Fuzzy Logic Implementation:
A Simple Water Level Control System Through Genetically Tweaked Fuzzy Rules

Inspired by the example of a complex problem solvable by fuzzy logic, I decided to model a fuzzy control system that aims to maintain a constant water level in a reservoir with variable and unpredictable inflow, by controlling the setting of an outflow valve fitted to the bottom of the reservoir. Since the system does not know exactly what the inflow is at any time, it would be quite difficult to create a traditional control system to solve this problem - but as we see, a fuzzy system can solve the problem very well using only a few simple rules and crucially, a genetic optimisation algorithm.

To improve the performance of my fuzzy system and solve the unavoidable problem of assigning values to vague membership functions, I've implemented a genetic algorithm which automatically adjusts the membership functions used by my fuzzy system to maximise performance.

Modelling the problem

The aim of my fuzzy control system is very simple, it attempts to maintain the water level in a reservoir at as close to 50 as possible, adjusting to counter the effects of a changing inflow. The water level is simulated in discrete steps, with the water inflow changing every five steps to some value in the range [0, 5], indicating that without any outflow enough water will enter to raise the level by that amount in a single step.

To counter this, the system is able to gradually open and close a valve at the bottom of the reservoir. The system is not able to instantaneously change the valve to any setting, by rather calculates a speed at which to open or close the valve each step, limited by both a maximum openness setting on the valve and a maximum speed at which it can open or close.

Outflow from this valve is calculated as a function of water level and valve cross sectional area, with the cross sectional area of the valve is calculated as a circle with radius equal to the valve setting.

Each step, the current level and the change in water level from the previous step are used to calculate the desired change in valve setting. The water level is then adjusted by the outflow and inflow, and the valve openness is adjusted before the next step begins.

The result of all this is a set of numbers, telling us the water level of the reservoir at each step of the simulation – through which we can easily calculate an average deviation from the desired level, and judge the performance of the system.

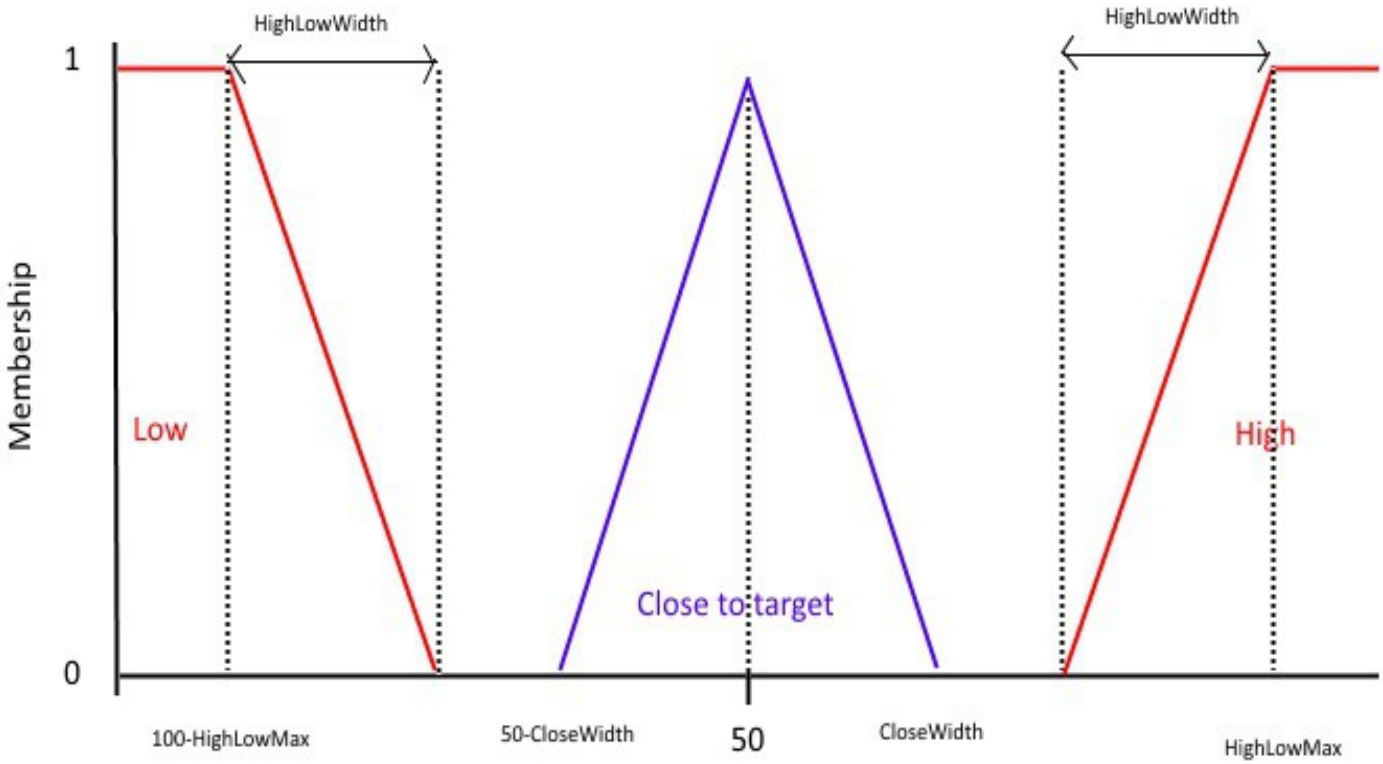
Fuzzy solution

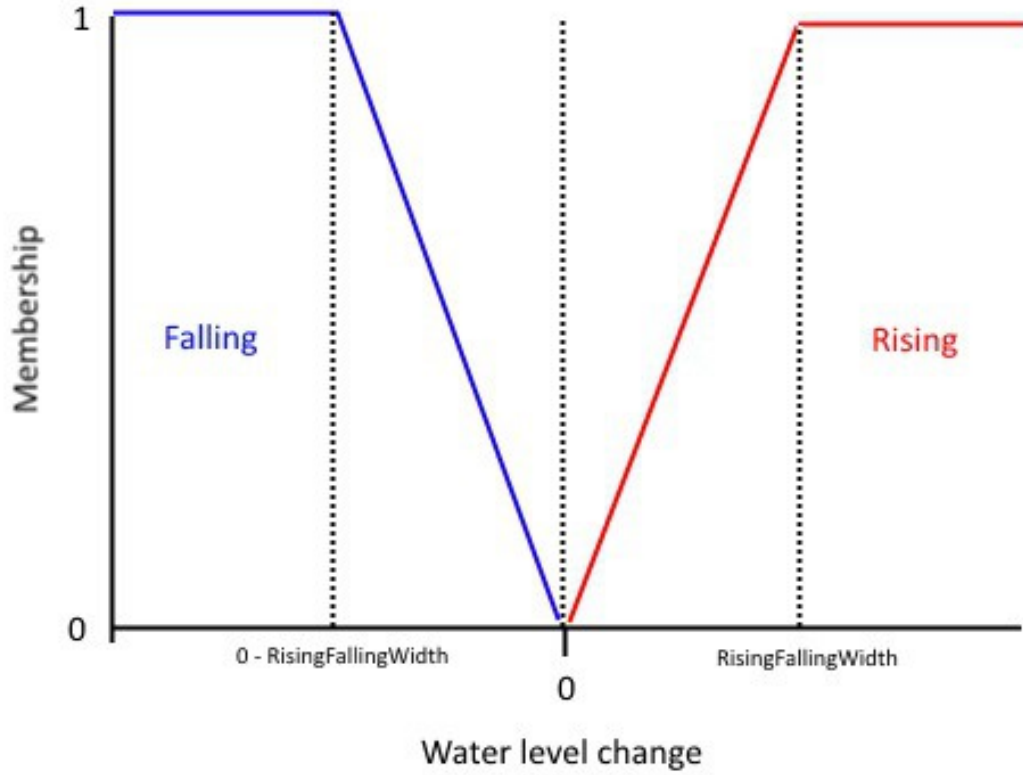
The control system that I implemented to solve this problem is based off of 5 simple rules:

-If water level is high, then open valve fast
-If water level is low, then close valve fast
-If water level is near target, then no change to valve

-If water is near to target and rising, then open valve slowly
-If water is near to target and falling, then close valve slowly

To implement these, I needed to define five fuzzy numbers for “high”, “low”, “near target”, “rising” & “falling”, as well as definitions for “fast” & “slow”, and a T-Norm to implement the AND function. For the fuzzy numbers, I created the following system – with the general shapes and symmetries predefined, but the exact values determined by variables to be tweaked at the next step:





I chose to make most of these numbers symmetrical with their opposite for both ease of computation, and because that seemed to be the most natural way to define numbers that describe opposite concepts.

The variables that determine the exact shape of these fuzzy numbers are given by a vector of doubles, this has the advantage of allowing me to tweak memberships easily with one line of code, rather than having to root through my code looking for every instance in which a value is used.

For the application of my fuzzy rules I used Sugeno-type fuzzy inference, so each rule had a firing strength calculated at each step, either through simple membership functions or through two membership values and the Mamdani AND operator (minimum of the two). “Open slowly”, “Open fast”, “Close slowly”, “Close fast” and “no change” were all then given a numerical value (through the same vector that defines membership functions) and a final output to the system was calculated based off of how strongly each rule fired. This output value was then used to determine exactly how much to adjust the valve at each stage of the simulation.

Tuning the system

To optimise my solution, I also implemented a genetic algorithm to adjust the membership functions of my fuzzy numbers and the output values of the Sugeno-type rule system. The genetic algorithm creates a population of somewhat randomised input vectors, then simulates the system these vectors produce over a set number of steps.

The next generation is then produced by the following steps:

-Assess the fitness of each solution in the previous generation by calculating the root mean squared deviation from the desired value. Then, take the inverse of this cubed – so that fitness grows exponentially as error decreases.

-Select members from the previous generation according to their fitnesses, so that members with a higher fitness are more likely to be selected for the next generation.

-Perform crossover on pairs of members, meaning that with some small probability entries in each vector are swapped with the corresponding one in the other vector.

-Perform mutation, meaning that with some small probability, entries in each vector are randomly adjusted by some small factor in a predefined range.

These steps are repeated for however many generations you want, however solutions seem to converge on a near optimal solution after only a few generations. Although the result is slightly different each time, after only a few generations this algorithm produced a solution with only a quarter of the average error of my initial solution.

Executing the main() function of GeneticTuner.java will cause it to output the results of this algorithm which you can see for yourself. Average error for the initial solution and final solution are both displayed, along with the actual vectors produced – which define the exact shape of the fuzzy membership functions for each solution.

Overall, I'm very pleased with the results. The fuzzy system correctly raises the water level to the desired value, and stabilises very well around this point. Although it performed somewhat well with my initial values for the membership functions, once the genetic algorithm optimised the solution the system performs excellently, consistently managing a very low average deviation from the desired value.

Implementation details

My implementation was coded entirely in Java, and is attached in both executable .jar form and as source code. There are three classes included, each performing a specific function:

ControlSystem.java

Accepts a vector of arguments in its constructor, and implements the fuzzy control sysyem defined by those arguments. Contains a number of methods, including many fuzzy membership functions and most importantly *generateValveSettingFromRules()* which calculates the exact valve adjustment that should be made from a given water level and change in water level. The arguments given correspond to properties like the width of the triangular numbers for the membership functions, and the locations of their maximum and minimums, among other things. A full list is available in ControlSystem.java.

Simulator.java

Handles the simulation of rising water level, inflow & outflow etc., taking an instance of *ControlSystem* as an argument, and running it through a specified number of steps, then finally outputting the performance of that system.

GeneticTuner.java

Implements the genetic tuning algorithm. Creates a number of vectors, then produces instances of *ControlSystem* from these, and simulates their performance through an instance of *Simulator,* applying the genetic algorithm described earlier. Contains the *main()* function and outputs the results of the program. This is where execution should be started.