FUZZY LOGIC C++ WORKSHEET

OVERVIEW

OBJECTIVES:

1. Add Input
2. Output Variables
3. Add Rule Base
4. Extend fuzzy system

# BONUS OBJECTIVES

1. Exploratory work with the FuzzyLogicUnity example

PRELIMINARY

Primarily this tutorial work will require the use of a C++ IDE. Visual Studio is the recommended software to use although any C++ development environment should be sufficient.

There is a Visual Studio Project on MyLearningSpace that you can download and use for this week’s work.

Use the link in MyLearningSpace to download the “*Fuzzy Logic - C++”* file. Unzip and open the Visual Studio Project.

The bonus objectives will require Unity (**version 2018.4+**). Older versions may work but I cannot aid with import errors if you use an older version. Unity will require you to create a Unity Account as well.

If you need help setting up the software, we have a tool called AppsAnywhere that allows you access to some of the software we use. You can use this to install some of the software we use. <https://myapps.abertay.ac.uk/> GETTING STARTED

STARTING CODE

#include <iostream>

#include <fl/Headers.h>

using namespace fl;

int main()

{

Engine\* engine = new Engine();

std::string status;

if (not engine->isReady(&status))

std::cout << "Engine is not ready" << std::endl << status << std::endl;

delete engine;

return 0;

}

Update the main() function to match the above code.

# RUN THE PROGRAM

Make sure the program compiles and runs correctly. The project should be limited to 64-bit instructions (Win32 is no longer available on the project) and it should compile in both Debug and Release.

In the initial state, the fuzzy logic engine has not been configured, so will report ‘Engine is not ready’.

The *isReady()* function is very useful for making sure the engine is properly configured before we start processing data through the fuzzy logic system. Make sure to utilise this code when you run any project using this library. Example code online suggests using this check to throw exceptions if the engine finds a problem during the ready check. Feel free to change the code to do this if you feel it is appropriate.

The full library, with examples, is available [here.](http://www.fuzzylite.com/download/windows-64-bit/)

If the program complains of fuzzylite.dll or fuzzylite-debug.dll as missing when you try and run the program, search through the project files for those DLLs and put them into the output folder of your compiler. If you cannot find them, download the library from the link above and they will be available there.

# ADD INPUT VARIABLE

## CREATING AN INPUT VARIABLE

Creating an input variable is necessary for allowing us to feed data into the engine for processing. Each input variable can be heavily customised within the library to manage to it handles the data given to it.

The first step is to simply create the input variable in the code and add it to the engine:

InputVariable\* obstacle = new InputVariable;

engine->addInputVariable(obstacle);

This adds a variable called *obstacle* to the program and adds this to the engine. But we haven’t specified how it processes the inputs yet.

## NAME & RANGE

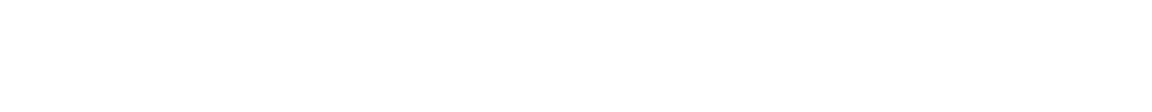
Two important pieces of information the input (and output) variable required is a name and a range which is will accept incoming data. The name is useful for parsing the ruleset later on, the range specifies the range of data the input will accept. This doesn’t mean it will not accept data outside those ranges, we can tell the variable to clamp incoming data to them, but it needs this specified so it can construct the graphs properly.

obstacle->setName("obstacle");

obstacle->setRange(0.000, 1.000);

The range can be anything you want. Imagine it as the X-axis on the graph that is used to calculate how incoming data is assigned to each term.



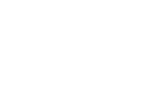


0.0



Range

1.0



1.0

## ADDING TERMS

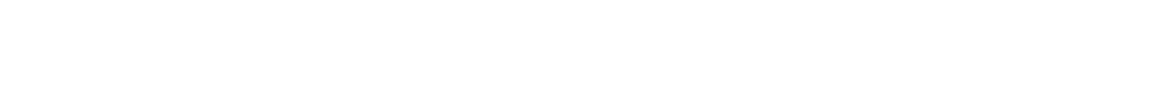
We need to add terms to the variable to state the different factors that the input can be part of. For example, if the variable was height, then we could use the terms short, tall, very tall, average, etc.

For this example we only care if the obstacle is to our left or our right. We can define this in the variable as follows:

obstacle->addTerm(new Triangle("left", 0, 0.05, 0.6));

obstacle->addTerm(new Triangle("right", 0.4, 0.95, 1.0));

This would create a graph like so, any input can be directly mapped to either the *left* or *right* terms.

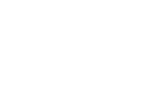


0.0

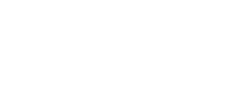
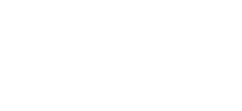
0.4

0.6

1.0



1.0



# OUTPUT VARIABLE

## CREATING OUTPUT VARIABLE

Creating the output variable is very similar to the input variables:

OutputVariable\* mSteer = new OutputVariable;

engine->addOutputVariable(mSteer);

mSteer->setName("mSteer");

mSteer->setRange(0.000, 1.000);

mSteer->setAggregation(new Maximum);

mSteer->setDefuzzifier(new Centroid(100));

mSteer->setDefaultValue(fl::nan);

mSteer->addTerm(new Triangle("left", 0, 0.3, 0.6));

mSteer->addTerm(new Triangle("right", 0.4, 0.7, 1.0));

Make sure there is a name setup for parsing of the rules and a range for the output. The aggregation method and defuzzifier method also have to be chosen and specified for the Output variables.

Adding terms in the same as the Input Variable, Ramp is the most straightforward but there are other shapes available.

# RULES

## RULES

The rules in FuzzyLite are collected together into Rule Blocks. You can use the *Rule::parse* method to write the rules out in English.

RuleBlock\* mamdani = new RuleBlock;

mamdani->setName("mamdani");

mamdani->setConjunction(fl::null);

mamdani->setDisjunction(fl::null);

mamdani->setImplication(new AlgebraicProduct);

mamdani->setActivation(new General);

mamdani->addRule(Rule::parse("if obstacle is left then mSteer is right", engine));

mamdani->addRule(Rule::parse("if obstacle is right then mSteer is left", engine));

engine->addRuleBlock(mamdani);

You can add multiple RuleBlocks, each one containing different methods of Conjunction, Disjunction, Implication, and Activation rules.

Conjunction operator is needed if you wish to AND rules together.

Disjunction operator is needed if you wish to OR rules together.

Implication operator is needed for state if X **then** Y

The activation is required as we need to tell the rule block how we want the rules to be activated. General activates all the rules in the order they are added to the RuleBlocks.

# INPUT

## ADDING INPUT TO THE SYSTEM

As the system above has a single input variable and a single output variable, we need to set the data being fed to the input variable and then run the engine to receive our output.

The following code gets data from our own console input, assigns it to the Input Variable (*obstacle*), tells the engine to process the data, then extracts the data from the output variable (*mStreer*).

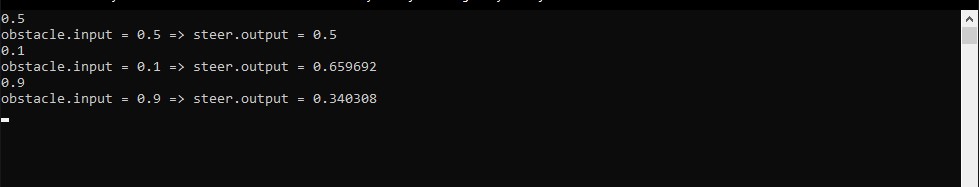
while (1)

|  |  |
| --- | --- |
| { |  |
|  | std::string input = ""; |
|  | std::cin >> input; |
|  | std::stringstream ss(input); |
|  | float number = 0.f; |
|  | ss >> number; |
|  | obstacle->setValue(number); |
|  | engine->process(); |
|  | std::cout << "obstacle.input = " << number << |
| } | " => steer.output = " << mSteer->getValue() << std::endl; |

The tutorial can also be found her[e https://www.fuzzylite.com/cpp/.](https://www.fuzzylite.com/cpp/)

## TEST THE SYSTEM

Compile and run the program. Test it out by entering a value between 0.0 and 1.0 and pressing the RETURN key. It should output something like the following:



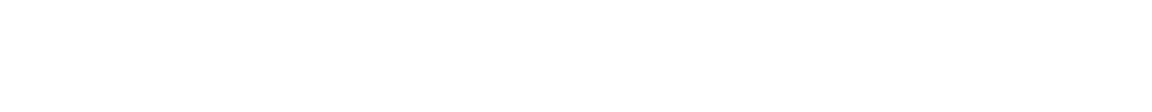
# EXERCISES

## ADD A NEW INPUT VARIABLE

Add a new input variable to account for (in a simulation) speed that we are currently moving to the left and the right.

It will be similar to the *obstacle* input variable but it will have three terms, moving\_left, moving\_right, and none. Make sure each term overlaps in the variable so that if plotted on a graph, it would look something similar to:



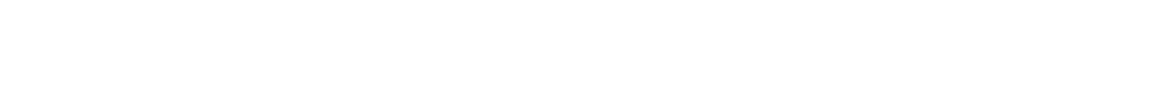


0.0

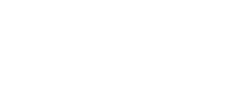
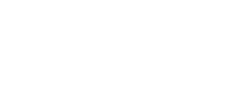
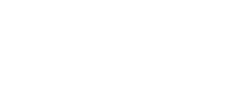


Range

1.0



1.0



Also make sure to be the Triangle instead of Ramp for the none term.

## ADD A CONJUCTION AND DISJUCTION OPERATOR TO THE RULEBLOCK

As we will be using the *and* and/or the *or* operator, make sure to set these two operators in the RuleBlock itself:

mamdani->setConjunction(new AlgebraicProduct);

mamdani->setDisjunction(new Maximum);

Feel free to explore how these work and experiment with how the different operators affect the final output of the system.

## ADDING NEW RULES

As we have a new set of inputs, we will need to add and edit the existing rules. For example, rather than saying:

"if obstacle is left then mSteer is right"

We may want something along the lines of:

"if obstacle is left or speed is left then mSteer is right"

You may also want to expand the output variables for different varieties of moving left and right, such as ‘slightLeft’ and ‘slightRight’ and add them to the input terms.

# ADDITIONAL EXERCISES

There are two additional projects on MyLearningSpace that can show you the use of Fuzzy Inference Systems.

One is a Unity project that makes use of a C# Fuzzy Inference Library. [https://github.com/davidgrupp/FuzzyLogic-Sharp.](https://github.com/davidgrupp/Fuzzy-Logic-Sharp)

There is a second worksheet called **Fuzzy Logic Unity Worksheet** that you can follow to explore the C# version of the code.

The other is a Fuzzy Inference System being used to control a car. Explore and experiment with the game.