University of Essex
School of Computer Science and Electronic Engineering

Assignment Report

Submitted as part of the requirements for:

CE801 Intelligent Systems and Robotics

Name: Ogulcan Ozer

Tutor: Prof. Hani Hagras

Date: 06 January 2019

Word Count:3039

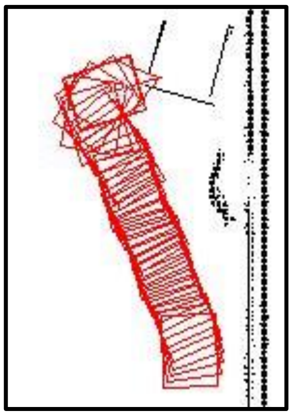
Table of Contents

1 Introduction

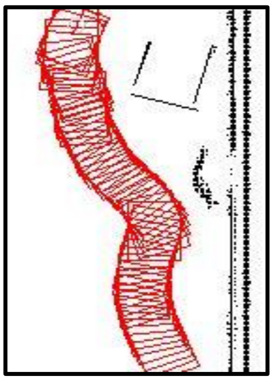
In the final week of the robotics laboratory implementations of the tasks; PID Controller for right edge following and Fuzzy Logic Controllers for right edge following and obstacle avoidance were demonstrated. This part of the report describes the implementations of the mentioned tasks. First part is the detailed explanation of the PID Controller. And the second part is about right edge and obstacle avoidance Fuzzy Logic Controllers together with their integration process.

1 PID Controller

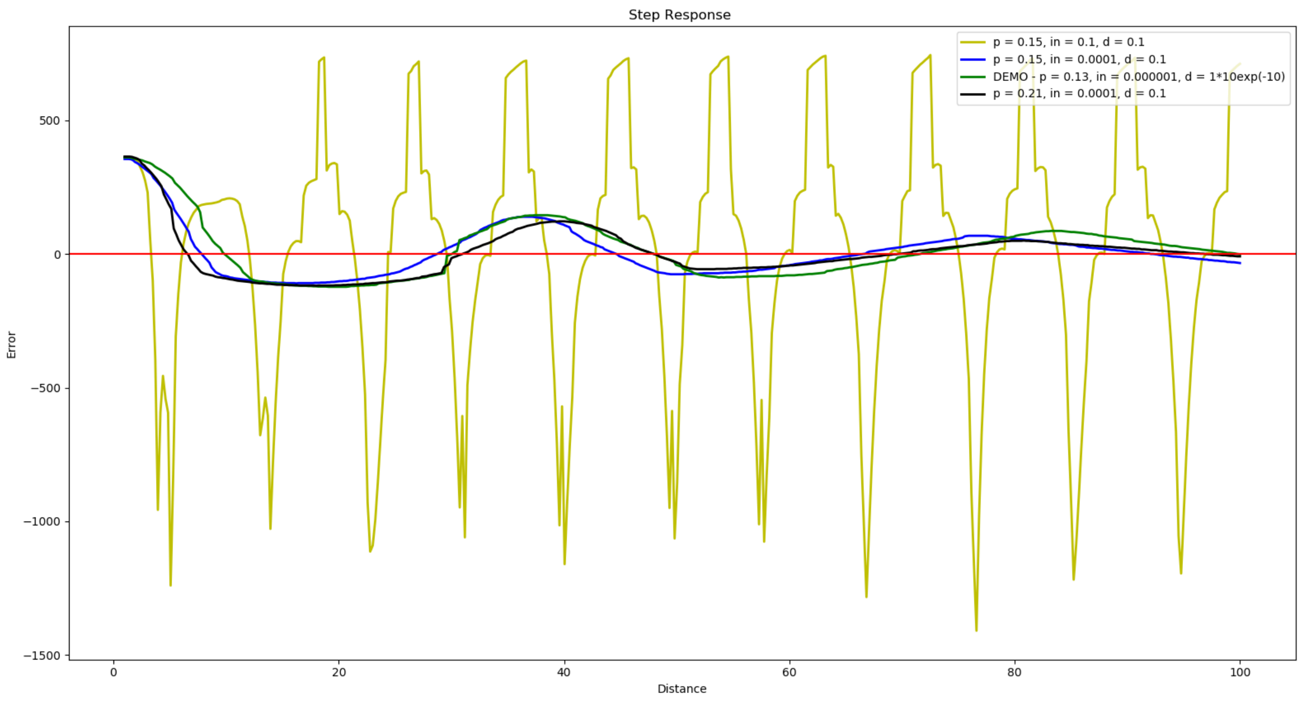
The theory of PID Controller was already covered in the lectures, but before starting the implementation, laboratory slides were used to learn the practical implementation of the controller and the initialization of the ARIA mobile robot. The first version of the implementation took very little time and it only used the rightmost sonar (7). First proportional error value chosen was 1, and the rest of the errors were set to 0. After some time, by trial and error, using the MobileSim the proper value for p was believed to be around [0.10,0.25]. Different integral error values were tested starting with 1, and values bigger than 0.001 created big oscillations. Thus, the Ki value was chosen as 0.0001, which resulted in smoother turns for the mobile robot. Lastly, the Kd was tested and the values around [0.1,0.2] seemed to improve the responsiveness of the robot. This first version using a single sensor was capable of following a straight right edge in a reasonable manner.


Robot failing because of a strong reaction, caused by a momentary big error.
(Kd = 0.2)

The second version of the implementation included the second sonar (6) from the right side, to allow the robot to follow any type of right edge, not just the straight and smooth ones. After the implementation, the new version was tested. Even though the straight edge following performance was similar, the robot performed poorly while following non-straight edges. Very strong and sudden reactions were observed. After trying different values for the parameters, the reason for this behavior was found to be the Kd value. After this, Kd was lowered to $1*10^{-10}$. At the end of these adjustments the controller performed reasonably, but since this quick implementation of the controller it became obvious that the real challenge was tuning the controller. At the beginning, the value chosen for Kp was 0.21. But after thoroughly testing the controller in MobileSim, it was realized that the robot still overreacted to some sharp turns, and it could also get stuck in narrow places. Thus, last version which was used in the demonstration was implemented.


Robot does not overreact to sudden errors.
(Kd =$1x10^{-10}$)

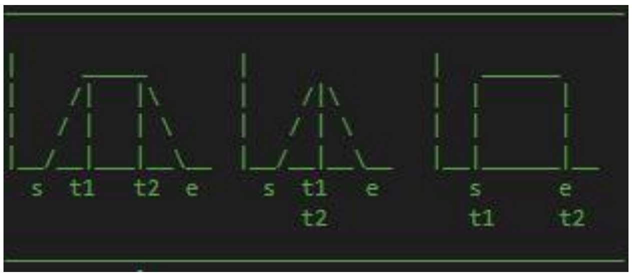
Another sonar (5) was added making it three in total, and Kp value was set as 0.13 providing smoother movement overall.



Step responses of experiments with various parameters including the final parameters used in the demonstration.

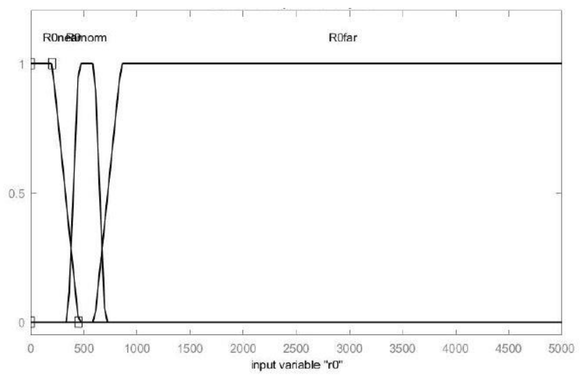
2 Fuzzy Logic Controllers and Context Based Blending

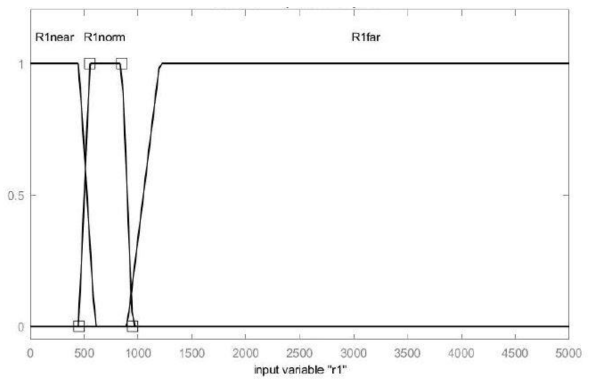
Before the implementation of the fuzzy logic controllers, a membership calculation function was created. It takes in a crisp value and the shape of the function, then return the membership value. At first these functions were defined separately, but soon realized that with the help of some if statements, every- linear- function could be stored in 4 integer values. Thus, more generalized membership function and a function struct was created. Function struct would hold the label and the start, top1, top2 and end values of a given function. The new membership function takes an array of functions- representing a fuzzy set- and an input struct as its inputs. This input struct holds a sensor reading and two arrays for the resulting labels and memberships. So, the membership function takes these two structs, calculates the membership values of the input from the fuzzy set array, then puts them into the empty array in the input struct.



To be able to calculate the output- by permutating the input labels array- a right edge rule base was created. For future convenience rule bases were implemented as string arrays, so that they can be read and used from external text files. For output calculations, the permutate function was implemented. This function takes a rule base and an array of inputs, then looks up the outputs in the rule base after permutating the input array. These acquired output rules then are placed inside an output structure, which has a label to hold the rule, value to hold the min composition of the inputs and an empty variable to hold the firing strength, which is calculated later. After getting the outputs, the structs are passed to the set fire function that calculates the firing strength of each output rule from their respective fuzzy set. These strength values then are placed into the empty variables in the output structs. Then the output array is passed to the get speeds function, which calculates the right and left wheel speed from the values and strengths in the output structs.

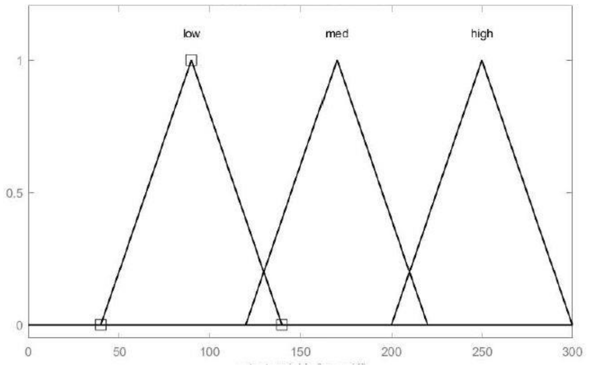
When the implementation of the right edge controller was finished, better values for the functions of the input fuzzy sets were chosen using MobileSim. Sonar values were recorded after placing the robot in desired positions. On the other hand, the output values were decided from previous experience with the robot. In the end controller worked correctly, but before tuning the functions and the rule base, obstacle avoidance controller was implemented. There were two additions to the code for the obstacle avoidance. First, a new rule base was created. Second, another function was implemented to permutate three input controller. For the rule base, a python script was written, that asks for user input for each permutation and outputs a rule base text file. And for the function, simply another outer loop was added to previous one to permutate three inputs. Again, the robot was tested only using the obstacle avoidance and the controller seemed to work properly. Before starting the implementation of context-based blending, left edge following was added. It did not require much work since it used the right edge rule base and fuzzy set. Differences being the sonar readings and the output speeds sent to wheels in reverse. Then left edge controller was tested and it also worked correctly.



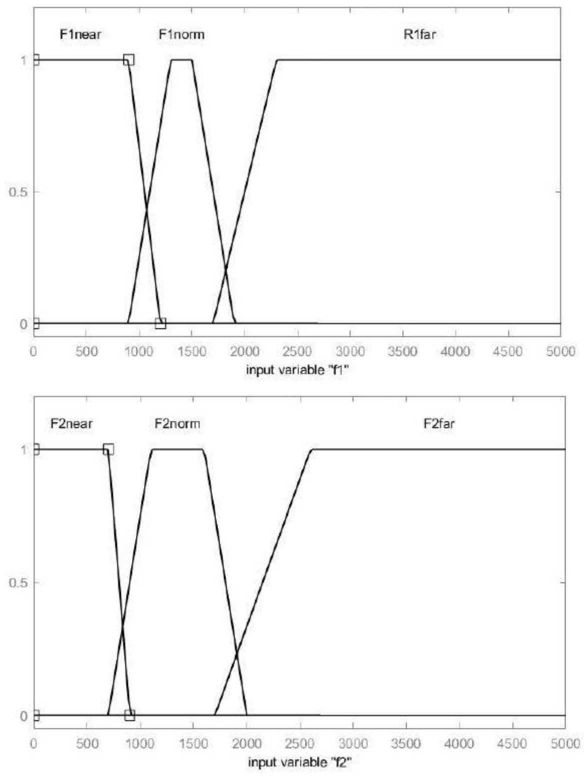


Functions r0 and r1 for right edge following, for sonars 7 and 6.

For the context-based blending three more functions left, right and front were written. Each of them started their activation at their respective FAR values and reached full membership at their respective NEAR values. Also, a two-line code was added to calculate the post context-based blending wheel speeds. Whole fuzzy system seemed to perform properly. Some rules were changed after a lot of testing and some of the function values were also tuned. Before the demonstration, the activation values of left edge following were lowered so that the robot prefers right edges. It was not removed so that the robot



Function for the outputs (wheel1 = wheel2)



Functions f1 and f2 (f0 = f2) for obstacle avoidance, for sonars 5, min (4,3) and 2.

could still react to very close obstacles from left side. But the robot did not perform as expected in the demonstration. It somehow did its job, but not in the way that it should have done, which was very different in MobileSim tests. I could not find the reason at that time, but after the demonstration, each line of code was reviewed and there was in fact a big mistake. In the line where the inputs for the context-based blending are created, left and right edge sonar readings were misplaced. The reason robot could still perform was because it still had both left and right edge following active. But as mentioned above, context-based blending activation values for left edge were lowered. This resulted in robot sometimes going straight instead of following the right edge after sharp turns, robot preferring left edges and not having enough distance to react to some obstacles coming from the right side of the robot. This mistake is now fixed in the latest version of the code. There are also some changes in the rule bases and the function values, but the inner workings of the program remain the same.

2 Literature Review

1 PID Controller

After it`s discovery in 1910, PID controllers became very successful. Because of its simple three term concept and efficiency, it was a good solution to many real-world problems. But in real world solutions, it was realized that the tuning of the parameters could be challenging. [1] It is mentioned in [1] that because of the miscommunication between the academia and the industry, Kd is believed to increase stability, but in real life applications this does not seem to be the case. It is claimed that in many applications, Kd is not even used. Later, it is explained that the term Kd increases the response of the controller to disturbances, but in real life solutions this might also cause problems. Then [1] compares different types of tuning methods and concludes that having different types of hardware and software controllers in the industry makes it harder for practitioners to achieve desired results. Which also results in not having a PID controller standard for control engineering.

There have been many different improvements made on PID controllers. One of them is [2], a fuzzy PID controller. It uses a conventional PID controller with constant coefficients, and generates an output using a fuzzy controller. Output is obtained by fuzzifying and fuzzifying PI and D parts separately and then combining them. Even though the PID parameters are constant and adjusted manually, the values for the fuzzy sets are generated by using a genetic algorithm, which according to [2] this type of fuzzy controller achieves greater results over other classical implementations.

Difficulty of tuning the parameters of PID controller was already mentioned. [3] proposes a auto-tuning PID module for robot motion systems. It is mentioned that, achieving an optimal performance in a dynamic system is difficult with constant PID parameters. Thus, [3] implements a conventional PID controller that receives its parameters from a recursive parameter estimator. This results in real-time parameter estimation which results in better step response and less disturbance making the mobile robot run smoother and precise.

2 Fuzzy Logic Controller

One of the earlier works, using fuzzy logic control for robot movement is [4]. It consists of multiple fuzzy logic controllers, that get the environment readings from 10 sensors in total. Outputs of these controllers are combined with an equation and sent to the wheels. Some of controller rules are: rules for dead alleys, obstacle avoidance, left and right edge following and more. In [4], it is concluded that having ten inputs makes it hard to create a proper rule base that covers all important behaviors, since the rule base grow rapidly with each extra input. It is also stated that having contrasting rules in the same rule base could cause errors that result in collisions with obstacles.

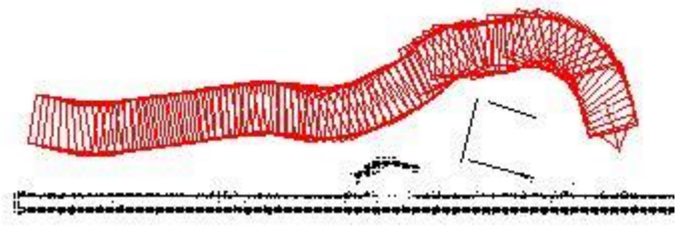
[5] proposes an improvement to the traditional fuzzy logic controller. It is argued in the article that, since classical models convert sensor readings directly to actions, they are prone to getting stuck in dead ends, not being able to navigate all the environment or reach a goal. [5] claims that having a grid-based map of the environment and a Path-Searching behavior together with classical obstacle avoidance and goal seeking, can prevent a mobile robot getting trapped in dead ends. The memory-grid holds the coordinates of seen obstacles and numbers of times a given location visited. Then the Path-searching behavior gives its own output by looking at the memory-grid, which results in mobile robot preferring unseen parts of the environment.

Another improvement over classical type-1 fuzzy logic controllers can be seen in [6], which proposes a type-2 fuzzy logic controller for autonomous mobile robots. In the paper, it is stated that, with type-1 fuzzy logic controller we can get satisfactory results in an uncertain environment, but type-1 controllers cannot completely handle the uncertainties that arise from changing environments because they use precise fuzzy sets, which in turn causes us to waste time tuning the controller specifically for the environment. [6] mentions that, since the type-2 fuzzy sets have a footprint of uncertainty, they can overcome these problems caused by changing environments, because type-2 sets can cover the same range as type-1 sets with fewer labels.

3 Results and Improvements

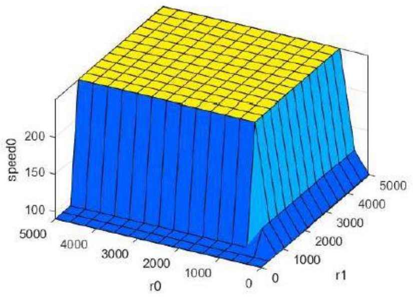
1 PID Controller

I think that the performance of the PID controller while following straight edges is satisfactory. It still has some steady state error, but overall it can get on track, move on quickly and properly. But I believe there is still a lot of room for improvement, since I do not find the non-straight edge performance good enough. There can still some oscillations and strong reactions be observed while testing the mobile robot. For future improvement, I strongly believe that there is a need for auto-tuning. Because manually tuning the parameters requires a lot of time and testing. Even though the tuning process looks intuitive and simple, it is difficult to choose proper values for parameters like Ki and Kd. Thus, having an algorithm or some type of machine learning technique for the tuning of the parameters is needed. Because the outcome parameters of this auto-tuning process might not look intuitive, but the combination of those parameters might preform really well in a given environment.


Movement of the mobile robot with PID controller, following a straight and non-straight edge.

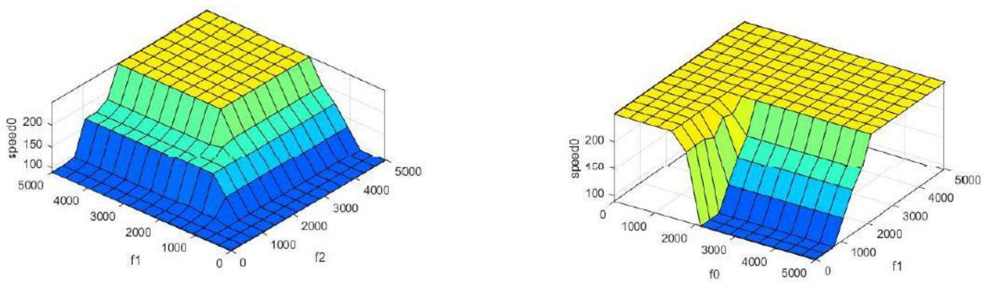
2 Fuzzy Logic Controller

The fuzzy logic implementation can perform surprisingly well despite having a really simple concept. If the structure of the environment is simple enough, it can wander around without any collisions with smooth movement. But on the other hand, it also has many weaknesses. First of them is not having a dead-end prevention. After extensive testing in MobileSim, it was observed that the mobile robot can get stuck in many different types of places in a very busy environment like the map Columbia. In this implementation, this issue was partly addressed by keeping the NEAR values of the obstacle avoidance set high. But this approach results in a different problem. Now the mobile robot prefers the emptiest paths instead of following a right edge and discovering anything on its path. I believe this can be prevented by having smaller NEAR values combined with dead-end prevention. Second weakness is having opposite rules like left and right edge following. Overall, these controllers



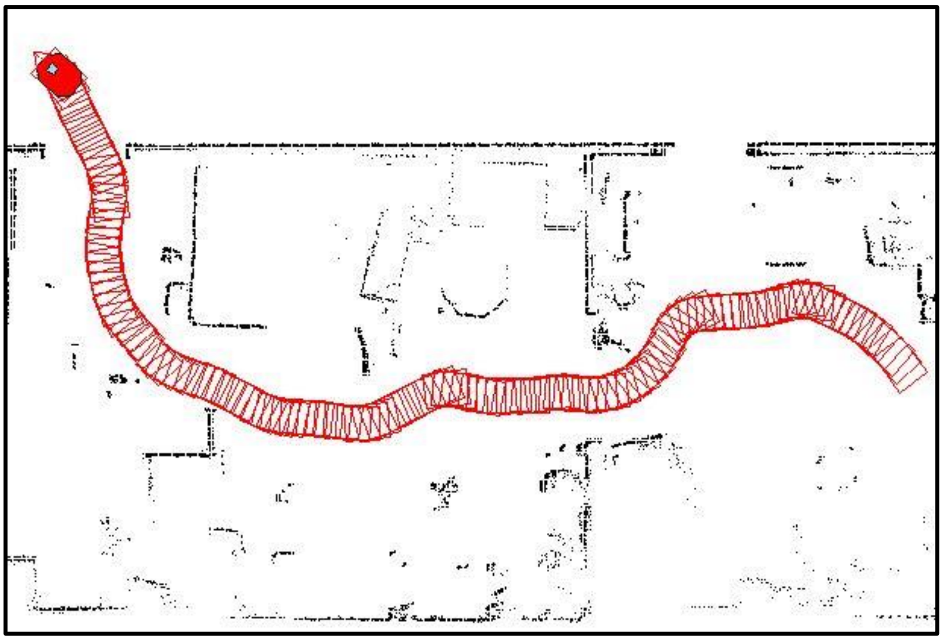
Surface of r0, r1 - output

are useful, but in some cases, they result in faulty navigation. I think this problem can be fixed after fine tuning the context-based blending values for these controllers.



Surfaces of f1, f2 – output and f0, f1 - output

Another weakness that was observed is the number of sonar sensors. As mentioned above, in simple environments the robot can perform well, but in a cluttered environment some small obstacles are not detected by the sonars on the robot, and these situations always end in collusions. Even though it could somehow be solved in software side, I think this can simply be addressed by adding more sonars or using a different type of rangefinder.



Movement of the mobile robot with fuzzy logic controllers in a part of the Columbia map in MobileSim.

4 Conclusions

The PID controller implemented in this assignment is very basic, using constant parameters that were adjusted manually. Even though the tuning of Kp and Ki did not cause much trouble, choosing a value for Kd proved to be very difficult. This can also be seen in [1] that, the chosen value for the Kd may not always give the desired result. Therefore, I believe that the PID controller should be accompanied by another mechanism like [2] or [3] to improve the performance of the mobile robot for navigation.

This classical type-1 fuzzy logic implementation works well in simple environments but struggles with dynamic obstacles and dead ends. Current number of rules do not cover all the required behavior that can keep the mobile robot navigating in the environment. Adding more and more rules could solve this problem, but as mentioned in [6] using a type-2 fuzzy logic controller would be a more robust and efficient solution. Type-2 controller could help us dealing with the robot getting stuck in dead ends, but I still think that adding a Path-Searching behavior like in [5] would be beneficial. Because even if we tune the type-2 sets perfectly, the robot might still not discover some of the places that are unknown.

And I still firmly believe that, for a mobile robot that can fully handle an uncertain and dynamic environment, eight sonars are not enough. Some small or narrow objects get ignored and result in the collision of the robot and the object. Which can be prevented by using more or different sensors.

5 References

[1] G. C. Kiam Heong Ang, Yun Li, "PID Control System Analysis, Design, and Technology," *IEEE TRANSACTIONS ON CONTROL SYSTEMS TECHNOLOGY,* vol., 13, no. 4, JULY 2005.

[2] M. K. S. Tang, Kim Fung Man, Guanrong Chen, Fellow, IEEE, and and M. Sam Kwong, IEEE, "An Optimal Fuzzy PID Controller," *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS,* vol.48, no. 4, AUGUST 2001.

[3] S. W. Huafei Xiao, "Auto-tuning PID Module of Robot Motion System," presented at the IEEE Conference on Industrial Electronics and Applications, 2011.

[4] I. Shigeki, "A Method of Indoor Mobile Robot Navigation by Using Fuzzy Control," *IEEE INTERNATIONAL WORKSHOP ON INTELLIGENT ROBOTS AND SYSTEMS,* 1991.

[5] J. N. K. L. Meng Wang, "Fuzzy logic-based real-time robot navigation in unknown environmentwith dead ends," *ROBOTICS AND AUTONOMOUS SYSTEMS,* vol. 56, pp. 625–643, 2008.

[6] H. A. Hagras, "A Hierarchical Type-2 Fuzzy Logic Control Architecture for Autonomous Mobile Robots " *IEEE TRANSACTIONS ON FUZZY SYSTEMS,* vol.12, no. 4, AUGUST 2004.

6 Appendix

Note**: Since the mistake was in 2 lines of code, only the latest version is included and the corrected part is highlighted.**

```
/*_______________________________________________________________________
__________
# CE801 Intelligent Systems and Robotics | Ogulcan Ozer. | 11 December
2018
      UNFINISHED. Read the rulebase from a text file.
_______________________________________________________________________
________*/
/*-----------------------------------------------------------------------
----------
#     CE801 Fuzzy logic system for right edge following and obstacle
avoidance
      with context-based blending for ARIA Robot.

      Usage: Upload the executable to the robot and run.
-------------------------------------------------------------------------
--------*/

#include <iostream>
#include<iostream>
#include<fstream>
#include<string>
#include <vector>
#include <string>
#include "Aria.h"
using namespace std;

/*-----------------------------------------------------------------------
----------
#     Data and structs.
-------------------------------------------------------------------------
--------*/
//RuleBase for right and left sensors, output is reversed for left edge
following.
static const vector<vector<string>> RuleBaseRF =
{
      {     //possible inputs
            "FAR,FAR",//1
            "FAR,NORM",//2
            "FAR,NEAR",//3
            "NORM,FAR",//4
            "NORM,NORM",//5
            "NORM,NEAR",//6
            "NEAR,FAR",//7
            "NEAR,NORM",//8
            "NEAR,NEAR"//9
      },
      {     //possible outputs
            "HIGH,LOW",//1
            "LOW,HIGH",//2
            "LOW,HIGH",//3
            "HIGH,MED",//4
            "HIGH,HIGH",//5
            "LOW,HIGH",//6
            "LOW,HIGH",//7
            "LOW,HIGH",//8
            "LOW,HIGH"//9
      }
};

//RuleBase for front sensors.
static const vector<vector<string>> RuleBaseOA =
{
      {
            //possible inputs
"NEAR,NEAR,NEAR",//1
"NEAR,NEAR,NORM",//2
"NEAR,NEAR,FAR",//3
"NEAR,NORM,NEAR",//4
"NEAR,NORM,NORM",//5
"NEAR,NORM,FAR",//6
"NEAR,FAR,NEAR",//7
"NEAR,FAR,NORM",//8
"NEAR,FAR,FAR",//9
"NORM,NEAR,NEAR",//10
"NORM,NEAR,NORM",//11
"NORM,NEAR,FAR",//12
"NORM,NORM,NEAR",//13
"NORM,NORM,NORM",//14
"NORM,NORM,FAR",//15
"NORM,FAR,NEAR",//16
"NORM,FAR,NORM",//17
"NORM,FAR,FAR",//18
"FAR,NEAR,NEAR",//19
"FAR,NEAR,NORM",//20
"FAR,NEAR,FAR",//21
"FAR,NORM,NEAR",//22
"FAR,NORM,NORM",//23
"FAR,NORM,FAR",//24
"FAR,FAR,NEAR",//25
"FAR,FAR,NORM",//26
"FAR,FAR,FAR"//27
      },
      {     //possible outputs
"LOW,HIGH",//28
"HIGH,LOW",//29
"HIGH,LOW",//30
"LOW,LOW",//31
"HIGH,LOW",//32
"HIGH,LOW",//33
"MED,MED",//34
"HIGH,LOW",//35
"HIGH,LOW",//36
"LOW,HIGH",//37
"LOW,HIGH",//38
"HIGH,LOW",//39
"LOW,HIGH",//40
"MED,HIGH",//41
"HIGH,MED",//42
"LOW,HIGH",//43
```

```
"HIGH,HIGH",//44
"HIGH,MED",//45
"LOW,HIGH",//46
"LOW,HIGH",//47
"LOW,HIGH",//48
"LOW,HIGH",//49
"MED,HIGH",//50
"MED,MED",//51
"LOW,HIGH",//52
"MED,MED",//53
"HIGH,HIGH"//54


      }
};


//Input struct to hold sensor readings their memberships.
struct Input
{
      vector<string> labels;
      double value;//Input from sonar.
      vector<double> memberships;//For possible memberships.
};

//Structure for holding a member of a Rule output.
struct OutRule
{
      string label;
      double value;
      double fire;


};

//Structure to hold a function's values for membership calculation.
//________________________________________________
//
//|     _____       |              |   ______
//|    /|   |\      |    /|\       |  |      |
//|   / |   | \     |   / | \      |  |      |
//|__/__|___|__\__  |__/__|__\__   |__|______|__
//  s  t1   t2  e     s  t1   e       s      e
//                       t2           t1     t2
//________________________________________________
struct Function
{
      string label;
      double start;
      double top[2];
      double end;

};

/*---------------------------------------------------------------------
----------
#     Variables for fuzzy-sets
```

```
-----------------------------------------------------------------------
--------*/
static const Function d1 = { "LEFT", 0,{ 0, 451 }, 1400 };// Functions
for context-based blending .
static const Function d2 = { "RIGHT", 0,{ 0, 1401 }, 5001 };//
static const Function d3 = { "FRONT", 0,{ 0 , 3000 }, 5001 };//

static const Function r0ne = { "NEAR", 0,{ 0, 200 }, 450 };//Functions
for right edge following, both for sensor 7 and 6.
static const Function r0no = { "NORM", 350,{ 450, 600 }, 700 };//
static const Function r0f = { "FAR", 600,{ 850, 5001 }, 5001 };//
static const Function r1ne = { "NEAR", 0,{ 0, 450 }, 600 };//
static const Function r1no = { "NORM", 450,{ 550, 850 }, 950 };//
static const Function r1f = { "FAR",  900,{ 1200, 5001 }, 5001 };//


//Functions for obstacle avoidance, for sensors 2, minimum of 3-4 and
5.
static const Function f0ne = { "NEAR", 0,{ 0, 700 }, 900 };//
static const Function f0no = { "NORM", 700,{ 1100, 1600 }, 2000 };//
static const Function f0f = { "FAR",  1700,{ 2600, 5001 }, 5001 };//
static const Function f1ne = { "NEAR", 0,{ 0, 900 }, 1200 };//
static const Function f1no = { "NORM", 900,{ 1300, 1500 }, 1900 };//
static const Function f1f = { "FAR",  1600,{ 2200, 5001 }, 5001 };//
static const Function f2ne = { "NEAR", 0,{ 0, 700 }, 900 };//
static const Function f2no = { "NORM", 700,{ 1100, 1600 }, 2000 };//
static const Function f2f = { "FAR",  1700,{ 2600, 5001 }, 5001 };//


static const Function yl = { "LOW", 40,{ 90, 90 }, 140 };//Functions
for motor speeds.
static const Function ym = { "MED", 120,{ 170, 170 }, 220 };//
static const Function yh = { "HIGH", 200,{ 250, 250 }, 300 };//

static const vector<Function> x0 = { r0ne,r0no,r0f };//Fuzzy sets for
right edge sensors.
static const vector<Function> x1 = { r1ne,r1no,r1f };//

static const vector<Function> f0 = { f0ne,f0no,f0f };//Fuzzy sets for
front sensors.
static const vector<Function> f1 = { f1ne,f1no,f1f };//
static const vector<Function> f2 = { f2ne,f2no,f2f };//

//static const vector<Function> y = { yl,ym,yh };//Fuzzy set for motor
speed.

static const vector<Function> o1 = { d1 };//Wrapper sets for context
based blending, since
static const vector<Function> o2 = { d2 };//membership function only
accepts Function vectors.
static const vector<Function> o3 = { d3 };//

 /*----------------------------------------------------------------------
-----------
 #    Function definitions
 ----------------------------------------------------------------------
---------*/
```

```
 //Permutation of activated set members for 2 input sets (Left edge and
right edge following).
vector< vector<OutRule> > permutate2(static const
vector<vector<string>> RuleBase, vector<Input> inputs);

//Permutation of activated set members for 3 input sets (Obstacle
avoidance).
vector< vector<OutRule> > permutate3(vector<Input> inputs);

//Return the index of a rule in the rule base.
int ruleIndex(vector<vector<string>> rules, string s);

//Get output speeds for left and right wheel from fire strengths.
vector<double> getSpeeds(vector<vector<OutRule>> func);

//Get the membership value of each input.
double getDegree(Input *in, const vector<Function> *func);

//Set the firing strength of each output.
void setFire(vector<vector<OutRule>> &func);


/*-----------------------------------------------------------------------
----------
#     Main Program
-------------------------------------------------------------------------
--------*/
int main(int argc, char **argv)
{
      //Initialize the robot.
      Aria::init();
      ArRobot robot;
      ArPose pose;
      ArSensorReading *sonarSensor[8];
      //Parse command line args.
      ArArgumentParser argParser(&argc, argv);
      argParser.loadDefaultArguments();
      ArRobotConnector robotConnector(&argParser, &robot);
      if (robotConnector.connectRobot()) {
            std::cout << "Robot Connected !" << std::endl;
      }
      robot.runAsync(false);
      robot.lock();
      robot.enableMotors();
      robot.unlock();
      //End of robot initialization.


      double minL, minR, minF;//For holding sensor minimums to be used
in context blending.

      //Main loop.
      while (true)
      {
            double sonarRange[8];//Read sonar values.
            for (int i = 0; i < 8; i++)
            {
                  sonarSensor[i] = robot.getSonarReading(i);
```

```
                    sonarRange[i] = sonarSensor[i]->getRange();

            }//


            cout << "Sensor R0:" << sonarRange[7] << endl;
            cout << "Sensor R1:" << sonarRange[6] << endl;
            cout << "Sensor F0:" << sonarRange[2] << endl;
            cout << "Sensor F1:" << min(sonarRange[3], sonarRange[4]) <<
endl;
            cout << "Sensor F2:" << sonarRange[5] << endl;
            minL = min(sonarRange[0], sonarRange[1]);//Get minimums for
each context
            minR = min(sonarRange[6], sonarRange[7]);//
            minF = min(min(sonarRange[5], sonarRange[2]),
min(sonarRange[3], sonarRange[4]));//
            //Wrap in Input structs.
            Input oR = { {}, minR };/* The fixed problem was, instead of
assigning minR to oR, minR was assigned to oL and vice versa. */
            Input oL = { {}, minL };
            Input oF = { {}, minF };
            //Get memberships.
            getDegree(&oL, &o1);
            getDegree(&oR, &o2);
            getDegree(&oF, &o3);
            //Wrap each sonar input.
            Input in1 = { {}, sonarRange[7] };
            Input in2 = { {}, sonarRange[6] };
            Input in3 = { {}, sonarRange[2] };
            Input in4 = { {}, min(sonarRange[3],sonarRange[4]) };
            Input in5 = { {}, sonarRange[5] };
            Input in6 = { {}, sonarRange[0] };
            Input in7 = { {}, sonarRange[1] };
            //Get membership values of each sonar.
            getDegree(&in1, &x0);
            getDegree(&in2, &x1);
            getDegree(&in3, &f0);
            getDegree(&in4, &f1);
            getDegree(&in5, &f2);
            getDegree(&in6, &x0);
            getDegree(&in7, &x1);


            //Group sensor outputs according to their use.
            vector<Input> inputsRE = { in1,in2 };//right
            vector<Input> inputsLE = { in6,in7 };//left
            vector<Input> inputsOA = { in3,in4,in5 };//obstacle
avoidance

            vector<vector<OutRule>> outRE = permutate2(RuleBaseRF,
inputsRE);//Get all the rules resulting from the input - for right edge
            vector<vector<OutRule>> outLE = permutate2(RuleBaseRF,
inputsLE);//-left edge
            vector<vector<OutRule>> outOA = permutate3(inputsOA);//-
obstacle avoidance
```

```
            //Set firing strengths of each group
            setFire(outLE);
            setFire(outOA);
            setFire(outRE);

            //Get speeds of the groups for each wheel
            vector<double>speedRE = getSpeeds(outRE);
            vector<double>speedOA = getSpeeds(outOA);
            vector<double>speedLE = getSpeeds(outLE);


            //Adjust speeds according to context-based blending.
            double lms = (((speedOA[0] * oF.memberships[0]) + 
(speedRE[0] * oR.memberships[0]) + (speedLE[1] * oL.memberships[0])) / 
(oF.memberships[0] + oL.memberships[0] + oR.memberships[0]));//Change 
the output of left
            double rms = (((speedOA[1] * oF.memberships[0]) + 
(speedRE[1] * oR.memberships[0]) + (speedLE[0] * oL.memberships[0])) / 
(oF.memberships[0] + oL.memberships[0] + oR.memberships[0]));//




            //Send them to the wheels.
            robot.setVel2(lms, rms);

            ArUtil::sleep(75);//Sleep

      }
      ////End of main program.




}

/*-----------------------------------------------------------------------
----------
#     Functions
-----------------------------------------------------------------------
--------*/

//Permutation of activated set members for 2 input sets (Left edge and 
right edge following).
vector< vector<OutRule> > permutate2(static const 
vector<vector<string>> RuleBase, vector<Input> inputs) {

      vector< vector<OutRule >> output;//output to be returned.
      output.resize(2);
      OutRule temp;

      //Repeat 9 times.
      for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) {
```

```
				double m1 = inputs[0].memberships[i];
				double m2 = inputs[1].memberships[j];

				if ((m1 != 0) && (m2 != 0)) {//For each input, if 
memberships are not 0, get the corresponding rule.
					int idx = ruleIndex(RuleBase, 
inputs[0].labels[i] + "," + inputs[1].labels[j]);
					int a = RuleBase[1][idx].find(",", 0);
					temp.value = min(m1, m2);
					temp.label = (RuleBase[1][idx]).substr(0, a);
					output[0].push_back(temp);

					temp.label = (RuleBase[1][idx]).substr(a + 1);
					output[1].push_back(temp);

				}



			}

		}

		return output;



}

//Permutation of activated set members for 3 input sets (Obstacle 
avoidance).
vector< vector<OutRule> > permutate3(vector<Input> inputs) {

		vector< vector<OutRule >> output;//output to be returned.
		output.resize(2);
		OutRule temp;

		//Repeat 27 times.
		for (int i = 0; i < 3; i++) {

			for (int j = 0; j < 3; j++) {

				for (int k = 0; k < 3; k++) {


					double m1 = inputs[0].memberships[i];
					double m2 = inputs[1].memberships[j];
					double m3 = inputs[2].memberships[k];

					if ((m1 != 0) && (m2 != 0) && (m3 != 0)) {//For 
each input, if memberships are not 0, get the corresponding rule.
						int idx = ruleIndex(RuleBaseOA, 
inputs[0].labels[i] + "," + inputs[1].labels[j] + "," + 
inputs[1].labels[k]);
						int a = RuleBaseOA[1][idx].find(",", 0);
						double t = min(m1, m2);
						temp.value = min(t, m3);
```

```
                              temp.label = (RuleBaseOA[1][idx]).substr(0, 
a);
                              output[0].push_back(temp);
                              temp.label = (RuleBaseOA[1][idx]).substr(a 
+ 1);
                              output[1].push_back(temp);



                        }
                  }


            }

      }

      return output;



}

//Return the index of a rule in the rule base.
int ruleIndex(vector<vector<string>> rules, string s)
{
      for (int i = 0; i < rules[0].size(); i++)
      {
            if (rules[0][i].compare(s) == 0) {
                  return i;
            }
      }
      return -1;
}

//Get output speeds for left and right wheel from fire strengths.
vector<double> getSpeeds(vector<vector<OutRule>> func)
{
      double leftu = 0, leftd = 0, rightu = 0, rightd = 0;

      for (int i = 0; i < func[0].size(); i++) {//Do for left

            leftu = leftu + func[0][i].fire;
            leftd = leftd + func[0][i].value;
      }
      for (int i = 0; i < func[1].size(); i++) {//Do for right
            rightu = rightu + func[1][i].fire;
            rightd = rightd + func[1][i].value;
      }
      vector<double> speeds = { leftu / leftd,rightu / rightd };

      return speeds;
}

//Get the membership value of each input.
double getDegree(Input *in, const vector<Function> *func)
{
	int c = 0;
	for (auto tr : *func) {
		in->labels.push_back(tr.label);

		// This part checks for special cases.
		if ((in->value <= tr.start) || (in->value >= tr.end))//Check
if the value is at the edges.
		{
			if (tr.start == in->value && tr.start == tr.top[0]) {
				in->memberships.push_back(1);//Step function, or
edge steps up at the beginning
			}
			else if (tr.end == in->value && tr.end == tr.top[1]) {
				in->memberships.push_back(1);//Step function, or
edge steps down at the end

			}
			else {
				in->memberships.push_back(0.0);//Outside of the
function bounds
			}


		}
		else {

			if (tr.start < in->value && in->value < tr.top[0])
				in->memberships.push_back((in->value - tr.start)
/ (tr.top[0] - tr.start));//If rising edge.
			if (tr.top[0] <= in->value && in->value <= tr.top[1])
				in->memberships.push_back(1); //If highest
point.
			if (tr.top[1] < in->value && in->value < tr.end)
				in->memberships.push_back((tr.end - in->value) /
(tr.end - tr.top[1]));//If falling edge.
		}
	}
	return 0;

}

//Set the firing strength of each output for a given output rule.
void setFire(vector<vector<OutRule>> &func)
{
	for (int i = 0; i < func[0].size(); i++) {//Do for left wheel.
		if (func[0][i].label.compare("LOW") == 0) {
			func[0][i].fire = func[0][i].value*(yl.end + yl.start)
/ 2;
		}
		else if (func[0][i].label.compare("MED") == 0) {
			func[0][i].fire = func[0][i].value*(ym.end + ym.start)
/ 2;
		}
		else {
			func[0][i].fire = func[0][i].value*(yh.end + yh.start)
/ 2;
		}
	}
```

```
      for (int i = 0; i < func[1].size(); i++) {//Do for right wheel
            if (func[1][i].label.compare("LOW") == 0) {
                  func[1][i].fire = func[1][i].value*(yl.end + yl.start)
/ 2;
            }
            else if (func[1][i].label.compare("MED") == 0) {
                  func[1][i].fire = func[1][i].value*(ym.end + ym.start)
/ 2;
            }
            else {
                  func[1][i].fire = func[1][i].value*(yh.end + yh.start)
/ 2;
            }
      }


}

/*------------------------------------------------------------------------
----------
#     End of program
--------------------------------------------------------------------------
--------*/
```