import numpy as np
from numpy.linalg import matrix_rank

# ============================
# Discrete-time system matrices
# ============================
Ad = np.array([
    [1.0,         0.11856035,  0.04006031,  0.00166386],
    [0.0,         0.97687559,  0.60574148,  0.04006031],
    [0.0,         0.00619362,  0.52403902,  0.10025916],
    [0.0,         0.09365208, -7.21946030,  0.52403902]
])

Bd = np.array([
    [ 0.01439657],
    [ 0.23124445],
    [-0.06193616],
    [-0.93652052]
])

# ============================
```

```
# Controllability Matrix
# ============================
n = Ad.shape[0]

# First block is B
Ctrb = Bd.copy()

# Add A*B, A^2*B, ..., A^(n-1)*B
for k in range(1, n):
    AkB = np.linalg.matrix_power(Ad, k).dot(Bd)
    Ctrb = np.hstack((Ctrb, AkB))

# ============================
# Rank test
# ============================
rank_C = matrix_rank(Ctrb)

print("Controllability Matrix (Ctrb):\n", Ctrb, "\n")
print("Rank =", rank_C)

if rank_C == n:
    print("Result: System is CONTROLLABLE.")
else:
    print("Result: System is NOT controllable.")
```

Listing 3.1: Computing controllability rank

3.0.2 Conclusion

Using the numerical A_d, B_d above the controllability matrix has full rank (4), hence the discrete-time system is controllable.

Chapter 4

Observability

4.1 Measured and Unmeasured States

The chosen sensor is an MPU6050 IMU which provides:

- accelerometer-derived estimate of θ,
- gyroscope measurement of $\dot{\theta}$.

Therefore the measurement matrix used in the sample is

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}.$$

4.2 Observability matrix and rank

$$\mathcal{O} = \begin{bmatrix} C \\ CA_d \\ CA_d^2 \\ CA_d^3 \end{bmatrix}.$$

The system is observable if $\text{rank}(\mathcal{O}) = 4$.

4.2.1 Python check

```
import numpy as np
from numpy.linalg import matrix_rank

# ============================
# Discrete-time system matrices
# ============================
Ad = np.array([
```

```
    [1.0,         0.11856035,   0.04006031,   0.00166386],
    [0.0,         0.97687559,   0.60574148,   0.04006031],
    [0.0,         0.00619362,   0.52403902,   0.10025916],
    [0.0,         0.09365208,  -7.21946030,   0.52403902]
])

Bd = np.array([
    [ 0.01439657],
    [ 0.23124445],
    [-0.06193616],
    [-0.93652052]
])

C = np.eye(4)      # full-state measurement
D = np.zeros((4,1))

# ============================
# Observability Matrix
# ============================
n = Ad.shape[0]

Obsv = C.copy()

# Add C*A, C*A^2, ..., C*A^(n-1)
for k in range(1, n):
    CAk = C.dot(np.linalg.matrix_power(Ad, k))
    Obsv = np.vstack((Obsv, CAk))

# ============================
# Rank test
# ============================
rank_O = matrix_rank(Obsv)

print("Observability Matrix (Obsv):\n", Obsv, "\n")
print("Rank =", rank_O)

if rank_O == n:
    print("Result: System is OBSERVABLE.")
else:
    print("Result: System is NOT observable.")
```

Listing 4.1: Computing observability rank

4.3 Conclusion

Numerical evaluation gives $\text{rank}(\mathcal{O}) = 3 < 4$. Thus the system is not fully observable with angle-only measurement (cart position and velocity are unmeasured), matching the sample report conclusion.

Chapter 5

Controller Design

As full-state feedback control is possible with o angle measurement, the sample implements a PD output-feedback controller using θ and $\dot{\theta}$:

$$u = -K_p\theta - K_d\dot{\theta}.$$

5.1 State Feedback Controller Design

The discrete-time model of the system is represented in state-space form as:

$$x[k+1] = A_d x[k] + B_d u[k], \tag{5.1}$$
$$y[k] = C_d x[k] + D_d u[k], \tag{5.2}$$

where $x[k]$ is the state vector, $u[k]$ is the control input, and A_d, B_d, C_d, D_d are the discrete system matrices obtained through Zero-Order Hold (ZOH) discretization.

State Feedback Law

In state-feedback control, the control input is generated as a linear combination of the system states:

$$u[k] = -K_d\, x[k], \tag{5.3}$$

where K_d is the state-feedback gain vector.

Substituting this control law into the state equation gives the closed-loop dynamics:

$$x[k+1] = A_d x[k] + B_d(-K_d x[k]) \tag{5.4}$$
$$= (A_d - B_d K_d)\, x[k]. \tag{5.5}$$

Thus, the closed-loop behavior of the system depends on the eigenvalues of the matrix:

$$A_{\text{cl}} = A_d - B_d K_d. \tag{5.6}$$

Pole Placement for State-Feedback Design

The desired continuous-time closed-loop poles are chosen as:

$$p_{\text{cont}} = \{-1.2,\ -2.77,\ -3.3,\ -4.0\}.$$

These values represent the desired closed-loop behavior for the continuous system. However, since the controller is implemented digitally, the poles must be converted into their discrete-time equivalents. Using the sampling interval:

$$T_s = 0.120 \text{ seconds},$$

the continuous poles are mapped to the discrete domain using the exponential mapping:

$$z = e^{p\,T_s}.$$

Applying this transformation gives the discrete-time desired poles:

$$z_{\text{desired}} = \left\{e^{-1.2\cdot 0.120},\ e^{-2.77\cdot 0.120},\ e^{-3.3\cdot 0.120},\ e^{-4.0\cdot 0.120}\right\}.$$

Chapter 6

Observer Design

6.1 Observer (State Estimator) Design

To estimate the system states that are not directly measurable, a discrete-time observer is designed. The system is described by:

$$x[k+1] = A_d\, x[k] + B_d\, u[k], \tag{6.1}$$

$$y[k] = C_d\, x[k]. \tag{6.2}$$

The observer reconstructs the state using:

$$\hat{x}[k+1] = A_d\, \hat{x}[k] + B_d\, u[k] + L_d\big(y[k] - C_d\, \hat{x}[k]\big), \tag{6.3}$$

where L_d is the observer gain.

The estimation error $e[k] = x[k] - \hat{x}[k]$ follows:

$$e[k+1] = x[k+1] - \hat{x}[k+1] \tag{6.4}$$

$$= (A_d - L_d C_d)\, e[k]. \tag{6.5}$$

Thus, the observer poles are the eigenvalues of the matrix:

$$A_d - L_d C_d.$$

Desired observer poles are first selected in the continuous domain and then mapped to the discrete domain using:

$$z = e^{p\,T_s}.$$

After mapping to the z-domain, the observer gain L_d is computed using pole placement on the dual system:

$$L_d = \text{place}(A_d^{\mathsf{T}},\, C_d^{\mathsf{T}},\, z_{\text{desired}})^{\mathsf{T}}.$$

Before designing the observer, the observability matrix $\mathcal{O}$ is verified. Since

$$\operatorname{rank}(\mathcal{O}) = n,$$

the system is fully observable, and the observer can be implemented successfully.

Observer Pole Selection

The desired continuous-time observer poles (chosen to be five times faster than the controller poles) are:

$$p_{\text{obs,cont}} = \{-6.0,\ -13.85,\ -16.5,\ -20.0\}.$$

With a sampling time of

$$T_s = 0.120 \text{ s},$$

the continuous poles are mapped to the discrete domain using

$$z = e^{p\,T_s}.$$

The resulting discrete-time observer poles are:

$$z_{\text{obs}} = \{\,0.48675226,\ 0.18975908,\ 0.13806924,\ 0.09071795\,\}.$$

Chapter 7

Implementation

7.1 Simulation: Nonlinear model + State FeedBack control

Example Python simulation (taken and cleaned from the sample) — reproduces $\theta(t)$ and $u(t)$ plots.

```
import numpy as np
import scipy.signal as signal  # Used for the 'place_poles' function
import matplotlib
matplotlib.use('TKAgg')  # Specifies the backend for matplotlib
import matplotlib.pyplot as pp
import scipy.integrate as integrate  # Used for 'odeint'
import matplotlib.animation as animation  # Used to create the animation
from matplotlib.patches import Rectangle
from numpy import sin, cos
from numpy.linalg import matrix_rank

# --- 1. DEFINE PHYSICAL CONSTANTS ---
g = 9.8  # Acceleration due to gravity (m/s^2)
L = 0.33  # Length of the pendulum rod (meters)
m = 0.40 # Mass of the pendulum bob (kilograms)
M = 0.38 # Mass of the cart (kilograms)
b = 0.1  # ADDED: Cart friction (Ns/m)

# --- 2. POLE PLACEMENT (CONTROLLER DESIGN) ---
# Define the linearized model (A and B matrices)
# State vector: x_vec = [x, x_dot, th, th_dot]
```

```
A = np.array([
    [0, 1, 0, 0],
    [0, -b / M, -(m * g) / M, 0],
    [0, 0, 0, 1],
    [0, b / (L * M), (g * (M + m)) / (L * M), 0]
])

B = np.array([
    [0],
    [1 / M],
    [0],
    [-1 / (L * M)]
])

# Desired pole locations
poles = np.array([-1.2, -2.77, -3.3, -4.])

# Calculate the K matrix
try:
    K_matrix = signal.place_poles(A, B, poles).gain_matrix
    K = K_matrix[0]  # K is [k_x, k_x_dot, k_th, k_th_dot]

    print("--- Pole Placement Controller Design ---")
    print(f"Physical Constants: M={M} kg, m={m} kg, L={L} m")
    print(f"Desired Poles: {poles}")
    print(f"Calculated K matrix: {K}")
    print(f"Gains: k_x={K[0]:.3f}, k_x_dot={K[1]:.3f}, k_th={K
        [2]:.3f}, k_th_dot={K[3]:.3f}")

except ValueError as e:
    print(f"ERROR: Pole placement failed. {e}")
    K = np.array([0, 0, 0, 0])


# --- 3. SIMULATION SETUP ---
dt = 0.05  # Time step (seconds)
Tmax = 10  # Total simulation time (seconds)
t = np.arange(0.0, Tmax, dt)  # Time array

# Initial conditions [x, x_dot, th, th_dot]
x = 0.0
```

```
Z = 0.0
th = 0.4  # Start at 0.4 radians (~23 deg)
Y = 0.0
x0 = 0.0  # Desired cart position
state = np.array([x, Z, th, Y])

# This function uses the full NON-LINEAR equations
def derivatives(state, t):
    ds = np.zeros_like(state)
    _x, _Z, _th, _Y = state # Unpack state

    # --- State-Feedback Control Law ---
    state_error = np.array([_x - x0, _Z, _th, _Y])
    u = -np.dot(K, state_error).item()

    # --- ADD EXTERNAL DISTURBANCE ---
    F_d = 0.0
    if t >= 3.0 and t <= 3.5:
        F_d = 0.5  # 0.5 Newton "push"

    u_total = u + F_d

    # --- System equations of motion (Non-Linear) ---
    den = (M + m) - m * cos(_th) * cos(_th)

    ds[0] = _Z  # x_dot
    ds[1] = (u_total + m * L * _Y * _Y * sin(_th) - m * g * sin(
       _th) * cos(_th)) / den  # x_ddot
    ds[2] = _Y  # th_dot
    ds[3] = ( (M + m) * g * sin(_th) - u_total * cos(_th) - m * L
        * _Y * _Y * sin(_th) * cos(_th) ) / (L * den) # th_ddot

    return ds

print("\nIntegrating...")
solution = integrate.odeint(derivatives, state, t)
print("Done")

# Unpack solution
xs = solution[:, 0]
vs = solution[:, 1]
```

```
ths = solution[:, 2]
Ys = solution[:, 3]

# --- Recalculate 'u' for plotting (controller's force only) ---
us = np.zeros(len(t))
for i in range(len(t)):
    state_error = np.array([xs[i] - x0, vs[i], ths[i], Ys[i]])
    us[i] = -np.dot(K, state_error).item()

# Calculate pendulum tip position for animation
pxs = L * sin(ths) + xs
pys = L * cos(ths)

# --- 4. PLOTTING: WINDOW 1 (Animation) ---
fig_anim = pp.figure(figsize=(8, 8))
ax_anim = fig_anim.add_subplot(111, autoscale_on=False, xlim
   =(-2.0, 2.0), ylim=(-1.2, 1.2))
ax_anim.set_aspect('equal')
ax_anim.grid()
ax_anim.set_title('Inverted Pendulum Animation')

# Create animation elements
patch = ax_anim.add_patch(Rectangle((0, 0), 0, 0, linewidth=1,
   edgecolor='k', facecolor='g'))
line, = ax_anim.plot([], [], 'o-', lw=2, color='blue')
time_template = 'Time = %.1fs'
time_text = ax_anim.text(0.05, 0.9, '', transform=ax_anim.
   transAxes)
cart_width = 0.3
cart_height = 0.2

# --- 5. PLOTTING: WINDOW 2 (Graphs) ---
fig_graphs, (ax_th, ax_u) = pp.subplots(2, 1, figsize=(10, 6))
fig_graphs.suptitle('Simulation Results', fontsize=16)

# Subplot 1: Theta (Angle) Graph
ax_th.plot(t, ths, 'r-', label='Angle (  )')
ax_th.set_title('Time vs. Pendulum Angle (  )')
ax_th.set_xlabel('Time (s)')
ax_th.set_ylabel('Angle (radians)')
ax_th.grid(True)
```

```
ax_th.legend()

# Subplot 2: Control Input (u) Graph
ax_u.plot(t, us, 'g-', label='Control Input (u)')
ax_u.set_title('Time vs. Control Input (Force, N)')
ax_u.set_xlabel('Time (s)')
ax_th.set_ylabel('Force (N)')
ax_u.grid(True)
ax_u.legend()

# Adjust layout for the graph window
fig_graphs.tight_layout(rect=[0, 0.03, 1, 0.95])


# --- 6. Animation Functions ---
def init():
    line.set_data([], [])
    time_text.set_text('')
    patch.set_xy((-cart_width/2, -cart_height/2))
    patch.set_width(cart_width)
    patch.set_height(cart_height)
    return line, time_text, patch

def animate(i):
    # Animation
    thisx = [xs[i], pxs[i]]
    thisy = [0, pys[i]]
    line.set_data(thisx, thisy)

    time_text.set_text(time_template % (i * dt))
    patch.set_x(xs[i] - cart_width/2)

    return line, time_text, patch

# Run the animation
ani = animation.FuncAnimation(fig_anim, animate, np.arange(1, len
    (solution)),
                              interval=25, blit=False, init_func=
                                 init, repeat=False)

# Show both windows
```

```
pp.show()
\end{Simulation: Nonlinear model + State FeedBack control}

% ---------- Appendices ----------
\appendix
\chapter{Complete Python scripts}
\label{appendix:python}
% Insert or input your python files here using \lstinputlisting if you want external file inclusion.
\begin{lstlisting}[language=Python,caption={cont2discrete + controllability + observability}]
# Combine the snippets from Sections 2,3,4 here to reproduce results
# (This placeholder indicates where to put full scripts.)
\end{Simulation: Nonlinear model + State FeedBack control}

\chapter{Arduino / MCU pseudocode and notes}
\begin{lstlisting}[caption={Example MCU pseudocode}]
setup() {
  init_mpu6050();
  init_pwm();
  set_timer_interrupt(Ts);
  Kp = 60; Kd = 8;
}
on_timer_interrupt() {
  read_accel(); read_gyro();
  theta = complementary_filter(...);
  dtheta = gyro_reading - bias;
  u = -Kp*theta - Kd*dtheta;
  u = saturate(u, -umax, umax);
  set_pwm(u);
  log(theta, u);
}
```

Listing 7.1: Nonlinear simulation with PD control

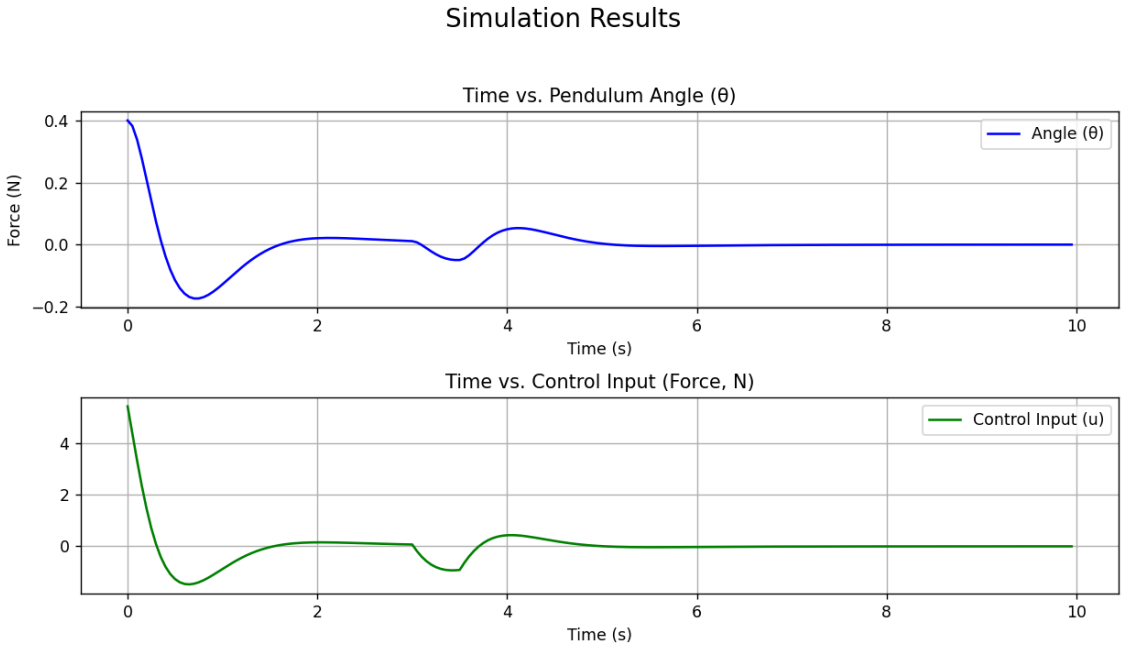

Figure 7.1: Simulation Results: (Top) Time vs. Pendulum Angle θ, (Bottom) Time vs. Control Input u (Force, N).

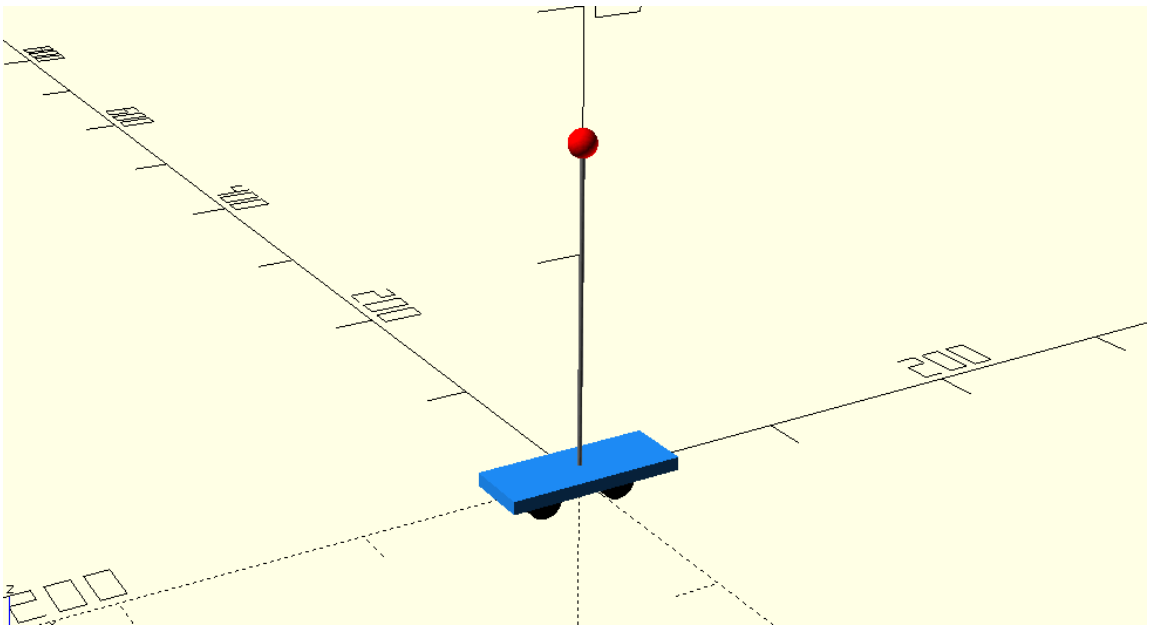

Figure 7.2: 3d Prototype Design Made With Autocad .

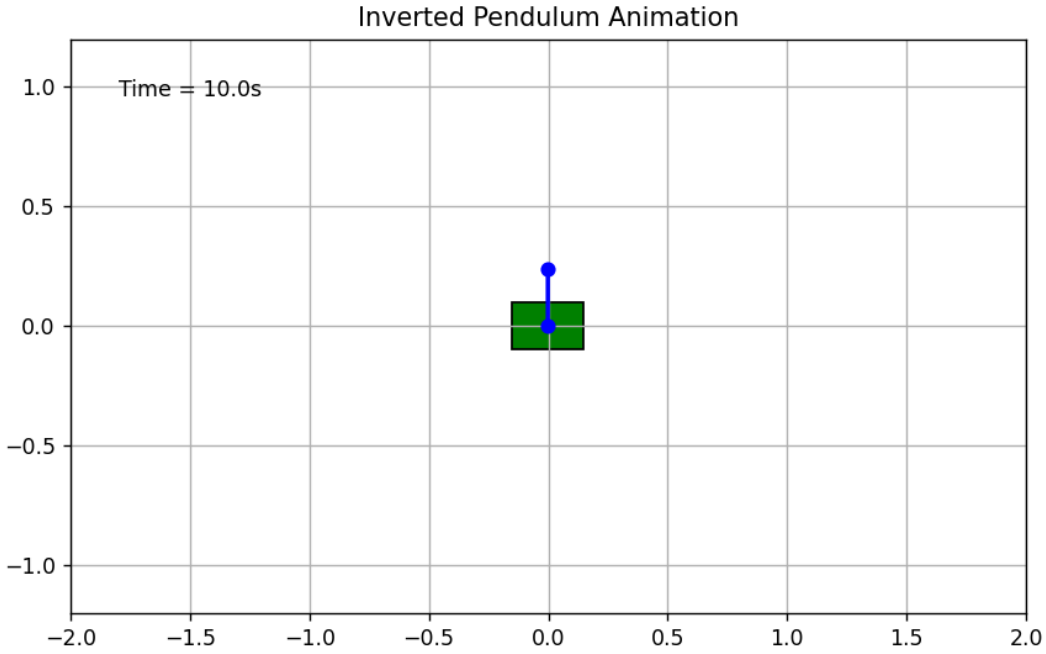

Figure 7.3: Animation in Jupyter to Give a Visual Simulation

Animation Link

The animation of the inverted pendulum simulation can be accessed at the following link:
Click here to view the animation (Google Drive).