Advanced Control 5 (ENG5009)

Lab Assignment

2127147b

March 21, 2019

Abstract

The following report outlines the development and testing of a waypoint following and obstacle avoidance system for the simulation of an autonomous robot in MATLAB. The system presented uses fuzzy logic controllers to generate desired turning commands and motor gains, a range of different input types were used alongside basic signal processing techniques to provide the fuzzy controllers with sufficient insight into the surrounding environment. The controller was found to produce successful results with the robot travelling to a specific coordinate with a 0.05m radius tolerance. Further development and fine-tuning was carried out to optimise the controller performance for a set of different scenarios. All code can be found on GitHub at [1], relevant code is included in the appendices.

1 Introduction

2 Methodology

2.1 Overview of System

Two cascaded fuzzy controllers are used, the first (path controller) determines a desired turn command based solely on the robot's current heading angle (ψ) and its angle relative to a desired waypoint (ψ_{ref}). The second controller takes the generated turn command and inputs relating to a nearby object (d_{wall}, $\bar{d}_{wall}$, Θ_{wall}, r) to determine appropriate gains for the left and right motors. A variable lowpass FIR filter is used for each motor gain output to smooth the voltages, its cutoff frequency is controlled by one of the fuzzy motor controller's outputs. A proportional drive voltage is applied to each motor that varies with the robot's distance from the waypoint, it remains constant until close to the waypoint. A block diagram fof the system can be seen in figure 1.

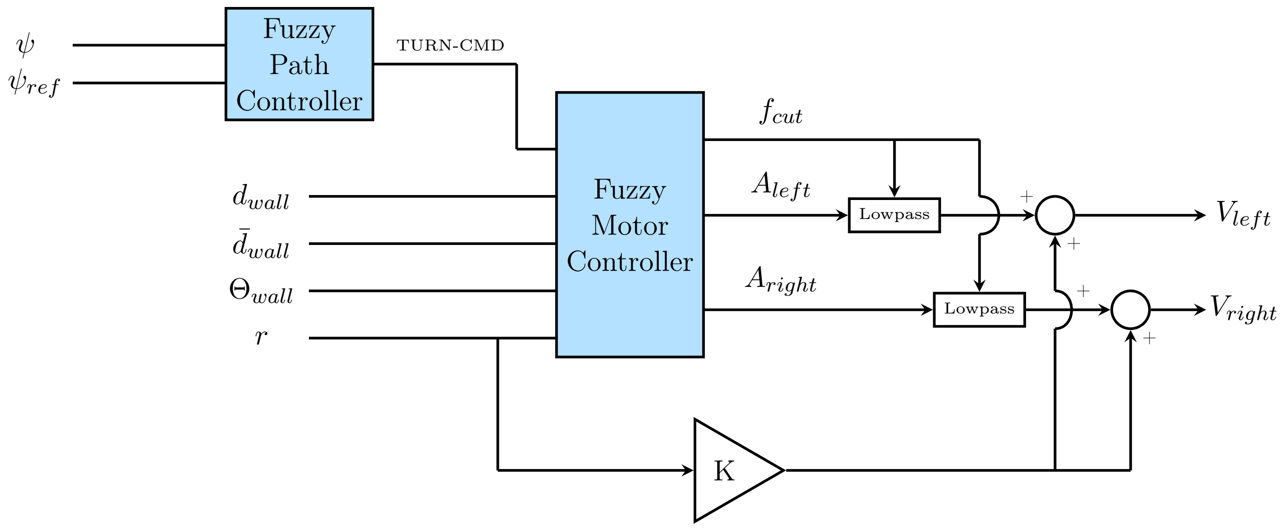


Figure 1: Block Diagram of Control System

2.2 Task 1: Waypoint Following

Overview

The aim of this task is to guide the robot to a set waypoint without obstacle avoidance. This was achieved using fuzzy logic to evaluate a desired turn command by comparing the robot's current heading angle (ψ) with its angle relative to the waypoint (ψ_{ref}). The fuzzy logic is implemented to drive the error between both angles to zero such that the robot travels in the direction of the waypoint.

A second fuzzy controller was created to control the gain applied to the motors, it takes the generated turn command and radius from the waypoint as inputs. As the robot approaches the waypoint, the rules are altered to enable coarser manoeuvres to allow it to stop within a 0.05m tolerance. A drive voltage is also proportionally reduced as the robot's position converges on the waypoint.

Fuzzy Sets

The input variables to the path controller are the heading and reference angles, they are measured from 0 rads (north), to either $-\pi$ or $+\pi$ rads (south) where a negative angle represents a counterclockwise angle and vice versa. Nine fuzzy input sets were derived as follows, {$S_{\text{-ve}}$, SW, W, NW, N, NE, E, SE, $S_{+\text{ve}}$}, where N is north, NE is north-east etc, these sets are identical for both the heading and reference angle inputs. $S_{\text{-ve}}$ and $S_{+\text{ve}}$ both represent a range around south as the angle jumps from -π to $+\pi$ and are therefore treated as the same set in the fuzzy rules. Trapezoidal membership functions were used for fuzzification, they can be seen for the heading angle input in figure 2, the sets for the reference angle input are identical.

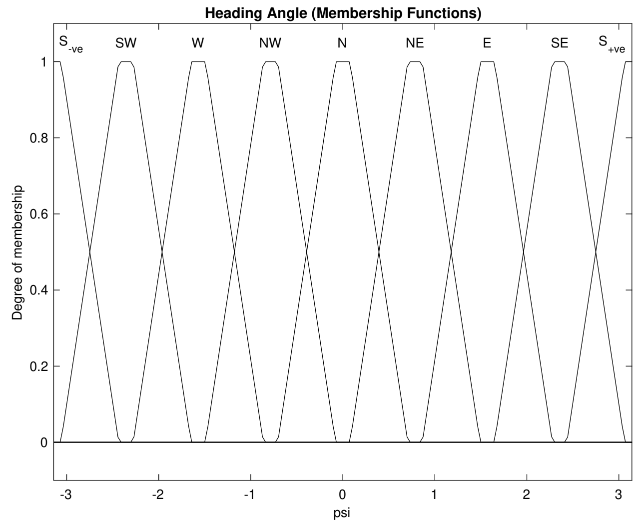


Figure 2: Membership Functions for Heading Angle Input

The path controller output variable is a turning command with the derived set, $\{L_{rev}, L_{rot}, L_{hard}, L_{soft}, FWD, R_{soft}, R_{hard}, R_{rot}, R_{rev}\}$, for reverse, rotate, hard, soft and forward manoeuvres respectively. These commands allow for a variety of coarse or fine turning adjustments to be made by the motor controller, trapezoidal membership were used for the fuzzification and can be seen in figure 3

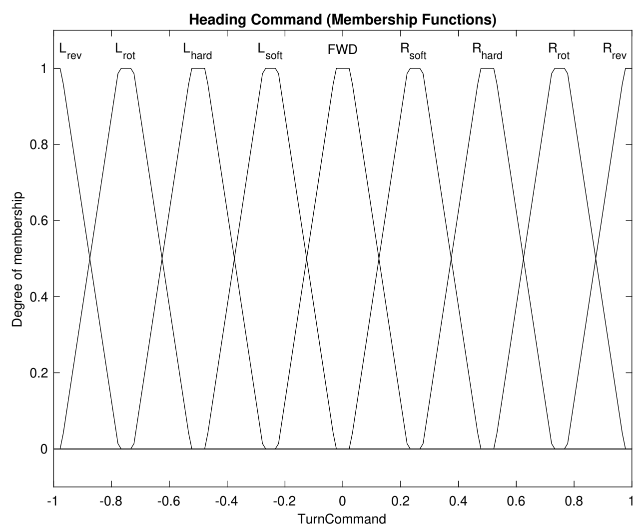


Figure 3: Membership Functions for Turn Command Output

The second input to the motor controller is the robot's radius to the waypoint with the following set, {VN, N, F, VF}, representing very-near, near, far and very-far respectively. This is used to execute tighter turning manoeuvres when close to the waypoint as when the robot is very-near to the waypoint, the turn command will begin to change much more rapidly. This therefore needs to be taken into account such that the robot can navigate to within 0.05m. The radius is also used in the main code to proportionally reduce the drive voltage on approach to the waypoint. Trapezoidal membership functions are used and can be seen in figure 4.

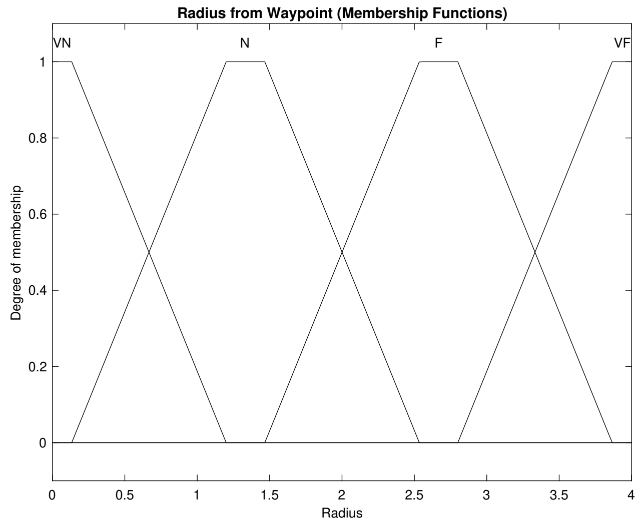


Figure 4: Membership Functions for Radius Input

The motor controller produces gains for the motors as outputs in the range of -1 to 1 with the derived set, $\{\mathrm{REV_{hard}}, \mathrm{REV_{soft}}, \mathrm{OFF}, \mathrm{FWD_{soft}}, \mathrm{FWD_{hard}}\}$, representing hard-reverse, soft-reverse, off, soft-forward and hard-forward manoeuvres repectively. These gains are scaled in the main code to an appropriate range and limited to the maximum range of ±7.4V. Triangular membership functions are used and can be seen in figure 5.

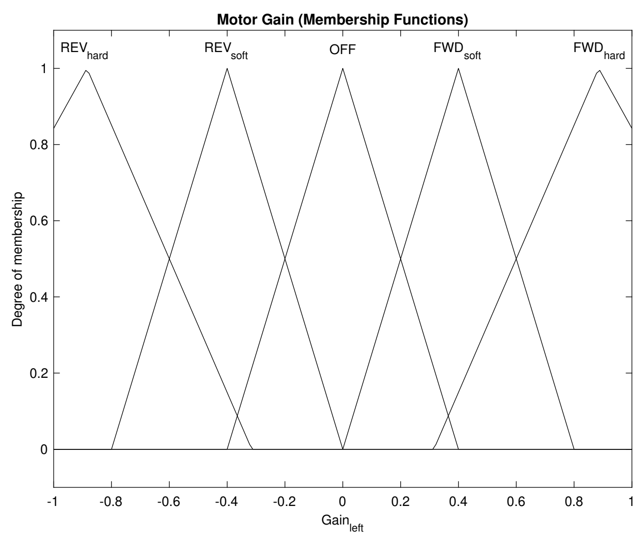


Figure 5: Membership Functions for Motor Gain Outputs

Rules

The path controller will produce a turning command that is appropriate for the robot's heading in relation to the its reference angle to the waypoint. For example: **IF** the robot is facing north (ψ=N) **AND** its bearing is north east (ψ_{ref}=NE), **THEN** turn soft right ($\mathrm{R_{soft}}$). In this situation only a soft right turn is required as angles in north and north-east are likely to be close together, a harder manoeuvre is more likely to result in an undesired overshoot. Rules were derived for each combination of headings and reference angles, a sample of the 81 rules can be seen in table 1 for heading of south-positive and south-west. The entire set of rules are included in appendix ???????????????????.

Table 1: Sample of Fuzzy Logic Rules for Path Controller (outputs in yellow)

ψ_{ref}	ψ	**TURN CMD**
$S_{\text{-ve}}$	$S_{\text{-ve}}$	FWD
$S_{\text{-ve}}$	SW	L_{soft}
$S_{\text{-ve}}$	W	L_{hard}
$S_{\text{-ve}}$	NW	L_{rot}
$S_{\text{-ve}}$	N	L_{rev}
$S_{\text{-ve}}$	NE	R_{rot}
$S_{\text{-ve}}$	E	R_{hard}
$S_{\text{-ve}}$	SE	R_{soft}
$S_{\text{-ve}}$	$S_{\text{+ve}}$	FWD
SW	$S_{\text{-ve}}$	R_{soft}
SW	SW	FWD
SW	W	L_{soft}
SW	NW	L_{hard}
SW	N	L_{rot}
SW	NE	L_{rev}
SW	E	R_{rot}
SW	SE	R_{hard}
SW	$S_{\text{+ve}}$	R_{soft}

The motor controller interprets these turn commands by applying appropriate gains to the left and right motors such that robot executes the requested turning manoeuvre. For example: **IF** the requested manoeuvre is a soft-right turn (TURN CMD = R_{soft}) **AND** the robot is not very-near to the waypoint (r!=VN), **THEN** A_{left} is FWD_{soft} **AND** A_{left} is OFF. If the robot is very-near to the waypoint then it will only execute rotational manoeuvres, for example: **IF** the requested manoeuvre is a soft-right turn **AND** the robot is very-near (r=VN), **THEN** A_{left} is FWD_{soft} **AND** A_{left} is REV_{soft}. These rules can be seen in table 2

Table 2: Truth table of motor controller rules (outputs in yellow)

TURN CMD	r	A_{left}	A_{right}
FWD	!VN	FWD_{soft}	FWD_{soft}
L_{soft}	!VN	OFF	FWD_{soft}
L_{hard}	!VN	REV_{soft}	FWD_{hard}
L_{rot}	!VN	REV_{hard}	FWD_{hard}
L_{rev}	!VN	REV_{hard}	REV_{soft}
R_{rev}	!VN	REV_{soft}	REV_{hard}
R_{rot}	!VN	FWD_{hard}	REV_{hard}
R_{hard}	!VN	FWD_{hard}	REV_{soft}
R_{soft}	!VN	FWD_{soft}	OFF
FWD	VN	FWD_{soft}	FWD_{soft}
L_{soft}	VN	REV_{soft}	FWD_{soft}
L_{hard}	VN	REV_{hard}	FWD_{hard}
L_{rot}	VN	REV_{hard}	FWD_{hard}
L_{rev}	VN	REV_{hard}	FWD_{hard}
R_{rev}	VN	FWD_{hard}	REV_{hard}
R_{rot}	VN	FWD_{hard}	REV_{hard}
R_{hard}	VN	FWD_{hard}	REV_{hard}
R_{soft}	VN	FWD_{soft}	REV_{soft}

Verification

2.3 Task 2: Obstacle Avoidance

Overview

Fuzzy Sets

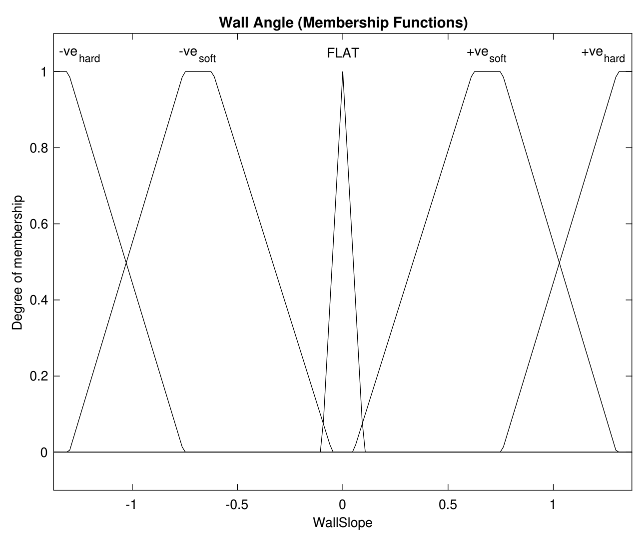


Figure 6: Membership Functions for Wall Angle Input

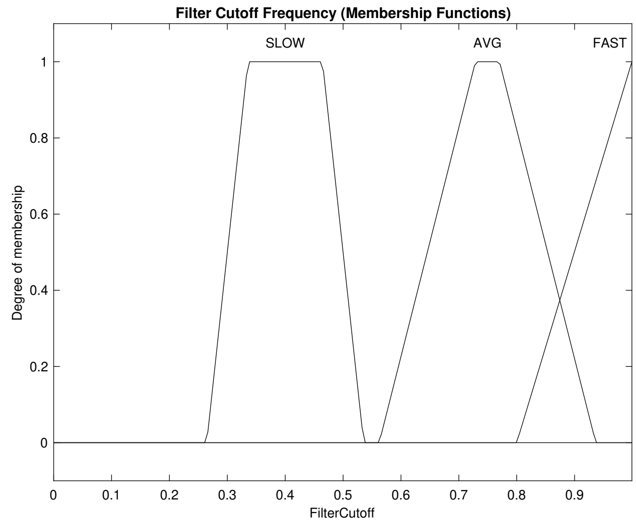


Figure 8: Membership Functions for Filter Cutoff Frequency Output

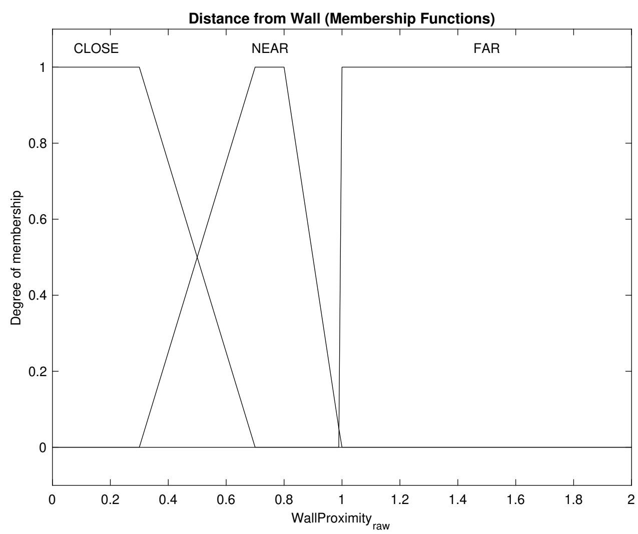


Figure 7: Membership Functions for Wall Proximity Input

Rules

Table 3: Truth table of motor controller rules when outwidth the proximity of a wall

TURN CMD	r	Θ_{wall}	d_{wall}	$\bar{d}_{wall}$	A_{left}	A_{right}	ω_{cut}
FWD	!VN	X	FAR	FAR	FWD_{soft}	FWD_{soft}	AVG
L_{soft}	!VN	X	FAR	FAR	OFF	FWD_{soft}	AVG
L_{hard}	!VN	X	FAR	FAR	REV_{soft}	FWD_{hard}	AVG
L_{rot}	!VN	X	FAR	FAR	REV_{hard}	FWD_{hard}	AVG
L_{rev}	!VN	X	FAR	FAR	REV_{hard}	REV_{soft}	AVG
R_{rev}	!VN	X	FAR	FAR	REV_{soft}	REV_{hard}	AVG
R_{rot}	!VN	X	FAR	FAR	FWD_{hard}	REV_{hard}	AVG
R_{hard}	!VN	X	FAR	FAR	FWD_{hard}	REV_{soft}	AVG
R_{soft}	!VN	X	FAR	FAR	FWD_{soft}	OFF	AVG
FWD	VN	X	FAR	FAR	FWD_{soft}	FWD_{soft}	AVG
L_{soft}	VN	X	FAR	FAR	REV_{soft}	FWD_{soft}	AVG
L_{hard}	VN	X	FAR	FAR	REV_{hard}	FWD_{hard}	AVG
L_{rot}	VN	X	FAR	FAR	REV_{hard}	FWD_{hard}	AVG
L_{rev}	VN	X	FAR	FAR	REV_{hard}	FWD_{hard}	AVG
R_{rev}	VN	X	FAR	FAR	FWD_{hard}	REV_{hard}	AVG
R_{rot}	VN	X	FAR	FAR	FWD_{hard}	REV_{hard}	AVG
R_{hard}	VN	X	FAR	FAR	FWD_{hard}	REV_{hard}	AVG
R_{soft}	VN	X	FAR	FAR	FWD_{soft}	REV_{soft}	AVG

Table 4: Truth table of motor controller rules when within proximity of a wall

TURN CMD	r	Θ_{wall}	d_{wall}	$\bar{d}_{wall}$	A_{left}	A_{right}	ω_{cut}
X	X	$+\text{ve}_{\text{hard}}$	NEAR	X	FWD_{hard}	FWD_{soft}	AVG
X	X	$+\text{ve}_{\text{soft}}$	NEAR	X	FWD_{hard}	REV_{soft}	AVG
X	X	$+\text{ve}_{\text{hard}}$	CLOSE	X	FWD_{hard}	FWD_{soft}	AVG
X	X	$+\text{ve}_{\text{soft}}$	CLOSE	X	FWD_{hard}	REV_{hard}	AVG
X	X	$-\text{ve}_{\text{hard}}$	NEAR	X	FWD_{soft}	FWD_{hard}	AVG
X	X	$-\text{ve}_{\text{soft}}$	NEAR	X	REV_{soft}	FWD_{hard}	AVG
X	X	$-\text{ve}_{\text{hard}}$	CLOSE	X	FWD_{soft}	FWD_{hard}	AVG
X	X	$-\text{ve}_{\text{soft}}$	CLOSE	X	REV_{hard}	FWD_{hard}	AVG
FWD	X	FLAT	!FAR	X	FWD_{hard}	REV_{soft}	FAST
L_{any}	X	FLAT	!FAR	X	REV_{soft}	FWD_{hard}	FAST
R_{any}	X	FLAT	!FAR	X	FWD_{hard}	REV_{soft}	FAST

Table 5: Truth table of motor controller rules when approximately parallel to wall

TURN CMD	r	Θ_{wall}	d_{wall}	$\bar{d}_{wall}$	A_{left}	A_{right}	ω_{cut}
R_{rev}	!VN	X	FAR	!FAR	FWD_{hard}	FWD_{soft}	AVG
R_{rot}	!VN	X	FAR	!FAR	FWD_{hard}	FWD_{soft}	AVG
L_{rev}	!VN	X	FAR	!FAR	FWD_{soft}	FWD_{hard}	AVG
L_{rot}	!VN	X	FAR	!FAR	FWD_{soft}	FWD_{hard}	AVG

3 Results and Testing

Assigned Waypoint

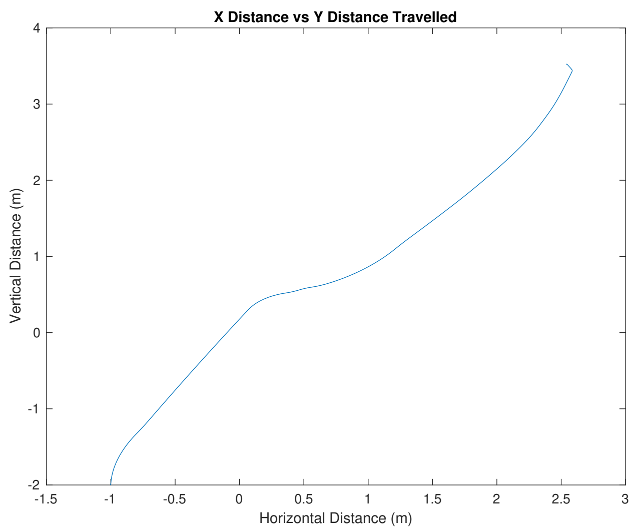
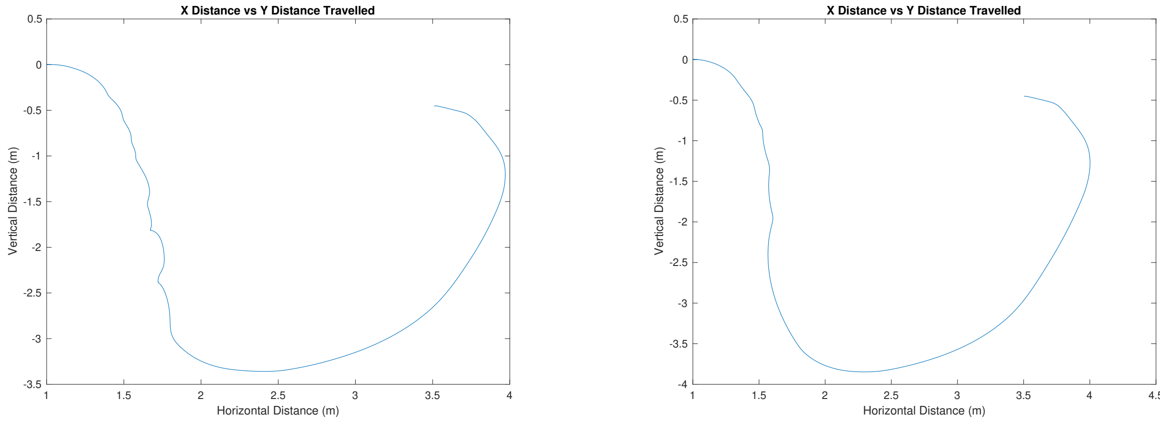


Figure 9: Robot path to waypoint (3.5, 2.5) from starting point (-2, -1)

Wall Tracking



(a) No filtered proximity input rules applied

(b) Robot path with filtered proximity input

Figure 10: Filtered proximity input rules applied

Perpendicular Approach

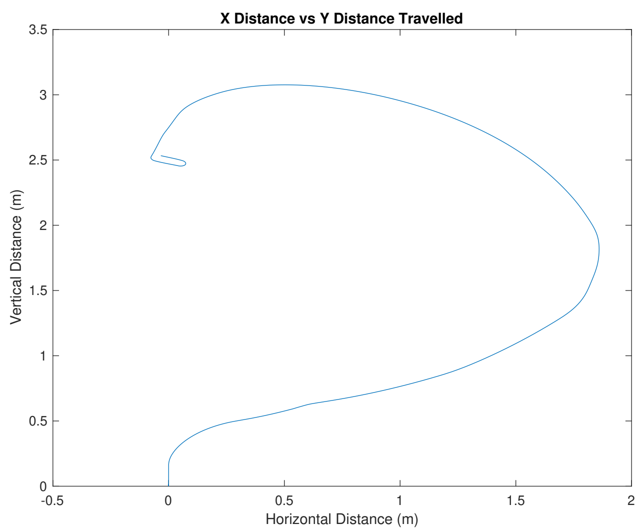


Figure 11: Path from (0, 2.5) to (0, 0) with perpendicular wall, (1.2, -1) to (1.2, 1)

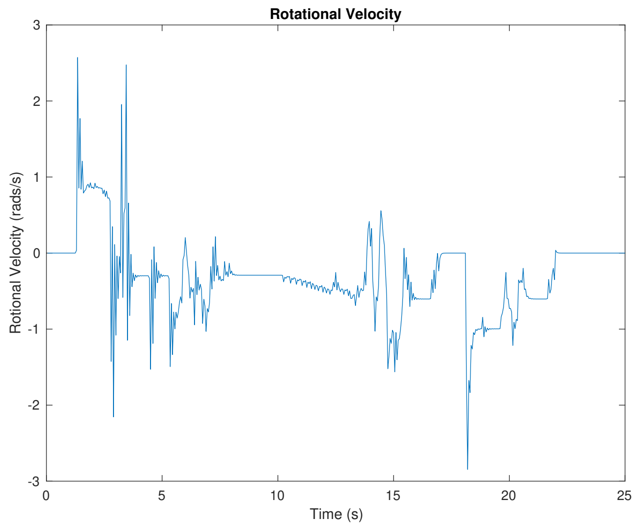


Figure 12: Rotational velocity when avoiding a perpendicular wall

4 Discussion

4.1 Evaluation

4.2 Further Work

References

[1] Jamie Brown. Git repository for advanced control 5 assignment. https://github.com/jamieb133/AdvancedControl5.

A Main Simulation Code

```
%
% Main simulation with control system
%
% Author: Jamie Brown
% File: run_model.m
%
% Created: 25/02/19
%
% Changes
%
%
%
%
%----------------------------------------------%
close all;
clear all;
clc;
%----------------------------------------------%

%----------------------------------------------%
%simulation config
sim_time = 25;
fs = 20; %sampling rate
fn = fs / 2; %nyquist
dT = 1 / fs;
xi = zeros(1,24); % intial state for x
xi(19) = -2; %starting x coordinate
xi(20) = -1; %starting y coordinate
LeftS = 0;
RightS = 0;
%----------------------------------------------%

%----------------------------------------------%
% Create Environment

max_x = 10;
max_y = 10;

Obs_Matrix = zeros(max_x/0.01,max_y/0.01);

wall = WallGeneration(-1, 1,1.2,1.2,'h');
wall2 = WallGeneration(-3, -3, -2, 2,'v');
wall3 = WallGeneration(2, 2, -3, 1,'v');
wall4 = WallGeneration(-3, -1, 4, 4,'h');

for x=1:length(wall)
    xpos = int16(wall(x,1)/0.01)+((max_x/2)/0.01);
    ypos = int16(wall(x,2)/0.01)+((max_y/2)/0.01);
    Obs_Matrix(ypos,xpos) = 1;
end

for x=1:length(wall2)
    xpos = int16(wall2(x,1)/0.01)+((max_x/2)/0.01);
    ypos = int16(wall2(x,2)/0.01)+((max_y/2)/0.01);
```

```
    Obs_Matrix(ypos,xpos) = 1;
end

for x=1:length(wall3)
    xpos = int16( (wall3(x,1)/0.01)+((max_x/2)/0.01) );
    ypos = int16( (wall3(x,2)/0.01)+((max_y/2)/0.01) );
    Obs_Matrix(ypos,xpos) = 1;
end

for x=1:length(wall4)
    xpos = int16( (wall4(x,1)/0.01)+((max_x/2)/0.01) );
    ypos = int16( (wall4(x,2)/0.01)+((max_y/2)/0.01) );
    Obs_Matrix(ypos,xpos) = 1;
end

%----------------------------------------------%

%----------------------------------------------%
%setup filters
n = 2;
fCut = fn/1.5; %filter cutoff
wn = fCut / (fs / 2) %normalise cutoff frequency to nyquist
filtType = 'low';
firCoeffs = fir1(n, wn, filtType);
leftFilter = FIRFilter(firCoeffs); %filter for right motor
rightFilter = FIRFilter(firCoeffs); %filter for left motor

sensorDelay = zeros(1, fs*2); %simple moving average buffer for wall proximity
%----------------------------------------------%

%----------------------------------------------%
ObjectAvoider = readfis('ObjectAvoider.fis');
HeadingController = readfis('HeadingsToTurnCmd.fis');
MotorController = readfis('TurnCommand.fis');

targetX = 3.5;
targetY = 2.5

%change these for different scenarios

xi(19) = 0
xi(20) = -0;
%xi(24) = pi/2;
targetX = 2.5;
targetY = -0;


targetWaypoint = [targetX, targetY];
simpleGain = 10/pi;
Vd = 2.5; %drive voltage
motorGain = 15;

time = zeros(1, sim_time/dT);
%----------------------------------------------%


%----------------------------------------------%
```

```
% MAIN SIMULATION LOOP

for outer_loop = 1:(sim_time/dT)

    %------------------------------------------%

    %obtain current reference and heading angles
    [atWaypoint, refAngle] = los_auto(xi(19), xi(20), targetWaypoint);
    headingAngle = xi(24);

    %calculate radius to target waypoint
    deltaX = xi(19) - targetX;
    deltaY = xi(20) - targetY;
    radius = sqrt(deltaX^2 + deltaY^2);

    if radius < 0.05
        %we are within tolerance of 5cm so stop
        Vl = 0;
        Vr = 0;
    else
        %obtain current distance to obstacle
        sensorOut = ObsSensor1(xi(19), xi(20), [0.2 0], xi(24), Obs_Matrix);

        %calculate wall angle and proximity
        wallAngle = atan( (sensorOut(:,2) - sensorOut(:,1)) / 0.2);
        if sensorOut(:,1) < sensorOut(:,2)
            wallProximity = sensorOut(:,1);
        else
            wallProximity = sensorOut(:,2);
        end;

        %this controller determines a desired turn command (headingCmd)
        %   based solely on reference and heading angle fuzzy input sets
        headingCmd = evalfis([refAngle, headingAngle], HeadingController);

        %take moving average value of wall proximity
        %   (allows the fuzzy motor controller to estimate whether
        %   or not it is parallel to a wall while the robot "snakes" alongside it)
        sensorDelay = circshift(sensorDelay, 1);
        sensorDelay(1) = wallProximity;
        wallProximityFiltered = mean(sensorDelay);
        %wallProximityFiltered = 1;

        %this controller takes a turn command from the heading controller
        % and determines the output motor voltages depending on whether or
        %   not a wall is detected or assumed to be parallel
        fuzzyOut = evalfis([headingCmd, radius, wallAngle, wallProximity,
    wallProximityFiltered], MotorController);

        %generate coefficients for new filter cutoff frequency
        newCoeffs = fir1(n, fuzzyOut(:,3), 'low');
        leftFilter.coeffs = newCoeffs;
        rightFilter.coeffs = newCoeffs;

        %apply lowpass filter to fuzzy motor gains to smoothen
        gainLeft = leftFilter.filter(fuzzyOut(:,1));
        gainRight = rightFilter.filter(fuzzyOut(:,2));
```

```

        %apply individual voltages calculated from fuzzy controller
        if radius > 1
            %apply an additional constant drive voltage when far from waypoint
            %   and not in viscinity of a wall
            Vl = Vd + (motorGain * gainLeft);
            Vr = Vd + (motorGain * gainRight);
        else
            if wallProximity < 1
                %while in viscinity of wall, reduce drive voltage proprtionally
                Vl = (Vd * wallProximity) + (motorGain * gainLeft);
                Vr = (Vd * wallProximity) + (motorGain * gainRight);
            else
                %when close to waypoint, reduce drive voltage proportionally
                Vd * radius;
                Vl = (Vd * radius ) + (motorGain * gainLeft);
                Vr = (Vd * radius ) + (motorGain * gainRight);
            end;
        end;

        %limit the outputs to max voltage range (+- 7.4V)
        if Vl > 14.8
            Vl = 14.8;
        elseif Vl < -14.8
            Vl = -14.8;
        end;

        if Vr > 14.8
            Vr = 14.8;
        elseif Vl < -14.8
            Vl = -14.8;
        end;

    end;

    %apply calculated output voltages to motors
    Va = [Vl/2; Vl/2; Vr/2; Vr/2];
    [xdot, xi] = full_mdl_motors(Va,xi,0,0,0,0,dT);

    %euler integration
    xi = xi + (xdot*dT);

    %store variables
    xdo(outer_loop,:) = xdot;
    xio(outer_loop,:) = xi;
    VlResults(outer_loop,:) = Vl;
    VrResults(outer_loop) = Vr;
    %---------------------------------------------%


    %---------------------------------------------%


    %---------------------------------------------%
    %draw robot on graph for each timestep
    figure(1);
    clf; hold on; grid on; axis([-5,5,-5,5]);
```

```
    drawrobot(0.2,xi(20),xi(19),xi(24),'b');
    xlabel('y, m'); ylabel('x, m');
    plot(wall(:,1),wall(:,2),'k-');
    plot(wall2(:,1),wall2(:,2),'k-');
    plot(wall3(:,1),wall3(:,2),'k-');
    plot(wall4(:,1),wall4(:,2),'k-');
    pause(0.001);

    time(outer_loop) = outer_loop*dT;
    %----------------------------------------------%

end
%----------------------------------------------%
disp(xi(19));
disp(xi(20));
%----------------------------------------------%
%PLOTS

figure(2);
plot(xio(:,19));
title('Y Distance Travelled');
xlabel('Timesteps');
ylabel('Distance (m)');

figure(3);
plot(xio(:,20));
title('X Distance Travelled');
xlabel('Timesteps');
ylabel('Distance (m)');

figure(4);
plot(xio(:,24));
title('PSI Angle');
xlabel('Angle (rads)');
ylabel('Time (s)');

figure(5);
plot(xio(:,20),xio(:,19));
title('X Distance vs Y Distance Travelled');
xlabel('Horizontal Distance (m)');
ylabel('Vertical Distance (m)');

figure(6);
plot(VlResults(:,1));
title('Right Motor Voltage');
xlabel('Time (s)');
ylabel('Voltage (V)');

figure(7);
plot(time, xio(:,18));
title('Rotational Velocity');
xlabel('Time (s)');
ylabel('Rotional Velocity (rads/s)');

%----------------------------------------------%
```

B FIRFilter Class

```
%
% Basic sample by sample FIR filter class
% File: FIRFilter.m
%
% Author: Jamie Brown
%
% Created: 25/02/19
%
% Changes
%
%
%
%
classdef (ConstructOnLoad = true) FIRFilter  < handle

    properties
        taps %number of filter coefficients
        coeffs %impulse response (array of coefficients)
        buffer %buffer containing previous samples
    end

    methods

        %constuctor
        function self = FIRFilter(coeffs)
            tapSize = size(coeffs)
            self.taps = tapSize(2)
            self.coeffs = coeffs
            self.buffer = ones(1, self.taps - 1)
        end

        %filters samples via convolution
        function outSample = filter(self, inSample)
            outSample = 0;

            %shift data along buffer by one sample
            self.buffer;
            self.taps;
            for count = self.taps:-1:2
                self.buffer(count) = self.buffer(count-1);
            end;

            %insert new sample
            self.buffer(1) = inSample;

            %convolve
            for count = 1 : (self.taps - 1)
                outSample = outSample + self.buffer(count) * self.coeffs(count);
            end
        end
    end
end
```

C Lab 1 Answer Sheet