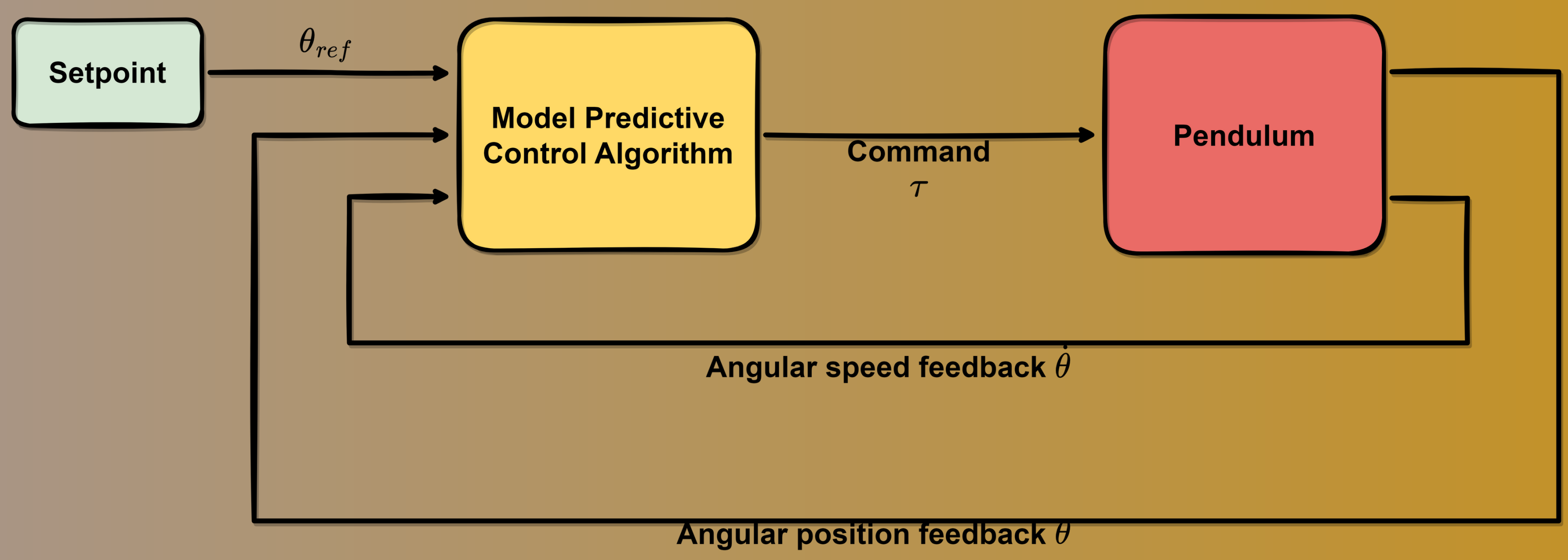
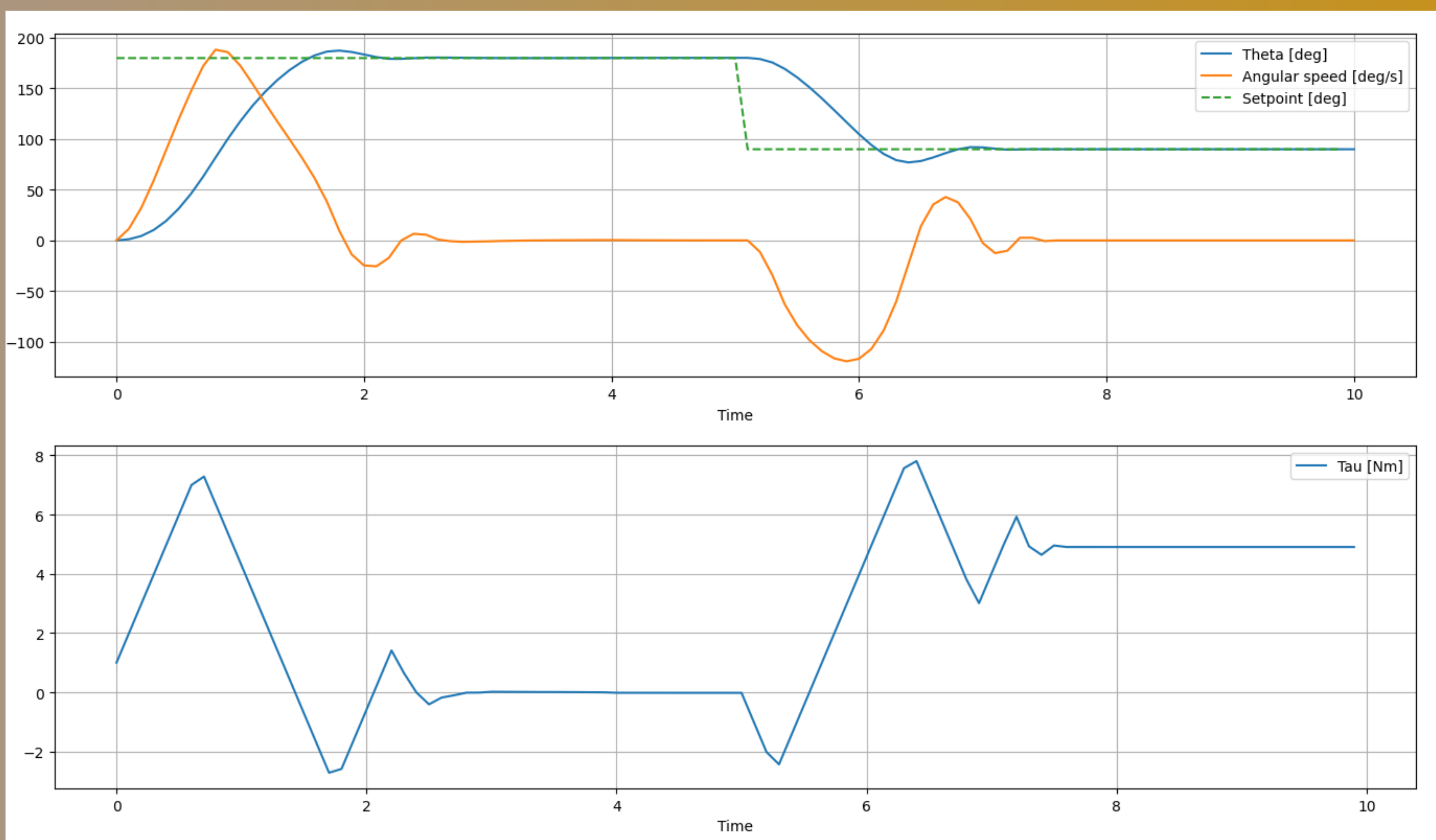
Model Predictive Control




```python
# Solve MPC optimization problem
def solve_mpc(theta_ref, theta, dtheta, tau_ini, l, k, m, g, dt, Q11, Q22, R, N, tau_max, delta_tau_max):

    # Linear constraints on the rate of change of tau: -delta_tau_max <= tau[i+1] - tau[i] <= delta_tau_max for all i in the N - 1
    # Implemented using LinearConstraint as -delta_tau_max <= delta_tau_matrix * tau <= delta_tau_max
    delta_tau_matrix = np.eye(N) - np.eye(N, k=1)
    constraint1 = LinearConstraint(delta_tau_matrix, -delta_tau_max, delta_tau_max)

    # We need a constraint on the rate of change of tau[0] respect to its previous value, which is tau_ini[0]
    first_element_matrix = np.zeros([N, N])
    first_element_matrix[0, 0] = 1
    constraint2 = LinearConstraint(first_element_matrix, tau_ini[0]-delta_tau_max, tau_ini[0]+delta_tau_max)

    # Add constraints
    delta_tau_constraint = [constraint1, constraint2]

    # Bounds --> -tau_max <= tau[idx] <= tau_max for idx = 0 to N-1
    bounds = [(-tau_max, tau_max) for idx in range(N)]

    # Starting optimisation point for theta and dtheta are the current measurements
    theta0 = theta
    dtheta0 = dtheta

    # Minimization
    result = minimize(mpc_cost, tau_ini, args=(theta_ref, theta0, dtheta0, l, k, m, g, dt, Q11, Q22, R, N), bounds=bounds, constraints=delta_tau_constraint)

    # Extract the optimal control sequence
    tau_mpc = result.x

    return tau_mpc
```



Simone Bertoni

Theory

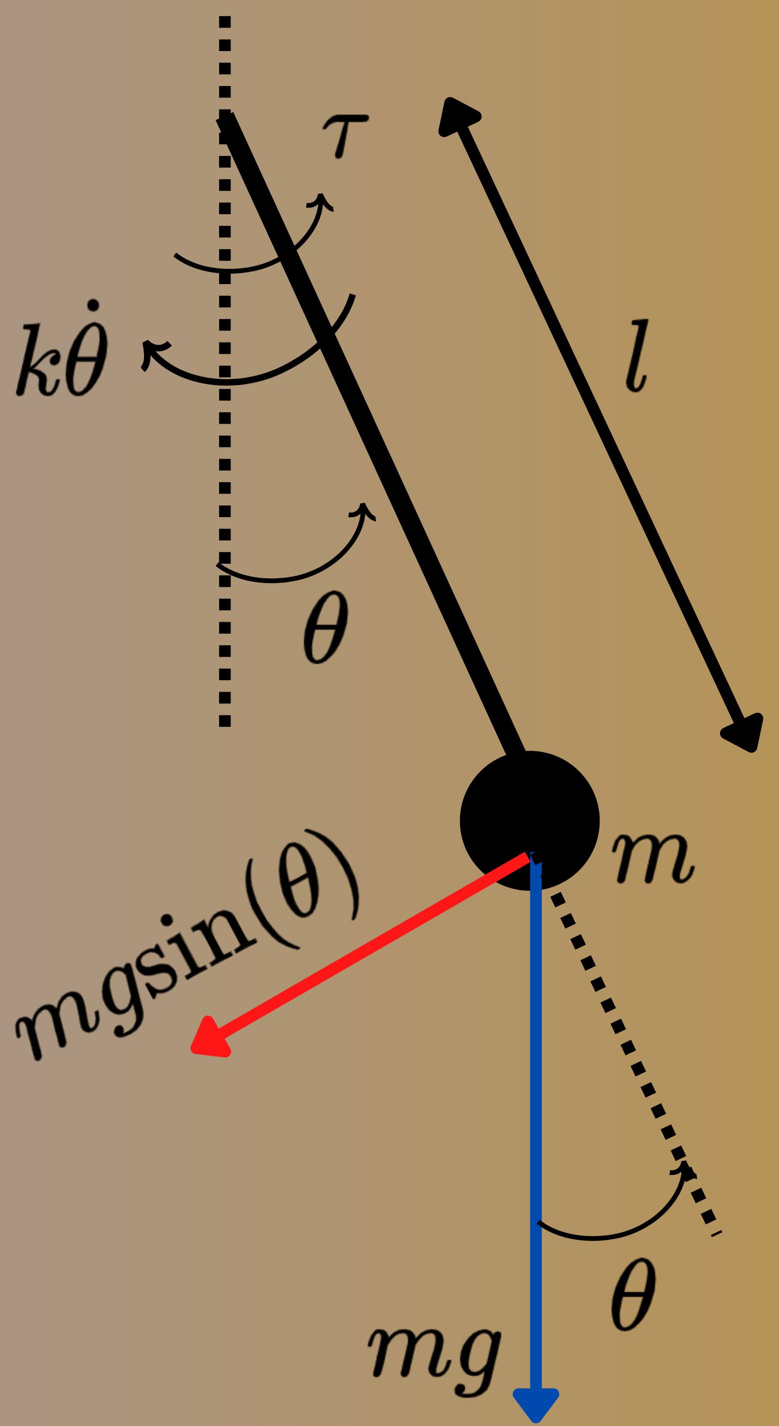
The idea is to find the optimal control sequence over a control horizon of N steps that minimises a cost function, for example:

$$J = \sum_{i=1}^{N-1} Q(r_i - y_i)^2 + Ru_i^2$$

where:

r: setpoint
y: controlled variable
r: command
Q: weighting coefficient related to y
R: weighting coefficient related to u

Plant: pendulum



$m = 0.5\ kg$

$l = 1\ m$

$k = 0.5\ Nms$

$$\tau = ml^2\ddot{\theta} + k\dot{\theta} + mglsin(\theta)$$

Simone Bertoni

MPC algorithm

Discretised plant model:

$$\ddot{\theta}(i+1) = \frac{\tau - k\dot{\theta}(i) - mglsin(\theta(i))}{ml^2}$$
$$\dot{\theta}(i+1) = \dot{\theta}(i) + dt\,\ddot{\theta}(i+1)$$
$$\theta(i+1) = \theta(i) + dt\,\dot{\theta}(i+1)$$

Cost function:

$$J = \sum_{i=1}^{N-1} Q_{11}\dot{\theta}_i^2 + Q_{22}(\theta_{ref_i} - \theta_i)^2 + R\tau_i^2$$

Constraints:

Max torque: $-\tau_{max} < \tau_i < \tau_{max}$

Torque rate of change: $-\Delta\tau_{max} < \tau_{i+1} - \tau_i < \Delta\tau_{max}$

Simone Bertoni

Python code – 1

```python
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.optimize import minimize, LinearConstraint

# Step of the pendulum system
def pendulum_step(theta, dtheta, tau, l, k, m, g, dt):

    # Calculate the change in angular acceleration
    ddtheta = (tau - k * dtheta - m * g * l * math.sin(theta)) / (m * l * l)

    # Update the angular velocity and angle
    dtheta_next = dtheta + ddtheta * dt
    theta_next = theta + dtheta_next * dt

    return (theta_next, dtheta_next)

# Cost function to be minimized
def mpc_cost(tau, theta_ref, theta0, dtheta0, l, k, m, g, dt, Q11, Q22, R, N):

    # Initialise cost = 0 and states to current measured states
    cost = 0
    theta = theta0
    dtheta = dtheta0

    # Simulate the next N steps of the system
    for idx in range(N):

        # Calculate the change in angular acceleration
        ddtheta = (tau[idx] - k * dtheta - m * g * l * math.sin(theta)) / (m * l * l)

        # Update the angular velocity and angle
        dtheta = dtheta + ddtheta * dt
        theta = theta + dtheta * dt

        # Update cost
        cost += Q11 * dtheta**2 + Q22 * (theta_ref - theta)**2 + R * tau[idx]**2

    return cost

# Solve MPC optimization problem
def solve_mpc(theta_ref, theta, dtheta, tau_ini, l, k, m, g, dt, Q11, Q22, R, N, tau_max, delta_tau_max):

    # Linear constraints on the rate of change of tau: -delta_tau_max <= tau[i+1] - tau[i] <= delta_tau_max for all i in the N - 1
    # Implemented using LinearConstraint as -delta_tau_max <= delta_tau_matrix * tau <= delta_tau_max
    delta_tau_matrix = np.eye(N) - np.eye(N, k=1)
    constraint1 = LinearConstraint(delta_tau_matrix, -delta_tau_max, delta_tau_max)

    # We need a constraint on the rate of change of tau[0] respect to its previous value, which is tau_ini[0]
    first_element_matrix = np.zeros([N, N])
    first_element_matrix[0, 0] = 1
    constraint2 = LinearConstraint(first_element_matrix, tau_ini[0]-delta_tau_max, tau_ini[0]+delta_tau_max)

    # Add constraints
    delta_tau_constraint = [constraint1, constraint2]

    # Bounds --> -tau_max <= tau[idx] <= tau_max for idx = 0 to N-1
    bounds = [(-tau_max, tau_max) for idx in range(N)]

    # Starting optimisation point for theta and dtheta are the current measurements
    theta0 = theta
    dtheta0 = dtheta

    # Minimization
    result = minimize(mpc_cost, tau_ini, args=(theta_ref, theta0, dtheta0, l, k, m, g, dt, Q11, Q22, R, N), bounds=bounds, constraints=delta_tau_constraint)

    # Extract the optimal control sequence
    tau_mpc = result.x

    return tau_mpc
```

Simone Bertoni

Python code – 2

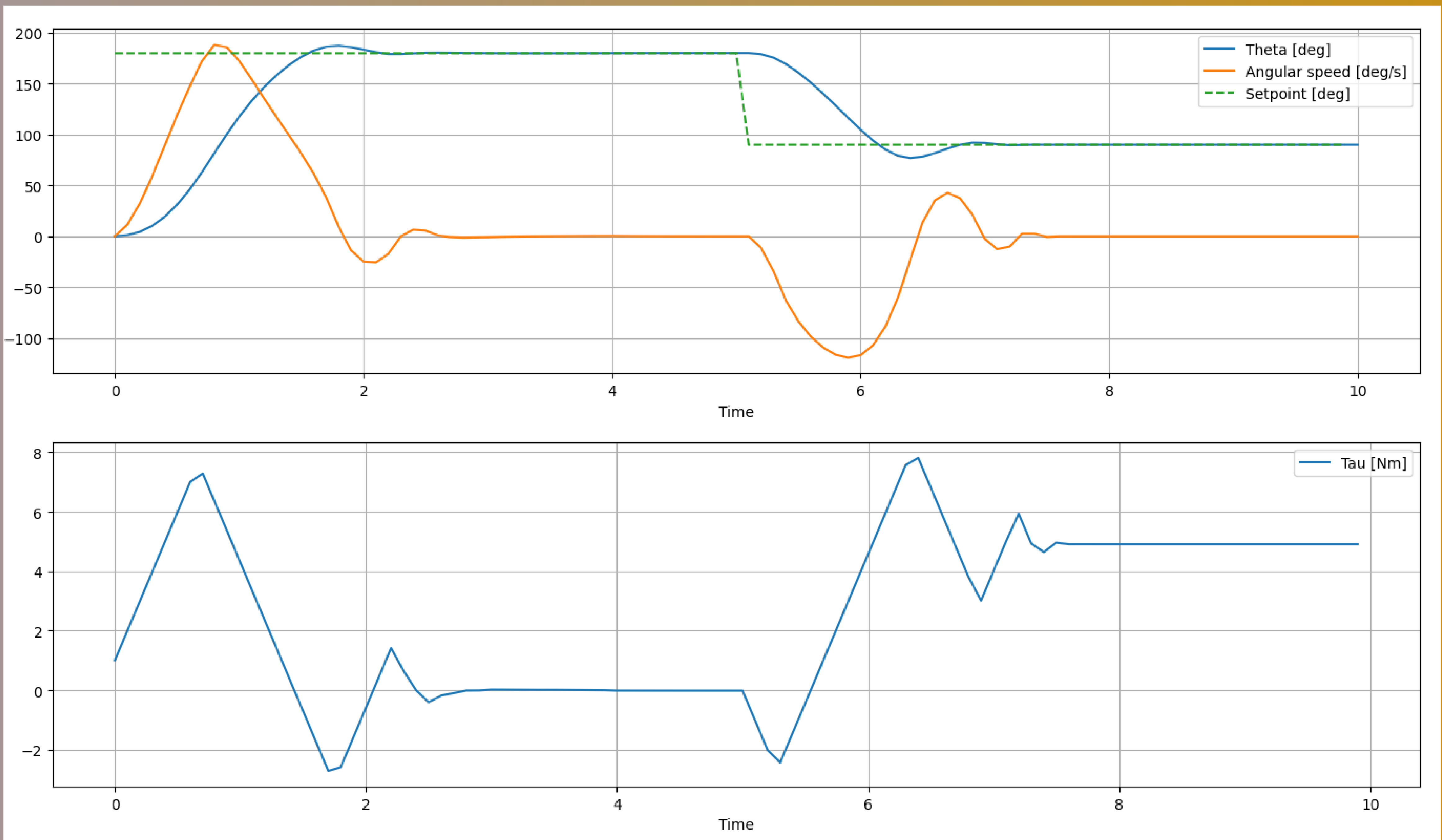
```
# ---------- SIMULATION INITIALISATION ----------

# Plant parameters
l = 1  # length of the pendulum
k = 0.5  # coefficient of friction
m = 0.5  # mass of the pendulum
g = 9.81  # acceleration due to gravity

# Time step
dt = 0.1

# Simulation time
time_range = 10

# Simulation steps
L = round(time_range/dt)

# Init time
time = 0

# Initial state
theta0 = 0
dtheta0 = 0

# Arrays for logging
theta = np.zeros(L + 1)
dtheta = np.zeros(L + 1)
tau = np.zeros(L)
theta_ref = np.zeros(L)

# Init arrays to initial state
theta[0] = theta0
dtheta[0] = dtheta0

# ---------- CONTROL SYSTEM CALIBRATION ----------

# Model predictive control horizon
N = 20

# Cost weights
Q11 = 0 # angulare speed ignored
Q22 = 1
R = 0 # control input ignored - constraints are in place

# Max torque allowed
tau_max = 10

# Max delta torque between two steps
delta_tau_max = 1

# Plant parameters - estimated
l_est = 1  # length of the pendulum
k_est = 0.5  # coefficient of friction
m_est = 0.5  # mass of the pendulum
g_est = 9.81  # acceleration due to gravity

# Array for initial solution for MPC
tau_ini = np.zeros(N)
```

Simone Bertoni

Python code – 3

```python
# ---------- SIMULATION ----------

for idx in range(L):

    # ---------- SIMULATE USER INPUT ----------

    # Generate reference setpoint
    if time < 5:
        theta_ref[idx] = math.pi # 180 deg
    else:
        theta_ref[idx] = math.pi*0.5 # 90 deg

    # Increment time
    time += dt

    # ---------- CONTROL SYSTEM LOOP ----------

    # Find optimal sequence for the next N steps
    tau_mpc = solve_mpc(theta_ref[idx], theta[idx], dtheta[idx], tau_ini, l_est, k_est, m_est, g_est, dt, Q11, Q22, R, N, tau_max, delta_tau_max)

    # Use first element of control input optimal solution
    tau[idx] = tau_mpc[0]

    # Initial solution for next step = current solution
    tau_ini = tau_mpc

    # ---------- SIMULATION LOOP  ----------

    # Run dynamic system - pendulum
    (theta[idx+1], dtheta[idx+1]) = pendulum_step(theta[idx], dtheta[idx], tau[idx], l, k, m, g, dt)

# Plot results

plt.subplot(2, 1, 1)
plt.plot(np.arange(L+1)*dt, theta[:]*180/math.pi, label="Theta [deg]")
plt.plot(np.arange(L+1)*dt, dtheta[:]*180/math.pi, label="Angular speed [deg/s]")
plt.plot(np.arange(L)*dt, theta_ref*180/math.pi, '--', label="Setpoint [deg]")
plt.xlabel("Time")
plt.legend()
plt.grid()

plt.subplot(2, 1, 2)
plt.plot(np.arange(L)*dt, tau, label="Tau [Nm]")
plt.xlabel("Time")
plt.legend()
plt.grid()
plt.show()
```

Simulation



Simone Bertoni