CE801: PID Controller and Fuzzy Logic Controller

Peter Gent  
Chelmsford  
plgent@essex.ac.uk

Reg no:1802559

*This report details the construction of fuzzy logic and PID controllers to the application of a right wall following robot. The background research for PID and fuzzy logic is considered before detailing the experimental progress and procedures. This concludes with a detailed analysis of the results for the fuzzy logic in terms of Matlab inputs and outputs.*

# Introduction (*Heading 1*)

This paper covers two projects that are a part of the CE801 assignment. The first part of the assignment is involved with writing a proportional-integrative-derivative (PID) controller robot that follows a wall on the right hand side. The second part of the assignment is involved with creating a fuzzy logic controller that follows a right hand wall and avoids obstacles. For both parts of the assignment there is a literary review of PID and fuzzy logic principles and an overall conclusion for both projects of lessons learnt from the process of developing a PID solution and a fuzzy logic solution.

# LITERARY REVIEW

## PID CONTROLLER

[1]

A PID or proportional, integral and derivative controller is made up of three elements that combine together to produce a solution that can control a robot. The first of these three controllers P can be considered in isolation:

[1]

With this part of a controller the error is multiplied by a constant. The effect of multiplying the error scales down or up the error to make the robot react more drastically compared to the error. With a PD controller the change in error between measurements is taken and then multiplied by the D constant to change how much the change in error affects the overall error. This can be surmised in the equation below.



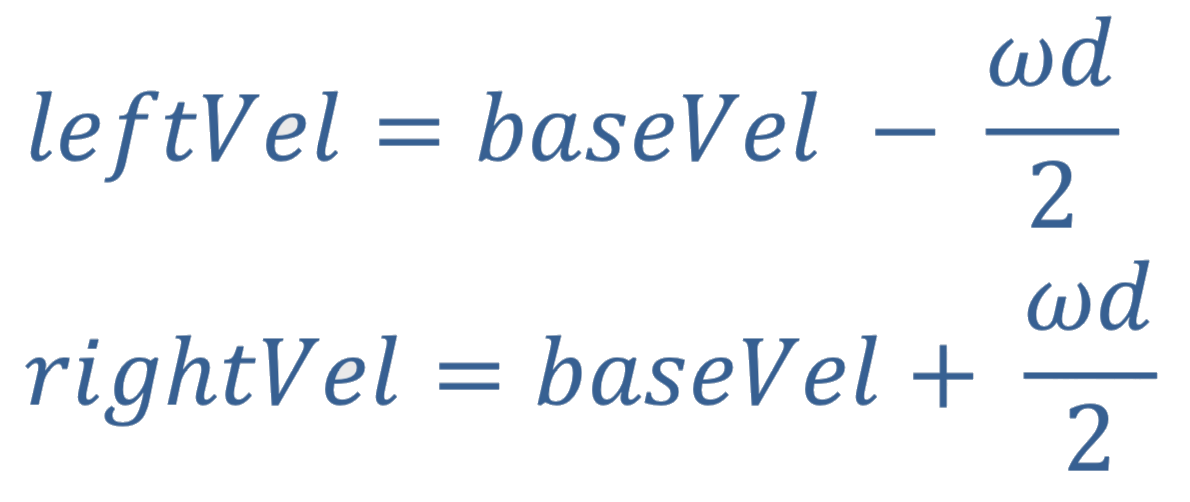
[1]

The effect of PD is that it will correct itself if error is reducing too fast or not fast enough. Finally with PI considered the history of errors are accumulated to ensure convergence on the expected value:

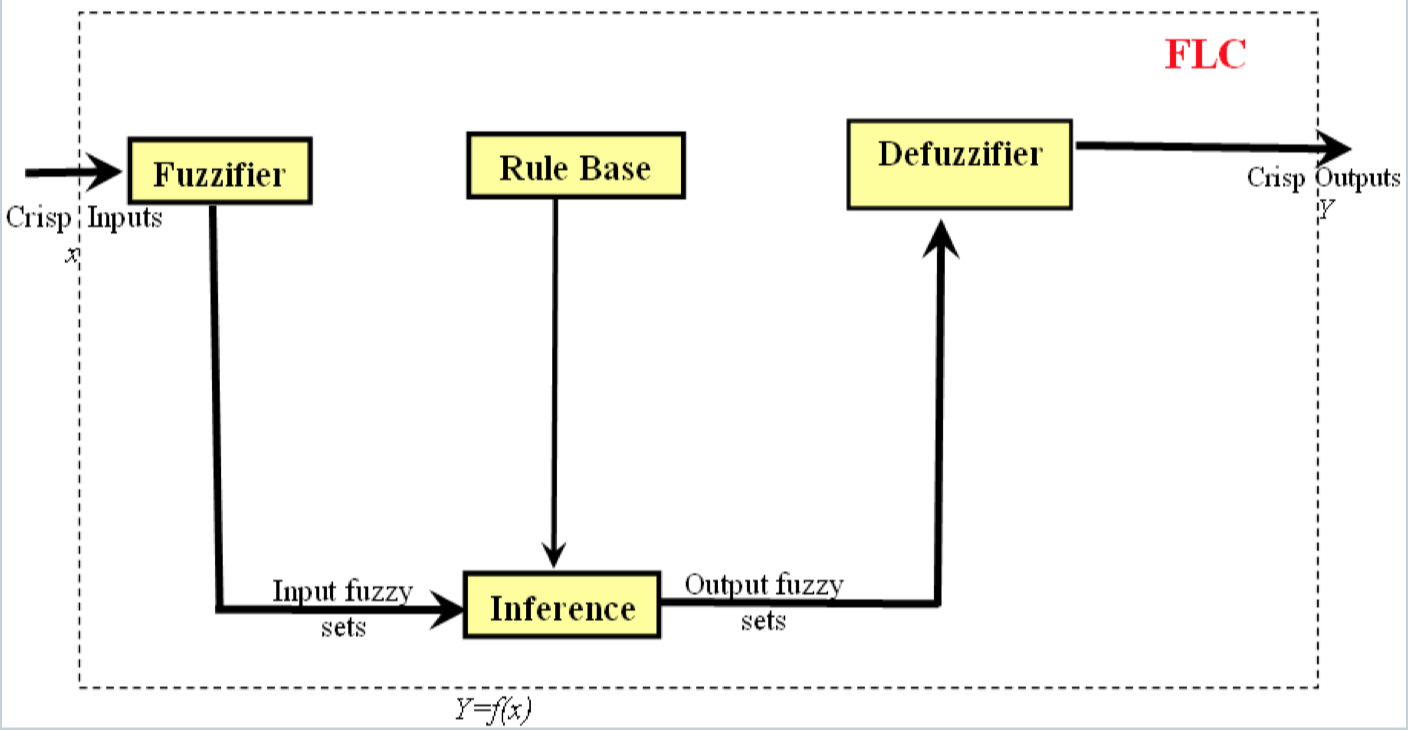
[1]

When combined into PID the result is an equation that provides a complete ‘off the shelf’ solution that balances the past error history, change in error and the current error:

[1]

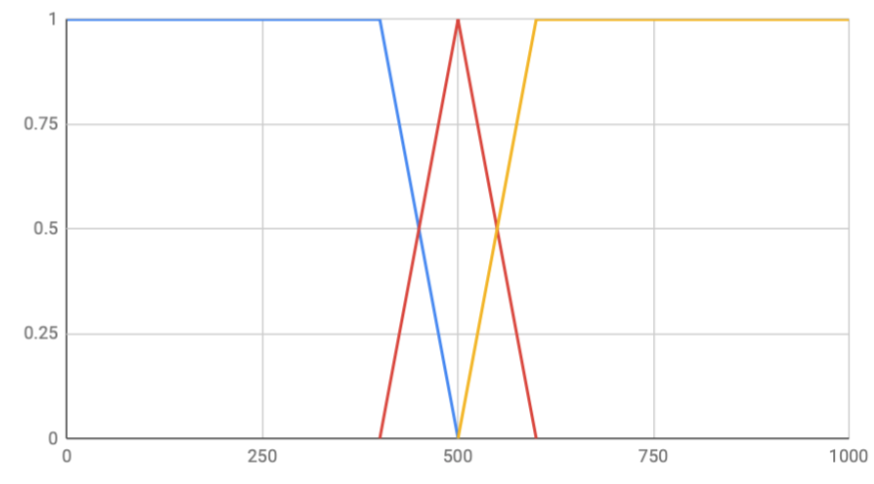
With PID calculated it is necessary to calculate the angular speed in order to move a robot in the correct direction. This can be calculated with the function above. It solves the left and right velocities separately by taking a base velocity and adding it to the change in angular velocity that factors in error from the PID controller divided by two [2].

## FUZZY LOGIC:



*Fig 1. architecture of a fuzzy logic based system* [3]

A fuzzifier system relies on a four stage process of fuzzifiying crisp inputs into fuzzy outputs, inferring the mapping of output rules from the rule base from input rules, and then defuzzifiying output values based upon the output rules that are inferred and the minimum or maximum firing strengths depending on the defuzzification technique chosen (height, centroid of set, or maxima amongst others)[4].

**

## fig 2. Membership graph

The first step of creating a fuzzy logic controller is to create a fuzzier that turns crisp values into approximate or fuzzy values that are normally better expressed in terms of natural language such as cold, warm or hot [5]. In a system using crisp values the system could only be either cold, warm, or hot with sharp boundaries between i.e. true or false for being a member of a temperature designation [6].

With a fuzzifier it is a simple matter of creating an input graph like that in figure 2. In this example blue may be cold, red may be warm and yellow may be hot. For the purpose of explaining the membership functions in further detail the values of 400, 410, 500 are taken. For 400 we get a firing strength of 1 of set cold and discount the 0 firing strength for warm. Likewise for 500 a firing strength of 1 for fuzzy set warm and 0 for values cold or hot. The interesting part of a fuzzy logic fuzzier is what happens in the transitions or grey area between fuzzy sets that are either leading or falling. For example to the layman with an x value of 410 may either belong to set A or set B so what can be done in this situation? For this purpose crisp values are used to generate a fuzzy set with two equations [7]:

X-A

\_\_\_

B-A

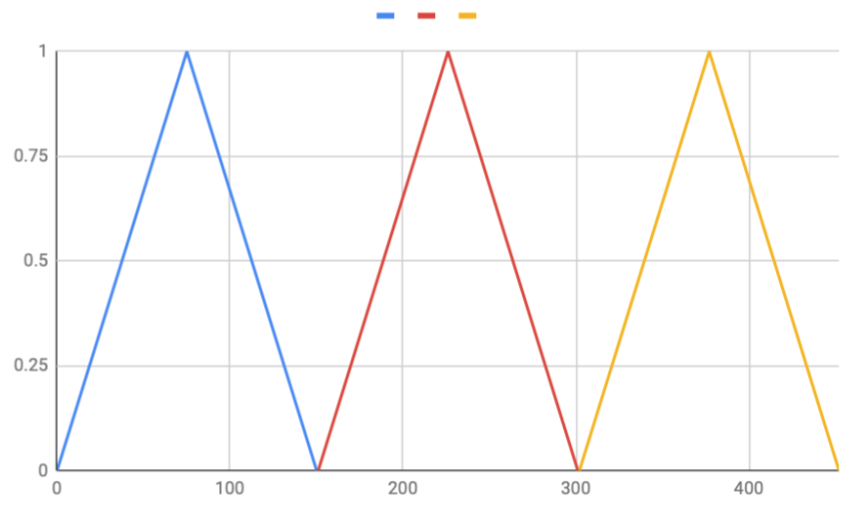
B-X

\_\_\_

B-A

In the first equation x-a the x is leading indicating a line that is ascending and in the second equation b-x it indicates a line that is falling. When the example 410 is used with both of these equations the firing strengths can be calculated as members to either of the shapes. It is with these values that the inference engine is used against rules held in a rule base.

## Figure 3. rule base table example with lookups for motor speed.



## Figure 4. Output graph

At this point permutations of the fuzzy inputs are created. For the crisp input of 410 for example creates a fuzzy set of low and medium. This creates the permutations low med and med low when considered in isolation. The final step in the inference engine is then to work out the minimum firing strengths for each. With the firing strengths selected it is then possible to defuzzify the outputs by using centroid of sets deffuzification. This simply involves multiplying the minimum shape strengths reading of the x axis with firing strengths ( see figure 4.) .

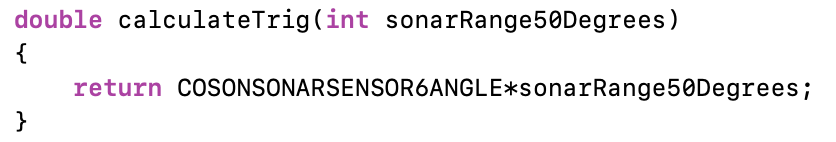
| MFS | LFS | RMS | LMS |
| --- | --- | --- | --- |
| Low | Low | Slow | Fast |
| Low | Med | Med | Slow |
| Low | High | Fast | Slow |
| Med | Low | Med | Med |
| Med | Med | Med | Slow |
| Med | High | Fast | Slow |

# PID CONTROLLER

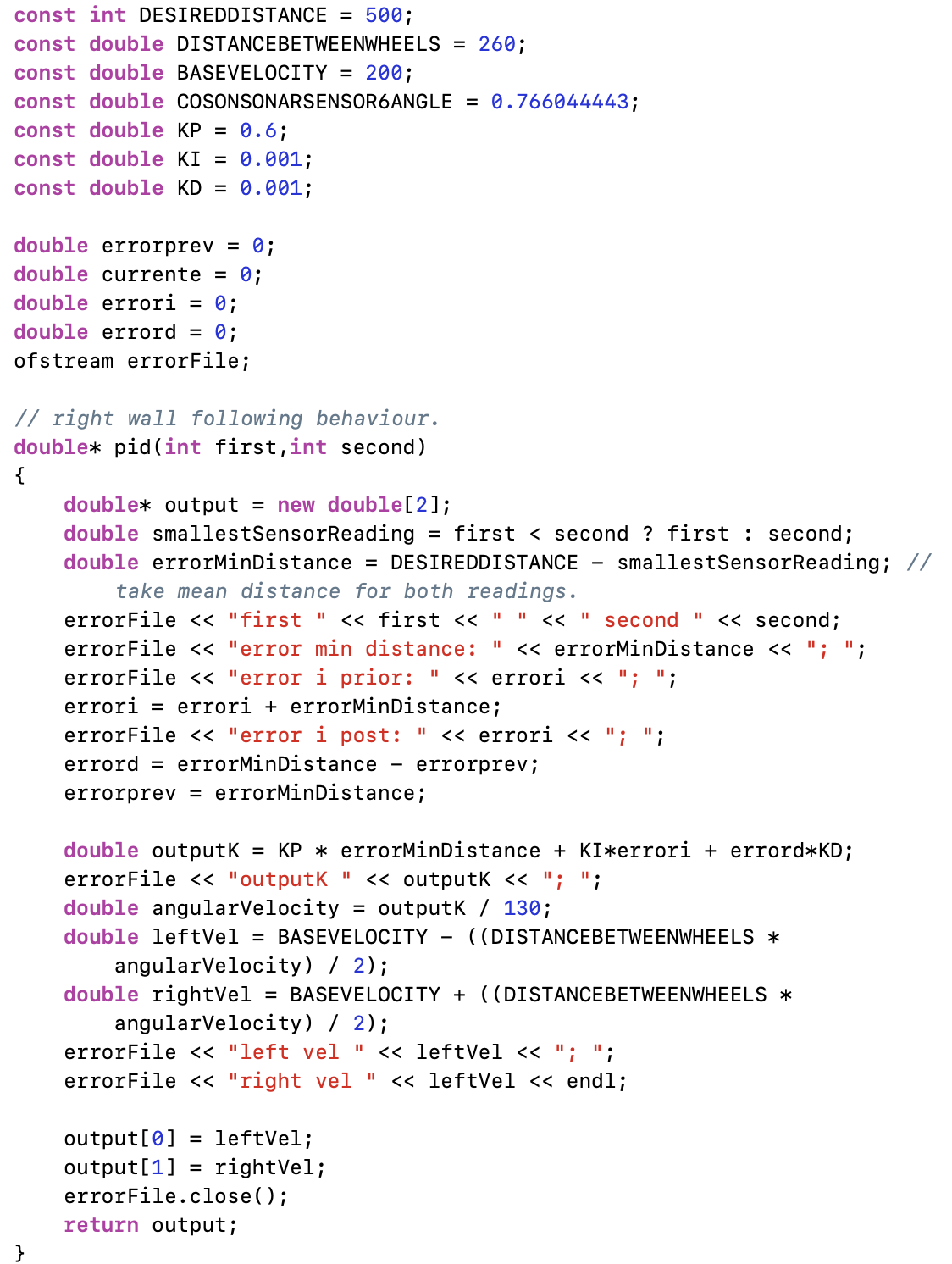
The code for the PID controller is relatively basic and understandable to the layman, but in the interest of clarity the code will be explained here in a run through from the start of the program in main.cpp.

## 

In order to prepare the robot for moving the robot has to be unlocked and the motors must have it’s motors enabled. After that it is just a simple case of getting the readings (6-7) for right hand side instead of 0-1 for left side wall following.



The distance to the wall must then be calculated for the right forward sensor by using some simple trigonometry (above). This is because only the hypotenuse is known from the front right sonar range and it doesn’t reflect the right angular distance from the wall.



Once the ranges are gathered the output function is called and this does most of the work to actually calculate a solution that has the expected velocity to steer the robot along and around the right edge wall. At this point it is worth noting that there are a number of constants, including desired distance, base velocity, distance between walls as well as tuning values for KP KI and KD to tune the PID controller. The values for KP KO and KD were tuned by following a procedure of implementing the code, running the code on the robot and then tuning KP KO and KD by a process of trial and error. In the end KI and KD were tuned to be negligible values because tuning these values any higher would lead to undesirable behaviour i.e. the robot would go in full reverse. Hence that is why KP is so large vs KI or KD.

In order to calculate the overall error outputK the error for each of the PID elements are calculated separately in the manner described in the literary review. The error for K is merely the distance between the desired distance and the actual distance, whereas the error for the integral is the integral plus min distance. Finally the derivative error is the change in error between two measurements. However as both KI and KD are set to negligible values both of the errors for these are made to be negligible and therefore do not require further consideration. The final thing that is done to get a desirable output is to work out the left and right velocities based upon the error for KP. The angular velocity calculated by dividing the outputK by 130. With the angular velocity its then possible to calculate the angular velocity for the right and left wheels by using the equation for wheel velocity from the literary review.

In summary the code for PID has been covered and so has the design process for creating a PID robot using trial and error. The result of the code is as aforementioned a robot that follows a right wall edge and does so around corners. In the next section of the report a fuzzy logic approach will be taken to achieve the same behaviour with a more sophisticated approach that has the potential to be applied to more tasks than a PID controller.

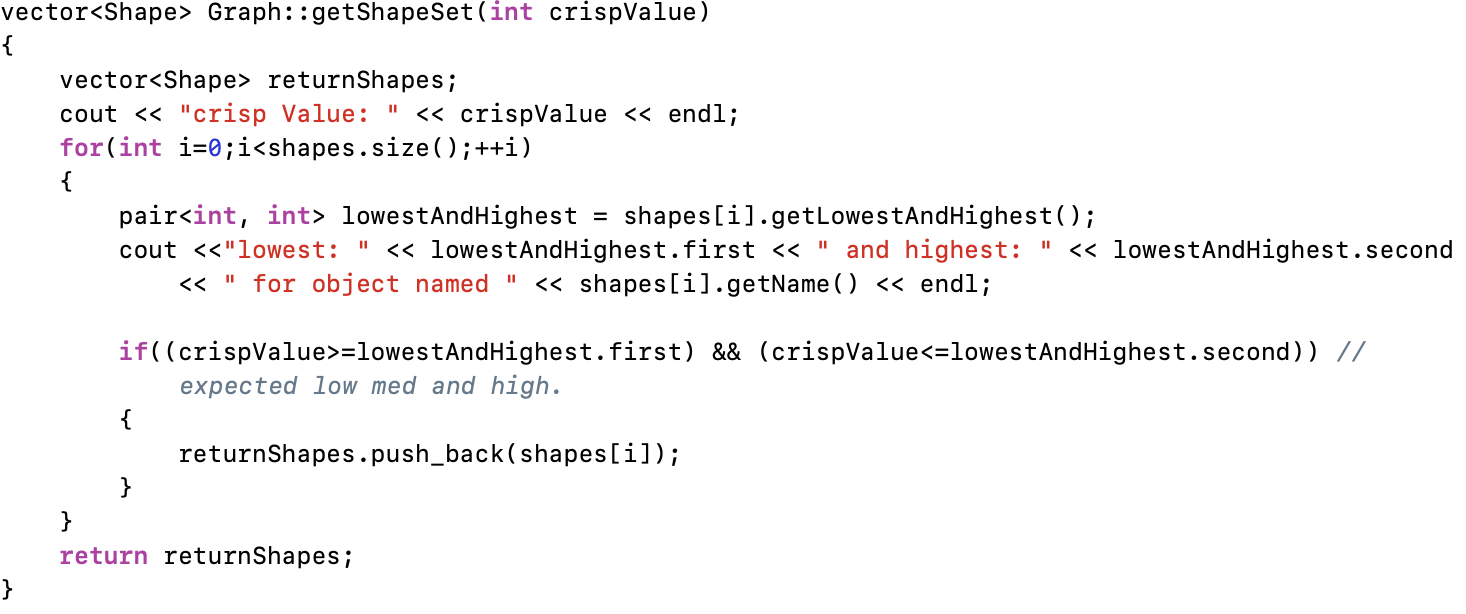
1. FUZZY LOGIC CONTROLLER

In the first part of this section a run through of the code will be given to demonstrate how a fuzzy logic system can be implemented.

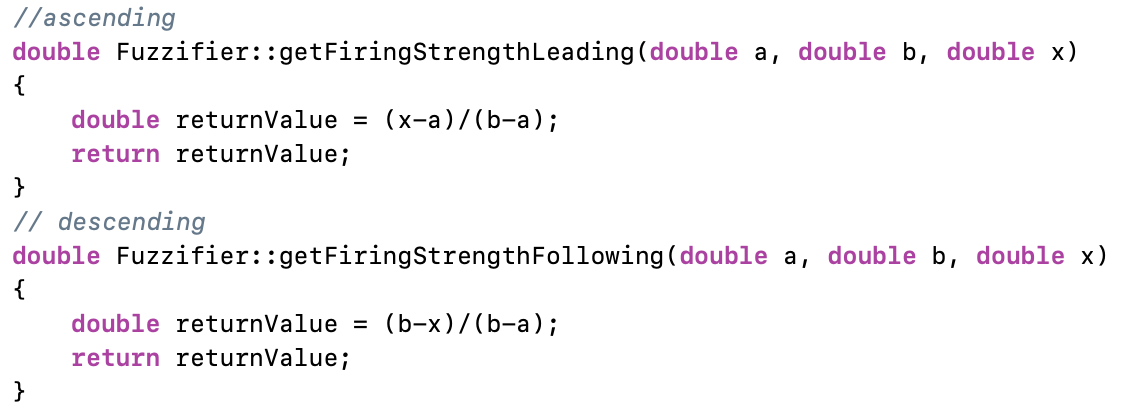
The main source code gives a top level view of the overall algorithm for calculating velocities with fuzzy logic based upon crisp inputs taken into the fuzzification part of the fuzzy logic system and centroid of sets output from the defuzzifier as pictured in the literary review. For this purpose several classes exist: a fuzzier to get fuzzy inputs, which contains a graph of shapes; an inference engine that acts like a lookup table to get the output rules that fire based upon the fuzzy inputs; and a defuzzifier that works out how fast either the left or right wheels should go based on centroid of sets before setting the velocity of each wheel. The three stage process will now be covered in more detail to explain any design patterns used and the structure of the code itself, including any object orientation.

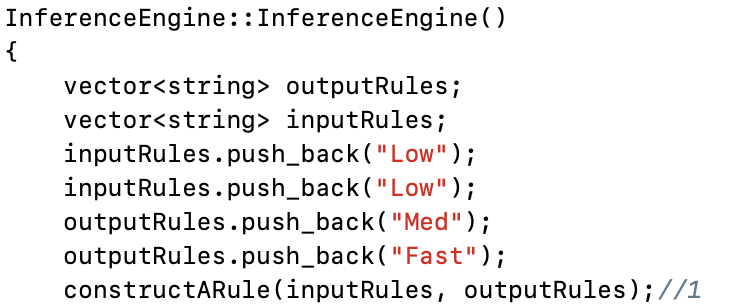
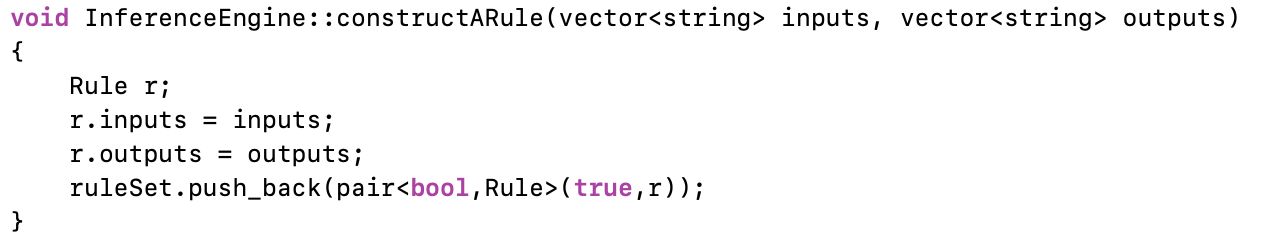
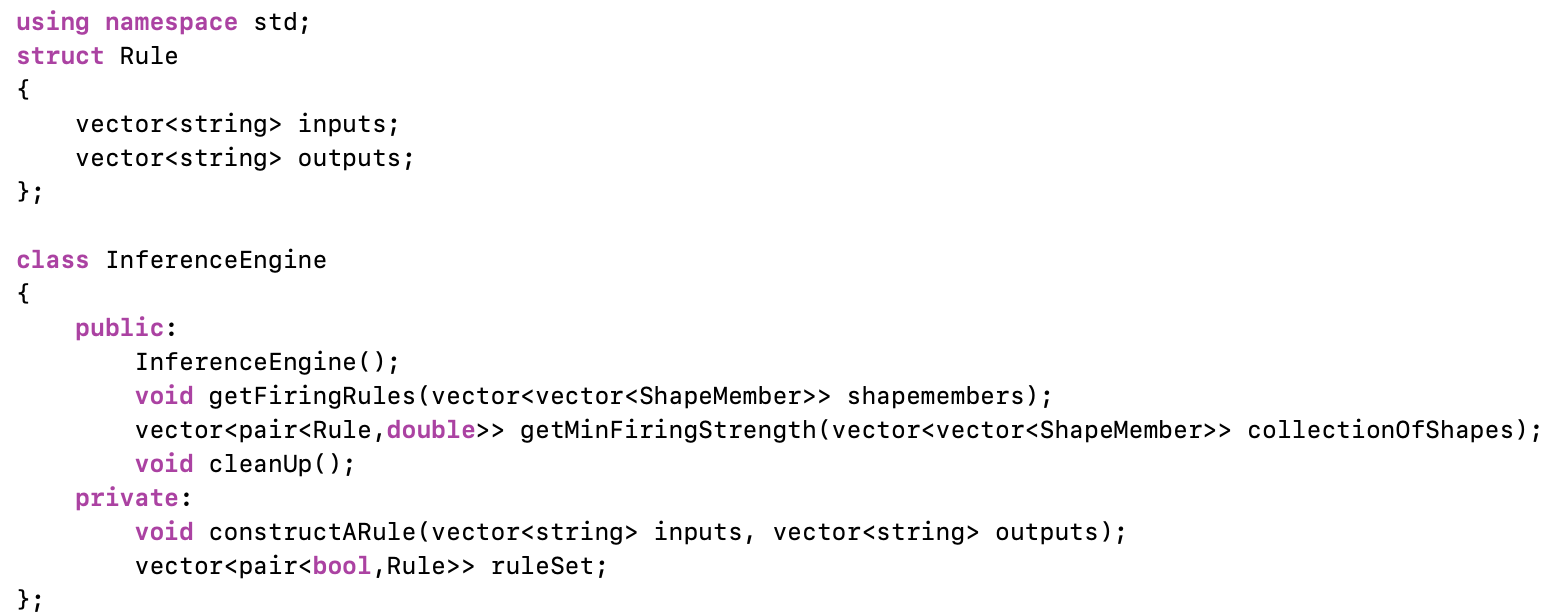
The first stage of the fuzzy logic process is to instantiate a RANGEGRAPH enum to recall the correct graph in the fuzzifier, when the firing strengths for each sensor are calculated. At this point like with the PID controller for the right forward sensor trigonometry is used to calculate the distance from the wall. The function retrieves instances of the ShapeMember class for each of the firing strengths stores the shape name, the crisp value and the firing strength retrieved for the sake of transparency so that any errors can be backtraced through the firing strength function.



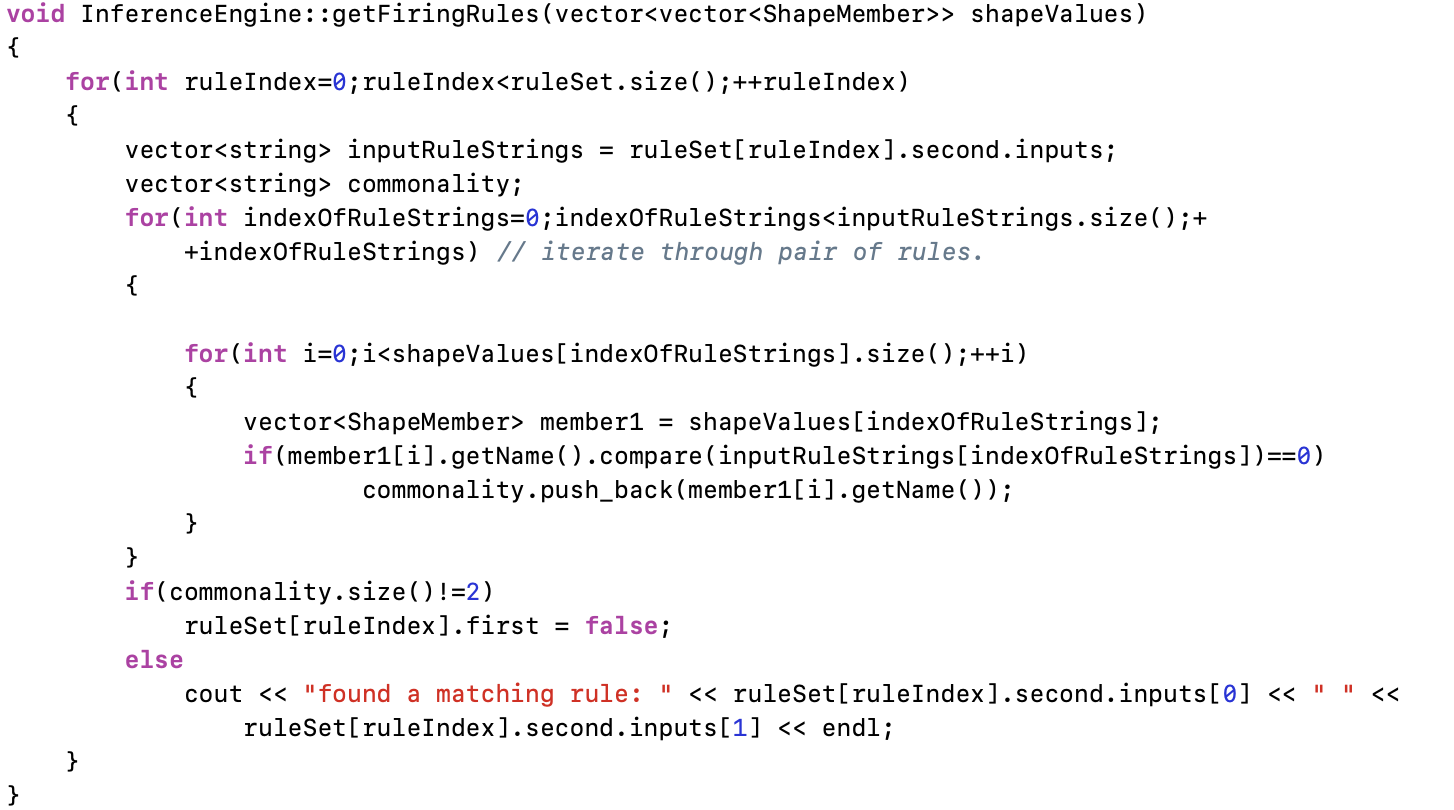


The firing strength function returns one of three possible outputs: one for the case where the fuzzified inputs only correspond to a single set; one where the inputs correspond to two shapes (AB or BA); and one where it corresponds to three or more shapes. The three outcomes rely on the getShapeSet function held in graph. A graph is an object that has a vector of shapes inside it in a composition, rather than inheritance relationship. The getShapeSet iterates through the shapes held in a graph and then adds to the vector of return shapes if the crisp value is between the highest and lowest values of x. In the most complex case where there is an intersection between two shapes one of two actions are undertaken. In the first case where the shape is on a turning point (i.e. on a corner with one shape of firing strength 1 and another 0) checkTurningPoints adds to and returns firingStrengths for only the shape which has a firing strength of one, discounting the a firing strength of 0. In the case where there is an intersection the ascending and descending (leading or following) shape have both of their firing strengths calculated and returned as shape members (see above).



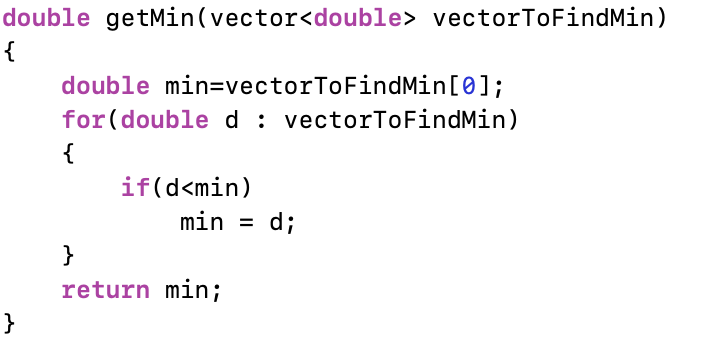


Following the calculation of firing strengths for both right and right forward sensors the inference engine is called to get the output rules for a given set of shapes encapsulated as ShapeMembers. In order to calculate the applicable rules a complete ruleset is created upon the construction of the inference engine, with each rule being constructed through a constructARule interface (see above). The combination of input rules and output rules as strings are used to construct a rule that encapsulates the input and output values against a boolean value.

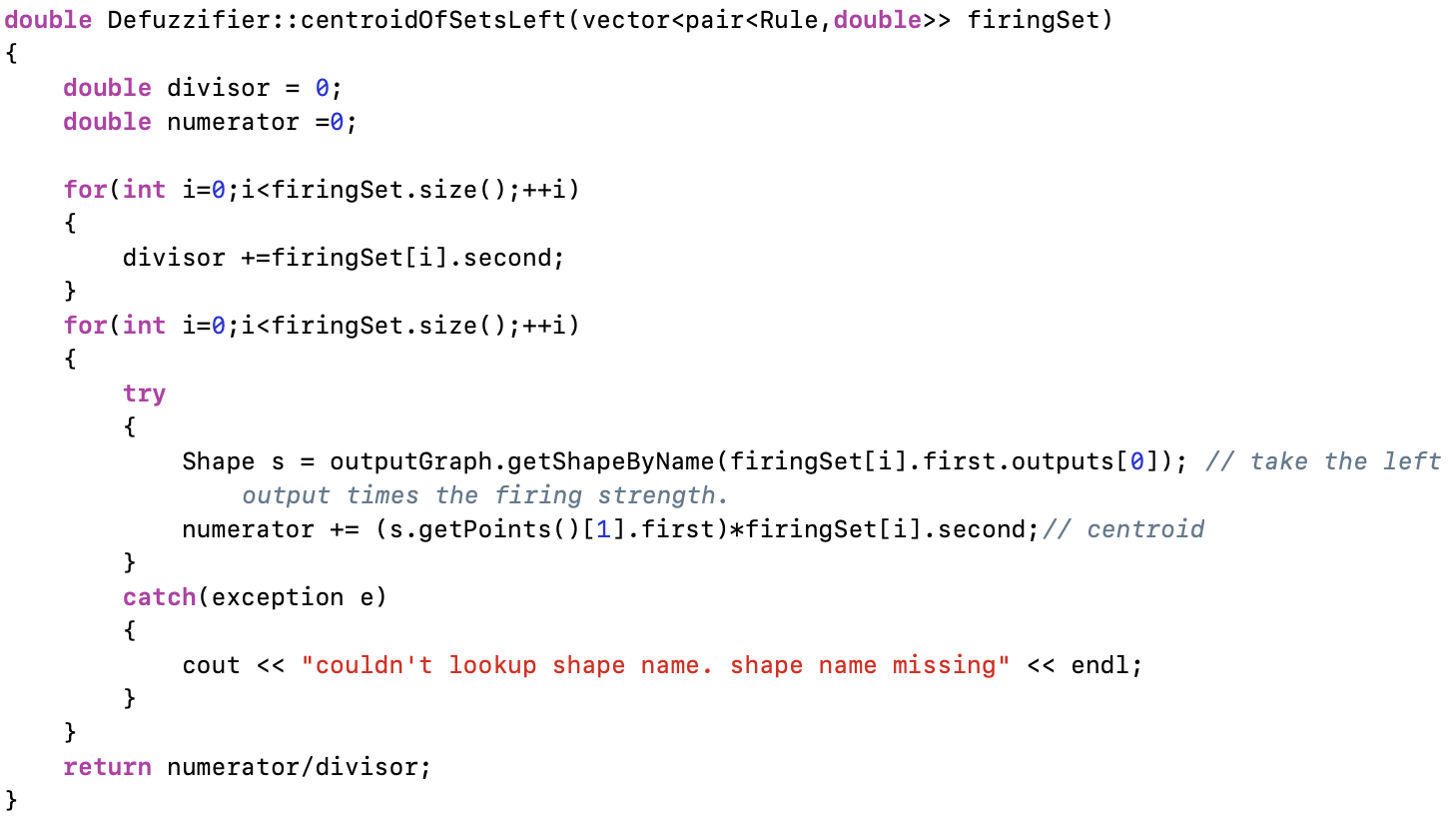


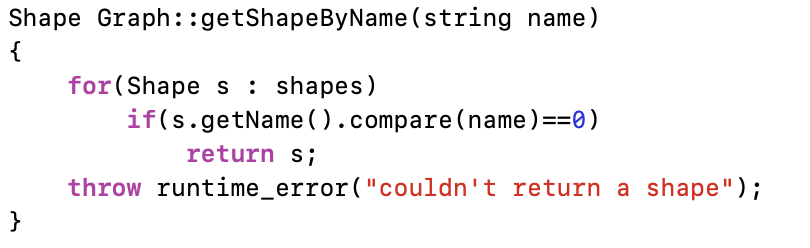
The first stage of inferring the output rules is to generate all combinations of input rules that can fire. This is done by first getting each rule in the inference engine to match against shape member names in inputRuleStrings. Then the rule is iterated through for each of the shape values and if two shapes are not matched then the rule is turned to false. If a rule passes on the other hand then the next call to the inference engine in the function getMinFiringStrength will look through the rules that matched the inputs.

The getMinFiringStrength function will then unpack a given shape member and push back the firing strengths for a given shape. A helper function getMin then finds the input shape affiliation with the smallest firing strength out of the rules attached. The output rule with the smallest firing strength inside getMinFiringStrength will then be aggregated with the output rule.



Subsequently to finding all of the output firing strengths the inference engine is cleaned up to turn all of the negated rules back to true ready for the next call to the inference engine. The next and final part of the code handles left and right defuzzification as a part of the Defuzzifier class.





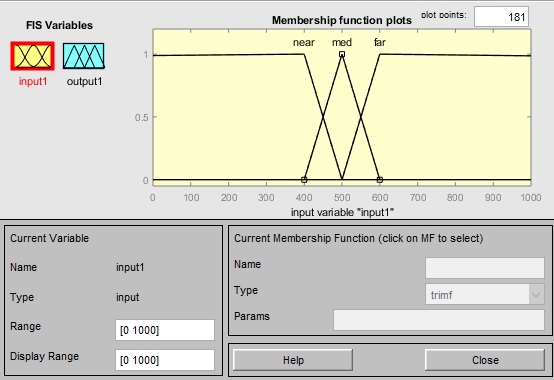
The only major difference between centroidOfSets left to right is whether or not the first or second value of the output shapes is taken. In this case for the sake of brevity only the centroidOfSetsLeft function is considered as the differences are minimal. In both cases the divisor is first calculated as the sum of firing strengths, followed by calculation of the numerator. The numerator is calculated by translating the string name of a firing rule with that held inside the outputGraph. This is done by calling the .getShapeByName function inside Graph.cpp.

This is a very simple function (see above) that simply goes through all of the shapes held in a graph such as outputGraph and returns a shape when it is found from the list of shapes that a graph contains. When it does so then the point where the output graph is highest (firing strength 1) is taken and then multiplied by the firing strength of the firing set. The numerator is finally divided by the divisor and this directly gives the respected left or right velocity for the centroid of sets function. With the code covered it is important to consider how the rules were designed and the processes behind creating an effective input and output ruleset.

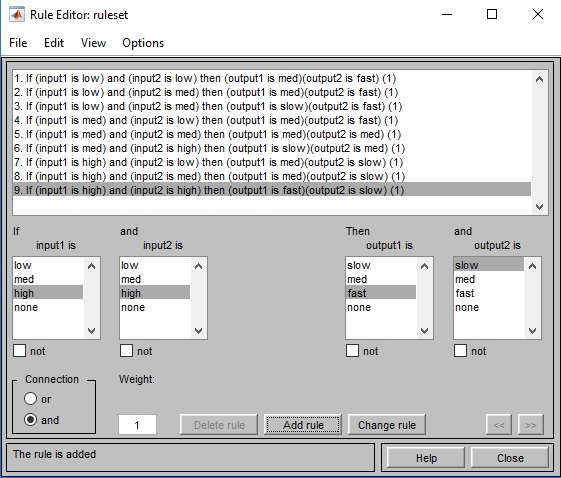
The first point of call for creating a comprehensive ruleset was to prototype the ruleset in matlab. This is a process that started with creating input and output functions for the fuzzy tool in matlab (see figure 5 and figure 6).

## 

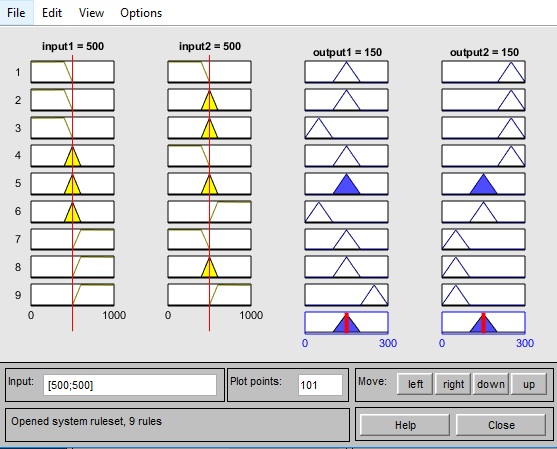
## Figure 5. creating an output function for



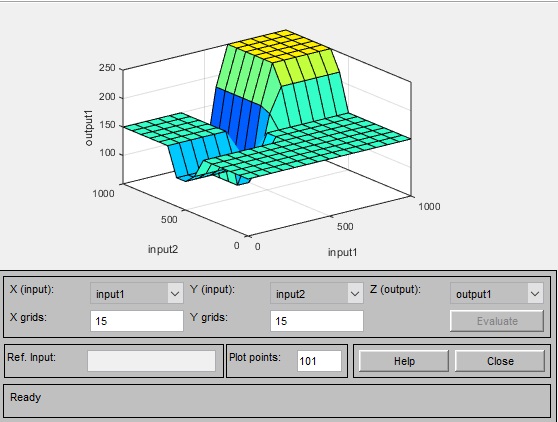
## fig 6 input graph



## fig 7 rulebase for two inputs and two outputs.

****

## fig 8. outputs for rules

****

## fig 9. surface view of rule base.

The process then involved creating firing rules for outputs against inputs (figure 7) by using trial and error. The final result of this can be seen in the surface view in figure 9. The result is a ruleset that when followed by the robot results in the robot navigating along and around corners on the right hand side. With further work more rules could be created to make a robot that not only follows the right wall but avoids obstacles as well. This would require the use of a front sensor to look for obstacles in front of the robot to be avoided and result in a much more complicated ruleset of 33 i.e. 27 rules. As time constraints were an issue the best that could be achieved in the time given was a robot that would navigate, albeit without a smooth movement along the right hand side. Further development would hopefully result in a more smooth motion and obstacle avoidance with either a subsumption architecture or context based blending. What context based blending provides over subsumption is a set of behaviours that may be active at the same time to produce fluid output [].

# CONCLUSION

This report has considered both PID and fuzzy logic controllers as two possible archetypes for solving a simple wall following behaviours. The first of these, the PID robot has been created navigates around a right wall flawlessly, all without a large integral or derivative. The second controller on the other hand, the fuzzy logic controller is able to navigate around a right wall but it lacks obstacle avoidance behaviour and exhibits a jerky motion whether following a straight wall or navigating around a corner. The main reason for this is suspected to be the shape of the input member operation. As can be seen in figure 6 the member function has a very fine line between medium, near and far functions. If there was a trapezoid with a larger catchment zone for medium distance the robot would probably follow the right wall with more ease in a straight line, rather than jerking forward bit by bit.

In both cases for the PID and fuzzy logic there was a defined process of design and development of the solutions. For the PID robot this was a case of running the robot, watching what it does and then begin tuning the kp variable until the robot demonstrated better behaviour. This followed by tuning the ki and kd variables until the robot could navigate along and around a right edge and right corners. The fuzzy logic used a much more lengthly process of creating input and output rules on a whiteboard before prototyping them in Matlab. As a starting point to this process it turned out relatively easy to work out that the number of rules required was three to the power N because of the three members that it might be a member of to the number of sensors to read them. With obstacle avoidance this would increase to three to the power three, giving 27 possibilities.

As obstacle avoidance was not implemented due to time constraints all other facets of the robot for both possible solutions can be considered to be successful. A lot was learnt during the process of creating a PID and fuzzy logic controller. In particular or more so than with PID there was a much more lengthly period of experimentation with different rules and shapes of membership operations for the input and output.

##### References

[1] H. Hadras, "CE801: Intelligent Systems and Robotics Lecture 4: Robot Control", University of Essex, 2018.

[2] H. Hagras, "LAB 3 – IMPLEMENTATION TUTORIAL - I CE801: INTELLIGENT SYSTEMS AND ROBOTICS Wall following behaviour using PID control", University of Essex, 2019.

[3] H. Hagras, "LAB 6 – IMPLEMENTATION TUTORIAL - III CE801: INTELLIGENT SYSTEMS AND ROBOTICS Wall following behaviour, Obstacle avoidance behaviour using Fuzzy Logic control", University of Essex, 2019.

[4] D. Samanta, "Defuzzification techniques", IIT Kharagpur, 2018.

[5] H. Hagras, "CE801: Intelligent Systems and Robotics Lecture 9: Fuzzy Logic Control", University of Essex, 2018

[6] H. Hagras, "CE801: Intelligent Systems and Robotics Lecture 6: Fuzzy Logic Control", 2018.

[7] H. Hagras, "CE801: Intelligent Systems and Robotics Lecture 7: Fuzzy Logic", University of Essex, 2018.