Implementation of discrete-time controllers for the Regbot platform

Carletti Simone
Politecnico di Torino & DTU
s242168@dtu.dk

Nordio Gianluca
Università di Padova & DTU
s243125@dtu.dk

***Abstract*—This report details the design and implementation of a discrete-time control system for the mobile robot *Regbot*, completed as part of the Digital Control course. The system was first designed and simulated in Simulink, then manually implemented in C++ on the onboard Teensy microcontroller. A path-following task was used to evaluate performance. The report highlights the design process, tuning methods, and challenges of transitioning from simulation to hardware implementation.**

***Index Terms*—discrete, controllers, mobile robotics, Simulink, Regbot, PI, Teensy, PlatformIO**

I. Introduction

The main goal of this project was to design and implement a series of controllers for the *Regbot* mobile robot, directly on the onboard μC (hence digital control techniques were used). In particular we were tasked with the goal of designing such controllers in order to make the robot complete the path of figure 1. We were also asked to make *the robot follow the route as fast as possible* and to *stay as close to the black marking as possible*.

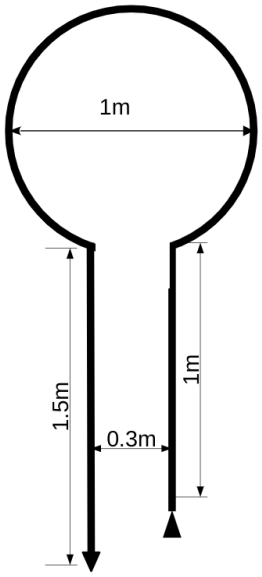

Fig. 1: The desired path

The code can be found at https://github.com/regi18/dtu-digital-control

II. Model

To begin this project, we started by setting up a model where we could do all of our testing, simulations and tuning.

A. Robot model

As the center of our Simulink we have the model of the robot (fig. 2), which simulates the inputs and outputs that we have available on the real platform. It takes as inputs the motor voltages (plus the rotational velocity, needed to simulate torque) and gives out the linear and angular velocities of the two wheels. The linear velocities are calculated just by multiplying the angular velocity with $\frac{\text{wheel radius}}{\text{gear ratio}}$.

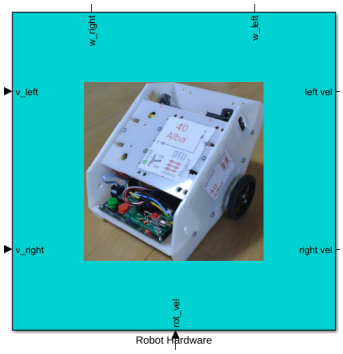

Fig. 2: Robot Hardware Block

Inside the block we have, apart from other things, two identical models for the left and right motors (see fig. 3). Since the real robot is going to be controlled in discrete-time, we also added a *zero order hold* to the input voltage. To have a more realistic representation we also added a transport delay to the output ω.

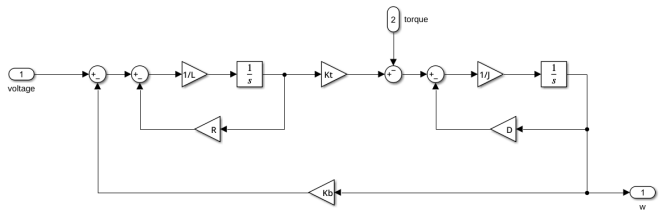

Fig. 3: Motor model

The latter is needed because the motor encoder has an inherit delay in its measurements, caused by the physical spacing of the magnet dipoles (only 12 dipoles on the entire encoder). For this reason we have a measurement only every:

$$\Delta t = \frac{2\pi}{12\omega_m} \tag{1}$$

which, especially for low speeds, gives a non negligible delay.

The parameters contained in the block diagram (fig. 3) can be calculated by taking some specific input/output measurements on the real robot. This is better described in section IV.

B. Pose calculation

To have a complete simulator we also added a block that, taken as inputs the left and right wheel velocities, calculates the pose and turn rate of our robot.

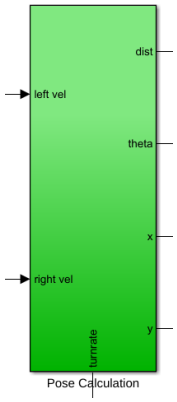

Fig. 4: Pose Calculation Block

The content is just the implementation of various formulas for a differential drive robot:

$$\theta = \int \frac{\dot{x}_R - \dot{x}_L}{B}\, dt \quad , \quad \dot{\theta} = \frac{\dot{x}_R - \dot{x}_L}{B}$$

$$d = \int \frac{\dot{x}_R + \dot{x}_L}{2}\, dt$$

$$x = \int \cos(\theta)\frac{\dot{x}_R + \dot{x}_L}{2}\, dt \quad , \quad y = \int \sin(\theta)\frac{\dot{x}_R + \dot{x}_L}{2}\, dt$$

III. Controllers

With the model at our disposal we can now turn to the implementation of all the regulators. We chose to use only PI controllers (fig. 5), as they proved sufficient to achieve the desired performances. The full Simulink is illustrated in fig. 14.

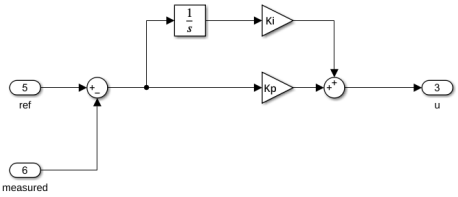

Fig. 5: PI controller

A. Velocity controllers

We first started by implementing two controllers (one for each wheel) so we could set the desired wheel velocities.
This regulator takes as input a reference angular velocity (for the wheel) and gives out a voltage command (which is saturated at $\pm 9v$ to protect the motors).

As stated before we have 2 identical of these controllers, one for each wheel. The input to the two are as follows:

$$ref_{left} = v_{ref} - u_{turnrate} \quad , \quad ref_{right} = v_{ref} + u_{turnrate}$$

where v_{ref} is the desired linear velocity of the robot, and $u_{turnrate}$ is the turnrate ($\dot{\theta}$) command.

In order to eliminate sliding and other unwanted phenomena we also added a limiter to the reference input, so to smooth out and saturate any problematic references - see fig. 6 (it implements a fast low-pass filter to eliminate abrupt changes, plus a saturation to act as a limiter).

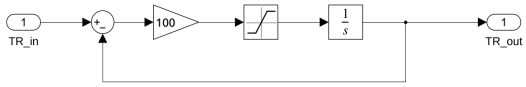

Fig. 6: Limiter block diagram

B. Turnrate controller

We than proceeded to implement a controller for the turnrate, so that we could make the robot turn as desired. We experimented by adding a *lead* term in the feedback, but didn't notice any improvements.

This controller takes as input the reference and the measured turnrate, and outputs the necessary command. Also, as for the velocity reference, we added a limiter to the reference input.

C. Heading controller

Following the same procedure as the turnrate controller, we also implemented a regulator to set a desired heading. Since this is not always used, we implemented a switch to bypass it (e.g. if we want the robot to make a continuous curve it's better to simply set a constant turnrate) - see fig. 7.

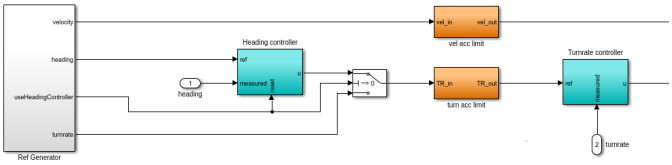

Fig. 7: Heading and Turnrate controllers

IV. Parameters

A. Motor

The robot uses two DC-motors to move. Such systems can be described considering both the electric and mechanical part. Generally a simplification can be performed based on whether the electrical system is considered faster than the mechanical, however here no such simplification is used.

As anticipated, it is necessary to find the various parameters for the block diagram of the motors. One possible way to do this is to consider the model piece by piece and, with some input/output measurements, and calculate the parameters.

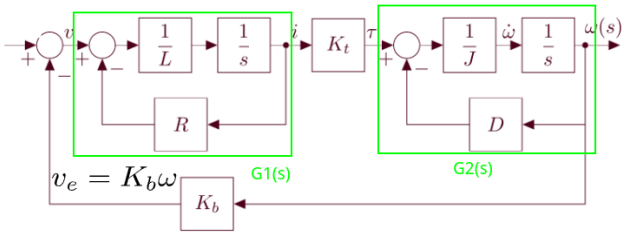

Fig. 8: Motor diagram sub-blocks

For example, we can isolate the first loop (relative to the electrical part - see fig. 8), which has the following transfer function:

$$G_1(s) = \frac{1}{Ls + R}$$

It has input v and output i. We can then try to measure the motor current, giving as input known voltages.

The same can be done with the other 2 loops:

$$G_2(s) = \frac{1}{Js + D}$$

$$G(s) = \frac{K_t}{JLs^2 + DLs + JRs + DR + K_tK_b} \tag{2}$$

with the first being the mechanical part and the latter one being the complete loop.

In our case, we did all the necessary measurements and calculations just to realize in the end that we had the wrong switch position on the robot. This caused the motors to only receive 5 volts (instead of the full 12v). Before redoing all the work we tried the parameters that the professors had, and we found that they worked perfectly well. During our testing we also switched multiple robots and did not notice any problems (from this we decided to stick with those parameters and not recalculate them).

B. Turn-rate transfer function

Before proceeding to design a controller we first needed to find the transfer function for the turnrate. To do this we considered only the robot model with the velocity controller (that as we will see later requires only the known motor t.f. to be designed) and proceed with system identification.

We set as input reference a unit step (for both turnrate and velocity) and measured the output turnrate (from the pose calculation block). With this data we then turned to Matlab to perform a t.f. estimation:

```
out = sim('model_no_turnrate_control.
    slx', 3);

% Resample with constant sample time
t = 0.5:Ts:1.5;
tsout = resample(out.turn_tf, t);

% Get in and out in separate vectors
idin = tsout.Data(:,1);
idout = tsout.Data(:,2);

% Collect in structure
idd = iddata(idout, idin, Ts);

% Estimate
sys = tfest(idd, 3, 2);
figure();
compare(sys32, idd);

% Bode
sys.InputDelay = transport_delay;
figure();
bode(sys)
```

Listing 1: Transfer function estimation

where `out.turn_tf` is the time-series of the input-output measurements (step vs turnrate) that we got from Simulink. The output figures are available appendix B.

C. Heading transfer function

A similar procedure can be done to estimate the t.f. of the heading. This time we kept all controllers except the one for heading, again taking input-output measurements for a unit step. The code is the same as the one shown in listing 1, with the only difference being the form of the t.f. (that in this case was `tfest(idd, 4, 3)`). The output figures are available appendix C.

V. Tuning

With the model of the motors defined and working, we focused on the design of the various controllers. We worked incrementally, first considering the velocity controllers, then adding the turnrate controller and finally the heading controller.

For the design technique we first tried to use manual bode plot methods, e.g. using Matlab's `sisotool`, but then we decided to just use the command `pidtune` of the *Control System Toolbox* since it gave similar results but much quicker.

A. Velocity controllers

The first design, as stated before, was for the velocity controllers. For doing so we simply defined the motor transfer function shown in eq. (2) and then used the pid tuner:

```
% Define the t.f.
Gn = Kt;
Gd = [J*L, D*L+J*R, D*R + Kt*Kb];
G = minreal(tf(Gn, Gd));
G.OutputDelay = transport_delay;
Gz = c2d(G, Ts,'zoh');

% Tune a PI controller with 75deg PM
pm = 75;
o = pidtuneOptions('PhaseMargin',pm);
[C, i] = pidtune(Gz, 'PI', o);
% C.Kp, C.Ki
```

Listing 2: Design of velocity controller

We can notice (as stated in a previous section) that we're using the *zero order hold* and a *transport delay*.
The resulting K_i and K_p can be than used in our Simulink model.

In figure 9 we can see the effect of the created PI controller, which correctly respects the required 75° of phase margin (since we have tuned the controller for a discrete t.f., in order to test it on the fly we also need to discretize the controller, which we have done with `Cz = c2d(C, Ts, 'tunstin')`).

B. Turnrate and heading controllers

The procedure is similar to what shown in listing 2. This time instead of defining the t.f. manually we're going to use the one we estimated in sections IV-B and IV-C.

```
phase_margin = 60;
wc = 25;
o = pidtuneOptions('PhaseMargin',
    phase_margin);
[C, i] = pidtune(G, 'PI', wc, opts);
```

Listing 3: Design of turnrate controller

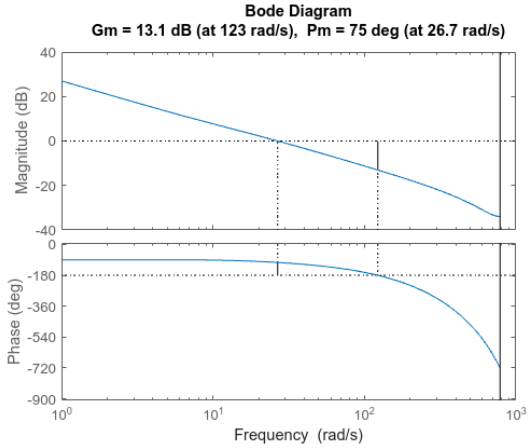

Fig. 9: Bode plot of controlled open-loop for the velocity

```
phase_margin = 50;
opts = pidtuneOptions('PhaseMargin',
    phase_margin);
[C, i] = pidtune(G, 'PI', opts);
```

Listing 4: Design of heading controller

Once again the resulting K_p and K_i can than be used in Simulink.

The step responses for the closed-loops can be seen in appendix D.

VI. C++ Implementation

The final step of this project is to convert all the controllers in C++, so that we can directly implement them on the onboard Teensy.

We can start by saying that, since we're using discrete controllers, all of the logic will be executed at a fixed interval. We choose a sampling time of $\Delta t = 4ms$.

A. PI

From the moment that all the controllers are PI we created a reusable class that defines a such a regulator (see `pi_controller.h`). The output command is calculated as follows:

$$e_i = ref - z$$
$$I = \sum_i \frac{\Delta t(e_i + e_{i-1})}{2}$$
$$u = K_p e_i + K_i I$$

where the value of the integral I is calculated using the *trapezoidal rule*, and it's updated every Δt. For the case of the velocity controller the output u is also saturated to $\pm 9v$. We then used this class for the 4 PI controllers, by setting the K_p and K_i we previously found.

B. Limiter

A similar thing has been done for the limiters. Following from the block diagram (fig. 6) we implemented the following logic in code (see `smoother.h`):

$$\tilde{e}_i = 100 \cdot (z_i - z_{i-1})$$
$$e_i = \max\left(-\text{limit}, \min\left(\tilde{e}_i, \text{limit}\right)\right)$$
$$u = \sum_i \frac{\Delta t(e_i + e_{i-1})}{2}$$

where again the integration is done via the *trapezoidal rule* and *limit* is the desired min/max value to be imposed.

C. Controllers

With all the necessary classes defined we proceeded to put everything together in the same order as the Simulink model.

```
// Heading + Turnrate controllers
double turnrateComand =
    getTurnrateComand(...);

double velRef = velSmoother.compute(
    sm.getVelocityReference());
double leftRef = velRef -
    turnrateComand;
double rightRef = velRef +
    turnrateComand;

// Velocity controllers
motorControlUpdate(&sm, &pi0, &pi1,
    leftRef, rightRef);
```

Listing 5: Composition of the controllers

where `getTurnrateComand()` implements the controllers of fig. 7 and `motorControlUpdate()` the two velocity controllers.

D. Pose calculation

We also implemented the *pose calculation* block by utilizing the discrete definitions of the formulas from section II-B; see `pose_tracker.h`. The only difference is in the calculation of the traveled distance (that then is used also for other calculation, recalling that $\int \dot{x} = x$):

$$S = \frac{\text{wheel radius}}{\text{gear ratio}} \frac{2\pi}{4 \cdot 12}$$
$$d_l = -(enc_i^l - enc_{i-1}^l) \cdot S$$
$$d_r = (enc_i^r - enc_{i-1}^r) \cdot S$$

where S is the distance per tick and enc^l, enc^r are the right and left wheel encoders. The formula for S is derived similarly to (1), recalling that every magnet is capable of 4 detections.

E. Reference generator

We then finished with one of the most important things: the definition of the references to correctly track the path. As briefly anticipated at the beginning we used both the heading and turnrate controllers individually.

The code contained in `state_machine.h` defines the current state of the robot (waiting, started) but also defines the reference sequence and which controller to use.

The following best-performing parameters have been obtained by manually tuning and testing the robot on the real path:

```
const double path[][4] = {
    {0.6, 0.5, 1, 0},
    {0.3, 0.5, 1, -30},
    {1.6, 0.5, 0, 0.5 / 0.45},
    {0.5, 0.5, 0, -0.5 / 1},
    {1.4, 0.5, 0, 0},
};
```

Listing 6: Best path

In the given array each line represent the following: meters to travel, linear velocity, useHeadingController (true/false), reference (heading or turnrate). The turnrate reference is calculated with $\omega = v/R$.

It can be noticed that, although in real life the robot followed the path almost exactly, the parameters are not exact (e.g. the last is 1.4m instead of 1.5m). This probably comes from the fact that the odometry data we're relying on is not precise.

VII. Results

To get a hold of the real-life results, we logged and plotted a series of measurements from the real robot. The best performing run (following the path of listing 6) is shown in fig. 10. As we can see we use the heading controller just for the first part, while we continue to use the turnrate controller for the rest.

The response, although not immediate (which is to be expected, since the robot needs some time to physically accelerate and move) tracks perfectly the given reference. This can be seen also in fig. 12 where we tried a different configuration activating again the heading controller towards the end. Here we can notice especially that that the heading controller is quite slow. This was to be expected, since from the bode of the controlled open loop we had an ω_c of only $2.49 rad$.

Once again we can notice the uncertainty of the odometry. Although in real life the run of figure 10 performed the best, the logged data shows a final heading of around 140° (as it is reflected on the pose chart, fig. 11), while in reality the robot followed a pretty straight line at around 180°. It has to

be noted that this data has been captured on one of the robot with thin wheels, which probably added to the error (since there is more slipping while turning).

In fig. 12 and 13 we can see logs of what apparently looks like a very good run. Instead, for the reason stated before, the robot goes quite out of line, with a real final angle much greater than 180°.

It is worth noting that the plot of the pose is calculated directly with the (discrete) formulas of section II-B, hence it inherently linked to the error of the wheel odometry (and it does not show the real pose).

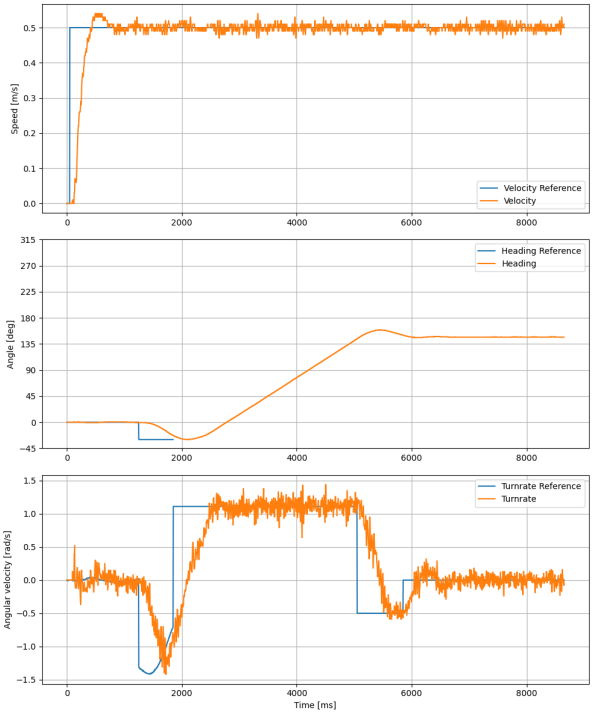

Fig. 10: Data from the best score run

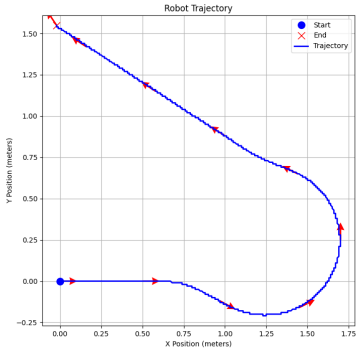

Fig. 11: Logged pose of the best score run

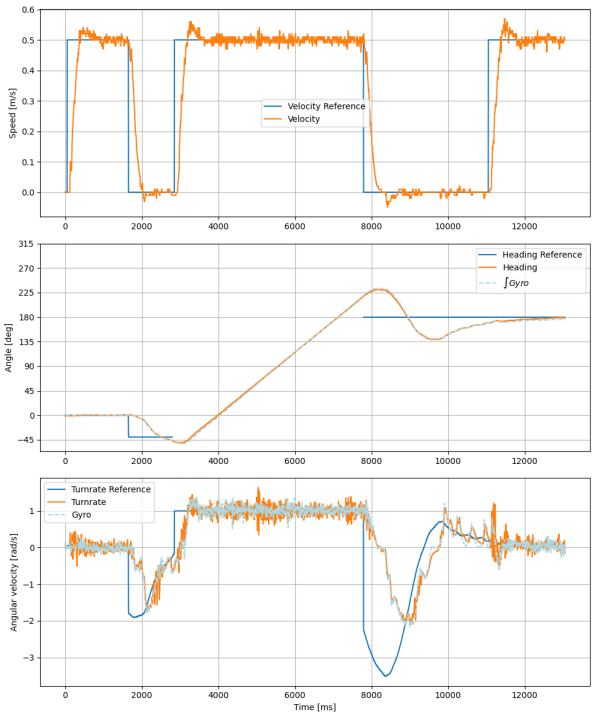

Fig. 12: Data of another run

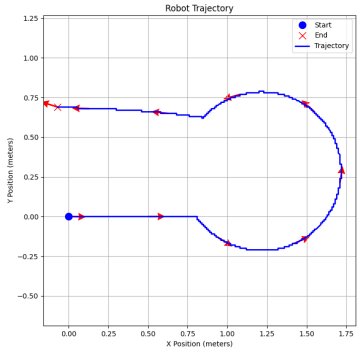

Fig. 13: Logged pose of another run

Appendix

Appendix A
Controllers

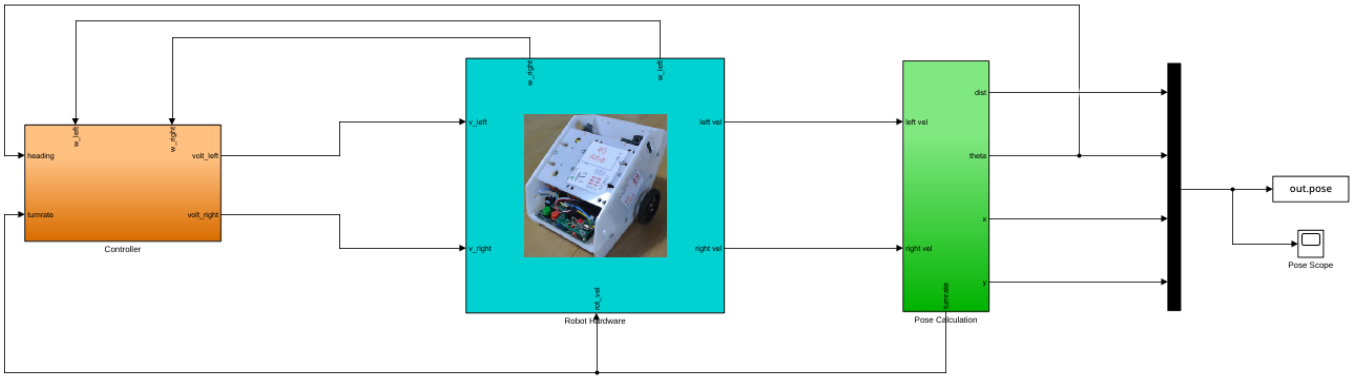

Fig. 14: Complete Simulink

Appendix B
Turn-rate transfer function

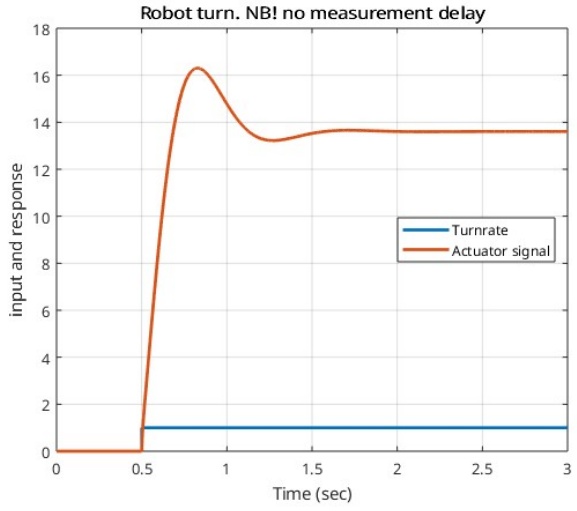

Fig. 15: Input-output measurements for the turnrate t.f. estimation

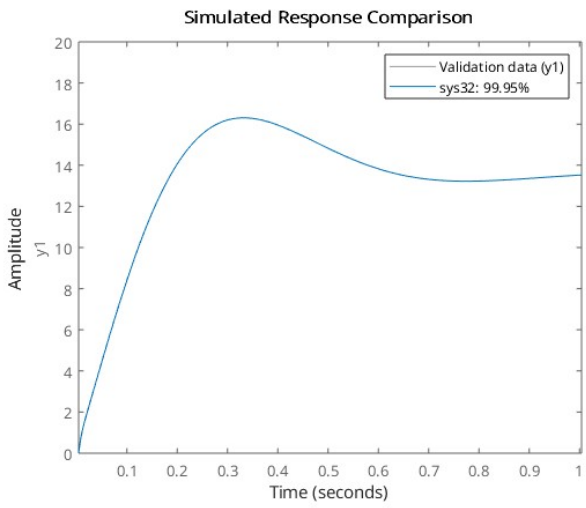

Fig. 16: `compare(sys, idd)` for the turnrate t.f.

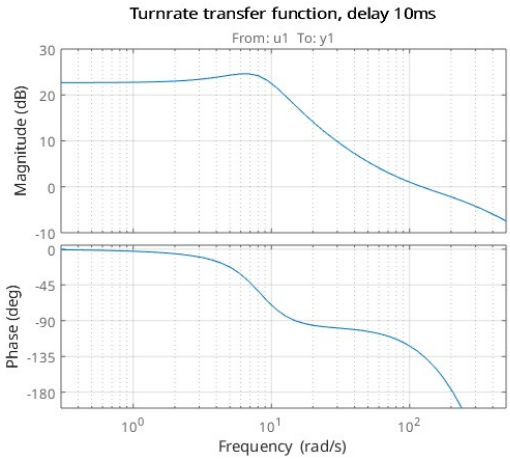

Fig. 17: Bode plot of the turnrate t.f.

APPENDIX C
HEADING TRANSFER FUNCTION

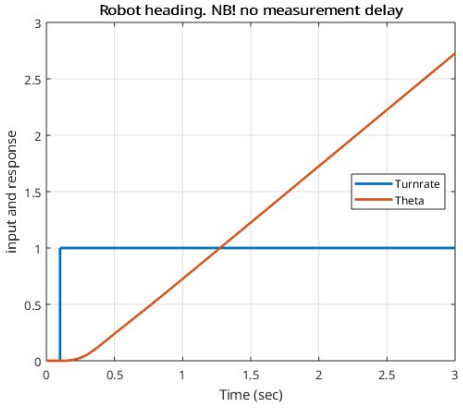

Fig. 18: Input-output measurements for the heading t.f. estimation

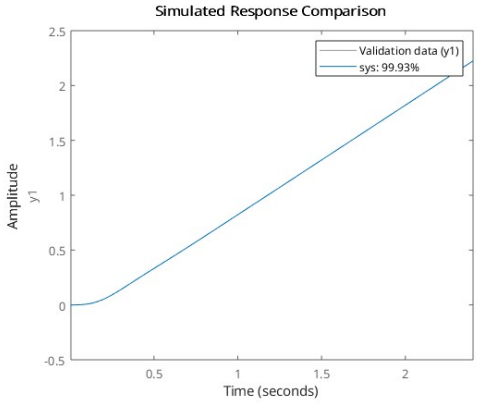

Fig. 19: `compare(sys, idd)` for the heading t.f.

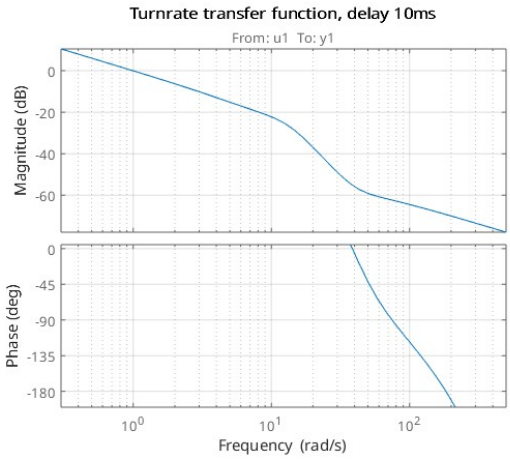

Fig. 20: Bode plot of the heading t.f.

APPENDIX D
CLOSED-LOOP STEP RESPONSES

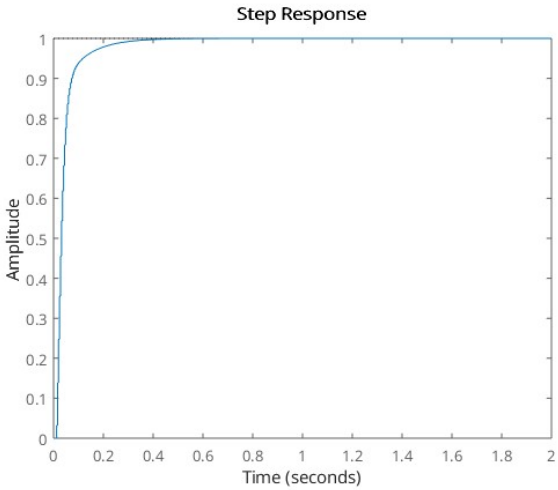

Fig. 21: Step output of closed-loop for the velocity controller

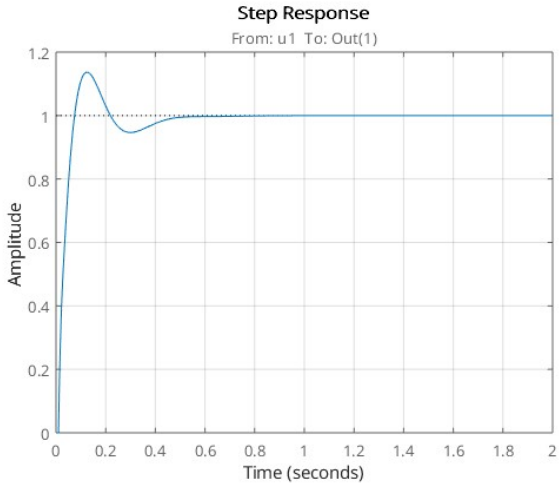

Fig. 22: Step output of closed-loop for the turnrate controller

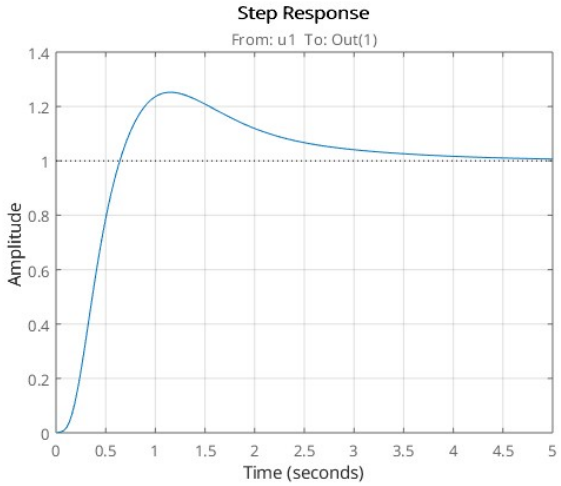

Fig. 23: Step output of closed-loop for the turnrate controller