CE801: Intelligent Systems and Robotics Assignment

Damian Machlanski
dm18389@essex.ac.uk
ID: 1802490

May 24, 2024

1 Introduction

The development of intelligent entities and autonomous control have been considered grand challenges for many years. Soon after robotics has come into being, it was natural to try to apply those concepts there given many potential benefits of having intelligent machines around. For instance, they could take care of heavy labour, mundane household chores, investigate contaminated environments or simply put humans out of hazardous jobs, just to name a few. Therefore, it is clear that robots need to behave intelligently in order to be able to operate effectively within complex environments and complete increasingly difficult tasks, which constitutes of most real-world problems.

There are many techniques that try to tackle the intelligent behaviour problem, but fuzzy logic [1] has been particularly successful lately, partly due to handling uncertainty quite well and its overall transparency. This is also true for robot control, where Fuzzy Logic Control (FLC) systems find their application [2]. For less complicated control tasks there are simpler methods, like PID controllers, though they are usually more time consuming to set up and more difficult to interpret.

The aim of this project is to design and implement two robot controllers for navigation problem by incorporating PID and fuzzy logic approaches. The PID controller is set to follow a wall by the right side of the machine. The FLC, on the other hand, is used to provide two behaviours, namely right edge following and obstacle avoidance. Those behaviours are then combined via either subsumption or context blending. This work describes all the design steps, implementation details, experimental setups and results, which are conducted both in simulation and on a physical robot.

This document is structured as follows. First of all, section 2 thoroughly explains techniques incorporated in this work, followed by their concrete application in section 3. Then, sections 4 and 5 present experimental setups and discuss their results respectively. Finally, section 6 concludes the report and suggests possible future work. In addition, appendix A and B contain all the code written for PID and fuzzy controllers respectively.

2 Background

This section describes techniques used to develop the control systems for the navigation problem as part of this work. These are: PID controller and fuzzy logic control system.

2.1 PID controller

PID controller is a close loop control system that keeps track of errors, that is, the difference between a desired value and the actual one, at each time step (see equation 1). The errors can then be taken into account when calculating new outputs, such as speed or steering, which in turn should decrease consecutive errors and reach stable equilibrium.

$$e_t = d_t - x_t \tag{1}$$

The controller consists of three main parameters: Proportional, Integral and Derivative. Each of them can be of different importance, depending on currently assigned value, which is reflected in equation 2. The P component is concerned about the current error only, the I factor takes into account all errors generated so far, whereas D part is interested in the difference between current and previous error. All three components are then summed together to form the output.

$$u_t = P \cdot e_t + I \cdot (e_0 + e_1 + ... + e_t) + D \cdot (e_t - e_{t-1}) \tag{2}$$

The exact values of P, I and D have to be found empirically through experiments and their possible values depend on the problem at hand. It is also worth noticing that by setting some parameters to zero, a simpler controller can be obtained. For instance, P (I=D=0), PI (D=0) or PD (I=0).

In general, PID controllers are relatively easy to design, but their parameters need to be tuned via empirical testing, making it a considerably time-consuming process depending on the nature of experiments. In addition, troubleshooting can be also challenging as it can be difficult to understand why the controller gave specific output given an input.

2.2 Fuzzy Logic

Fuzzy logic aims at handling information uncertainty through language and allows to have intermediate areas between absolute values. In other words, an item can be a member of a function to some degree, ranging from 0 to 1. Moreover, one item can be a member of more than just one function. This gray area between sharp boundaries enables smooth transitions from one area to the other. The use of linguistic rules, on the other hand, make underlying logic understandable to humans as they can be written in the form of IF-THEN rules. This fact greatly helps with the design process of fuzzy logic solutions and their overall transparency.

When it comes to controllers, they usually operate on precise inputs and outputs, such as sensor readings and desired speed respectively. Fuzzy logic control systems map such inputs and outputs

through imprecise linguistic rules. This type of reasoning aims to mimic the way humans handle uncertainty. A typical FLC architecture is presented by figure 1.

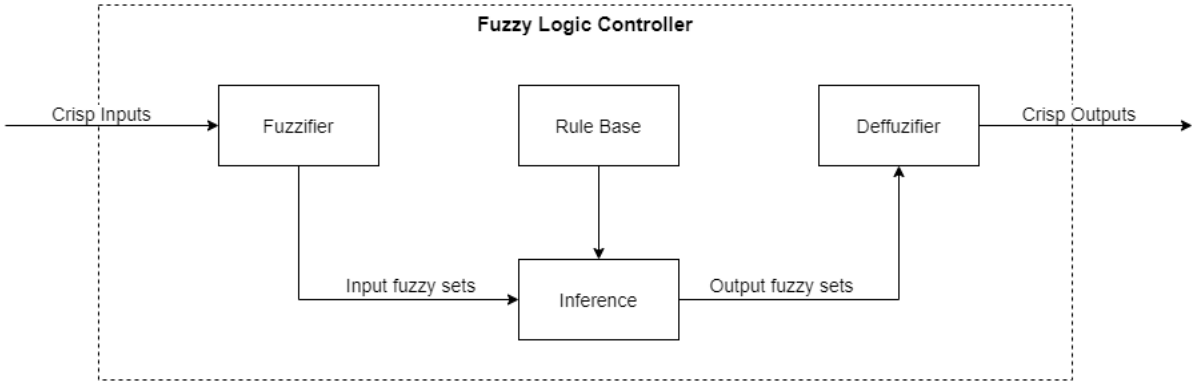


Figure 1: Fuzzy Logic Controller diagram.

First, crisp inputs are fuzzified through input fuzzy sets into linguistic values, like low, medium or high (see example in figure 2). One of the most common methods is a singleton fuzzifier, where the input value (x-axis) goes directly to the membership function it belongs to, giving a firing strength for that function. As an input can belong to more than one function, all combinations of inputs and membership functions are considered, together with their respective firing levels. Then, all linguistic input values are matched with rule base table, an example of which is presented in table 1, in order to obtain linguistic output values.

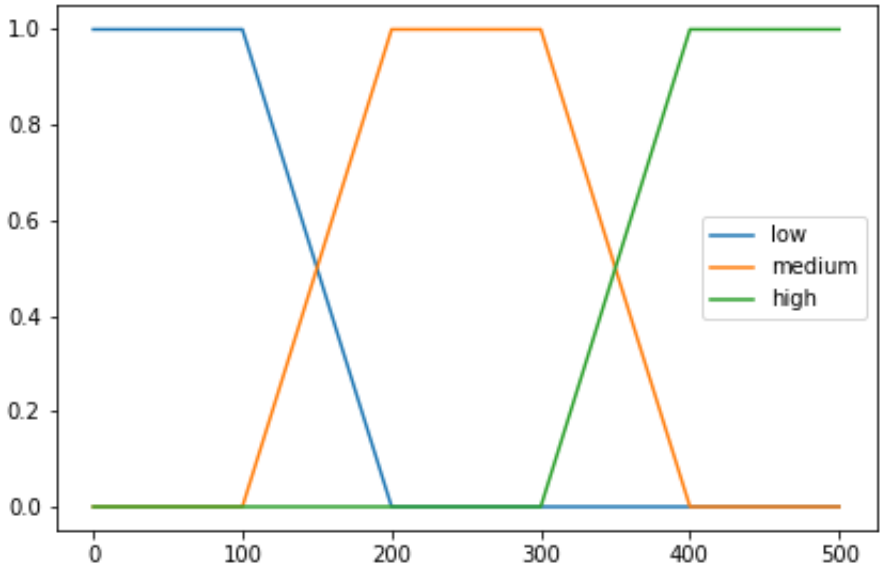


Figure 2: Example fuzzy set with three membership functions: low, medium and high.

Distance	**Steering**
Low	Left
Medium	Straight
High	Right

Table 1: Example rules of a simplified right edge following behaviour, where Distance represents a sensor input and Steering a desired output action.

Each fired rule has a strength, exact value of which depends on the operator used. One of the most popular is AND operator, which returns minimum firing level of considered inputs as its output. The next step, defuzzification, requires an output fuzzy set. There are different defuzzification approaches available, but one of the simplest is a combination of centroid defuzzifier and symmetrical triangular membership functions within the fuzzy set. This is because it is relatively straightforward to find the centre of a symmetrical triangle with respect to the x-axis.

Once centroid values and corresponding rule strengths are known, a concrete output value can be calculated using the formula 3, where s_n denotes the strength of an activated rule and c_n its matching centroid value.

$$output = \frac{s_1 * c_1 + s_2 * c_2 + ... + s_n * c_n}{s_1 + s_2 + ... + s_n} \tag{3}$$

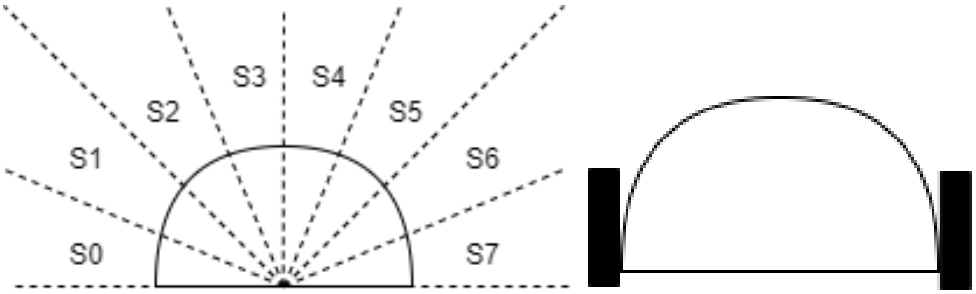
Fuzzy logic controllers generally handle information uncertainty very well, are considered highly transparent due to linguistic rules and overall do not require much fine-tuning given a thorough design process. However, the design part must be often handled manually, requiring in-depth domain expertise on the topic related to a task at hand. Furthermore, even though rules are easily readable by humans, the amount of them increases exponentially when dealing with complex behaviours, making them unfeasible to write by hand. This, however, can be mitigated by splitting such behaviours into smaller ones with separate rule sets and then combining their outputs. The most common combination techniques are subsumption and context blending. The former switches between behaviours based on defined conditions, while the latter assigns weights to each behaviour. The downside of subsumption architecture is the fact that it allows only one behaviour to be active at a time, resulting in sharp transitions between them. Context blending, on the other hand, can execute many actions at the same time with varied importance. This enables smooth transitions from one behaviour to another.

3 Methodology

This part of the document discusses overall methodology to robot navigation problem. First, it describes the model of the robot and utilised sensors. Then, PID and fuzzy logic controllers are presented, together with final parameter values and how they were obtained.

3.1 Robot navigation

The robot navigation modelled for this project is based on sonar sensors, readings of which are then put into controllers as inputs. Next, controllers return outputs that are taken into account when setting the speed of wheels. The model of the robot considered here is expected to have two wheels with the ability to set a desired speed for each of them separately. Figure 3 depicts the expected model, showing sonars' and wheels' locations.



(a) (b)

Figure 3: The model of the robot showing the location of (a) sonars and (b) wheels.

Such a model has its own consequences, like the fact that it is highly dependent on sonars quality and environment conditions. This is because controllers do not have the model of the environment hence the only information about the external world and robot's situation can be inferred from sensors. Unfortunately, sonars can be unreliable at times as they are prone to cross-talk issues, sensitive to temperature changes or just plain inaccurate due to low quality of the hardware. Those difficulties have to be handled well by controllers in order for the robot to navigate around effectively.

3.2 PID

The PID controller was designed to provide the robot with a right edge following behaviour only. To achieve that, sonars S6 and S7 were utilised. The desired distance from the wall was set to 45cm. Due to unreliable sonar readings, a history of past three values were stored for each sensor, which were then averaged to decrease the impact of big differences in readings, so the behaviour is more stable.

As the controller expects only one input variable (current wall distance), it was necessary to combine two sensor readings into one. But before that, it was crucial to adjust sensor 6 value, so it falls into the same range values as the other one. This was accomplished by multiplying sensor 6 readings by the value of cosine 40 degrees as sonar 6 is rotated by 40 degrees when compared to sonar 7. Then, both sonar readings were averaged together, given that both of them returned meaningful values, that is, less than 500cm; otherwise only one of them was taken into account.

PID controller is very sensitive to high numbers as this is immediately reflected in big error values and general robot unstability. To counteract this unwanted behaviour, current wall distance

obtained from sonars was further limited by certain minimum and maximum values, 25cm and 65cm respectively. To increase the stability even further and help with convergence, only ten last errors were kept for the purpose of calculating the "I" part of the PID formula.

After a lot of empirical testing and fine-tuning, the following PID values were obtained: P=0.5, I=0.01 and D=0.4. Using formula 2, together with PID constants and calculated error values, the final output was computed, which was in turn added to the base speed of the right wheel of the robot. The other wheel was set to a constant base speed.

3.3 Fuzzy Logic Control

Fuzzy logic control was utilised to provide the robot with two behaviours: right edge following and obstacle avoidance. Both of them were designed and implemented separately and then combined together via subsumption and context blending.

3.3.1 Right Edge Following

This behaviour consists of two inputs (sonar 6 and 7) and two outputs (left and right motor speed), an overview of which is presented by figure 4. The objective is to move alongside a wall at a specified distance and adjust the position accordingly if the distance is too small or too big.

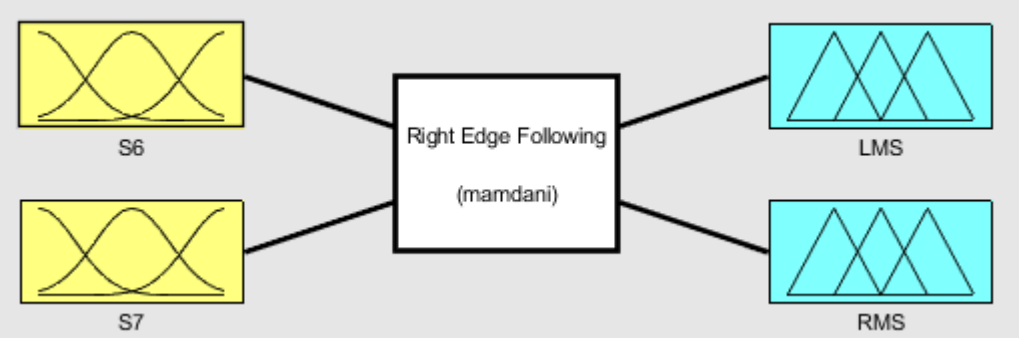
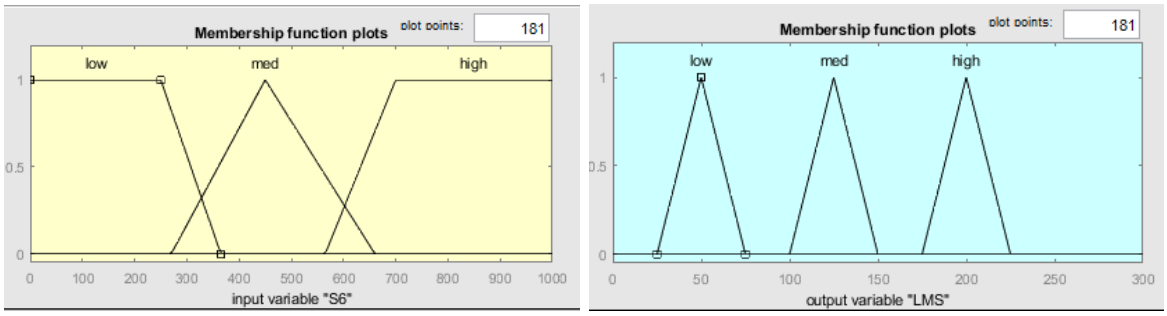


Figure 4: Overview of the right edge following behaviour design. S6 - sonar 6, S7 - sonar 7, LMS - left motor speed, RMS - right motor speed.

With regards to fuzzy sets, all inputs share the same input fuzzy set. Likewise, outputs use the same output fuzzy set. Both fuzzy sets are presented in figure 5. Both fuzzy sets consist of three membership functions: low, medium and high. In terms of input, there should be no surprise as to why low and high functions utilise trapezoid shapes as they need to cover a wide range of low and high values. A triangular shape was chosen for the medium function as the objective of the behaviour is to follow a wall at a specific distance. An alternative choice of a trapezoid shape would mean defining a distance as a range of values, which in turn would result in unwanted oscillations in behaviour. The functions within output fuzzy set were defined as symmetrical triangles to keep the calculations of defuzzification simple.



(a) (b)

Figure 5: Input (a) and output (b) fuzzy sets of the right edge following behaviour.

Given two inputs and three membership functions inside input fuzzy set, there are nine (3^2) linguistic rules that map inputs to outputs, all of which are presented in table 2.

S6	**S7**	**LMS**	**RMS**
L	L	L	H
L	M	L	H
L	H	L	H
M	L	M	H
M	M	M	M
M	H	M	M
H	L	M	H
H	M	M	M
H	H	H	L

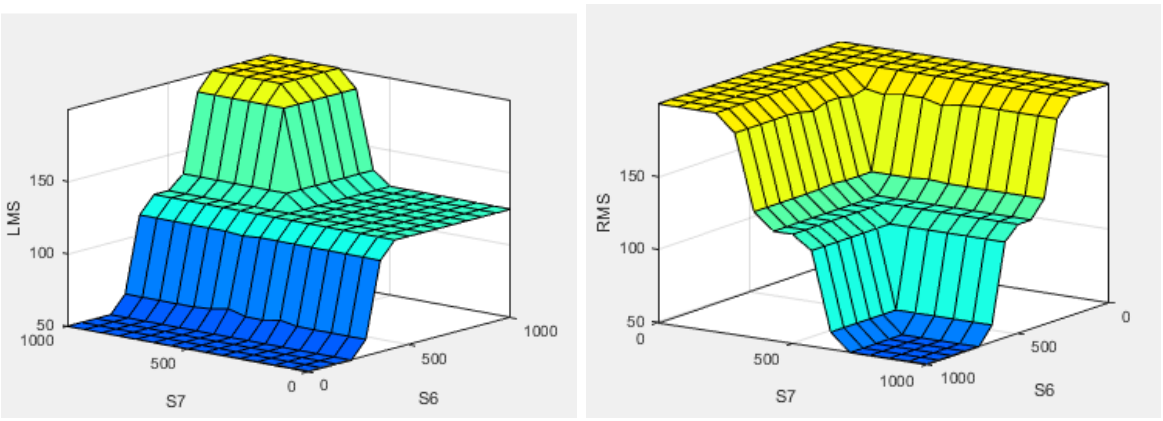
Table 2: Rule base table of the right edge following behaviour. S6 - sonar 6, S7 - sonar 7, LMS - left motor speed, RMS - right motor speed.

The last step of the design involved fine-tuning fuzzy sets, or more specifically, the values of membership functions in order to achieve satisfactory smoothness in terms of transitions between the functions. As demonstrated in figure 6, it is difficult to obtain high quality smoothness given only three membership functions, but the fine-tuning step was worthwhile doing nevertheless.

3.3.2 Obstacle Avoidance

The objective of this behaviour is to avoid objects in front of the robot. Although it uses four sensors, only three readings serve as actual inputs as the bigger value of either sensor 3 or 4 is discarded. Similarly to right edge following, this behaviour also has two outputs, left and right motor speeds. The overview of the design in presented in figure 7.

In terms of fuzzy sets of this behaviour, all of them comprise three membership functions: low,



(a) (b)

Figure 6: 3D plots showing relationships between both sensors and motor speed outputs. S6 - sonar 6, S7 - sonar 7, LMS - left motor speed, RMS - right motor speed.

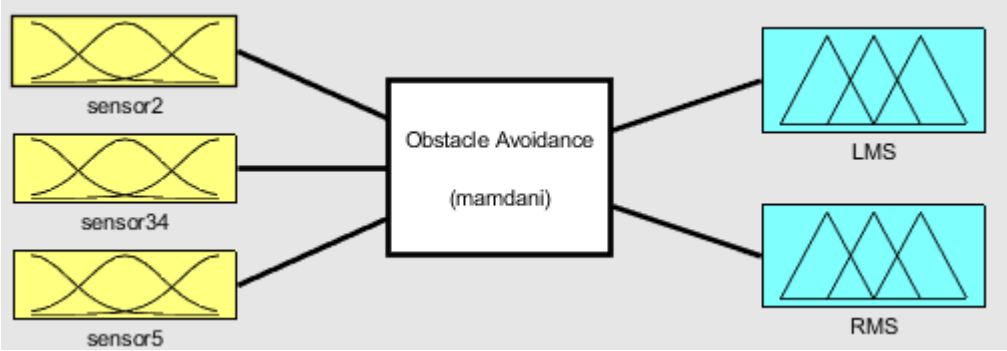


Figure 7: Overview of the obstacle avoidance behaviour design. LMS - left motor speed, RMS - right motor speed.

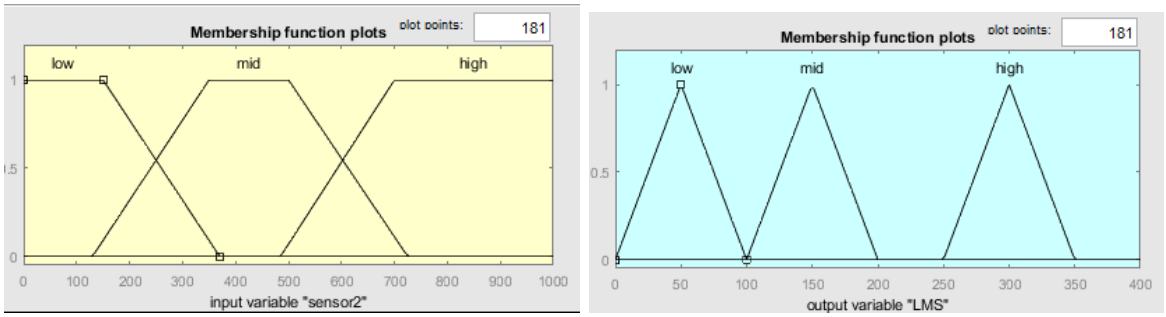
medium and high. Furthermore, all three inputs share the same fuzzy set. The same applies to outputs as there is only one output fuzzy set defined. Both fuzzy sets are illustrated in figure 8. The medium function in input fuzzy set was defined as a trapezoid shape in this case as it makes sense to have a range of values that account for medium distance from an obstacle. The high function in output fuzzy set was set to a slightly higher value compared to the other two in order to give the robot the ability to turn sharply in case an obstacle is very close.

Obstacle avoidance includes three inputs and three membership functions hence the total number of rules is 27 (3^3), all of which are presented in table 3.

The last step of the behaviour design was concerned about achieving better smoothness of transitions between membership functions. Similarly to right edge following fine-tuning, it was difficult to achieve quality smoothness due to only three membership functions. Figure 9 shows plots related to left motor speed, whereas figure 10 presents those connected to right motor speed.

S2	**S34**	**S5**	**LMS**	**RMS**
L	L	L	L	H
L	L	M	L	H
L	L	H	H	L
L	M	L	L	H
L	M	M	L	H
L	M	H	H	L
L	H	L	M	M
L	H	M	M	M
L	H	H	H	M
M	L	L	L	H
M	L	M	L	H
M	L	H	H	L
M	M	L	M	H
M	M	M	M	H
M	M	H	H	M
M	H	L	M	H
M	H	M	M	M
M	H	H	H	M
H	L	L	L	H
H	L	M	L	H
H	L	H	L	H
H	M	L	M	H
H	M	M	M	H
H	M	H	M	H
H	H	L	M	H
H	H	M	M	H
H	H	H	M	M

Table 3: Rule base table of the obstacle avoidance behaviour. S2 - sonar 2, S34 - minimum of sonar 3 and 4, S5 - sonar 5, LMS - left motor speed, RMS - right motor speed.



(a) (b)

Figure 8: Input and output fuzzy sets of the obstacle avoidance behaviour.

3.3.3 Combination

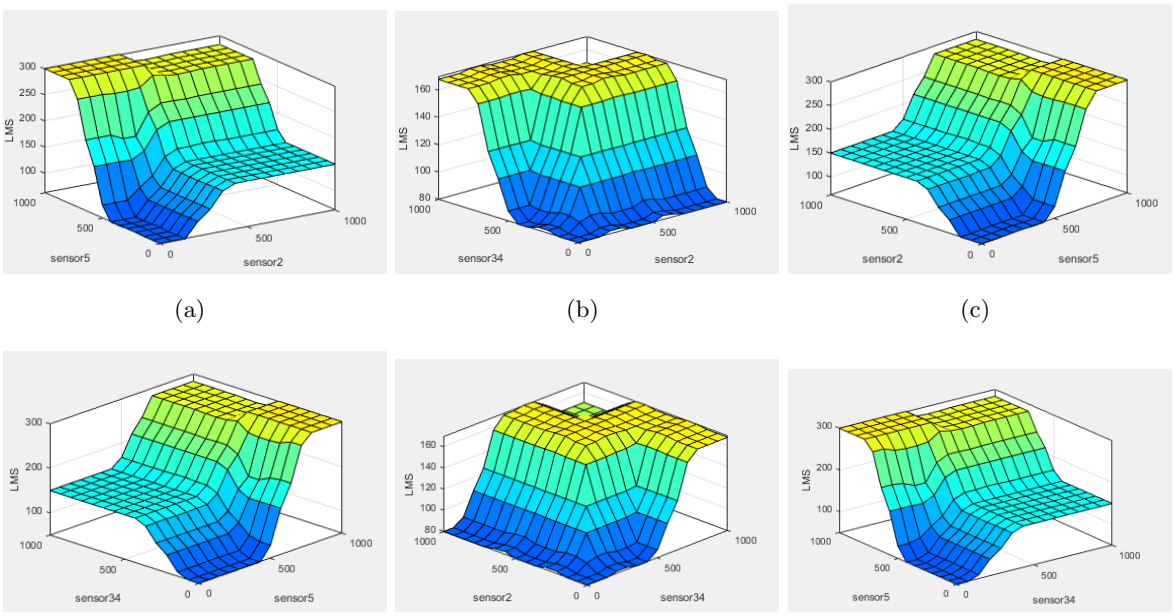
Once separate behaviours were completed, it was possible to combine them into more complex actions, so the robot can deal with more challenging environments. The subsumption architecture simply switches between behaviours based on defined condition. In this case, if the smallest reading from sensors 2, 3, 4 or 5 is lower than 800 then obstacle avoidance is activated. Otherwise, only right edge following behaviour is active.

The context blending approach is more sophisticated as it allows more than one behaviour to be active at a time, which in turn enables smooth transitions between behaviours. This is achieved by utilising yet another fuzzy set but this time each membership function represents each activity. As demonstrated in figure 11, depending on the minimum value of front sensors (sonars 2, 3, 4 and 5), both behaviours are active to some degree. In addition, obstacle avoidance function was set to zero for inputs lower than 190 to prevent the robot from hitting objects that are too close to be avoided.

4 Experiments

In order to evaluate the performance of implemented controllers, a series of experiments were performed using a Pioneer robot, presented on figure 12, within a prepared testing environment, showed in figure 13. The robot is equipped with eight sonars and two wheels on robot's sides, both of which can be manipulated individually.

Testing environments depend on controller being tested as PID was only designed to follow a wall, excluding obstacle avoidance. Thus, the setup designated for PID controller consisted of just a long wall, though not perfectly straight. As FLC solution was more sophisticated, it was possible to test it within more interesting setups. Both of them were of rectangular shape but the starting setting involved no obstacles hence the goal was to evaluate wall following behaviour exclusively, though in a bit more challenging setup than in PID case as this required robot to make 90 degrees turns. The second setup was similar to the first one but included many objects that forced the robot to change its course while following the wall. To make the task even more difficult, some areas of the environment



(a) (b) (c)

(d) (e) (f)

Figure 9: 3D plots showing relationships between given sensors and left motor speed outputs.

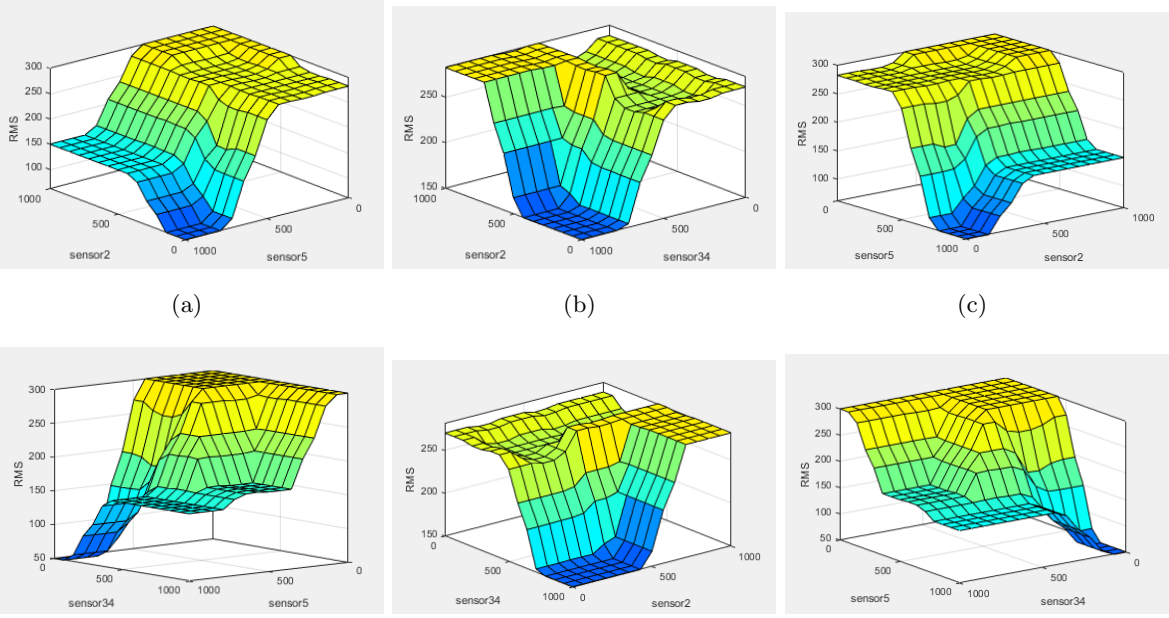
challenged the machine to go through narrow corridors to push its abilities to the limits.

4.1 PID

Running experiments as part of finding suitable parameters was a crucial part of developing a PID controller. Even apart from P, I and D parameters, other variables greatly impacted performance as well, such as base speed of wheels or the buffer size of historical sensor readings. What had the biggest impact on the stability of the system was perhaps the limitation of current wall distance by certain lower and upper values. Otherwise the controller was highly unstable or prone to reaching high wheel speeds if the robot was very far from the wall.

After finding a set of relatively stable parameters combined with careful preprocessing of sonar readings, the controller started to perform quite satisfactory. When it was placed too far from the wall, but within a reasonable distance, it was able to get closer to the wall eventually. Likewise, when putting it too close to the wall, the robot quite reliably increased the distance. Moreover, the robot was also able to make a "U turn" at the end of the wall most of the time.

One of the downsides of the controller was its relatively slow turns, mostly due to limits put on current wall distance. Furthermore, even though the robot was able to increase or decrease the distance from the wall as needed, the change in the course was usually slow, which in turn resulted in slow oscillations. Both issues are clearly the result of a trade-off between stability and responsiveness. As a consequence, the robot sometimes was not able to increase the distance from the wall on time.



(a) (b) (c)

(d) (e) (f)

Figure 10: 3D plots showing relationships between given sensors and right motor speed outputs.

4.2 FLC

The experiments evaluating fuzzy logic controller consisted of four subsets of tests, each of them checking different parts of the system. More precisely, first two experiments examined right edge following and obstacle avoidance behaviours in separation and then the other two tests aimed at observing the combination of those activities achieved through subsumption and context blending.

Right edge following behaviour did not have much problems with moving alongside a wall in general, regardless of the starting position. The corners of the rectangular wall also did not cause any major issues, although occasionally the robot turned too sharply, losing the wall from its sight and doing a full rotation as a consequence. However, what is more important is the fact that robot's turns were not perfectly smooth as it was relatively easy to notice the difference between sharp and slow turns. This in turn sometimes led to visible oscillations when robot was trying to locate itself in the desired distance from the wall. Lack of high quality smoothness is not a surprise as the behaviour includes only three membership functions for both inputs and outputs.

Obstacle avoidance activity was a bit more challenging to test in separation as the robot did not follow any designated path but rather moved forward until encountering objects in front of it. Overall, the general ability to avoid objects in front of the robot was fulfilled, though the turns were usually not very elegant. This is partly because in order to ensure the robot turns on time it was necessary to increase turn speed, which further led to bigger differences between membership function values and less graceful movements in general. A usual trade-off that needs to be taken into account unless having richer fuzzy sets is an option.

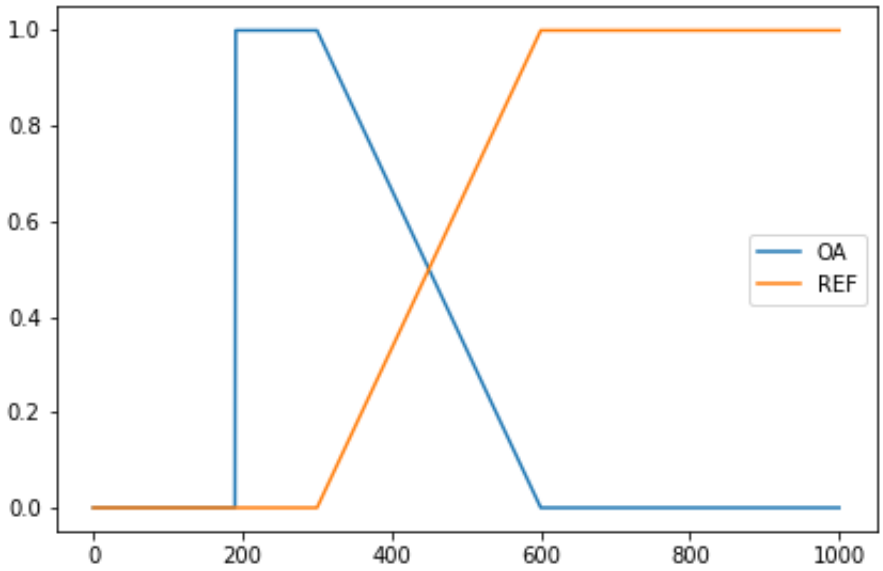


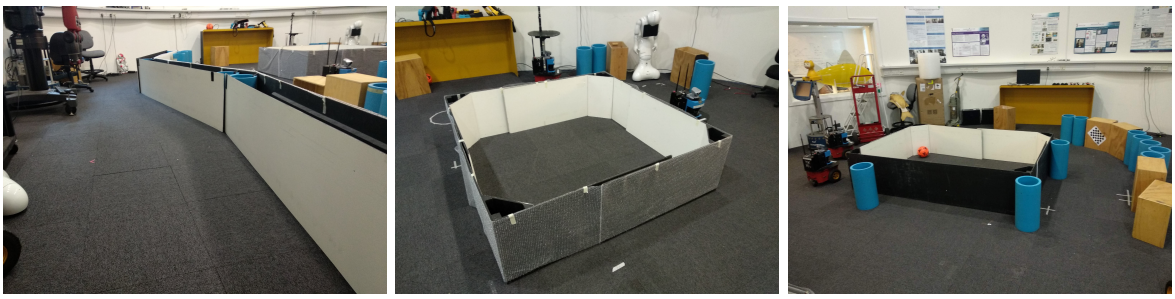
Figure 11: Fuzzy set of the context blending. OA - obstacle avoidance, REF - right edge following.

(a) (b) (c) (d) (e)

Figure 12: Robot used for real-world experiments.

Subsumption architecture finally glued together both behaviours into one system, which could follow a wall and avoid objects on its path if necessary. The obvious downside of this method is that only one activity can be active at a time, which was relatively easy to notice after a short observation. Although the robot was able to achieve what was expected, many turns seemed to be too sharp than they needed to be, possibly due to often activation of obstacle avoidance behaviour.

Context blending provided marked improvement when compared to previous composition as it was more challenging to discern which behaviour, and to what extent, moves the robot in some scenarios. The robot was even capable of going through narrow paths surrounded by different objects most of the time. Nevertheless, more careful examination of certain cases still exposed poor transitions between activities. Moreover, the robot sometimes performed shaky movements, possibly due to switching between behaviours back and forth.



(a) (b) (c)

Figure 13: Testing environments. (a) was used to evaluate PID controller, while (b) and (c) to test FLC.

5 Discussion

The PID controller performed reasonably well for the simple task of right edge following, proving its usefulness for not so complicated problems. However, achieving something similar to FLC via this controller would probably be significantly more challenging, if not unfeasible. Some of the clear drawbacks of this approach is its numerical sensitivity, so every minor change in parameters may greatly impact controller's performance. Thus, a careful preprocessing of inputs is vital to achieve reasonable stability. Another clear downside is time-consuming fine-tuning of the parameters, which, depending on the project and equipment, may be a deciding factor of whether to pursue this method. One possible improvement to this solution could be to use more reliable sensors.

The fuzzy logic controller showed its potential in autonomous robot navigation and overall superiority over the other controller. Very little fine-tuning, the ability to combine behaviours and transparency of its linguistic rules make it an attractive solution. However, there are still downsides to this method that need to be considered. First of all, the design process can become quite complicated, especially when dealing with many sensors that will inevitably generate big rule set tables. There is also a problem with smoothness and potentially growing number of membership functions. Perhaps the most promsing solution to this issue would be to incorporate Type-II fuzzy logic.

6 Conclusions and future work

In conclusion, the PID controller proved to be a good enough solution for simple control tasks as it is relatively easy to implement, though requires thorough parameter tuning. Fuzzy logic control provides excellent capabilities to be employed in more complex scenarios, particularly when combined with context blending. Although it is useful to fine-tune Type-I FLC systems, it might be necessary to upgrade to more sophisticated Type-II approach if smoothness is of great importance.

In terms of possible future work, some immediate improvements should be observed if higher quality sensors could be incorporated. It would be also interesting to see how the controller handles

even more challenging environments, or even somewhere outside the building in the dynamic world. This work only investigated two behaviours, but there are many more that could be interesting to try as well, such as goal seeking or area mapping. In order to tackle the smoothness issue, it would make sense to implement Type-II fuzzy logic and see how it improves for the same behaviours and similar experimental settings.

References

[1] L. A. Zadeh, "Fuzzy sets," *Information and control*, vol. 8, no. 3, pp. 338–353, 1965.

[2] A. Saffiotti, "The uses of fuzzy logic in autonomous robot navigation," *Soft Computing*, vol. 1, no. 4, pp. 180–197, 1997.

Appendix A

This appendix contains the code for PID controller.

```
#include "Aria.h"
#include <cstdio>
#include <iostream>
#include <fstream>
#include <deque>
#include <queue>

using namespace std;

// Gets the minimum of the numbers in the collection.
int getMinReading(deque<int> buffer)
{
        // Max sensor reading is 5000, so this is okay
        int result = 6000;

        deque<int>::iterator it = buffer.begin();
        while (it != buffer.end())
        {
                if (*it < result)
                {
                        result = *it;
                }
                *it++;
```

```
        }

        return result;
}

// Get the average of numbers in the collection.
double getAverageReading(deque<int> buffer)
{
        double sum = 0;
        double div = double(buffer.size());

        deque<int>::iterator it = buffer.begin();
        while (it != buffer.end())
        {
                sum += *it;
                *it++;
        }

        return sum / div;
}

int main(int argc, char **argv)
{
        // Initialisations
        Aria::init();
        ArRobot robot;
        ArSensorReading *sonarSensor[8];

        ArArgumentParser argParser(&argc, argv);
        argParser.loadDefaultArguments();

        // Connect to robot (and laser, etc)
        ArRobotConnector robotConnector(&argParser, &robot);
        if (robotConnector.connectRobot())
        {
                std::cout << "Robot connected!" << std::endl;
        }
```

```
robot.runAsync(false);
robot.lock();
robot.enableMotors();
robot.unlock();

// Safe start
robot.setVel2(0, 0);

// The value of cos(40) to adjust sensor readings
const double cos40 = 0.766;

// Desired distance from the wall
const double wall_distance = 450;

// The margin from desired wall distance that is taken into account when u
// This is to prevent numbers from being too small or too high, making calc
// robot behaviour more stable.
const double wall_distance_margin = 200;

// Upper allowed bound for sensor readings
const double max_current_distance = wall_distance + wall_distance_margin;

// Lower allowed bound for sensor readings
const double min_current_distance = wall_distance - wall_distance_margin;

// The base wheel speed
const double base_speed = 150;

// Max difference allowed between readings
const double max_diff = 500;

// P constant of the controller
const double kp = 0.5;

// I constant of the controller
const double ki = 0.01;

// D constant of the controller
```

```
const double kd = 0.4;

// Error values
double ep = 0, ep_prev = 0, ei = 0, ed = 0;

// The amount of past readings to keep
int sonar_buffer_size = 3;

// Collections of past sonar readings
std::deque<int> buffered_sonar6;
std::deque<int> buffered_sonar7;

// The amount of past Ei errors to keep
int ei_buffer_size = 10;

// Collection of past Ei errors
std::queue<double> buffered_ei;

// Diagnostics - dump sensor readings to a file
//ofstream file;
//file.open("data.csv", std::ios_base::app);

while (true)
{
        // Run (main logic)
        int sonarRange[8];
        for (int i = 0; i < 8; i++)
        {
                sonarSensor[i] = robot.getSonarReading(i);
                sonarRange[i] = sonarSensor[i]->getRange();
        }

        // Diagnostics
        //file << sonarRange[6] << "," << sonarRange[7] << endl;

        // Insert new readings into collections
        // Take into account sensors orientation
        buffered_sonar6.push_back(double(sonarRange[6]) * cos40);
```

```
buffered_sonar7.push_back(sonarRange[7]);

// Remove the oldest readings if exceeded desired buffer size
if(int(buffered_sonar6.size()) > sonar_buffer_size)
{
        buffered_sonar6.pop_front();
}
if(int(buffered_sonar7.size()) > sonar_buffer_size)
{
        buffered_sonar7.pop_front();
}

// Get average readings based on all readings stored
int sonar6 = getAverageReading(buffered_sonar6);
int sonar7 = getAverageReading(buffered_sonar7);

double current_distance;

if(sonar6 < 5000 && sonar7 < 5000)
{
        // Both sonars return sensible values
        if(abs(sonar6 - sonar7) > max_diff)
        {
                // But they are too different, so take the smalles
                current_distance = min(sonar6, sonar7);
        }
        else
        {
                // Acceptable difference - take the average of both
                current_distance = (sonar6 + sonar7) / 2.0;
        }
}
else
{
        // At least one reading returns max value (5k), so take the
        if(sonar6 < 5000)
        {
                current_distance = sonar6;
```

```
        }
        else
        {
                current_distance = sonar7;
        }
}

// Ensure current distance is within reasonable numbers to prevent
if(current_distance > max_current_distance)
{
        current_distance = max_current_distance;
}
else if(current_distance < min_current_distance)
{
        current_distance = min_current_distance;
}

// Calculate Ep error
ep = wall_distance - current_distance;

// Calculate Ei error
// ei stores current sum of buffered_ei collection
ei += ep;
buffered_ei.push(ep);
if(int(buffered_ei.size()) > ei_buffer_size)
{
        ei -= buffered_ei.front();
        buffered_ei.pop();
}

// Calculate Ed error
ed = ep - ep_prev;
ep_prev = ep;

// Calculate the output
double output = (kp * ep) + (ki * ei) + (kd * ed);

// Set wheel speeds
```

```
                // Left = constant speed
                // Right = base speed + output from calculations
                robot.setVel2(base_speed, base_speed + output);

                ArUtil::sleep(100);
        }

        // Termination
        robot.lock();
        robot.stop();
        robot.unlock();
        Aria::exit(0);
        return 0;
}
```

Appendix B

This appendix contains the code for Fuzzy Logic Controller.

```
#include "Aria.h"
#include <iostream>
#include <fstream>
#include <deque>
#include "ObstacleAvoidance.h"
#include "RightEdgeFollowing.h"
#include "Subsumption.h"
#include "BlendedControl.h"

using namespace std;

// Gets minimum value from a given collection
int getMinReading(deque<int> buffer)
{
        int result = 6000;

        deque<int>::iterator it = buffer.begin();
        while (it != buffer.end())
        {
                if (*it < result)
```

```
                {
                        result = *it;
                }
                *it++;
        }

        return result;
}

int main(int argc, char **argv)
{
        // Initialisations
        Aria::init();
        ArRobot robot;
        ArSensorReading *sonarSensor[8];

        ArArgumentParser argParser(&argc, argv);
        argParser.loadDefaultArguments();

        // Connect to robot (and laser, etc)
        ArRobotConnector robotConnector(&argParser, &robot);
        if (robotConnector.connectRobot())
        {
                std::cout << "Robot connected!" << std::endl;
        }

        robot.runAsync(false);
        robot.lock();
        robot.enableMotors();
        robot.unlock();

        // Safe start
        robot.setVel2(0, 0);

        // Number of readings to keep
        int sonar_buffer_size = 1;

        // Collections for sonar readings
```

```
deque<int> buffered_sonar2;
deque<int> buffered_sonar3;
deque<int> buffered_sonar4;
deque<int> buffered_sonar5;
deque<int> buffered_sonar6;
deque<int> buffered_sonar7;

// Cos(40) value for adjustment purposes
const double cos40 = 0.766;

// Mode:
// - 1: Right edge following
// - 2: Obstacle avoidance
// - 3: Subsumption
// - 4: Blended control
int mode = 4;
RightEdgeFollowing ref;
ObstacleAvoidance oa;
Subsumption subsumption(800);
BlendedControl blended;

// Diagnostics - dump readings to a file
//ofstream file;
//file.open("data.csv", std::ios_base::app);

while (true)
{
        // Run (main logic)
        int sonarRange[8];
        for (int i = 0; i < 8; i++)
        {
                sonarSensor[i] = robot.getSonarReading(i);
                sonarRange[i] = sonarSensor[i]->getRange();
        }

        // Update readings
        buffered_sonar2.push_back(sonarRange[2]);
        buffered_sonar3.push_back(sonarRange[3]);
```

```
buffered_sonar4.push_back(sonarRange[4]);
buffered_sonar5.push_back(sonarRange[5]);
buffered_sonar6.push_back(sonarRange[6]);
buffered_sonar7.push_back(sonarRange[7]);

// Remove the oldest reading if exceeded collection size
if (int(buffered_sonar2.size()) > sonar_buffer_size)
{
        buffered_sonar2.pop_front();
}
if (int(buffered_sonar3.size()) > sonar_buffer_size)
{
        buffered_sonar3.pop_front();
}
if (int(buffered_sonar4.size()) > sonar_buffer_size)
{
        buffered_sonar4.pop_front();
}
if (int(buffered_sonar5.size()) > sonar_buffer_size)
{
        buffered_sonar5.pop_front();
}
if (int(buffered_sonar6.size()) > sonar_buffer_size)
{
        buffered_sonar6.pop_front();
}
if (int(buffered_sonar7.size()) > sonar_buffer_size)
{
        buffered_sonar7.pop_front();
}

// Get minimum readings
double sonar2 = getMinReading(buffered_sonar2);
double sonar3 = getMinReading(buffered_sonar3);
double sonar4 = getMinReading(buffered_sonar4);
double sonar5 = getMinReading(buffered_sonar5);
double sonar6 = getMinReading(buffered_sonar6);
double sonar7 = getMinReading(buffered_sonar7);
```

```
        // Diagnostics
        //file << sonar2 << "," << sonar3 << "," << sonar4 << "," << sonar5

        // Take the minimum of two front sensors
        double sonar34 = min(sonar3, sonar4);

        // Takes into account sonars orientation
        sonar6 *= cos40;

        array<double, 2> output;

        switch (mode)
        {
        case 1:
                output = ref.GetOutput({ sonar6, sonar7 });
                break;;
        case 2:
                output = oa.GetOutput({ sonar2, sonar34, sonar5 });
                break;
        case 3:
                output = subsumption.GetOutput({ sonar2, sonar34, sonar5, s
                break;
        default:
                output = blended.GetOutput({ sonar2, sonar34, sonar5, sonar
                break;
        }

        // Set wheels speed
        robot.setVel2(output[0], output[1]);

        ArUtil::sleep(100);
}

// Termination
robot.lock();
robot.stop();
robot.unlock();
```

```
        Aria::exit(0);
        return 0;
}

#pragma once
#ifndef BEHAVIOUR_H
#define BEHAVIOUR_H

#include "FuzzySet.h"
#include <map>
#include <array>

class Behaviour
{
public:
        Behaviour();
        array<double, 2> GetOutput(vector<double> input);

protected:
        FuzzySet FSet;
        map<string, vector<string>> Rules;
        map<string, double> Centroids;
        virtual vector<string> GetRule(vector<FiringStrength> input) = 0;

private:
        vector<vector<FiringStrength>> _getCombinations(vector<vector<FiringStrengt
        void _permute(vector<vector<FiringStrength>> &input, vector<vector<FiringSt
        double _getMin(vector<FiringStrength> &input);
};

#endif // !BEHAVIOUR_H

#include "Behaviour.h"

Behaviour::Behaviour()
{
        // Does nothing
}
```

```
array<double, 2> Behaviour::GetOutput(vector<double> input)
{
        // Expected number of outputs
        const int outCount = 2;
        array<double, outCount> result;
        double numerators[outCount] = { 0 };
        double denominators[outCount] = { 0 };

        // Get firings for each input
        vector<vector<FiringStrength>> firings;
        for (double x : input)
        {
                firings.push_back(FSet.GetFiringStrength(x));
        }

        // Get all combinations of firings
        auto combinations = _getCombinations(firings);

        for (auto c : combinations)
        {
                // Get output rules according to the rule table
                auto rules = GetRule(c);

                // Apply MIN to the combination of firings
                double min_value = _getMin(c);

                // Min value * centroid value of the rule
                // Divided by the min value
                for (int i = 0; i < rules.size(); i++)
                {
                        numerators[i] += min_value * Centroids[rules[i]];
                        denominators[i] += min_value;
                }
        }

        // Calculate the final output
        for (int i = 0; i < outCount; i++)
        {
```

```
                result[i] = denominators[i] != 0 ? numerators[i] / denominators[i]
        }

        return result;
}

vector<vector<FiringStrength>> Behaviour::_getCombinations(vector<vector<FiringStre
{
        vector<FiringStrength> tmp;
        size_t n = input.size();
        vector<vector<FiringStrength>> result;
        _permute(input, result, n, 0, tmp);

        return result;
}

void Behaviour::_permute(vector<vector<FiringStrength>>& input, vector<vector<Firin
{
        if (row < width)
        {
                for (int c = 0; c < input[row].size(); c++)
                {
                        tmp.push_back(input[row][c]);
                        _permute(input, output, width, row + 1, tmp);
                        tmp.pop_back();
                }
        }
        else
        {
                output.push_back(tmp);
        }
}

double Behaviour::_getMin(vector<FiringStrength>& input)
{
        // Firings are expected to be within <0; 1> range
        double result = 2.0;
        for (auto fs : input)
```

```
        {
                if (fs.Strength < result)
                {
                        result = fs.Strength;
                }
        }

        return result;
}

#pragma once
#ifndef BLENDEDCONTROL_H
#define BLENDEDCONTROL_H

#include "ObstacleAvoidance.h"
#include "RightEdgeFollowing.h"

class BlendedControl
{
public:
        BlendedControl();
        array<double, 2> GetOutput(vector<double> input);

private:
        ObstacleAvoidance _oa;
        RightEdgeFollowing _ref;
        MembershipFunction _mf_REF, _mf_OA;
};

#endif // !BLENDEDCONTROL_H

#include "BlendedControl.h"

BlendedControl::BlendedControl()
{
        string names[] = { "OA", "REF" };

        _mf_OA = MembershipFunction(names[0]);
        _mf_OA.Points.push_back(Point(0, 0));
```

```
        _mf_OA.Points.push_back(Point(190, 0));
        _mf_OA.Points.push_back(Point(191, 1));
        _mf_OA.Points.push_back(Point(300, 1));
        _mf_OA.Points.push_back(Point(600, 0));
        _mf_OA.Points.push_back(Point(6000, 0));

        _mf_REF = MembershipFunction(names[1]);
        _mf_REF.Points.push_back(Point(0, 0));
        _mf_REF.Points.push_back(Point(300, 0));
        _mf_REF.Points.push_back(Point(600, 1));
        _mf_REF.Points.push_back(Point(6000, 1));
}

array<double, 2> BlendedControl::GetOutput(vector<double> input)
{
        // Take the minimum of front readings
        double oa_min = min(min(input[0], input[1]), input[2]);

        // Get firing strengths based on minimum front reading
        double f_OA = _mf_OA.GetValue(oa_min);
        double f_REF = _mf_REF.GetValue(oa_min);

        // Get outputs for OA and REF behaviours
        auto speed_OA = _oa.GetOutput({ input[0], input[1], input[2] });
        auto speed_REF = _ref.GetOutput({ input[3], input[4] });

        // Calculate wheels speed (safe divide to avoid zero division problem)
        double final_LSpeed = ((f_REF * speed_REF[0]) + (f_OA * speed_OA[0])) / (f_
        double final_RSpeed = ((f_REF * speed_REF[1]) + (f_OA * speed_OA[1])) / (f_

        array<double, 2> result = { final_LSpeed, final_RSpeed };
        return result;
}

#pragma once
#ifndef FIRINGSTRENGTH_H
#define FIRINGSTRENGTH_H

#include <string>
```

```
struct FiringStrength
{
public:
        std::string Name;
        double Strength;

        FiringStrength(std::string name, double strength);
        friend std::ostream& operator<<(std::ostream & str, const FiringStrength& v
};

#endif // !FIRINGSTRENGTH_H
```

```
#include "FiringStrength.h"

FiringStrength::FiringStrength(std::string name, double strength)
{
        Name = name;
        Strength = strength;
}

std::ostream& operator<<(std::ostream & str, const FiringStrength& v)
{
        str << "Name: " << v.Name << ", Strength: " << v.Strength;
        return str;
}
```

```
#ifndef FUZZYSET_H
#define FUZZYSET_H

#include "MembershipFunction.h"
#include "FiringStrength.h"

using namespace std;

struct FuzzySet
{
public:
        vector<MembershipFunction> Functions;
```

```
        FuzzySet();
        vector<FiringStrength> GetFiringStrength(double x);
        vector<string> GetFunctionNames();
};

#endif // !FUZZYSET_H
```

```
#include "FuzzySet.h"

FuzzySet::FuzzySet()
{
        // Does nothing
}

vector<FiringStrength> FuzzySet::GetFiringStrength(double x)
{
        vector<FiringStrength> results;
        for (int i = 0; i < Functions.size(); i++)
        {
                if (Functions[i].ContainsX(x))
                {
                        double strength = Functions[i].GetValue(x);
                        results.push_back(FiringStrength(Functions[i].Name, strengt
                }
        }

        return results;
}

vector<string> FuzzySet::GetFunctionNames()
{
        vector<string> result;

        for (int i = 0; i < Functions.size(); i++)
        {
                result.push_back(Functions[i].Name);
        }
```

```
        return result;
}

#pragma once
#ifndef MEMFUNC_H
#define MEMFUNC_H

#include "Point.h"
#include <vector>

using namespace std;

struct MembershipFunction
{
public:
        string Name;
        vector<Point> Points;

        MembershipFunction();
        MembershipFunction(string name);
        bool ContainsX(double x);
        double GetValue(double x);

private:
        Point _findStartingPoint(double x);
        Point _findEndingPoint(double x);
        double _getMinX();
        double _getMaxX();
};

#endif // !MEMFUNC_H

#include "MembershipFunction.h"
#include <iostream>

MembershipFunction::MembershipFunction()
{
}
```

```
MembershipFunction::MembershipFunction(string name)
{
        Name = name;
}

bool MembershipFunction::ContainsX(double x)
{
        return x >= _getMinX() && x <= _getMaxX();
}

double MembershipFunction::GetValue(double x)
{
        //Needs at least two points, otherwise return 0
        if (Points.size() < 2 || !ContainsX(x))
        {
                return 0;
        }

        // Assumes points are entered in ascending order
        Point startP = _findStartingPoint(x);
        Point endP = _findEndingPoint(x);

        double result = -1;

        //What kind of line is it?
        if (endP.Y > startP.Y)
        {
                //Arising
                result = (x - startP.X) / (endP.X - startP.X);
        }
        else if (endP.Y < startP.Y)
        {
                //Falling
                result = (endP.X - x) / (endP.X - startP.X);
        }
        else
        {
                //Flat
```

```
                result = startP.Y;
        }

        return result;
}

Point MembershipFunction::_findStartingPoint(double x)
{
        Point result = Point(-1, -1);

        if (Points.empty())
        {
                return result;
        }

        for (int i = 0; i < Points.size(); i++)
        {
                if (x >= Points[i].X)
                {
                        result = Points[i];
                }
                else
                {
                        break;
                }
        }

        return result;
}

Point MembershipFunction::_findEndingPoint(double x)
{
        Point result = Point(-1, -1);

        if (Points.empty())
        {
                return result;
        }
```

```
        for (int i = Points.size() - 1; i >= 0; i--)
        {
                if (x <= Points[i].X)
                {
                        result = Points[i];
                }
                else
                {
                        break;
                }
        }

        return result;
}

double MembershipFunction::_getMinX()
{
        if (Points.empty())
        {
                return -1;
        }

        double minX = Points[0].X;
        for (int i = 1; i < Points.size(); i++)
        {
                if (Points[i].X < minX)
                {
                        minX = Points[i].X;
                }
        }

        return minX;
}

double MembershipFunction::_getMaxX()
{
        if (Points.empty())
```

```
        {
                return -1;
        }

        double maxX = Points[0].X;
        for (int i = 1; i < Points.size(); i++)
        {
                if (Points[i].X > maxX)
                {
                        maxX = Points[i].X;
                }
        }

        return maxX;
}

#pragma once
#ifndef OA_H
#define OA_H

#include "Behaviour.h"

class ObstacleAvoidance : public Behaviour
{
public:
        ObstacleAvoidance();

protected:
        vector<string> GetRule(vector<FiringStrength> input) override;
};

#endif // !OA_H

#include "ObstacleAvoidance.h"

ObstacleAvoidance::ObstacleAvoidance()
{
        string names[] = { "L", "M", "H" };
```

```
//** LXX **
// LLX
Rules.insert(pair<string, vector<string>>("LLL", { "L", "H" }));
Rules.insert(pair<string, vector<string>>("LLM", { "L", "H" }));
Rules.insert(pair<string, vector<string>>("LLH", { "H", "L" }));
// LMX
Rules.insert(pair<string, vector<string>>("LML", { "L", "H" }));
Rules.insert(pair<string, vector<string>>("LMM", { "L", "H" }));
Rules.insert(pair<string, vector<string>>("LMH", { "H", "L" }));
// LHX
Rules.insert(pair<string, vector<string>>("LHL", { "M", "M" }));
Rules.insert(pair<string, vector<string>>("LHM", { "M", "M" }));
Rules.insert(pair<string, vector<string>>("LHH", { "H", "M" }));

//** MXX **
// MLX
Rules.insert(pair<string, vector<string>>("MLL", { "L", "H" }));
Rules.insert(pair<string, vector<string>>("MLM", { "L", "H" }));
Rules.insert(pair<string, vector<string>>("MLH", { "H", "L" }));
// MMX
Rules.insert(pair<string, vector<string>>("MML", { "M", "H" }));
Rules.insert(pair<string, vector<string>>("MMM", { "M", "H" }));
Rules.insert(pair<string, vector<string>>("MMH", { "H", "M" }));
// MHX
Rules.insert(pair<string, vector<string>>("MHL", { "M", "H" }));
Rules.insert(pair<string, vector<string>>("MHM", { "M", "M" }));
Rules.insert(pair<string, vector<string>>("MHH", { "H", "M" }));

//** HXX **
// HLX
Rules.insert(pair<string, vector<string>>("HLL", { "L", "H" }));
Rules.insert(pair<string, vector<string>>("HLM", { "L", "H" }));
Rules.insert(pair<string, vector<string>>("HLH", { "L", "H" }));
// HMX
Rules.insert(pair<string, vector<string>>("HML", { "M", "H" }));
Rules.insert(pair<string, vector<string>>("HMM", { "M", "H" }));
Rules.insert(pair<string, vector<string>>("HMH", { "M", "H" }));
// HHX
```

```
        Rules.insert(pair<string, vector<string>>("HHL", { "M", "H" }));
        Rules.insert(pair<string, vector<string>>("HHM", { "M", "H" }));
        Rules.insert(pair<string, vector<string>>("HHH", { "M", "M" }));

        MembershipFunction func1(names[0]);
        func1.Points.push_back(Point(0, 1));
        func1.Points.push_back(Point(150, 1));
        func1.Points.push_back(Point(370, 0));

        MembershipFunction func2(names[1]);
        func2.Points.push_back(Point(130, 0));
        func2.Points.push_back(Point(350, 1));
        func2.Points.push_back(Point(500, 1));
        func2.Points.push_back(Point(725, 0));

        MembershipFunction func3(names[2]);
        func3.Points.push_back(Point(485, 0));
        func3.Points.push_back(Point(700, 1));
        func3.Points.push_back(Point(5000, 1));

        Behaviour::FSet.Functions.push_back(func1);
        Behaviour::FSet.Functions.push_back(func2);
        Behaviour::FSet.Functions.push_back(func3);

        Behaviour::Centroids.insert(pair<string, double>(names[0], 50));
        Behaviour::Centroids.insert(pair<string, double>(names[1], 150));
        Behaviour::Centroids.insert(pair<string, double>(names[2], 350));
}

vector<string> ObstacleAvoidance::GetRule(vector<FiringStrength> input)
{
        string inputs = input[0].Name + input[1].Name + input[2].Name;

        return Rules[inputs];
}

#pragma once
#ifndef POINT_H
#define POINT_H
```

```
#include <iostream>

struct Point
{
public:
        double X;
        double Y;

        Point(double x, double y);
        friend std::ostream& operator<<(std::ostream& str, const Point& v);
};

#endif // !POINT_H
```

```
#include "Point.h"

Point::Point(double x, double y)
{
        X = x;
        Y = y;
}

std::ostream& operator<<(std::ostream& str, const Point& v)
{
        return str << "(" << v.X << ", " << v.Y << ")";
}
```

```
#pragma once
#ifndef REF_H
#define REF_H

#include "Behaviour.h"

class RightEdgeFollowing : public Behaviour
{
public:
        RightEdgeFollowing();
```

```
protected:
        vector<string> GetRule(vector<FiringStrength> input) override;
};

#endif // !REF_H

#include "RightEdgeFollowing.h"

RightEdgeFollowing::RightEdgeFollowing()
{
        string names[] = { "L", "M", "H" };

        Rules.insert(pair<string, vector<string>>("LL", {"L", "H"}));
        Rules.insert(pair<string, vector<string>>("LM", {"L", "H"}));
        Rules.insert(pair<string, vector<string>>("LH", {"L", "H"}));

        Rules.insert(pair<string, vector<string>>("ML", {"M", "H"}));
        Rules.insert(pair<string, vector<string>>("MM", {"M", "M"}));
        Rules.insert(pair<string, vector<string>>("MH", {"M", "M"}));

        Rules.insert(pair<string, vector<string>>("HL", {"M", "H"}));
        Rules.insert(pair<string, vector<string>>("HM", {"M", "M"}));
        Rules.insert(pair<string, vector<string>>("HH", {"H", "L"}));

        MembershipFunction func1(names[0]);
        func1.Points.push_back(Point(0, 1));
        func1.Points.push_back(Point(250, 1));
        func1.Points.push_back(Point(365, 0));

        MembershipFunction func2(names[1]);
        func2.Points.push_back(Point(270, 0));
        func2.Points.push_back(Point(450, 1));
        func2.Points.push_back(Point(660, 0));

        MembershipFunction func3(names[2]);
        func3.Points.push_back(Point(565, 0));
        func3.Points.push_back(Point(700, 1));
        func3.Points.push_back(Point(5000, 1));
```

```
        Behaviour::FSet.Functions.push_back(func1);
        Behaviour::FSet.Functions.push_back(func2);
        Behaviour::FSet.Functions.push_back(func3);

        Behaviour::Centroids.insert(pair<string, double>(names[0], 50));
        Behaviour::Centroids.insert(pair<string, double>(names[1], 150));
        Behaviour::Centroids.insert(pair<string, double>(names[2], 280));
}

vector<string> RightEdgeFollowing::GetRule(vector<FiringStrength> input)
{
        string inputs = input[0].Name + input[1].Name;

        return Rules[inputs];
}
```

```
#pragma once
#ifndef SUBSUMPTION_H
#define SUBSUMPTION_H

#include "RightEdgeFollowing.h"
#include "ObstacleAvoidance.h"

class Subsumption
{
public:
        Subsumption(int threshold);
        array<double, 2> GetOutput(vector<double> input);

private:
        RightEdgeFollowing _ref;
        ObstacleAvoidance _oa;
        int _behaviourThreshold;
};

#endif // !SUBSUMPTION_H
```

```
#include "Subsumption.h"
```

```
Subsumption::Subsumption(int threshold)
{
        _behaviourThreshold = threshold;
}

array<double, 2> Subsumption::GetOutput(vector<double> input)
{
        // input0 - sonar2
        // input1 - min(sonar3, sonar4)
        // input2 - sonar5
        // input3 - sonar6
        // input4 - sonar7

        array<double, 2> result;

        // If the minimum reading of front sensors is lower than defined threshold
        // then use OA; REF otherwise.
        if (min(min(input[0], input[1]), input[2]) < _behaviourThreshold)
        {
                result = _oa.GetOutput({ input[0], input[1], input[2] });
        }
        else
        {
                result = _ref.GetOutput({ input[3], input[4] });
        }

        return result;
}
```