POLITECNICO
MILANO 1863

School of Industrial and Information Engineering

MSc in Mechanical Engineering

Autonomous Vehicles

2022-2023 Academic Year

Project 3

Isabel Pollini

Introduction

To complete this assignment, a path of at least 6 waypoints must be defined in an empty Gazebo world. Between each waypoint, spheres have to be generated, either of red colour or green.

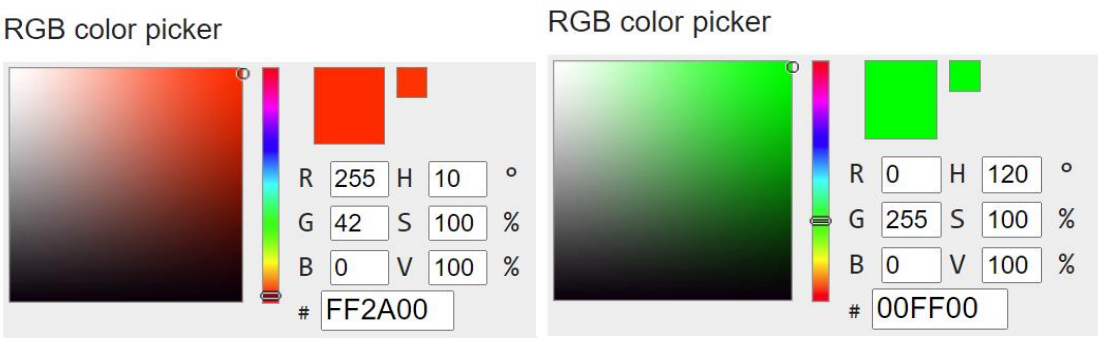
The request consists in the implementation of a feedback control that can lead the robot to pass through all the waypoints, stopping at the last one, while overtaking the red spheres on the left and the green spheres on the right.

World Settings

Firstly, the Gazebo world has to be set up correctly. The chosen path is formed by the following waypoints:

Waypoint	X [m]	Y [m]	θ [rad]
1	4	0	π/2
2	4	4	π/2
3	4	8	0
4	8	8	3π/2
5	8	4	3π/2
6	8	0	3π/2

For the spheres, the following RGB colour combinations where chosen:



The Matlab vectors that correspond to these specific colours are the following:

- Green: [0 1 0 1]
- Red: [255 42 0 1]

This shade of red was chosen because it has a Hue value different from 0, differently from the standard red [1 0 0 1], which allows to perform the HSV colour detection based on the hue for both colours.

Each sphere is placed exactly half-way through each pair of subsequent waypoints, alternating in colour, as shown in Figure 1.

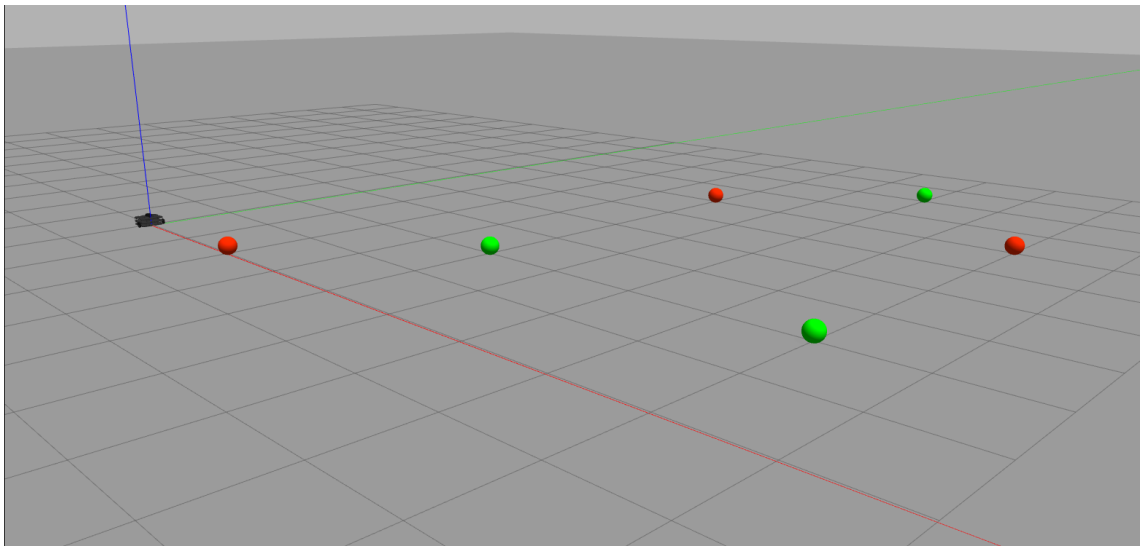


Figure 1. Gazebo world setting.

Simulink Model

Figure 2 provides an overview of the complete model realized in Simulink to complete this project.

It is composed by:

- Waypoint definition
- Colour detection subsystem
- Odometry subsystem
- Scan subsystem
- Velocity control function
- Velocity Publication subsystem
- Check colour function

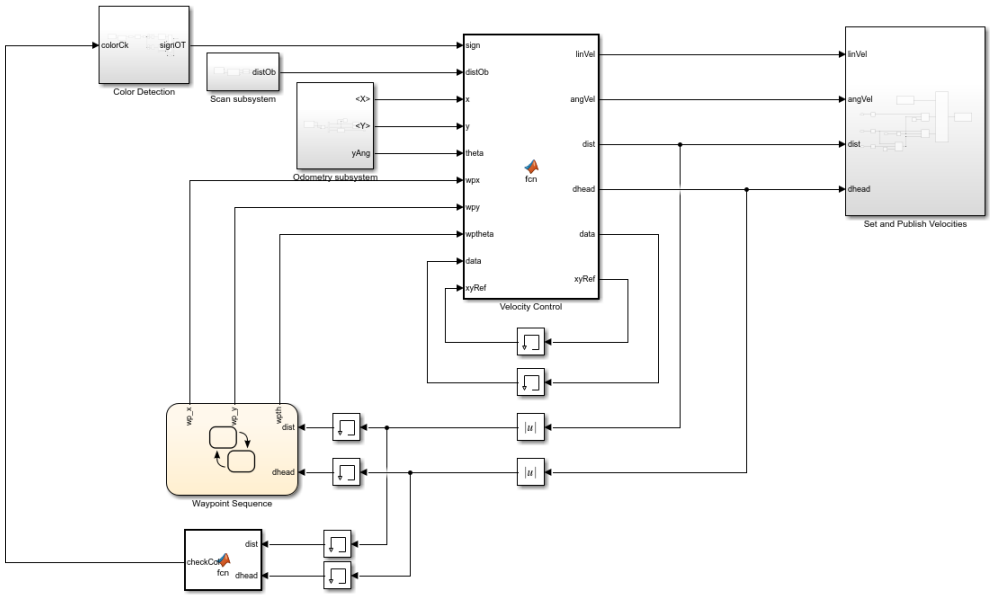


Figure 2. Complete Simulink model

The waypoints are defined by a state-flow block, that sets the next waypoint's coordinates (X, Y, θ) at the entry of each state and exits when the distance from the waypoint is less than 0.05m and the difference in orientation is less than 0.05rad. Once reached the last state, no operation is requested, therefore the robot stops.

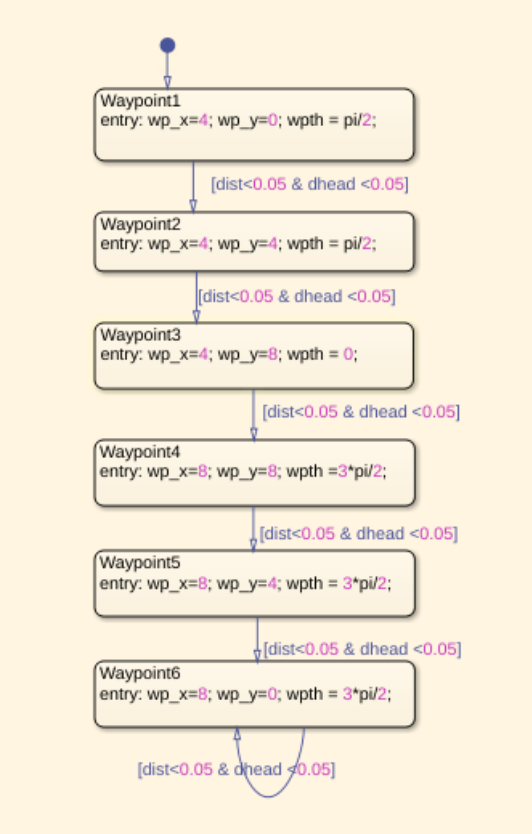


Figure 3. Waypoint sequence as a state-flow

The system is subscribed to the /odom topic, from whom it gets the real time position and orientation of the robot during the simulation. The orientation is communicated by ROS in quaternions, therefore it needs to undergo a transformation to eulerian angle before entering the control function. These processes take place inside the Odometry subsystem, shown in Figure 4, with the conversion having its dedicated subsystem, shown in Figure 5..

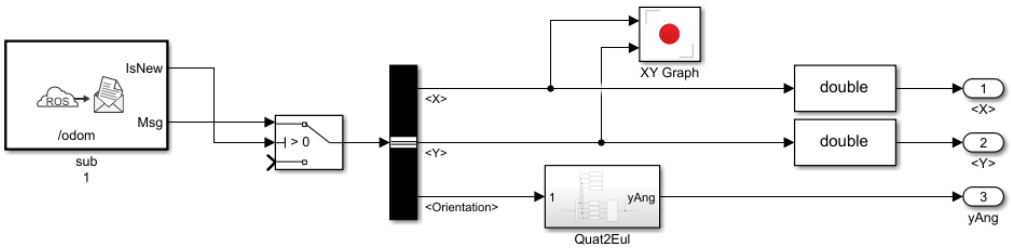


Figure 4. Odometry subsystem

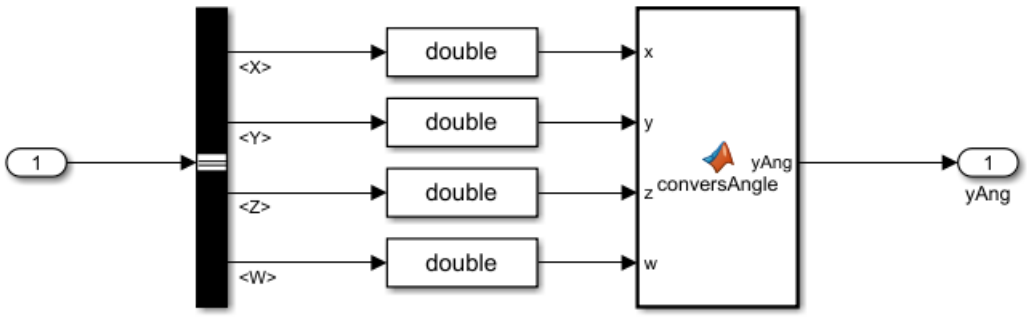


Figure 5. Quaternions-Euler angles conversion subsystem

The function that converts the orientation from quaternions to Euler angles is the following:

```
function yAng = conversAngle(x,y,x,w)
    eulAngle = quat2eul([w,x,y,z])
    yAng = eulAngle(1);
    check = isnan(yAng);
    if check == 1
        yAng = 0;
    end
end
```

To implement the obstacle avoidance, the system must be subscribed to the /scan topic, in order to get information on the nearby obstacles. In the Scan subsystem, shown in Figure 6, the messages are read by the Read Scan block, that gives the ranges and angles messages as input.

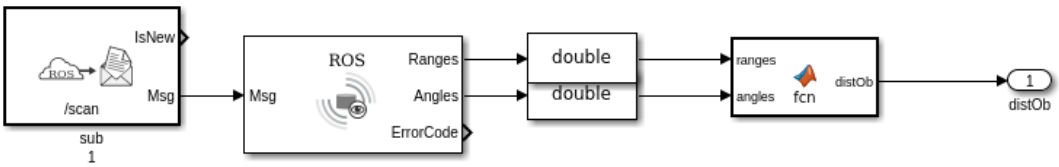


Figure 6. Scan subsystem

Starting from these messages it is possible to compute the distance to the closest obstacle with the following Matlab function.

```
function distOb = fcn(ranges,angles)
    if any(ranged>0)
        xOb    = ranges.*cos(angles);
        yOb    = ranges.*sin(angles);
        xObVal = xOb(isfinite(Ob));
        yObVal = yOb(isfinite(Ob));
        distOb = min(sqrt(xObVal.^2 + yObVal.^2));
    else
        distOb = 0;
```

```
    end
end
```

Since the behaviour of the obstacle avoidance control depends on the colour of the spheres, a colour detection subsystem is needed (Figure 6). After having subscribed to the bot's camera, the system reads the image and filters it for the hue values of the shade of red and green assigned to the spheres, to then analyse the area of each of the detected spheres to determine the closest for both colours.

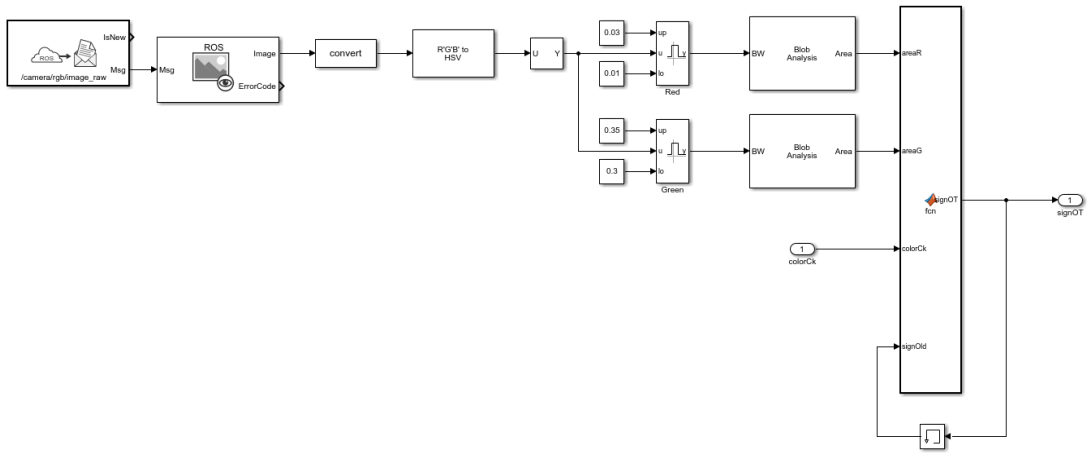


Figure 7. Colour detection subsystem

A Matlab function compares the areas of the closest red sphere and the closest green sphere, and determines the closest of the two, providing then as output an indicator of the direction in which the overtake must be performed, which will be fed to the control function.

```
function signOT = fcn(areaR,areaG,colorCk,signOld)
[maxAreaG,iG] = max(areaG);
[maxAreaR,iR] = max(areaR);

if colorCk == 1
    if ~isempty(iG) && ~isempty(iR)
        if maxAreaR > maxAreaG
            signOT = -1;
        elseif maxAreaG > maxAreaR
            signOT = 1;
        else
            signOT = signOld;
        end
    else
        signOT = signOld;
    end
else
    signOT = signOld;
end
end
```

It can be seen that this Matlab function only updates the indicator when requested by a Boolean variable, implemented in the following function, which makes sure that the update happens in the first instants of a new state.

```
function checkCol = fcn(dist,dhead)
    if dist < 0.05 && dhead < 0.05
        checkCol = 1;
    else
        checkCol = 0;
    end
end
```

The distance from the closest obstacle, the overtake indicator, the position and orientation of the bot and the coordinates of the waypoints are inputs to the control function. The other input variables, 'data' and 'xyRef', are vectors constituted by indicators that activate or disactivate different parts of the code.

The control is mainly made of three stages:

- Moving towards the waypoint, obtained with a proportional control on the linear and angular velocities, both with proportionality coefficients equal to 0.07. This action is performed before and after the overtaking process. The exit condition for this stage is given by the proximity to an obstacle being lower than 1 m.
- First stage of the overtaking process, which consists in the rotation of 90° towards direction indicated by the colour detection subsystem. Linear velocity is equal to 0, while the angular velocity is controlled by a proportional control with the same coefficient reported before. The exit condition for this stage is given by the orientation difference between current angle and target angle, which must be lower than 0.03 rad.
- Second stage of the overtaking process, which consists in a uniform circular motion, with the angular velocity set at the constant value of 0.06 rad/s and the linear velocity calculated as the product between the angular velocity and 'R', an estimate of the distance of the robot from the centre of the sphere. The exit condition for this stage is given by the distance of the robot from the starting point of this stage, i.e., the point in which the first stage of the overtaking process took place, which must be at least 30% longer than 'R'.

The control function code is here reported.

```
function [linVel,angVel,dist,dhead,data,xyRef] =
fcn(sign,distOb,x,y,theta,wpx,wpy,wptheta,data,xyRef)
check   = data(1);
checkP  = data(2);
R       = data(3);
distOT  = data(4);
distRef = data(5);
theta0  = data(6);

dist    = sqrt((x - wpx)^2 + (y - wpy)^2);
course  = atan2(wpy - y,wpx - x);

kpv     = 0.07;
kpw     = 0.07;
```

```
if distOb < 1 && distOb > 0
     distOT = distOb;
end

if distOT < 1 && distRef <= 1.3*R
    if checkP == 0
        theta0 = theta;
        checkP = 1;
    end
    if check == 0
        ang     = -sign*pi/2 + theta0;
        perp    = abs(ang - theta);
        if abs(perp) <= 0.03
            linVel      = 0;
            angVel      = 0;
            xyRef(1)    = x;
            xyRef(2)    = y;
            R           = distOT + 0.1;
            distRef     = sqrt((x - xyRef(1))^2 + (y - xyRef(2))^2);
            check       = 1;
        else
            linVel  = 0;
            angVel  = -sign*kpw*perp;
            check   = 0;
        end
    else
        angVel  = sign*0.06;
        linVel  = abs(angVel)*R;
        distRef = sqrt((x - xyRef(1))^2 + (y - xyRef(2))^2);
        check   = 1;
    end
    dhead = 1;
else
    if dist <= 0.05
        if theta < 0
            wptheta = wptheta - 2*pi;
        end
        dhead = wptheta - theta;
        check = 0;
        checkP = 0;
        distRef = 0;
        distOT = 1;
        theta0 = 0;
    else
        dhead0 = course - theta;
        if dhead0 < -pi
            dhead = dhead0 + 2*pi;
        elseif dhead0 > pi
            dhead = dhead0 - 2*pi;
        else
            dhead = dhead0;
        end
    end
    linVel  = kpv*dist;
    angVel  = kpw*dhead;
end
```

```
data(1) = check;
data(2) = checkP;
data(3) = R;
data(4) = distOT;
data(5) = distRef;
data(6) = theta0;
end
```

Finally, once the desired linear and angular velocities have been calculated, they must be properly set and published to ROS through the /cmd_vel topic. The velocities coming out from the function are firstly saturated, to ±0.1m/s for the linear velocity and to ±0.4 for the angular velocity. They then pass through a series of switch blocks, that ensure that the waypoint has not yet been reached. If the waypoint has been reached, both in position and orientation, then the published linear and angular velocities will be equal to 0. These processes take place in the Publishing subsystem, shown in Figure 8.

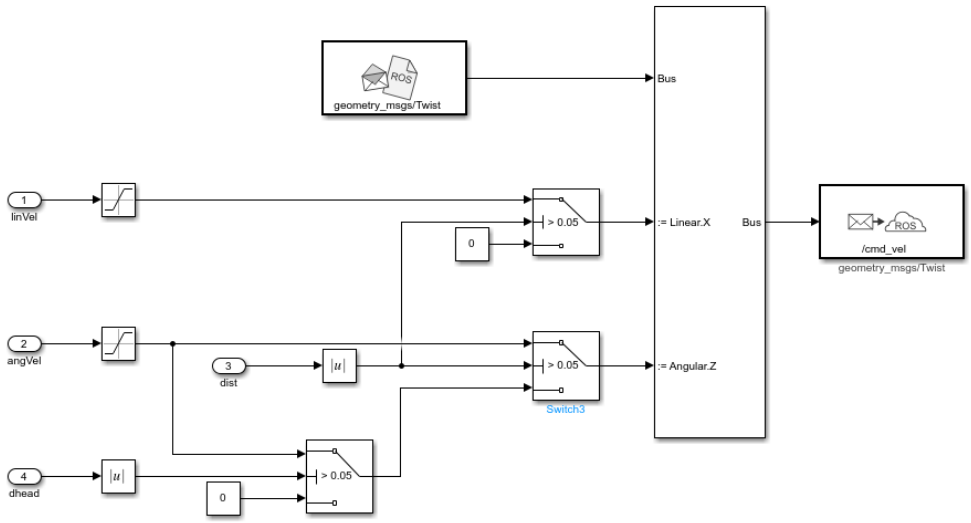


Figure 8. Publishing subsystem

Results

Figure 9 reports the trajectory of the turtlebot along the complete simulation. The black points highlight the position of the waypoints, demonstrating that the bot has indeed passed through each one of them. The position of the spheres, with their relative colour, is also reported in the figure.

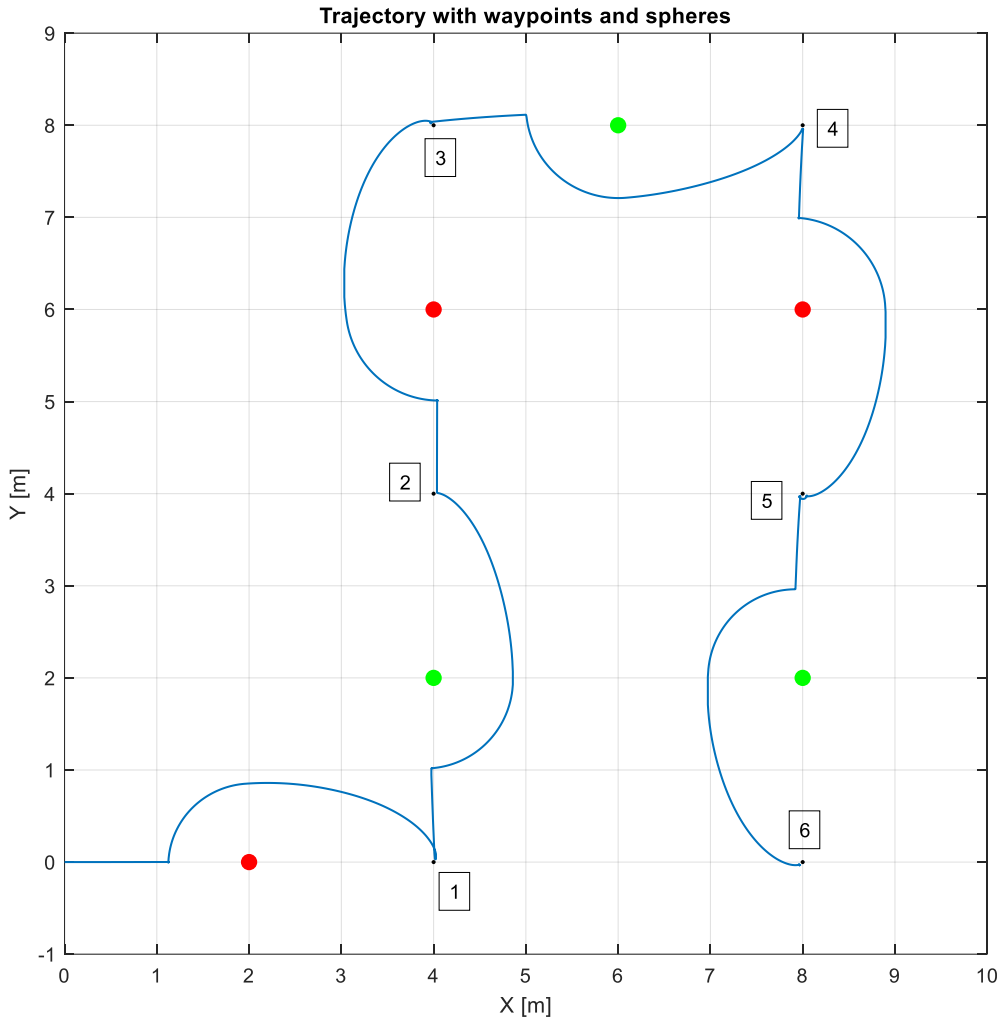


Figure 9. Trajectory of the robot, with waypoints and spheres.

Finally, the linear and angular velocity commands, published to the turtlebot through the /cmd_vel topic, are reported in Figure 10 plotted against time.

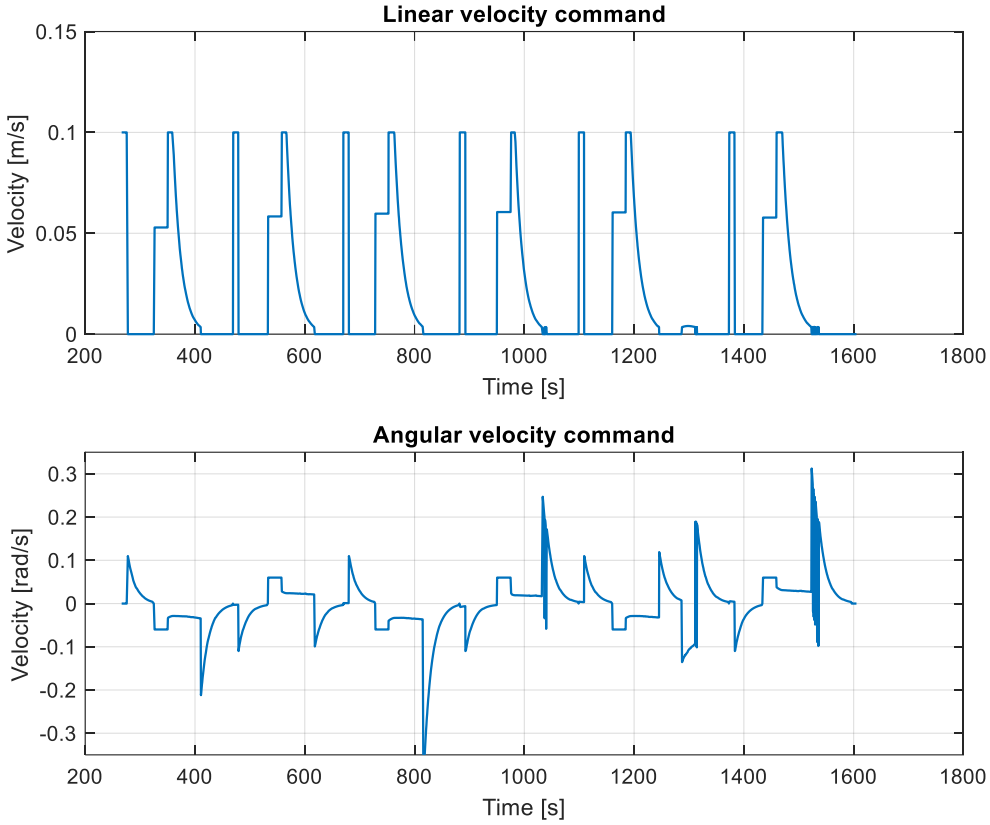


Figure 10. Linear and angular velocity commands

Conclusions

The presented control system is able to meet the requests, as shown in the results. However, it could still be improved. The control function requires many looping variables, that need memory blocks, which slow the simulation down: using state-flow to define the different moving stages would solve this problem, making the simulation more efficient.