Lab 6

Closed Loop Control

Author:

Jonas Lussi, Daniel Steinmann, Alexandre Mesot, Naveen Shamsudhin and Prof. Bradley J. Nelson

Institute of Robotics and Intelligent Systems

Date: 2021

Version: 1.0

Summary: The purpose of this week's lab is to model and implement a PID controller for our ball balancing system.

6.1 Background

This Lab will focus on designing and implementing a PID controller for our ball balancing system. The ball balancing system consists of 3 servo motors, joints that are connecting the motors to a plate, as well as a camera for visual feedback. The system is depicted in 6.1 The logical flow for the control of the system can be seen in Fig. 6.2

Note: The design of this system was inspired by Johann Link: https://www.instructables.com/id/Ball-Balancing-PID-System

6.2 Camera

The camera we use is a Pixy cam. It has integrated object detection and can directly send the position of the ball to the Udoo. The camera is designed for visual servoing and has a minimal latency of around 20 ms (=signal delay) and a high fps (frames per second = signal frequency). If we would use a standard camera, the images would need to be compressed for sending, decompressed on the Featherboard/PC and then analyzed with a computer vision process. This would lead to a large delay in the system which would make it impossible to control the ball with a PID approach. You will investigate how the delay impacts the performance of the system further in the Prelab.

Further information about the camera can be found here: https://pixycam.com

6.3 Kinematic Chain

To drive the plate that balances the ball, we use three servo motors that are connected to the plate via three separate two-link arms. This allows us to control the two rotational degrees of freedom of the plate, as well as its height.

Figure 6.1: The Ball Balancing System enjoys the view on the CLA rooftop.

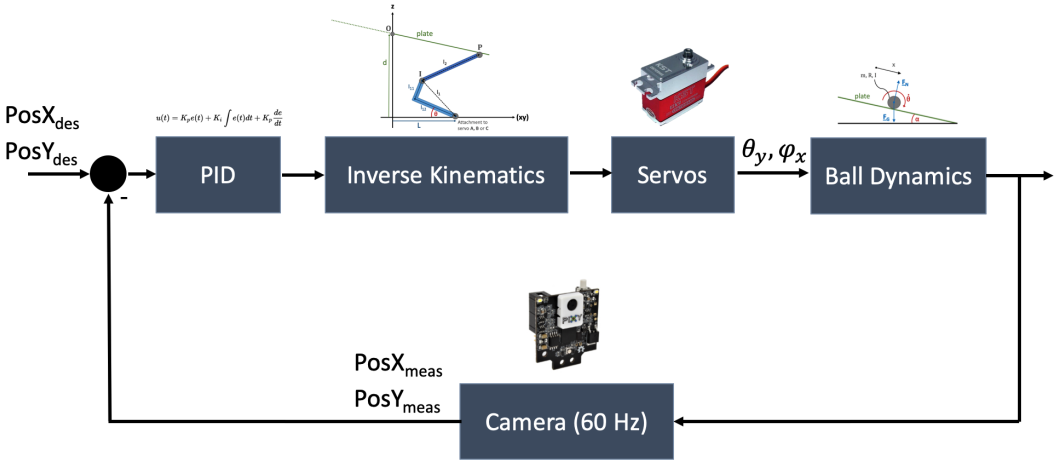

Figure 6.2: Control flow of the ball balancing system

6.3.1 Modeling

6.3.1.1 Ball on Plate

Although PID control is a model-free approach, it is useful to have a dynamic model of the ball at hand. It allows the simulation of the system and is essential to obtain an estimate of a feasible controller. To model the plant (ball dynamics), we first have to derive the differential equations describing the motion of the ball with respect to the input to the plant, which is the plate angle. The ball is assumed to roll without slipping on an inclined plane. For

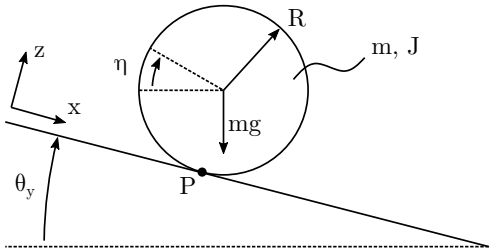

Figure 6.3: Ball Dynamics

simplicity, drag is omitted.

Figure 6.3 depicts the ball on the tilted plate. The movement of the ball can be analyzed in 2D, as $\ddot{x}$ only depends on the corresponding angle θ_y. Similarly the acceleration in y-direction $\ddot{y}$ only depends on the corresponding angle φ_x. Every other movement is a linear combination of the two. Therefore, we first derive the equations of motion along the x-axis and then apply the same equations to the orthogonal y-axis, paying attention to sign changes.

We apply the angular momentum principle in the contact point P between the ball and the plate. The moment of inertia of a hollow sphere is $J = \frac{2}{3}mR^2$. Using Steiner's theorem, the moment of inertia J_P around P is

$$\begin{aligned} J_P &= J + mR^2 \\ &= \frac{2}{3}mR^2 + mR^2 \\ &= \frac{5}{3}mR^2. \end{aligned} \tag{6.1}$$

Using the angular momentum principle, we get

$$J_P\ddot{\eta} = mgR\sin(\theta_y). \tag{6.2}$$

We state the no-slip condition at the contact point P as

$$x = R\eta. \tag{6.3}$$

Taking the second derivative of Equation 6.3, we can substitute $\ddot{\eta}$ in Equation 6.2 and obtain

$$\ddot{x} = \frac{3}{5}g\sin(\theta_y) \tag{6.4}$$

Now we can also formulate the same equation for the y-coordinate. Since we are using Euler angles, which are defined as positive rotations about the unit axes, we have to switch the sign of the y-coordinate. Intuitively, this is clear when looking at the definition of the plate angles, where a positive change in φ_x leads to the ball rolling in $-y$ direction. Therefore, the equations of motion are stated as

$$\begin{aligned} \ddot{x} &= \frac{3}{5}g\sin(\theta_y) \\ \ddot{y} &= -\frac{3}{5}g\sin(\varphi_x). \end{aligned} \tag{6.5}$$

6.3.1.2 Linearized Equations

To represent our system as a linear, time-invariant (LTI) system, we have to linearize the equations of motion around an equilibrium point. Linearization enables us to use transfer functions, state space models, and generally all tools of linear system theory.

$$\ddot{x} = \frac{3}{5}g\sin(\theta_y) = f(\theta_y) \tag{6.6}$$

The linearized equations can be calculated by using the Taylor series expansion of the function f around the equilibrium point $\theta_e = 0$. Note that now x, y, θ_y, φ_x represent small pertubations around the equilibrium. The linearized equations can then be written as:

$$\begin{aligned} \ddot{x} &= f(\theta_e) + f'(\theta_e)\theta_y \\ &= \frac{3}{5} g \theta_y \end{aligned} \tag{6.7}$$

and similarly :

$$\ddot{y} = \frac{3}{5} g \varphi_x$$

Note: We used the same symbols for the nonlinear and the linear systems for simplicity, but the systems must be kept apart.

6.4 PID Controller

A proportional–integral–derivative controller (PID) is one of the most common controllers in the world. It is widely used in industry for its simple, model-free implementation and its versatility. A PID controller calculates the error $e(t)$ between a desired setpoint of a process variable (e.g. postion) and the measurement of said signal and applies a proportional, integral and differential operation to this time-dependent error. The output of the PID controller is then used as an input to the plant (e.g. actuator) to reach the desired setpoint of the process variable. In the frequency domain, the transfer function for an analog PID controller is:

$$C(s) = K_P + \tfrac{1}{s} K_I + s K_D$$

where K_P, K_I and K_D are the proportional, integral and derivative gains respectively.

Figure 6.4 shows the diagram for a PID controller with anti-windup. When the actuator saturates, the anti-windup scheme will be activated and prevent the controller output from wandering away. In our case this could happen when we give the motor commands that they cannot achieve due to their limited speeds. For this Lab however, we will omit the anti-windup and focus on the PID controller itself. This is just for your personal reference.

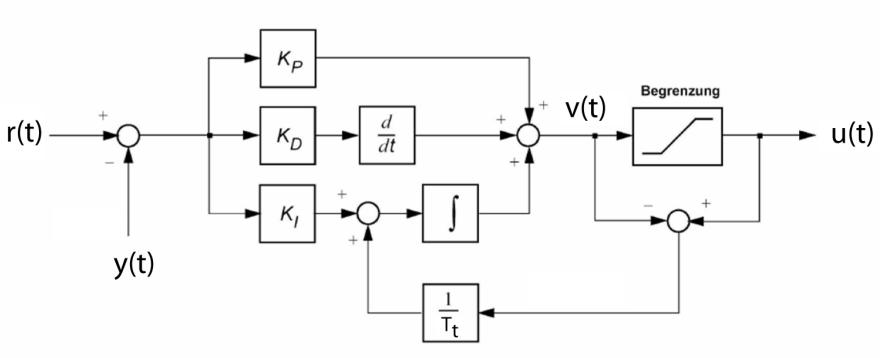

Figure 6.4: PID controller with windup protection.

6.4.1 Discrete Implementation

To state the PID as a transfer function $C(s)$ is useful for simulation with LTI systems, it allows to use classical control design methods such as bode plots and nyquist diagrams. However, if a PID is to be implemented, a discrete formulation is required. If we apply the inverse Laplace transform to the controller $C(s)$, we translate from frequency-domain to time-domain. First we express the input-output relation of the controller according to 6.5,

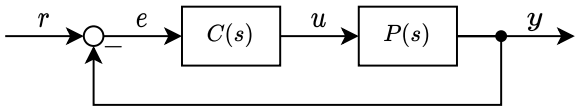

Figure 6.5: Feedback loop with controller $C(s)$ and Plant $P(s)$. The output y is subtracted from the reference r to obtain the error e. The controller computes an input u which is fed to the plant.

where $U(s)$ is the output and $E(s)$ the input:

$$C(s) = \frac{U(s)}{E(s)} \quad \Leftrightarrow \quad U(s) = C(s)E(s) = K_P E(s) + \frac{1}{s} K_I E(s) + s K_D E(s) \tag{6.8}$$

Then we apply the inverse Laplace transform to both sides of the equation to obtain the time-domain representation. Here we assume all zero initial conditions:

$$u(t) = K_P\, e(t) + K_I \int_0^t e(t)\,\mathrm{d}t + K_D\, \dot{e}(t) \tag{6.9}$$

From the time domain, we can obtain a discrete time approximation. The error $e(t)$ becomes $e[k]$ at timestep k. $r[k]$ and $y[k]$ denote the reference and measured position at timestep k respectively. We can then write

$$e(t) \approx e[k] = r[k] - y[k] \coloneqq u_P[k] \tag{6.10}$$

Now we have to replace the integral by a discrete sum. Reformulation allows to increment the integral time at each timestep instead of computing the entire sum every time. This approximation is called the Forward Euler method with sampling time T_s.

$$\begin{aligned} \int_0^t e(t)\,\mathrm{d}t &\approx T_s \sum_{i=0}^{k} e[k] \coloneqq u_I[k] \\ &\approx u_I[k-1] + T_s e[k] \end{aligned} \tag{6.11}$$

Finally, the error derivative can be formulated using the Euler backward method as

$$\dot{e}(t) \approx \frac{e[k] - e[k-1]}{T_s} \coloneqq u_D[k] \tag{6.12}$$

The error derivative can also be reformulated as a difference of derivatives, which lets us express the error derivative in terms of velocity and velocity reference. This is practical for our case, since we will work on a filter to obtain a good velocity estimate, which we can then directly use in the PID.

$$\begin{aligned} \dot{e}(t) &= \frac{\mathrm{d}}{\mathrm{d}t}(r(t) - y(t)) \\ &= \dot{r}(t) - \dot{y}(t) \\ &= v_{\text{ref}}(t) - v(t) \\ &\approx v_{\text{ref}}[k] - v[k] \coloneqq u_D[k] \end{aligned} \tag{6.13}$$

where

$$v_{\text{ref}}[k] = \frac{r[k] - r[k-1]}{T_s}$$

$$v[k] = \frac{y[k] - y[k-1]}{T_s}$$

Now we have discretized every element of the controller, and we can therefore formulate the PID control law at

step k with the expressions obtained above.

$$\begin{aligned} u_{\text{PID}}[k] &= K_P u_P[k] + K_D u_D[k] + K_I u_I[k] \\ &\text{where} \\ u_P[k] &= e[k] \\ u_I[k] &= u_I[k-1] + T_s e[k] \\ u_D[k] &= \frac{e[k] - e[k-1]}{T_s} = v_{\text{ref}}[k] - v[k] \end{aligned} \tag{6.14}$$

For the implementation, make sure you initialize all necessary variables. In the main loop, the general order of operations could look like this:

1. Measure elapsed time T_S
2. Measure position and use calibration function
3. Obtain velocity using filtering function
4. Set position and velocity reference depending on task
5. Update u_P, u_D, u_I and compute u_{PID}
6. Compute servo angles and send servo commands
7. Log all interesting states

Note: There are other possibilities to implement the PID controller. For this lab you are free to implement your own variation of the PID controller. For example, a common choice for $u_I[k]$ is also

$$u_I[k] = u_I[k-1] + T_s \frac{e[k] + e[k-1]}{2}$$

6.4.2 Step Response

Depending on the damping in a system second-order system and the controller gain that is used, the step response can be oscillatory (under-damped), critically damped or over-damped.

Figure 6.6 shows the step response for an arbitrary second-order system (not the ball balancing system) for different gain values of $K_p > K_{cr}$ (underdamped), $K_p = K_{cr}$ (critically damped) and $K_p < K_{cr}$ (overdamped).

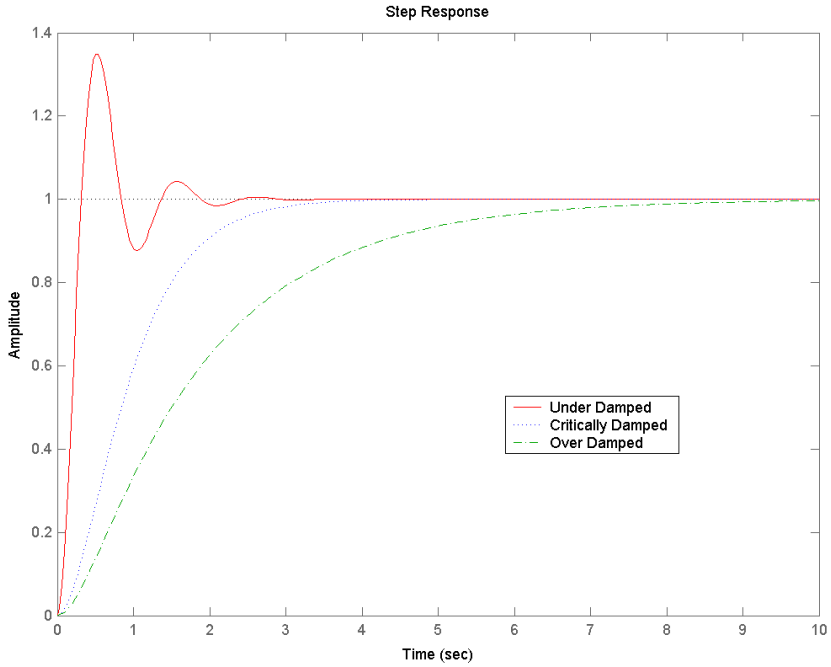

Figure 6.6: response for under damped, critically damped and over damped cases.

6.5 Prelab Procedure

Note: The Prelab quiz on Moodle must be done before reporting to the lab. The submission deadlines are on Moodle. Late submissions will not be corrected. Each group member has to complete the quiz on their own, however you are allowed to work with students in your group to solve the quiz (and we absolutely recommend you to do so). Please remember to submit your quiz when you have answered all of the questions.

6.6 Postlab Procedure

6.6.1 Introduction

Last week you have implemented the camera calibration and the inverse kinematics of the ball balancing robot. Now we will use those components together with a filter and a PID controller to control the ball at any desired location on the plate.

Note: Do not flash the Arduino Code. This might cause damage to the servos.

Note: The tasks continue from last week's lab. you can just copy paste your old code in the new skeleton.

6.6.2 Task 3: Velocity Filtering

- As you have seen in the introduction, we will work with a PID controller. In the last lab, you have already put in place a function which provides the x and y coordinates of the ball. Additionally, we will also need the v_x and v_y ball velocities for the derivative part of the controller. The most straightforward way of obtaining a velocity estimate is to use the first-order discrete derivative, also called the Euler backward method, where Δt is the sampling period and x_i the position of the ball at timestep i:

$$v_{x,i} \approx \frac{x_i - x_{i-1}}{\Delta t} \tag{6.15}$$

The file `pos_logfile.csv` contains a trajectory which was generated by moving the plate by hand and logging the ball coordinates. We will use it to test our velocity signal before we implement it in the C-language controller. Use the provided Matlab skeleton `velocity_filtering_skeleton.m` to compute the Euler backward estimate of v_x with a sampling frequency of $f = 60\,\text{Hz}$. Hand in a plot with the velocity estimate in the time range $t \in [10, 15]$ s. Describe your observations, and how they might impact the controller performance. **(PostLab Q1)**

- In the IRM lecture, you have learned about the Moving Average (MA) filter. We will use it to improve our velocity estimate. For a window size n and unfiltered values x, the filtered output y is given by

$$y_i = \frac{x_{i-n+1} + \cdots + x_i}{n} \tag{6.16}$$

Use the Matlab skeleton to compute the filtered velocity signal for $n = [1, 5, 10, 20, 30, 50, 100]$ and compare the resulting signals. Explain the influence of smaller and larger window sizes on the filter performance and associated delay. Assuming that the delay of the filtered signal should not exceed 0.15 s, propose a filter window size which should be implemented in the controller. Provide a plot with $t \in [10, 12]$ s with the following elements **(Postlab Q2)**:

  - The unfiltered velocity signal.
  - Three filtered velocity signals:
    * An example of a small window size.
    * Your proposed window size.
    * An example of a large window size.
  - A comprehensive legend.

- Now that we have found a starting value for the velocity filter window size, we have everything we need to implement the filter in C. Implement your *movingAverage* filter function in `util.c`.Note that you will find further information about all the functions you have to implement in the `util.h` file. With a filtering function in place, it is not much of an effort to filter the position signal as well. Provide a short reasoning why this could be a good idea, and whether you would choose a larger or a smaller window size for the position filter **(Postlab Q3)**:

6.6.3 Task 4: PID controller

- We now have all the components we need to implement a closed-loop PID controller. Implement the PID controller in the `main.c` file under "Task 4". You should make use of the functions we wrote so far. If you want to increase readability of your code you can also write additional functions in `util.c` to achieve this task. However, since there are many parameters to pass, it might be quicker for you to implement everything in the `main.c` file. **(Postlab Q4)**

  Implement a routine with the following specifications:

  - Before the loop starts, the user is asked to enter the PID parameters. The user can also choose to use a default set of parameters.
  - The ball should be controlled in the center of the plate. Therefore we have $x_{\text{ref}} = y_{\text{ref}} = v_{x,\text{ref}} = v_{y,\text{ref}} = 0$.
  - Use the provided logger function to store all specified variables in a text file and to display the relevant parameters on the terminal.
  - The task is complete if you can push the ball off-center and it returns and remains within the small center circle within 5 s. Make sure to submit a short video that displays this behaviour.

For your report, hand in a video in which you demonstrate the system stability according to the last specification in the list.

Hint 1: First make sure that your input signals are correct. After implementing the velocity filter, check whether the data makes sense.

Hint 2: Try to use a scaling constant for the PID output to obtain controller weights of the order of 1, they are easier to remember (at least for me).

Hint 3: Before spending hours on tuning the PID weights, maybe check if the calibration of your system and especially the camera still work as expected.

Hint 4: You are not the first ones to implement a PID. Use online resources and examples to set up the controller structure before you delve into the details.

6.6.4 Task 5: Step Response

- You are provided with a function that generates a step reference in x-direction. Implement a switch which lets the user choose whether they would like to see a centered ball or a step response. Further, you are provided with a Matlab script that plots the contents of the logfiles. Tune your PID-weights to produce the following step-response plots: **(PostLab Q5)**
  - Underdamped step response, steady state error $e_{ss} < 5\,\text{mm}$
  - A critically damped step response, $e_{ss} < 5\,\text{mm}$, overshoot within this error tolerance.
  - An Overdamped step response, with no overshoot

  For your report hand in these three plots with the proper label as well as a short video showing the step response.

  Note: It is sufficient to check if your responses meet the criteria visually. The error tolerance is marked automatically on the plots that are created with the provided matlab script. The figures at the end of this manual 6.7, 6.8, 6.9 show what your results should look like approximately.

6.6.5 Task 6: Circular Trajectory

- Implement another use-case which makes the ball trace a circular trajectory with radius $R = 80\,\text{mm}$ and a period of $p = 4\,\text{s}$. Program the function *circularTrajectory* in `util.c` . Use the Matlab plotting script to evaluate your trajectory. The task is solved if the ball remains within an error bound of 20 mm of the trajectory, therefore within the marked path on the plate. Make at least two turns. Overshoot while entering and exiting the trajectory is allowed. Figure 6.10 is a good example of how your trajectory should look like. **(PostLab Q6)**

  Hint: How does a parametrized curve for a circle look like? What is the analytic expression for the velocity of the moving point?

  Hint: You might need to re-tune your controller for this fast tracking task. Try deactivating the integrators first, since they slow down the controller performance.

  For your report hand in the plot as well as a short video showing the step response.

- NOTE: This is a bonus question. It does not have to be solved to achieve max points in the lab - however it can be used to make up for lost points in the postlab.
  You have learned in the IRM lecture that the moving average filter's implementation is simple but its performance is not always optimal. Try implementing a second-order Butterworth filter, which attenuates high-frequency components more efficiently than than the moving average filter.
  For your report, make a short video which demonstrates that the system shows a smoother balancing behaviour without any loss of speed. **(PostLab Q7 BONUS)**

Hint 1: Check out the Matlab documentation of the function `butter()`. It returns two arrays of coefficients $\vec{a}$ and $\vec{b}$.

Hint 2: Read the section *Difference equation* in this Wikipedia article to learn about how these coefficients are used for the implementation.

Hint 3: Start with a normalized cut-off frequency of $f_c = 0.08$ in Matlab, compute the coefficients and copy them into your filter implementation in C.

6.7 Postlab and lab report

1. Upload a single PDF-file with your solutions to moodle. The file should contain as all the code you implemented in the μP C files, namely submit the util.c, the main.c and your parameters.c file. Also make sure to show all your obtained plots for Postlab Q1-Q6 in the report. Please upload your solution in time, late submissions will not be corrected. Please also share the videos for Postlab Q4 - Q7 with your assistants on slack.
2. Come prepared with the Prelab procedure for the next lab.

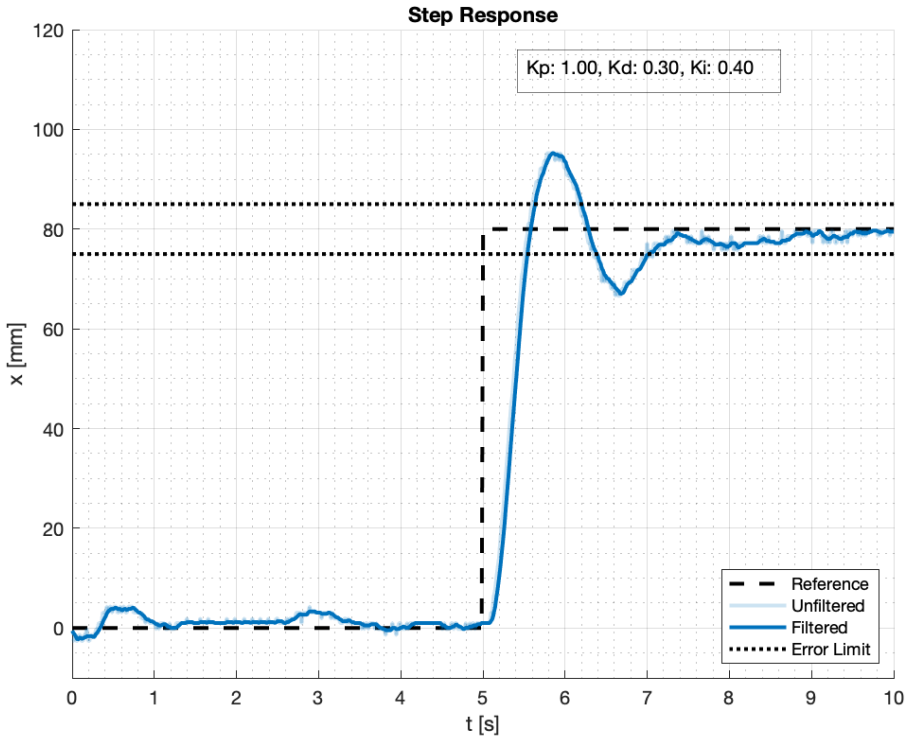

Figure 6.7: underdamped response

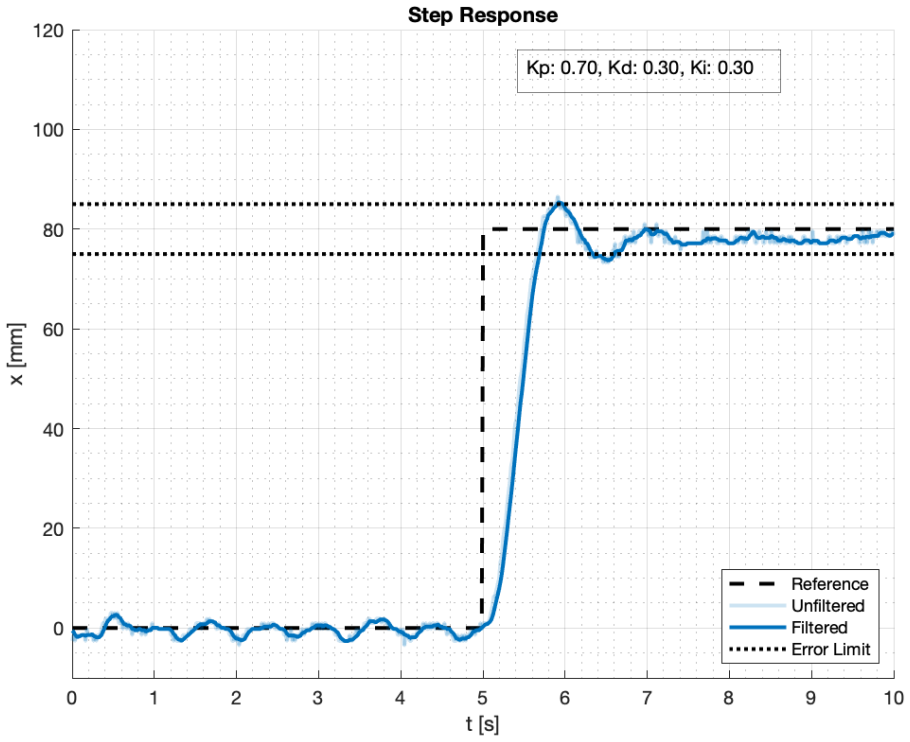

Figure 6.8: critically damped response

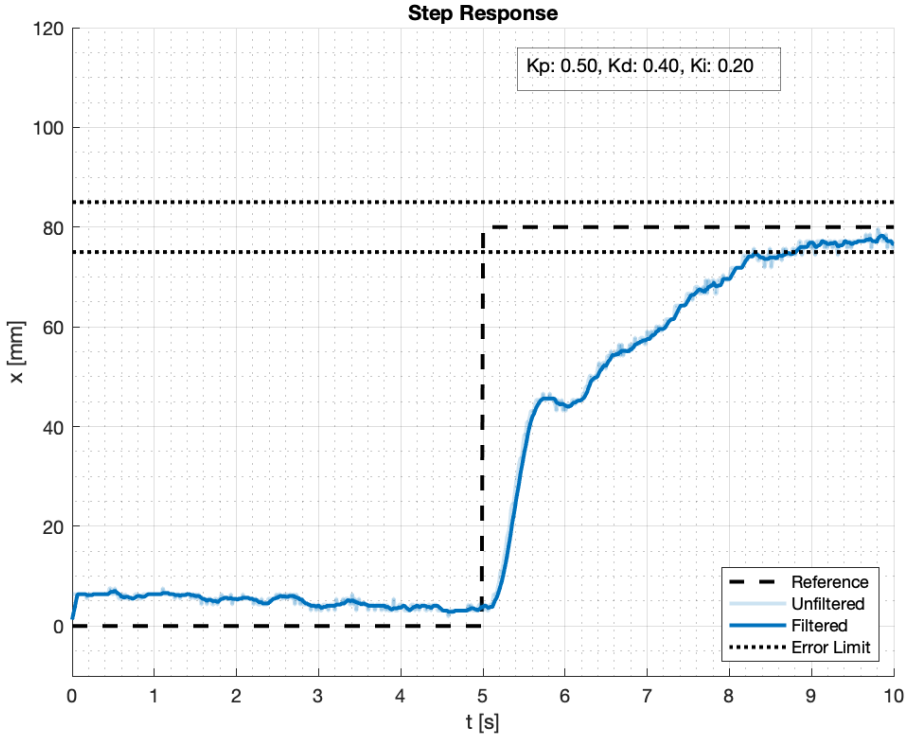

Figure 6.9: overdamped response

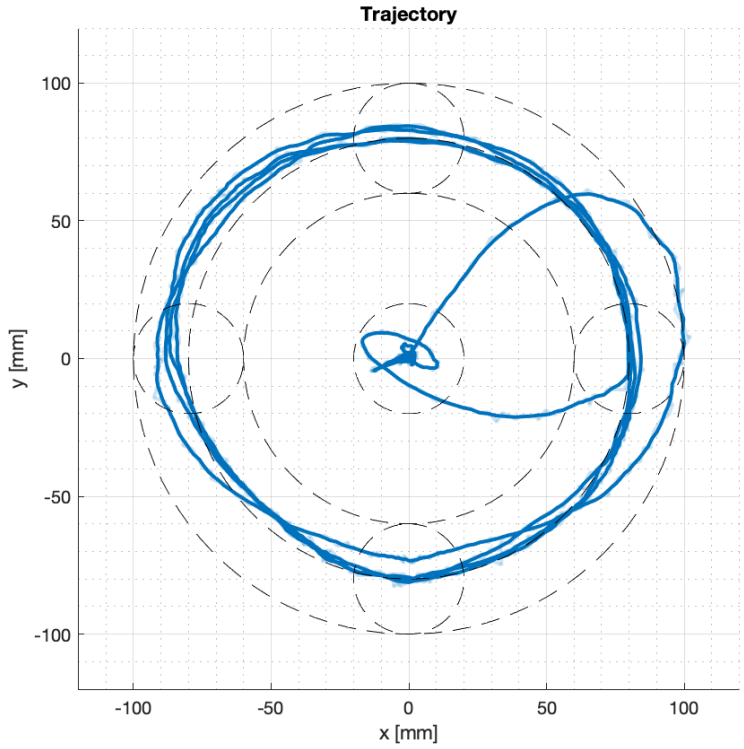

Figure 6.10: circular trajectory