Universiteit Antwerpen
Faculteit Wetenschappen

Modelling Of Software Intensive Systems

Assignment 1: Modelica

1st Master computer science
2024-2025

Liam Leirs
Robbe Teughels

Plant Model Creation:

We created a model of a plant that has a trolly with a rope attached to it. The goal of this model is to evaluate both the pendulum and the trolly. More specific, the (angular) speed and the displacement. This can be tuned by the Damping factors *Dp* and *Dc*. For controlling the model, we use the control signal *u* that gives the desired displacement.

We tested the model with two intuitive tests: no displacement, set displacement. When we require the cart to stay fixed in place, the entire model stays fixed as we would expect. When we set the control system to a desired location, we see the cart move to the exact location, given some time. The pendulum reacts on the movement of the cart as required. Given these tests, we assume that the model is representative to the real object for velocity and displacement.

```
model pendulum
  // Types
  type Factor = Real(unit = "");
  // Parameters
  parameter Modelica.Units.SI.Mass m = 0.2;     // Mass of the pendulum, (kg)
  parameter Modelica.Units.SI.Mass M = 10;      // Mass of the cart, (kg)
  parameter Modelica.Units.SI.Length r = 1;     // Length of rope, (m)
  parameter Factor dp = 0.12;                   // Damping factor of the pendulum, no unit
  parameter Factor dc = 4.79;                   // Damping factor of the cart, no unit
  constant Modelica.Units.SI.Acceleration g = 9.80665;  // Gravitational acceleration on Earth, (m/(s^2))
  // Variables
  Modelica.Units.SI.Length x;                   // displacement of the cart, (m)
  Modelica.Units.SI.Velocity v;                 // Velocity of the cart, (m/s)
  Modelica.Units.SI.Angle th;                   // Angular displacement of the pendulum, (rad)
  Modelica.Units.SI.AngularVelocity ohm;        // Angular velocity of the pendulum, (rad/s)
  Real u;                                       // Control signal to move the cart, no unit

  initial equation
  x=0;
  v=0;
  th=0;
  ohm=0;


  equation
  if time < 0.5 then u = 100; else u = 0; end if;

  der(x) = v;
  der(th) = ohm;
  der(v) = (r*(dc*v - m*(g*sin(th)*cos(th) + r*sin(th)*ohm^2) -u)-(dp*cos(th))*ohm) / (-r*(M+m*sin(th)^2));
  der(ohm) = (dp*ohm*(m+M)+(m^2*r^2*sin(th)*cos(th)*ohm^2) + m*r*(g*sin(th)*(m+M)) + (cos(th)*(u-dc*v))) / ((m*r^2)*(-M-(m*sin(th)^2)));


end pendulum;
```

Plant Model Calibration

Now that the plant model was created, it still needed to be calibrated (via the damping factors Dp & Dc). To do this, 2 new models had to be created where the effects of one of these factors is eliminated allowing us to estimate the other one.

For estimating Dc we will “lock” the pendulum essentially reeling it in:

```
model calibration_dc
  //Types
  type Factor = Real(unit = "");
  // Parameters
  parameter Modelica.Units.SI.Mass M = 10; // Mass of the cart, (kg)
  parameter Factor dc = 2; // Damping factor of the cart, no unit
  // Variables
  Modelica.Units.SI.Length x; // displacement of the cart, (m)
  Modelica.Units.SI.Velocity v; // Velocity of the cart, (m/s)
initial equation
  x = 0;
  v = 5;
equation
  der(x) = v;
  der(v) = -(dc/M)*v;
end calibration_dc;
```

For estimating Dp the trolley’s movement will be locked:

```
model calibration_dp
  // Types
  type Factor = Real(unit = "");
  // Parameters
  parameter Modelica.Units.SI.Mass m = 0.2; // Mass of the pendulum, (kg)
  parameter Modelica.Units.SI.Length r = 1; // Length of rope, (m)
  parameter Factor dp = 0.5; // Damping factor of the pendulum, no unit
  constant Modelica.Units.SI.Acceleration g = 9.80665; // Gravitational acceleration on Earth, (m/(s^2))

  // Variables
  Modelica.Units.SI.Angle th; // Angular displacement of the pendulum, (rad)
  Modelica.Units.SI.AngularVelocity ohm; // Angular velocity of the pendulum, (rad/s)
initial equation
  th = Modelica.Constants.pi/6;
  ohm = 0;
equation
  der(th) = ohm;
  der(ohm) = -((dp*ohm) + (m*g*r*sin(th)))/(m*r^2);
end calibration_dp;
```

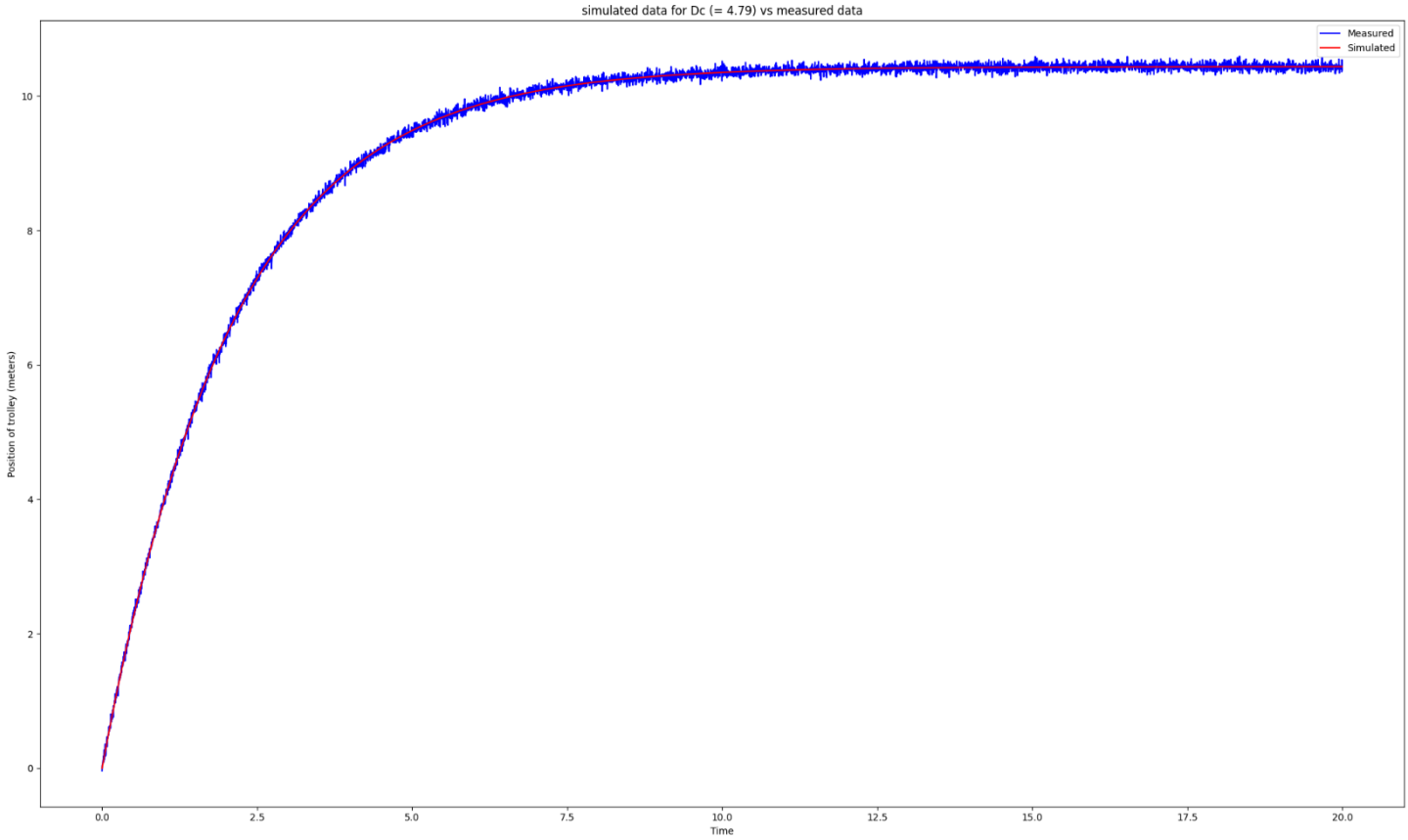
To calibrate these factors, we will use a collection of real-life measured data to compare our simulated data to. The damping coefficient Dp & Dc can take on values in the interval (0, 5] (precise up to 2 decimals). For each possible value the model will be simulated compared to the real-life data by calculating the sum of squared errors. The simulation with the smallest error will be the best one.

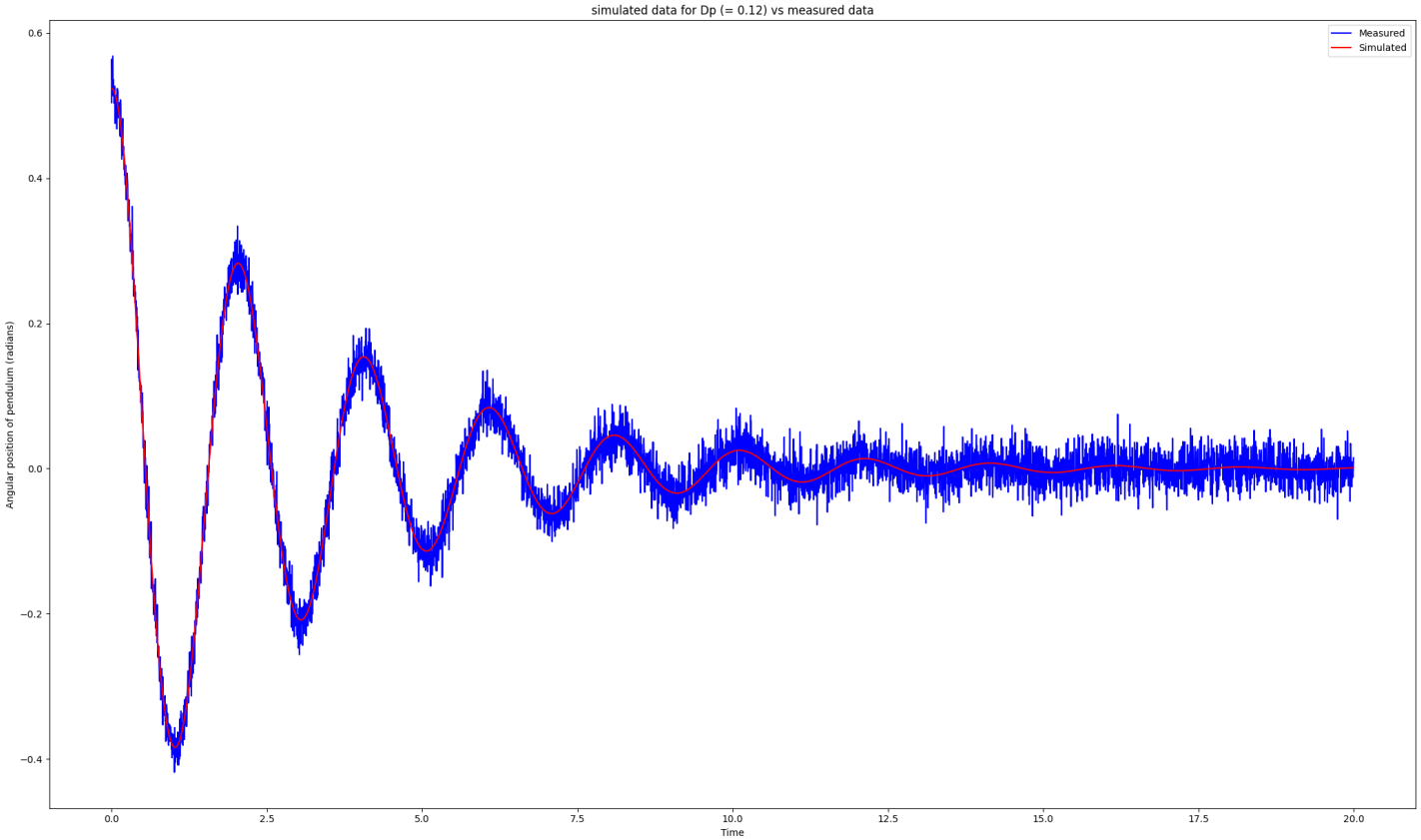
Code for tuning the parameters (parameter_tuning.py):

```python
from simulate import singleSimulation
import pandas as pd
import numpy as np
from matplotlib import pyplot
import os

def compute_vals(param: str):
    # If results already generated return
    if os.path.exists(param + "_results"):
        return
    os.mkdir(param + "_results")
    data = pd.read_csv("calibration_data_" + param[0] + "_" + param[1] + ".csv", names=["measured"])
    data = data.drop(index=data.index[0], axis=0)
    # For each possible value simulate & store alongside measured values
    for val in np.arange(0, 5, 0.01).round(2):
        if val == 0:
            continue
        values = singleSimulation("pendulum/Assignment1.calibration_" + str(param), "calibration_" + str(param), {param: val})
        if param == "dp":
            th_vals = values["th"]
        else:
            th_vals = values["x"]
        simulation_df = pd.DataFrame(th_vals, columns=["simulation"])
        data["simulation"] = simulation_df["simulation"]
        data.to_csv(param + "_results/" + param + "_" + str(val))

def sum_of_squared_errors(param: str):
    # Loops through all values of param and calculates
    # sum of squared errors for each one
    # => smallest one is considered best
    os.chdir(param + "_results")
    min = float("+inf")
    best = 0.01
    for val in np.arange(0, 5, 0.01).round(2):
        if val == 0:
            continue
        df = pd.read_csv(param + "_" + str(val))
        measured = np.array(df["measured"].tolist())
        simulation = np.array(df["simulation"].tolist())
        sum = np.sum((measured-simulation)**2)
        if sum < min:
            best = val
            min = sum
    os.chdir("..")
    return best

def calibrate_dp():
    compute_vals("dp")
    return sum_of_squared_errors("dp")
```

```python
def calibrate_dc():
    compute_vals("dc")
    return sum_of_squared_errors("dc")

if __name__ == "__main__":
    time_vals = np.arange(0, 20, 0.004).round(3)
    dc = calibrate_dc()
    dc_df = pd.read_csv("dc_results/dc_" + str(dc))
    dc_measured = dc_df["measured"].to_list()[:5000]
    dc_simulated = dc_df["simulation"].to_list()[:5000]
    pyplot.plot(time_vals, dc_measured, label="Measured", color="blue")
    pyplot.plot(time_vals, dc_simulated, label="Simulated", color="red")
    pyplot.xlabel("Time")
    pyplot.ylabel("Values")
    pyplot.legend()
    pyplot.show()
    dp = calibrate_dp()
    dp_df = pd.read_csv("dp_results/dp_" + str(dp))
    dp_measured = dp_df["measured"].to_list()[:5000]
    dp_simulated = dp_df["simulation"].to_list()[:5000]
    pyplot.plot(time_vals, dp_measured, label="Measured", color="blue")
    pyplot.plot(time_vals, dp_simulated, label="Simulated", color="red")
    pyplot.xlabel("Time")
    pyplot.ylabel("Values")
    pyplot.legend()
    pyplot.show()
    print(round(dp, 2))
    print(round(dc,2))
```

The above code gave the values 0.12 for Dp and 4.79 for Dc. Plotting the simulation data alongside the measured data for these values generates the following plots:





Controller Model Creation

Our final goal is to make a controller for the physical plant. We can use our finetuned plant and contain it within a block for a simple interface with in and puts that we also have in the real plant.

Our controlees will be a PID-controller. We define the core of the controller as a block so its ready for later. It will use the desired displacement of the plant and the actual displacement to smoothen/optimize the movement.

We have all our blocks ready to create a fully functional plant. By connecting the plant and the controller with the right in and outputs we have a controller controlling the plant. We can change the behavior of the controller by Kp, Ki, Kd of the PID-controller.

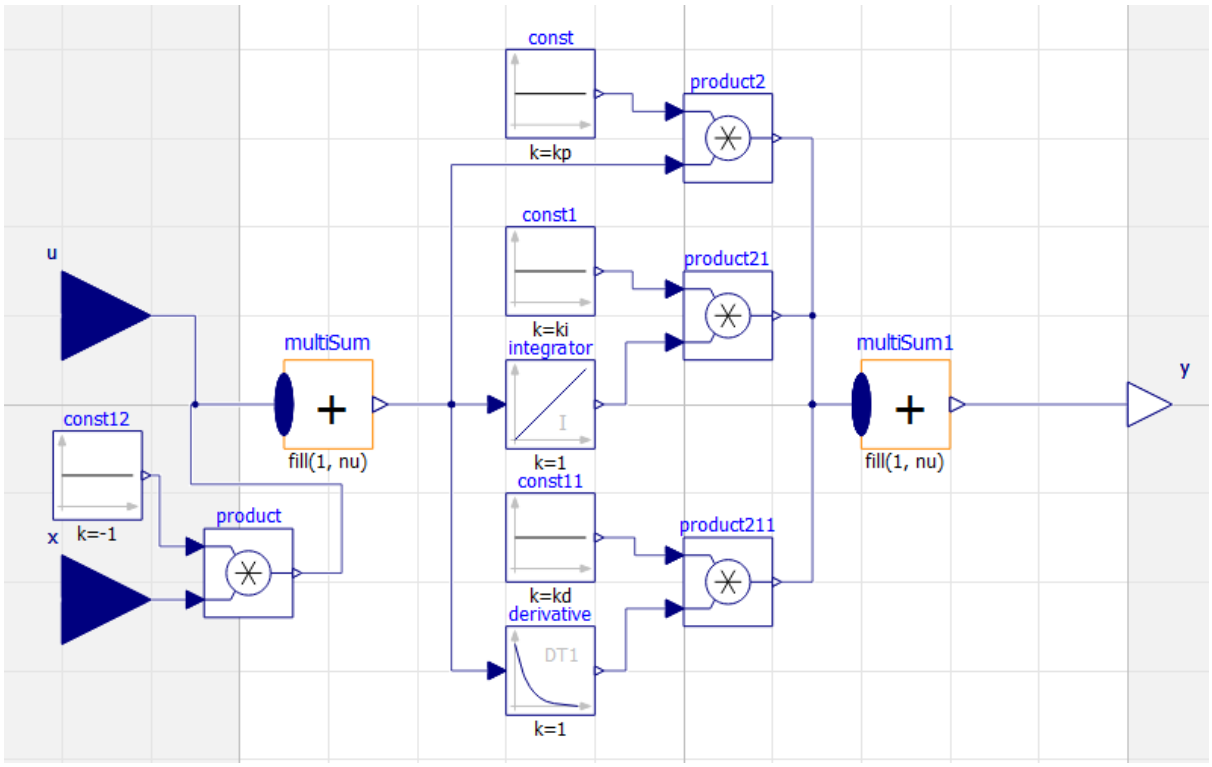
Kp:
: This variable determines how much the controller reacts to the error. High values make the controller more responsive and aggressive resulting in overshooting. Low values at the other hand may take more time to reach the destination and takes even longer to make the final adjustments to the exact location.

Ki:
: By remembering past errors, we can increase accuracy. High values results can cause overshooting and creates oscillations in the control system that can make the system unstable. Low values will result in a more stable system and better settling times.

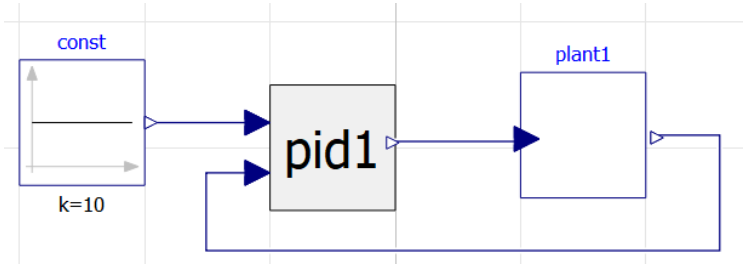
Kd:
: By predicting future errors, we can add a damping effect. High values will reduce overshoots and improve settling time at the cost of a less responsive controller. Low values will result in a less stable system with more overshoots and oscillations.

```
block plant
  extends Modelica.Blocks.Icons.Block;
  extends pendulum;
  Modelica.Blocks.Interfaces.RealInput u_input "Input signal connector" annotation(
    Placement(transformation(origin = {-120, 0}, extent = {{-20, -20}, {20, 20}}), iconTransformation(origin = {-90, -6}, extent = {{-20, -20}, {20, 20}})));
  Modelica.Blocks.Interfaces.RealOutput x_output "Output signal connector" annotation(
    Placement(transformation(origin = {110, 0}, extent = {{-10, -10}, {10, 10}}), iconTransformation(origin = {118, -4}, extent = {{-10, -10}, {10, 10}})));
equation
  u_input = u;
  x_output = x;
end plant;
```

Block of Plant



PID controller



PID-control loop

Controller Model Tuning

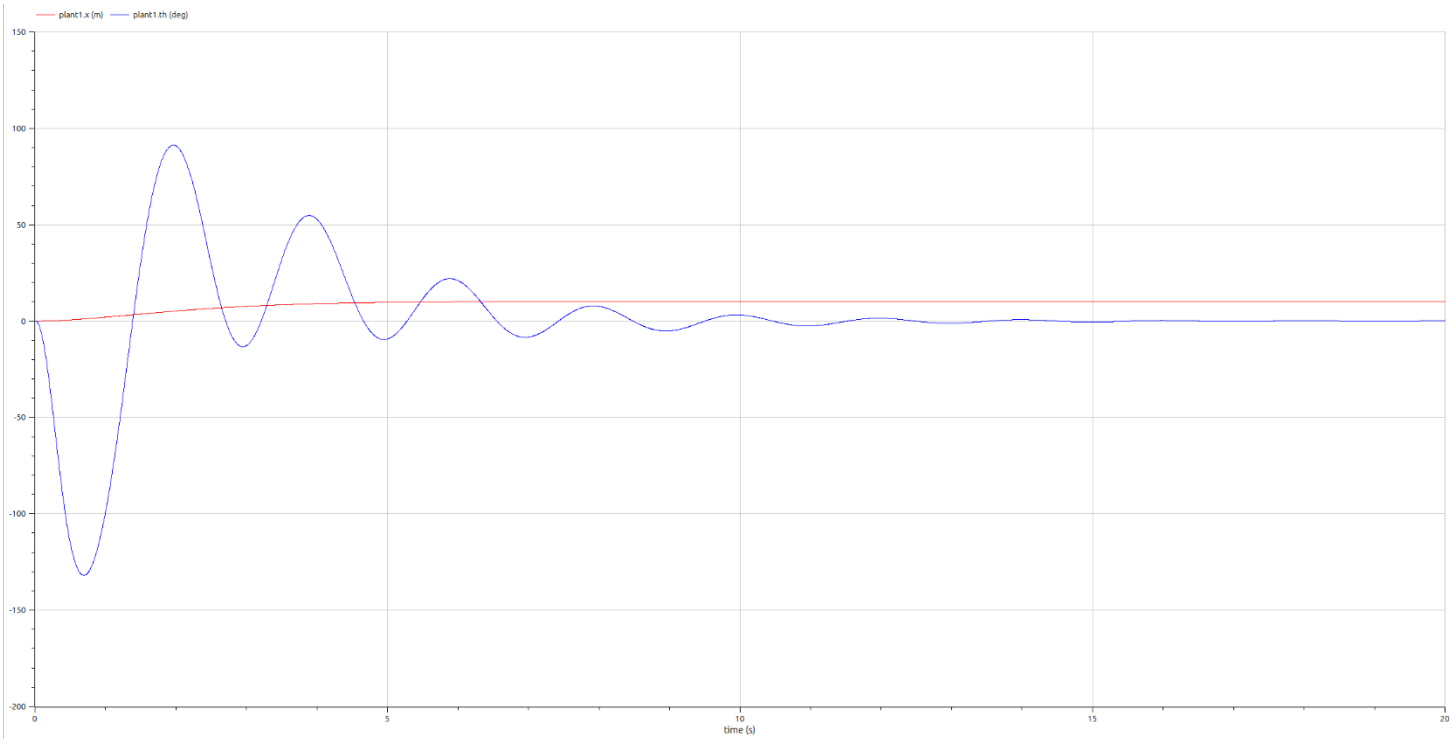
Now that we have a model for the pid controller and the control loop, it's time to calibrate it to find the optimal values for Kp, Ki & Kd using the cost function a*th_max + b*t_task where th_max is the maximum angular displacement of the pendulum and t_task is the moment when the gantry reaches the desired set-point of 10 meters (with 10 cm accuracy thus reaching 9.9 meters) and the pendulum's angular displacement remains within the acceptable range of 10 degrees.

Using this cost function it's clear that a PD controller will suffice thus Ki will be 0. The values for a & b are weight coefficients based on the student ID. Plugging the last four digits of our student ID's (1127 & 0395) into the provided python script we got the values 15 for a & 22 for b.

Code for finding values with lowest cost (pid_tuning.py):

```python
def calculate_cost(a, b, th_vals, time_vals, x_vals):
    th_max = max(th_vals)
    x_i = 0
    # Find time when gantry displacement passes 9.9m
    for i in range(len(x_vals)):
        if x_vals[i] >= 9.9:
            x_i = i
            break
    t_gantry = time_vals[x_i]
    th_bound = 0.1745
    t_oscillation = t_gantry
    # Find time when theta passes the 10 degree bound for the last time
    for i in range(x_i, len(time_vals)):
        if abs(th_vals[i]) > th_bound:
            t_oscillation = time_vals[i]
    t_task = max(t_gantry, t_oscillation)
    cost = a*th_max + b*t_task
    return cost

def simulate_pid_loop():
    # Iterate through all values of kp & kd and save simulation result in csv files
    os.mkdir("pid_results")
    for kp in range(1, 41):
        for kd in range(10, 501, 10):
            vars = singleSimulation("pendulum/Assignment1.pid_control_loop", "pid_control_loop", {"pid1.kp": kp, "pid1.kd": kd})
            df = pd.DataFrame({"time": vars["time"],
                               "displacement gantry": vars["plant1.x"],
                               "angular displacement": vars["plant1.th"]})
            df.to_csv("pid_results/kp_" + str(kp) + "_kd_" + str(kd) + ".csv")

def tune_pid(a: int, b: int):
    # If simulation results already present, skip to calculating cost
    if not os.path.exists("pid_results"):
        simulate_pid_loop()
    kp_best = 0
    kd_best = 0
    min_cost = float("+inf")
    for kp in range(1, 41):
        for kd in range(10, 501, 10):
            df = pd.read_csv("pid_results/kp_" + str(kp) + "_kd_" + str(kd) + ".csv")
            th_vals = df["angular displacement"].to_list()
            time_vals = df["time"].to_list()
            x_vals = df["displacement gantry"].to_list()
            cost = calculate_cost(a, b, th_vals, time_vals, x_vals)
            # If cost of simulation smaller than all previous ones, consider it the best one
            if cost < min_cost:
                min_cost = cost
                kp_best = kp
                kd_best = kd
    return kp_best, kd_best
```

Executing the script above gave us the optimal values 7 for Kp and 10 for Kd. Simulating again with these values gives the following plot:



The trace makes sense given the calibrated parameters, the oscillation of the pendulum gradually reduce and stay within an acceptable range while the cart moves to the setpoint and makes sure it doesn't overshoot.