DYNAMICS AND MOTION CONTROL

MF 2007

**Workshop B**

Servo control, code generation and robustness

Diego González Morín

Sofía Navarro Heredia

Philipp Rothenhäusler

Table of content

1. Introduction2
2. Parameter identification3
   1. Linear friction and inertia3
   2. Parameters of the nonlinear friction model6
3. Velocity control1
   1. Discrete time controller 2
   2. System responses 2
4. Position control4
   1. Discrete time controller 2
   2. Continuous time controller 2
5. **Introduction**

This project is focused on 3 important techniques need to design a robust and efficient controller.

In the first, a Servo controller is developed, using a trajectory planner, in order to control the position of a model of a DC motor, setting the values of the acceleration or speed. In the next section, this Servo controller is implemented in Matlab language to emulate how the implementation of a controller could be approached in an actual embedded system. The generated code is tested both in the model and the actual DC motor.

Last but not least,

PHILIPPPPP

The objective of this report is then to describe how all these methods have been implemented, and perform a later quantitative and qualitative analysis.

1. **Servo control**

Servo controllers are generally implemented in three occasions: the reference signal varies over a large range, we observe some signals of saturation in the process or we need to control not only the position, but also the speed and acceleration. By the addition of a trajectory planner we make sure that the controller do not saturates under normal operation conditions. In order to control the position, velocity and acceleration, is necessary to implement a model following feed forward control law.

In the case of our controller, we implemented both a trajectory planner and a model following feed forward control law. Both the designing process and experimentation will be described in this section.

* 1. Trajectory planner

As it was stated before, one problem that can appear while designing a controller is saturation of the input signal. This means that the controller’s output is too low/high, and the input to the models will not be the one calculated by it. This will lead to not desired response. In order to get rid of this issue, a trajectory planner can be implemented.

In the case of the DC motor that have been used in this project is that it saturates easily when a PID controller is applied. To avoid this saturation of the controller, the maximum values of the acceleration and speed were calculated:

1. First it was obtained the maximum current and voltage given in the datasheet of the motor (24 V and 182 mA). We chose to use as the maximum Only the 80% of these values of the current and voltage were used as the maximum to calculate the maximum speed and acceleration.

MaxCurrent = 0.182\*0.8;

MaxVoltage = 24\*0.8;

The maximum speed and acceleration were calculated using the total inertia (Jeq), the torque constant (Kt) and the back EMF constant (Kemf)

Max\_Speed = MaxVoltage/Kemf;

Max\_Accel = Kt\*MaxCurrent/Jeq;

1. In order to design the reference trajectory planner, it was taken into account that there were any dynamic specifications, therefore, the response of the motor did not have to be fast. The easiest way to approach the trajectory planner was to design the acceleration signal and then integrate it to obtain the speed and position. Also, as there were no dynamic specifications, the trajectory planner was designed such in a way that it accelerates for a time t, has 0 acceleration for a time t and then decelerates for t. Even that it was not necessary to develop a quick system, the time t was chosen to reduce the settling time having a velocity and acceleration inside the maximum boundaries. The objective was to reach a certain position after the acceleration, steady time and deceleration.

This was done by setting the time t (in the case of this project 0.5 s) and calculating the acceleration needed to reach the aimed position.

The aimed position that were going to be used during the experimentation were 10 rad and 100 rad. Once the acceleration calculated, the only step left was to compute the signal of the acceleration. To do so, the signal builder block was used. After the acceleration was integrated, the trajectories shown in the figures 1 and 2 for the acceleration, velocity and position were obtained. This integration was done in the discrete time, as the controller was going to be designed in the discrete time.

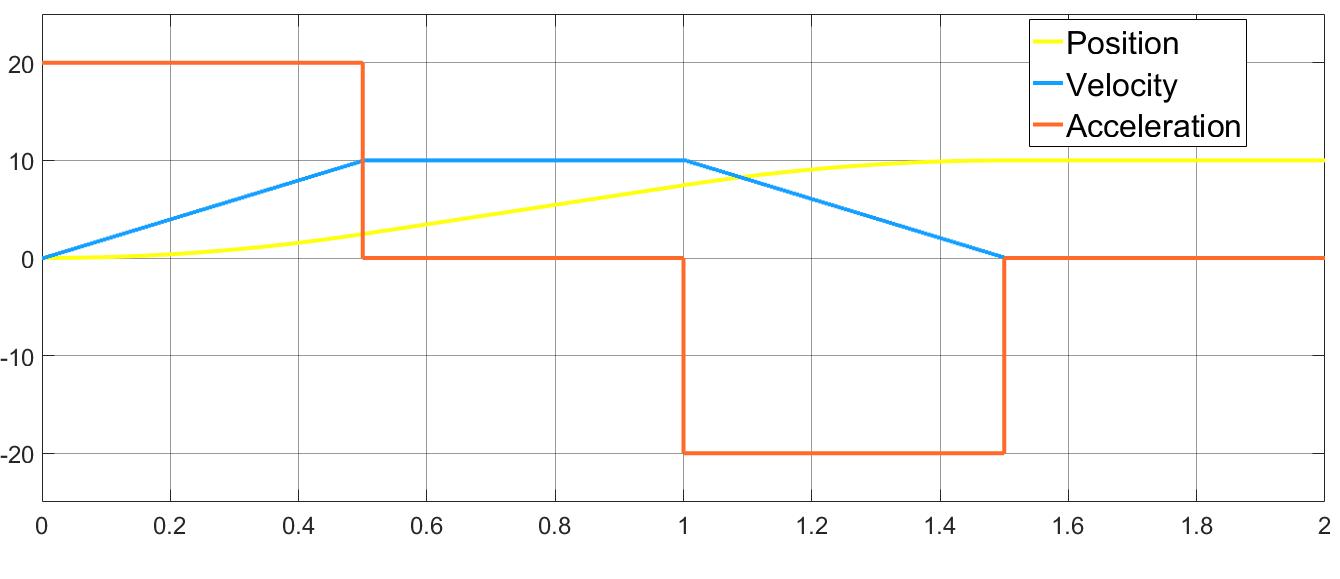


Figure 1: Trajectory planner out for aimed position of 10 rad

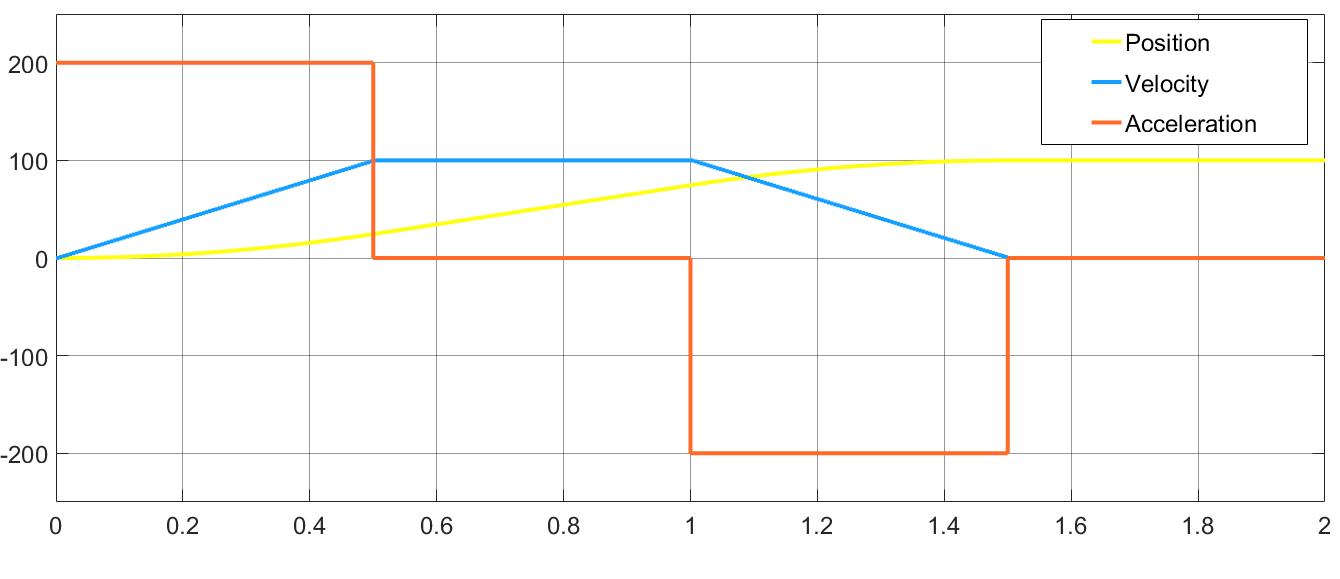


Figure 2: Trajectory planner out for aimed position of 10 rad

* 1. Model following control

As it was introduced before, the model following control was born as a method to allow the simultaneous control of the position, velocity, and acceleration. This can be done as the position can be several times differentiable in the case of the DC motor model. The objective now is to invert the process model in order to calculate the voltage we need to follow the trajectories of the acceleration, speed, and position.

In the case of the DC motor, having the voltage as the input, the model of the system is:

(1)

(2)

(3)

(4)

(5)

From these equations we get that:

(6)

The implementation of these equation in a Simulink block is direct once we have the speed and acceleration, and it was done as it is shown in the figure 3.

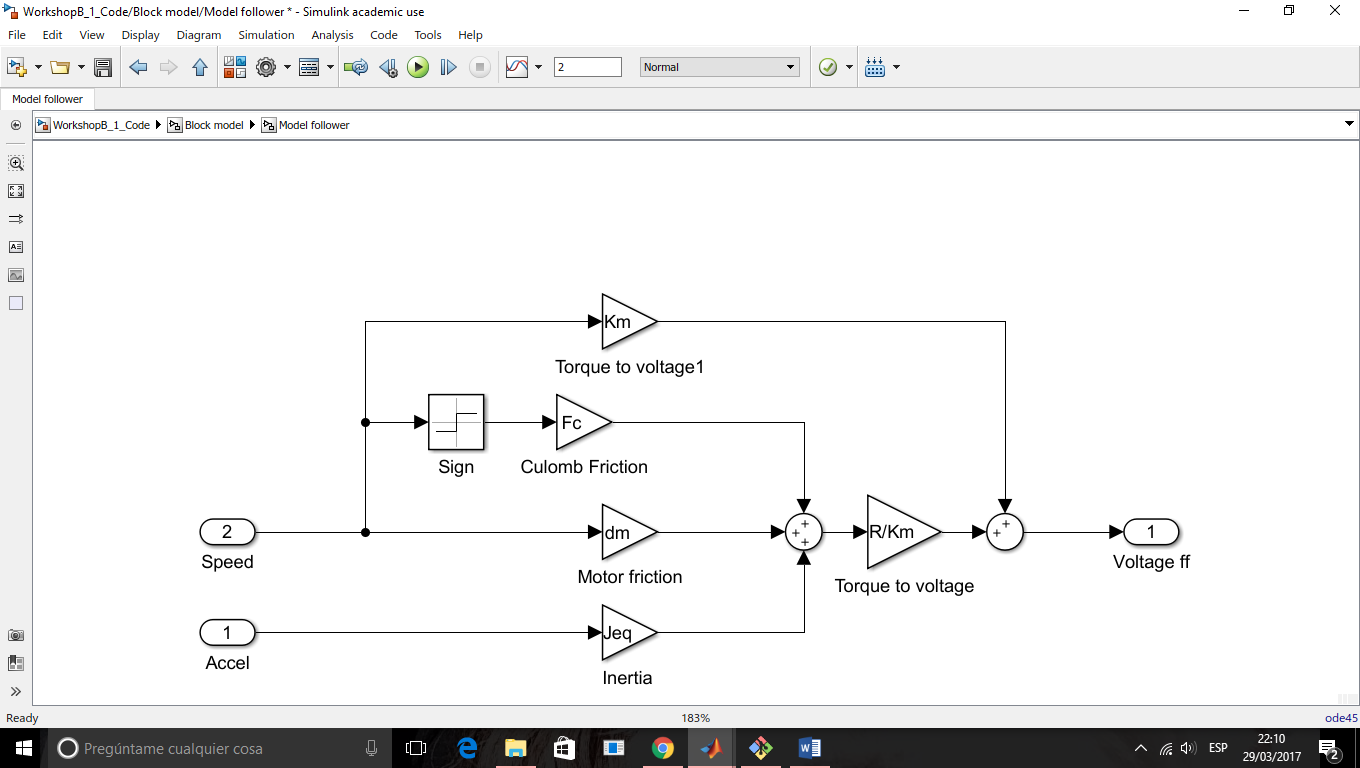


Figure 3: Simulink implementation of the model following controller

* 1. Implementation of the servo control.

The simultaneous implementation of both the trajectory planner and the model follower has to have the same structure as the one shown in the figure 4.

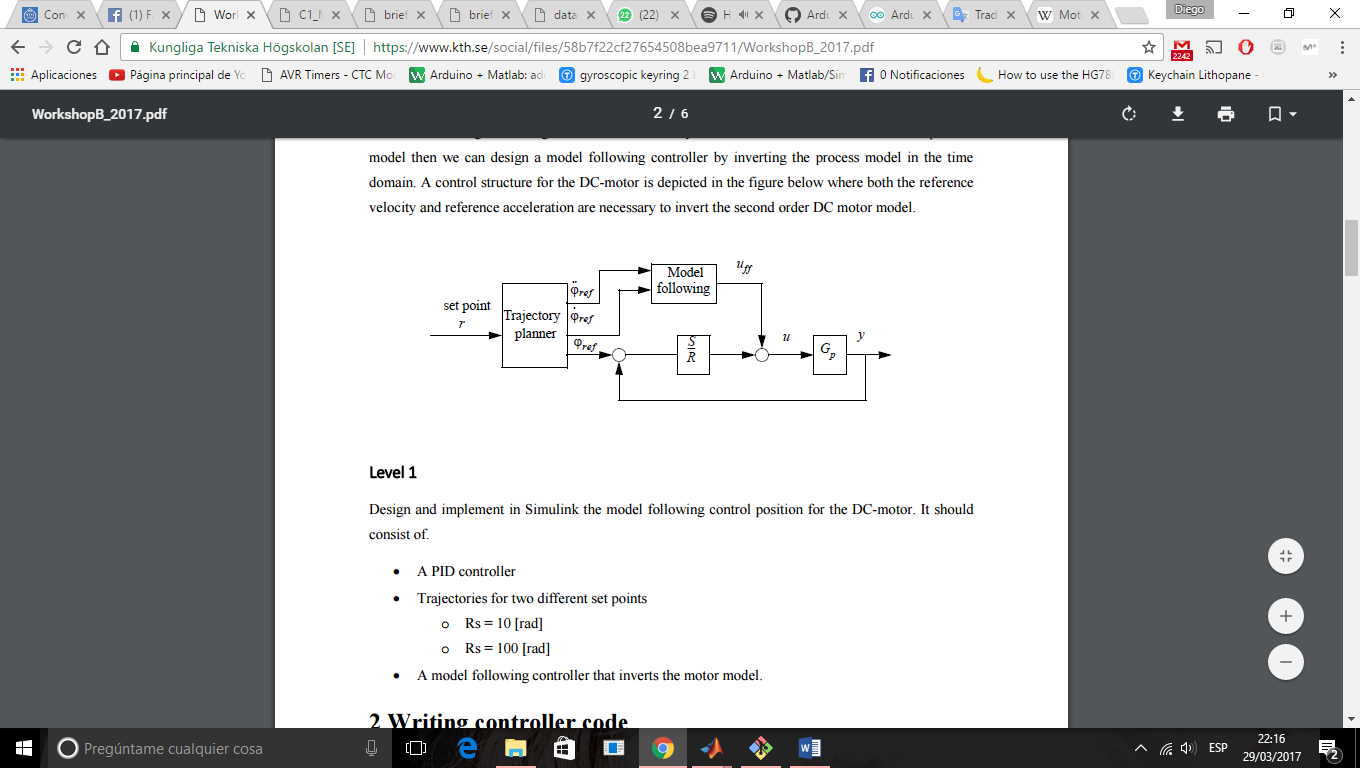


Figure 4: Structure of the servo controller

Both the Trajectory planner and the Model follower have already been implemented. The last step is then to estimate a proper PID error controller. This controller has to be designed in the discrete time, as the measurement of the position is taken by a discrete encoder. Taking into account that the system did not have to have a quick response, the following dynamic specifications were chosen:

* ts <= 0.2 s
* Overshood < 0.01%

Using pole placement, and a sampling time of 5 ms, the following controller was calculated:

Once this regulator has been implemented in Simulink, the last step bas to build everything together. The final system is shown in the figure 5.

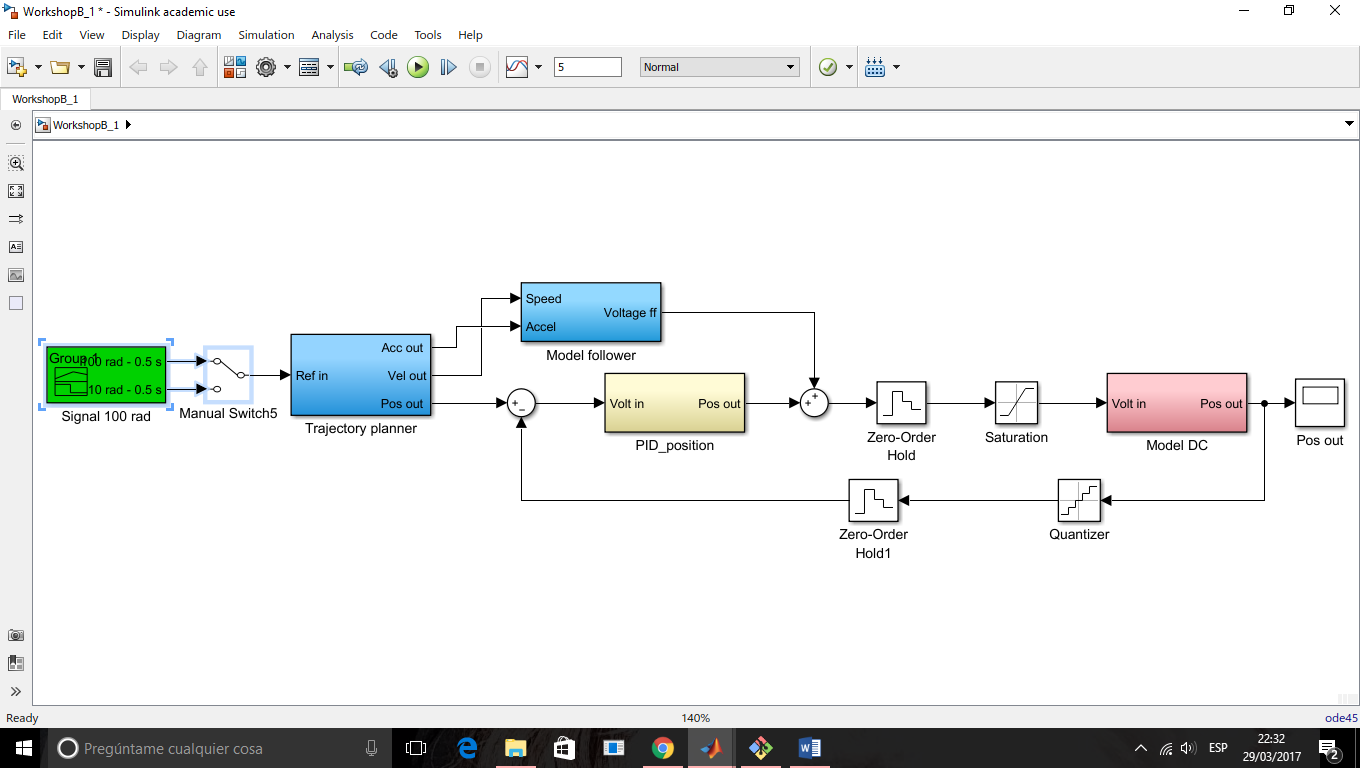


Figure 5: Final structure of the servo controller

Applying the acceleration trajectories need to get a final position of 10 and 100 we got the final output represented in the figures 6 and 7.

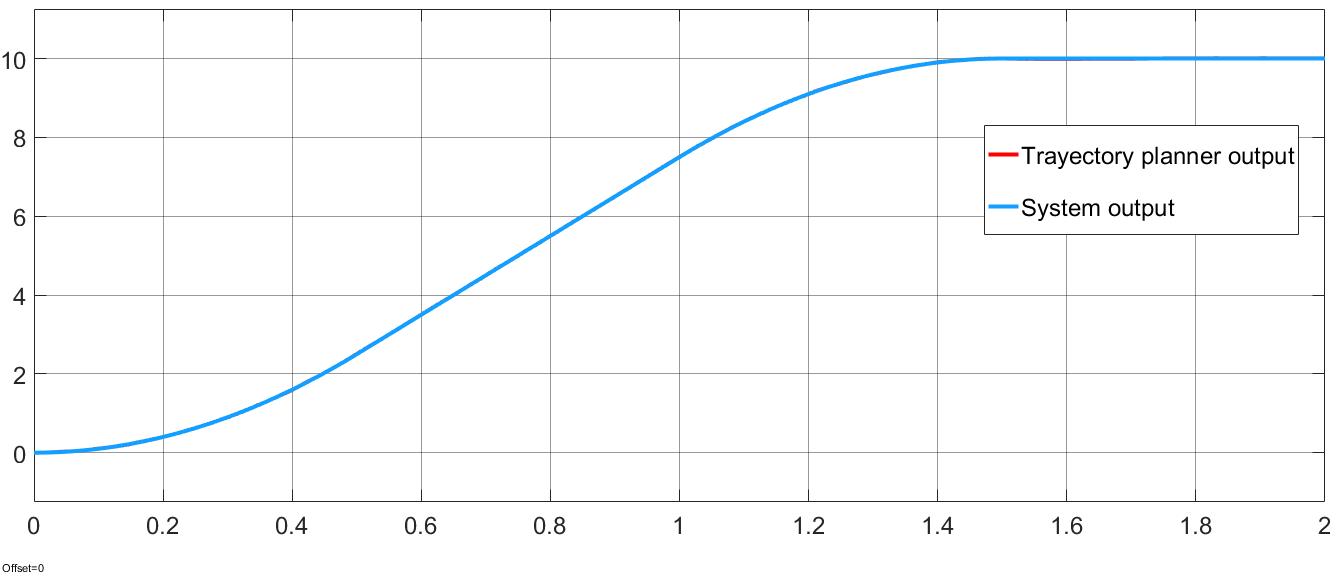


Figure 6: Output of the system compared to the trajectory planner position output for aimed position of 10 rad

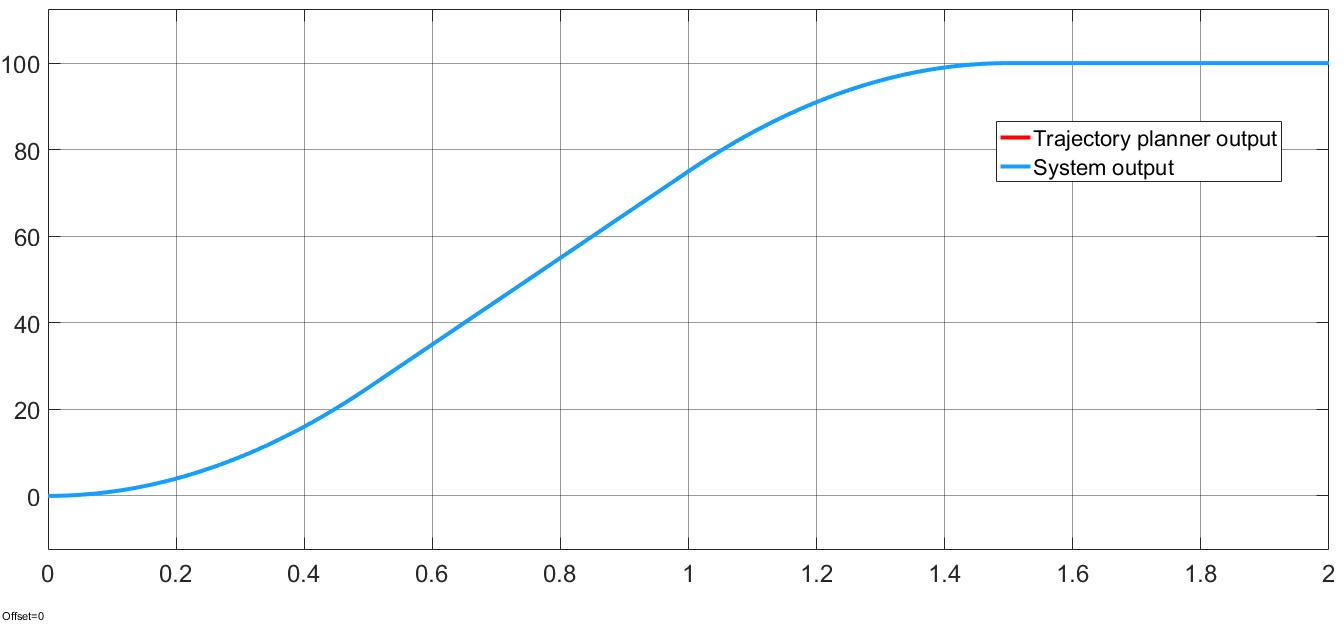


Figure 7: Output of the system compared to the trajectory planner position output for aimed position of 10 rad

Is easy to check that the system is able to follow the planned trajectory. It is also interesting to study the voltage output from the Model follower, and the entire Servo controller. In the figures 8 and 9, we can observe the output of the Model follower. This figure looks as expected, due to the fact that only the velocity and acceleration were used to invert the model.

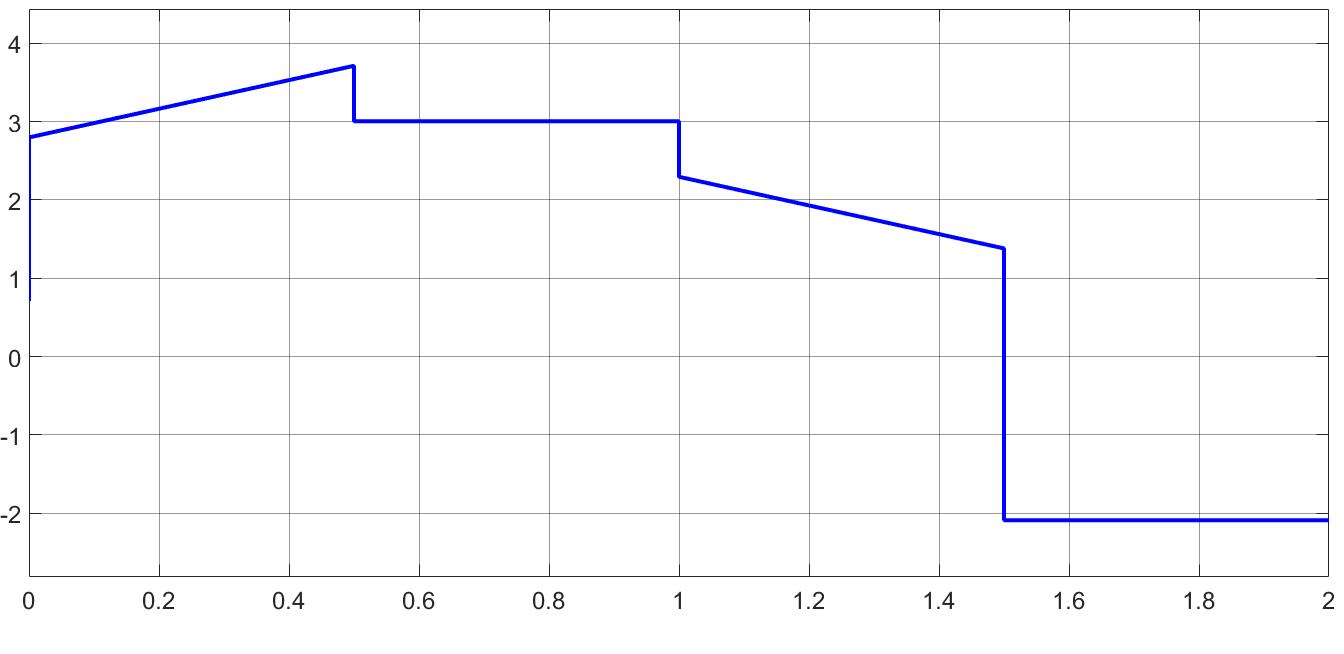


Figure 8: Voltage output from Model follower for aimed position of 10 rad

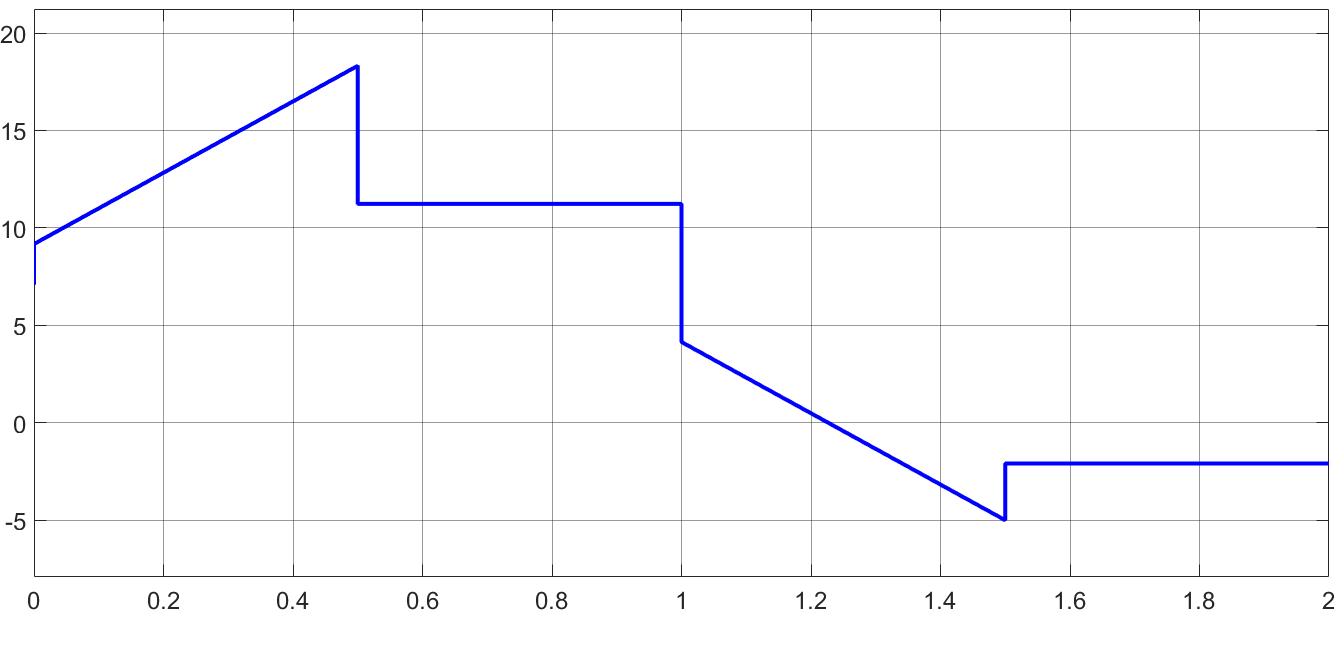


Figure 9: Voltage output from Model follower for aimed position of 100 rad

In the figure 10 is easy to observe that the voltage does not reach the maximum voltage in any of the cases.

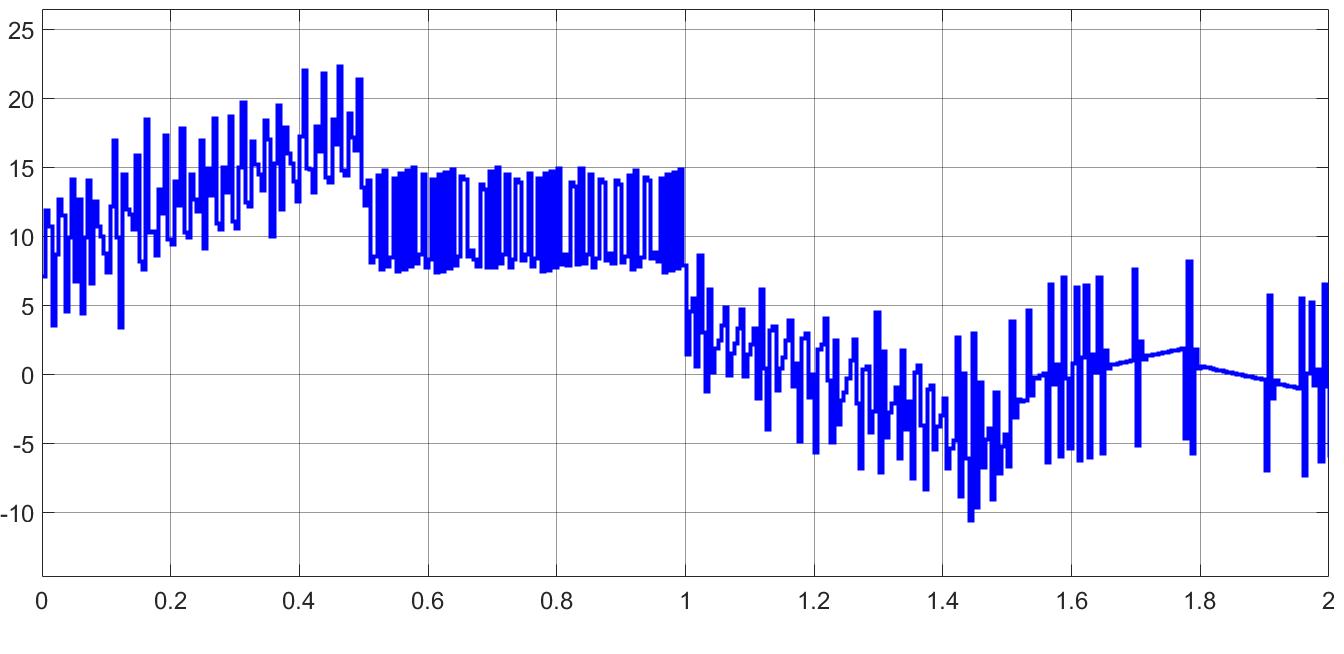
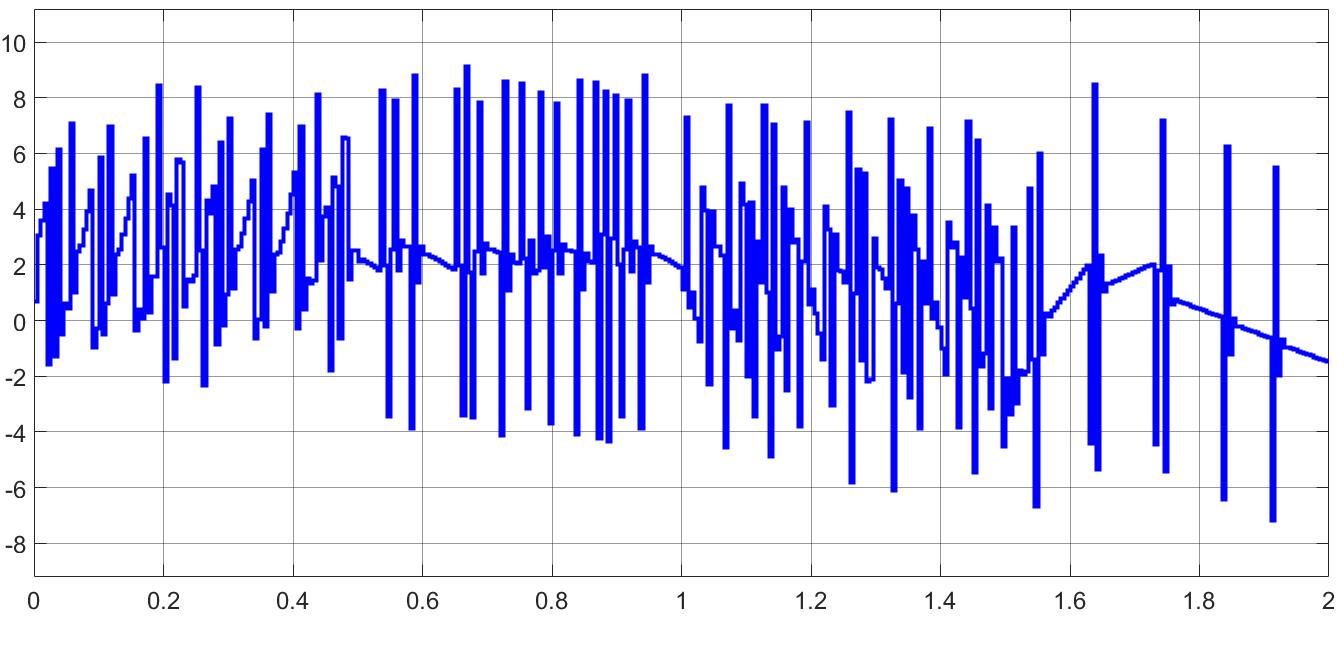


Figure 10: Voltage output of the Servo controller for 10 and 100 rad respectively

1. **Writing controller code**

The objective now is to implement the controller developed in the previous section using written code, trying to emulate how embedded systems work and are programmed. The code is implemented in a Matlab function playing the role of the microcontroller (figure 11).

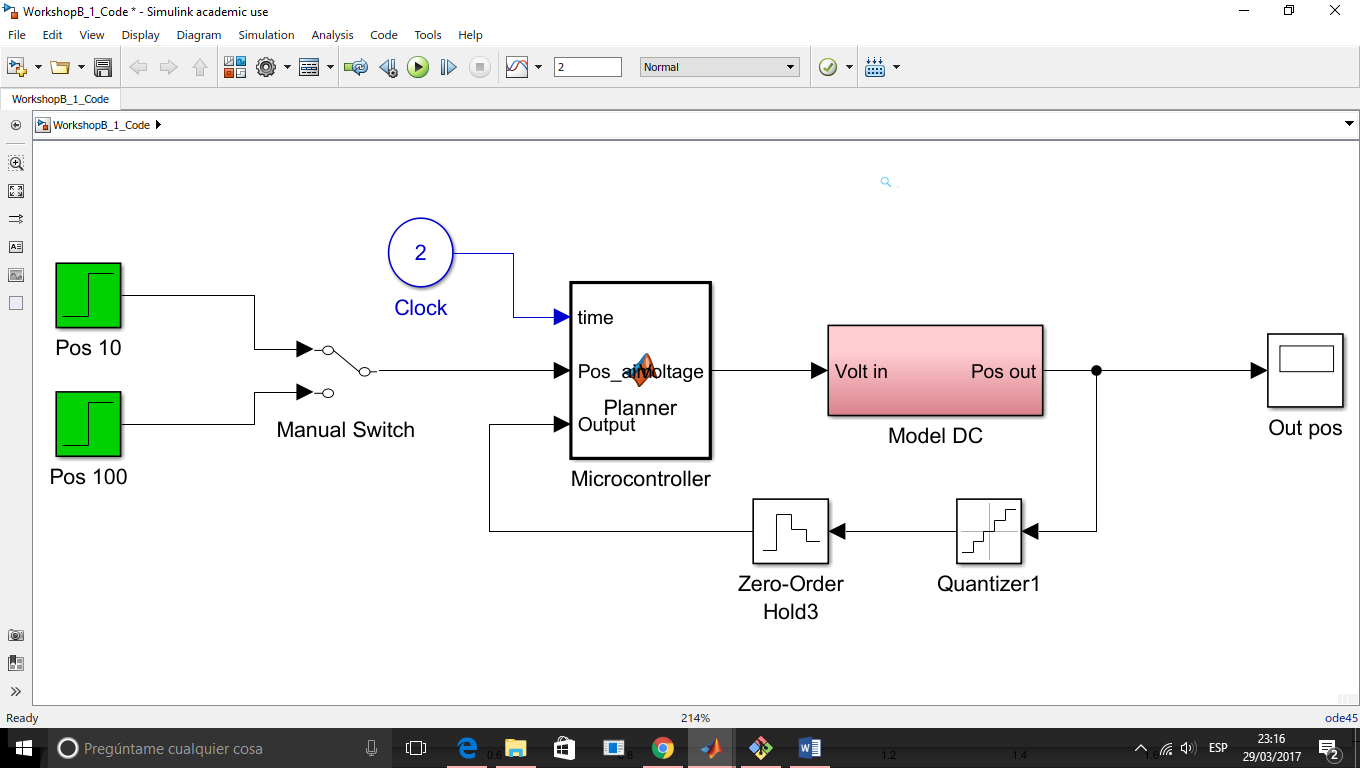


Figure 11: Structure of the system using a Matlab function as the microcontroller

The clock is used to count the time, needed to generate the correct trajectory in the Trajectory planner.

The code inside the function is the following:

function Voltage = Planner(time, Pos\_aim,Output)

%%%%%%%%%%%%%%%%%%%%%%%%%

%%Parameter declaration%%

%%%%%%%%%%%%%%%%%%%%%%%%%

R = 112;

Km = 69.7E-3;

Kemf = 1/(137\*2\*pi/60);

dm = 3.8E-6;

Jm = 7.46/(1000\*100^2);

J1 = 1.8E-5;

J2 = 1.8E-5;

n = 1;

Jeq = Jm + (J1+J2)/(n^2);

%Calculated parameters for the DC motor used in the workshops

eps\_Jeq = 0.6; % Scaling factor for inertia

eps\_dm = 3.56; % Scaling factor for dumping

Jeq = eps\_Jeq\*Jeq;

dm = dm\*eps\_dm;

%Coulomb friction

Fc = 0.0013;

%Sampling time

Ts = 5e-3;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%Trayectory planner - Signal generation%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Once the rising time is defined to 0.5 s.

%The motor will be accelerating during 0.5s.

%Then it will have 0 acceleration over 0.5 s. Last step would

%be that it accelerate in the opposite direction for 0.5 s. This

%acceleration will be one such that it will never pass the maximum

%acceleration( calculated in the following lines) and that ensures

%that the final position is the one desired.

%Motor characteristics - from datasheet. We use a margin of 20% as

%the PID controller will need some margins for operate correctly.

MaxCurrent = 0.182\*0.8;

MaxVoltage = 24\*0.8;

%Max dynamics

Max\_Speed\_ref = MaxVoltage/Kemf;

Max\_Accel\_ref = Km\*MaxCurrent/Jeq;

%Definition of trising

trising = 0.5;

%We calculate the speed and acceleration that we need to get to

%the position given the trising time.

Max\_Speed = Pos\_aim/(2\*trising);

Max\_Accel = Max\_Speed/trising;

%Check that we are inside the Max\_acceleration and

%Max\_speed boundaries

if Max\_Speed > Max\_Speed\_ref || Max\_Accel > Max\_Accel\_ref

disp('Error, Speed/Accel too high')

pause();

end

%Definition of the time that the motor will have 0 acceleration

t\_est = (Pos\_aim - Max\_Speed\*trising)/Max\_Speed;

%If we haven't reached the trising time, the motor accelerates

if time < trising

signal\_accel = Max\_Accel;

%If we reach this values, and have not reached t\_est time,

%the motor has 0 acceleration

elseif time >= trising && time < (trising + t\_est)

signal\_accel = 0;

%If we pass t\_est, then it accelerates again but inversely

elseif time >= (trising + t\_est) && time < (2\*trising + t\_est)

signal\_accel = - Max\_Accel;

%when it has accelerated during trising, the acceleration remains

%steady in 0.

else

signal\_accel = 0;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%Trayectory planner - Acceleration integration%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%We need to integrate the acceleration: once to get the

%velocity and twice to get the position. This is done,

%in the discrete time:

%

% - Speed = (Ts/(z-1))\*Accel

% - Pos = (Ts/(z-1))\*Speed

%

%Translated this to code-style language, we have:

%

% - Speed(n) = Ts\*Accel(n) + Speed(n-1)

% - Pos(n) = Ts\*Speed(n) + Pos(n-1)

%We need to store values from previous calls of this function:

%Declaration of Speed(n\_1)

persistent outspeed\_n\_1;

%Declaration of Pos(n\_1)

persistent outpos\_n\_1;

%Initialization to 0 of these variables

if isempty(outspeed\_n\_1)

outspeed\_n\_1 = 0;

end

if isempty(outpos\_n\_1)

outpos\_n\_1 = 0;

end

%The acceleration is not changed, we just need it to calculate the

%Speed and the postion. Storing the values for the next time step

Speed = signal\_accel\*Ts + outspeed\_n\_1;

outspeed\_n\_1 = Speed;

Position = Speed\*Ts + outpos\_n\_1;

outpos\_n\_1 = Position;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%Model planner - inverse model%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% We inverse the motor model to get the voltage we need to get a

% certain acceleration and voltage.

Voltage\_Planner = (Speed\*dm + signal\_accel\*Jeq + **. . .**

sign(Speed)\*Fc)\*R/Km + Km\*Speed;

%%%%%%%%%%%%%%%%%%%%

%%PID - controller%%

%%%%%%%%%%%%%%%%%%%%

%

%We calculated a PID controller for the DC motor model using Pole

%Placement, implemented as a matlab function for both workshops A

%and B. We did not need a fast-response controller. The controller

%we designed is:

%

% 758.5 z^2 - 1393 z + 640.5

% PID = ---------------------------

% z^2 - 0.9268 z - 0.07323

%

%Rewriting this in a code-style, we get:

%

%Out(n) = In(n)\*758.5 + In(n-1)\*(-1393) + In(n-2)\*640.5 –

% Out(n-1)\*(-0.9268) - Out(n-2)\*(-0.07323);

%We need again to declare some variables as persistent, as they

%have to be used in different iterations of the function

persistent Out\_n\_1;

persistent Out\_n\_2;

persistent In\_n\_1;

persistent In\_n\_2;

%Initialization of persistent variables

if isempty(Out\_n\_1)

Out\_n\_1 = 0;

end

if isempty(Out\_n\_2)

Out\_n\_2 = 0;

end

if isempty(In\_n\_1)

In\_n\_1 = 0;

end

if isempty(In\_n\_2)

In\_n\_2 = 0;

end

%Declaration of the parameters of the PID

PID\_den = [1 -0.9268 -0.07323];

PID\_num = [758.5 -1393 640.5];

%Is an error PID so we need to substract the actual position

%from the aimed position.

Error = Position - Output;

%Main PID function

Voltage\_Controller = Error\*PID\_num(1) + In\_n\_1\*PID\_num(2) + **. . .**

In\_n\_2\*PID\_num(3) - Out\_n\_1\*PID\_den(2) **. . .**

- Out\_n\_2\*PID\_den(3);

%Store the values for next iterations

In\_n\_2 = In\_n\_1;

In\_n\_1 = Error;

Out\_n\_2 = Out\_n\_1;

Out\_n\_1 = Voltage\_Controller;

%The last step will be to add both voltages, from the

%controller and the trayectory planner

Voltage = Voltage\_Controller + Voltage\_Planner;

end

Once the function has been implemented, the next step is to compare the performance between the Simulink block-based controller and the written Matlab function. (Figure 12 and 13)

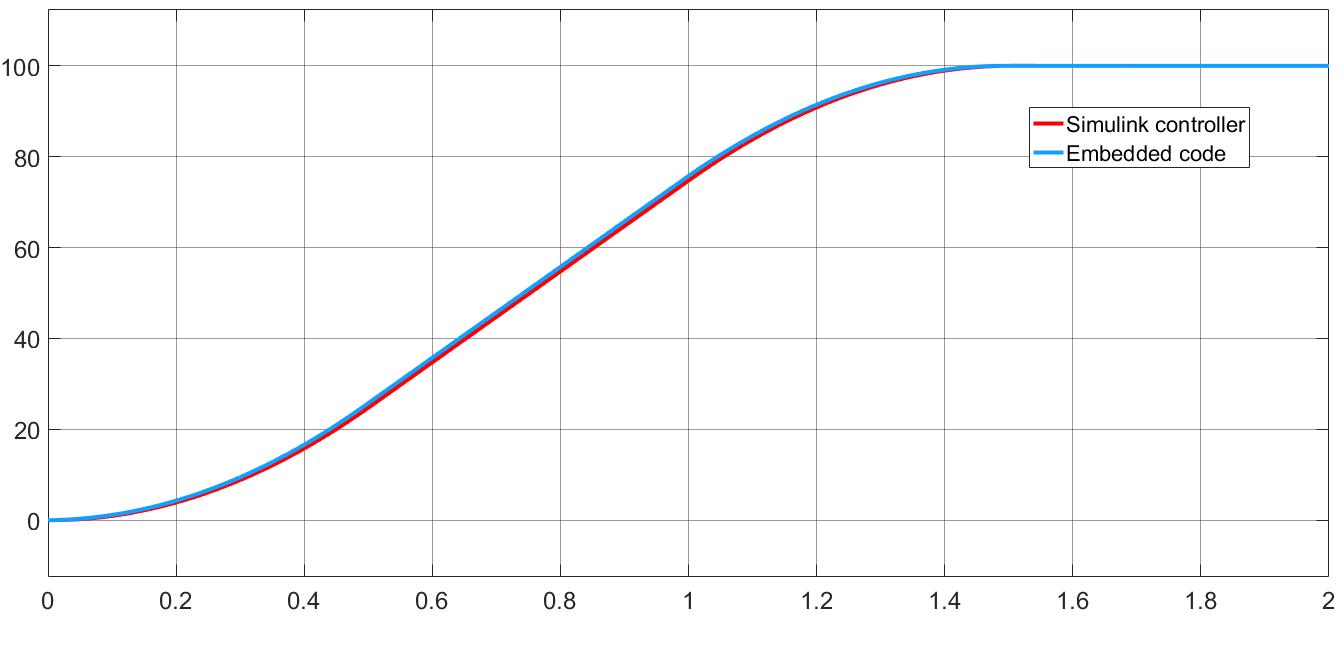


Figure 12: Comparison between embedded code and Simulink model for 100 rad.

In this figures is easy to observe that the output is not exactly the same. The main reason is that in the embedded code, when the integration of the Acceleration is done, the output is one period (Ts) quicker. (Figure 14). This is due to the fact that Simulink does the conversion of the signal from the Signal generator from continuous to discrete (using ZOH) while in the Embedded code, the integration is done directly in discrete time. This is why in the figure 14 we observe that the position and velocity have quicker output in the embedded case than in the Simulink case, remaining the same in the case of the acceleration, as this signal is not integrated nor transformed.

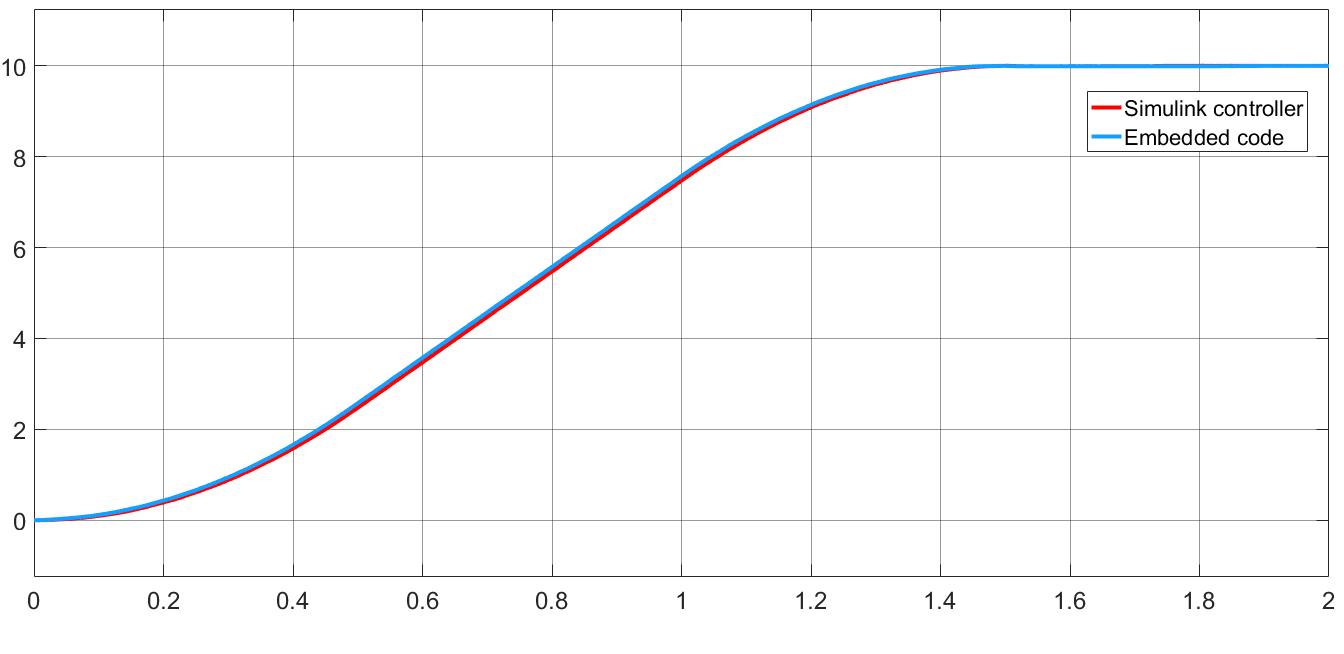


Figure 13: Comparison between embedded code and Simulink model for 10 rad.

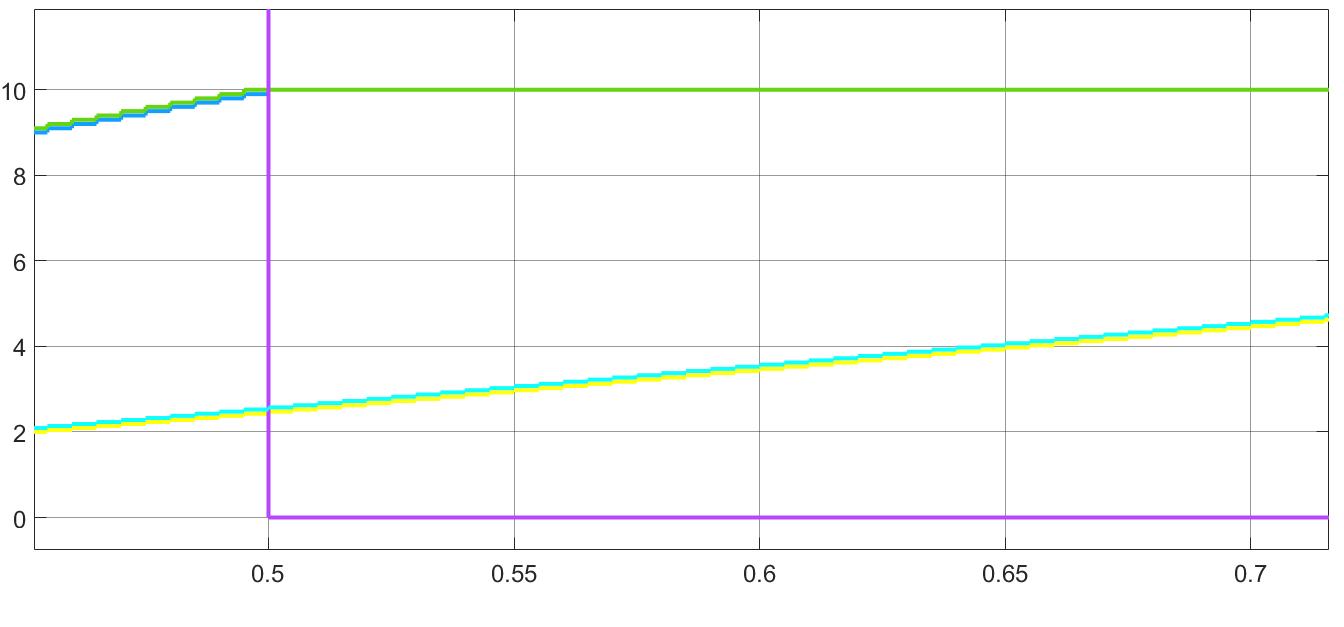


Figure 14: Detail of the outputs of the trayectory planner between the Simulink model and the Emedded code model