Glen Browne

EE 476 - Professor Tsang

May 13, 2018

Fuzzy Logic Project

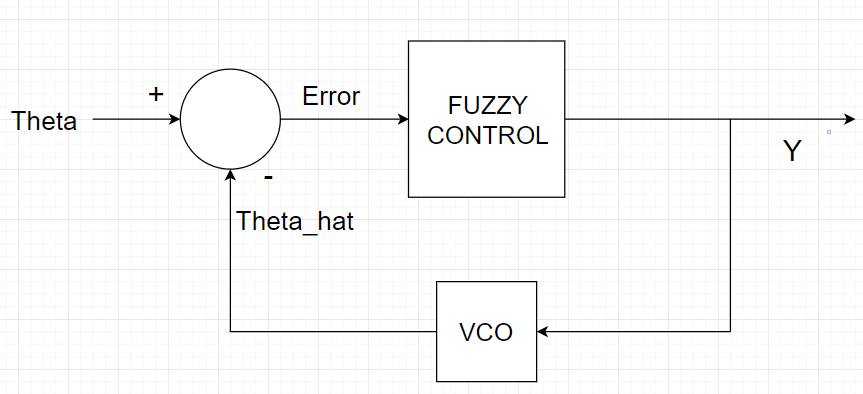
Simulating a Phase-Locked Loop with Fuzzy Logic



1. **The Problem: Creating a Phase-Locked Loop with Fuzzy Logic**

The problem to be solved by this project is to implement a phase-locked loop using fuzzy logic. A phase locked loop is a negative-feedback control system that adjusts its output to match the phase of the input. In this fuzzy-rule based system, varying levels of error, and vary rates of the change in error produce varying levels of correction to the output phase of the system. The objective of the system is to keep the output “in lock” with the input by synchronizing the phases of the input and output.

The fuzzy controller is only part of the PLL:



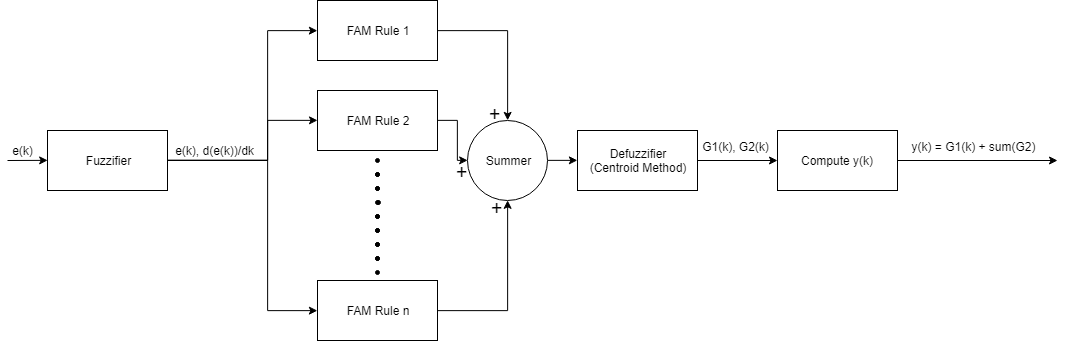
The entire system will take an input of a sine wave with angle theta and frequency fc. The fuzzy controller will take as inputs the error of the angle (theta - theta\_hat) and the rate of change of the error. The fuzzy controller will use these inputs to determine how to correct the output to produce an output that is synchronized with the input.

**b) Fuzzy Logic Control Method**

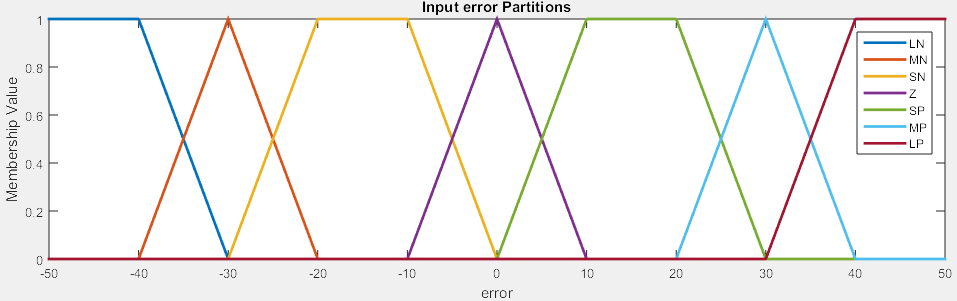
The control method I chose was quite simple. I simulated the behavior of a PI controller with fuzzy logic. The two input variables, error, and derivative of error, resulted in two output variables, G1 and G2. G1 is analogous to the proportional control of a PI controller, and G2 is analogous to the integral control of a PI controller. The output Y from the fuzzy controller is y(k) = G1(k) + sum(G2), where sum(G2) is the summation of all the past values and present value of G2. I considered using G1 and G2 as constants in the following equation: y = G(k)\*error + G2(k)\*sum(error), but this method proved redundant and did not work well. The redundancy killed the efficacy of this model because G1 and G2 were already determined by fuzzy logic, and they have the error already factored into their value. Using the error twice (once to determine G1 and and G2, and once to multiply G1 and G2 by the error and the summation of error) results in an unstable system. A much better way, I found was to use G1 as the proportional term, and the summation of G2 as the integral term, without multiplying them by the error, since they were already determined with FAM rules based on the error. G1 is proportional because it uses fuzzy inference rules to determine its value: it is small and negative if the output needs to adjust that way, large and positive if the output needs to adjust that way, and so forth. Sum(G2) is an integral term because it keeps track of the total error over time, and adjusts the output properly using fuzzy inference rules the same way as does G1.

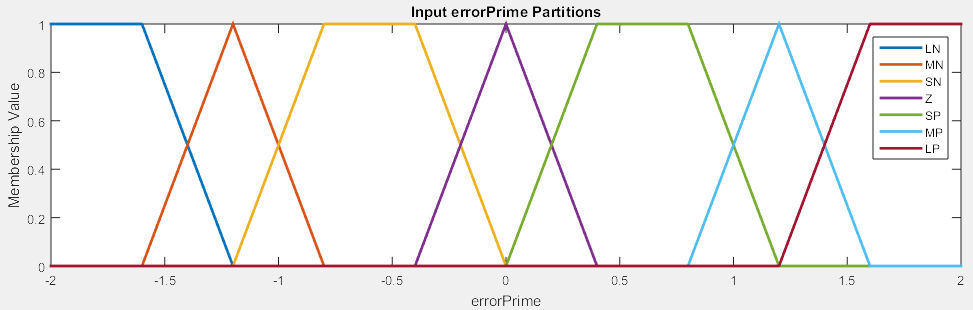
Either term alone will not be sufficient to lock the output to the input. Using G1 alone would produce a steady-state offset error, and using sum(G2) alone would result in a forever-oscillating system. Using both terms in tandem creates a controller that can reduce the error to zero and therefore can lock the output to the input.

Fuzzy Controller:

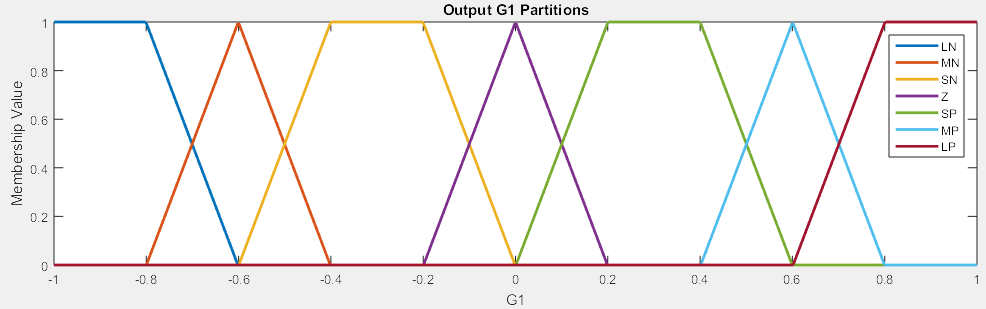


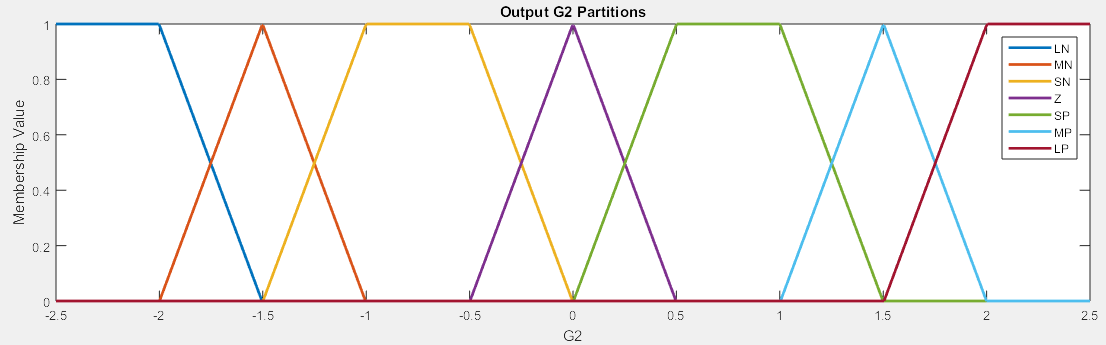
The two inputs, error, and derivative of error, have the following membership functions:





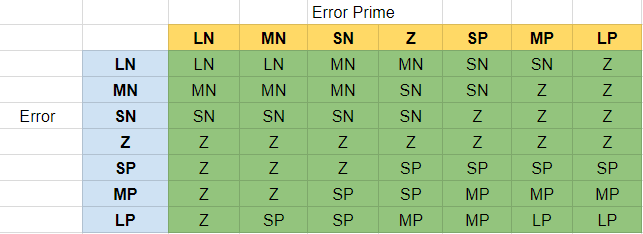
The two outputs, G1 and G2, have the following membership functions:



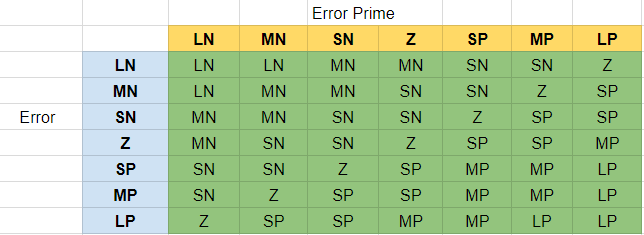


The universes of discourse for G1 and G2 were carefully chosen. The maximum magnitude of 1 for G1 prevents the system from getting into a situation of oscillation, which would happen with a larger value for G1. A smaller universe of G1 than -1 to 1 would cause a slow system response. The universe of discourse for G2, -2.5 to 2.5, is the maximum domain G2 can have without causing a significant steady-state oscillation. At -3 to 3, the steady state error is about 0.0001. At -2.5 to 2.5, the steady-state error is about 1 x 10^-13 at maximum, which is about 1 billion times less error, and is essentially zero. The universe of -2.5 to 2.5 takes a few more iterations to get the system to its minimum error, but I think the extra iterations are worthwhile since the error drops so dramatically.

**FAM Tables:**



G1 FAM Table



G2 FAM Table

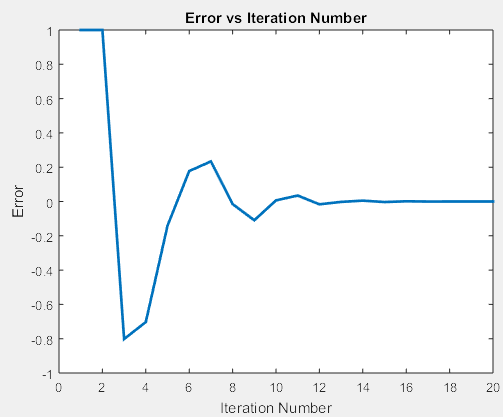
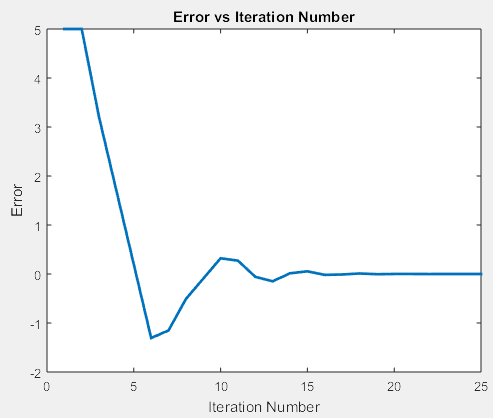
These FAM rules optimize the performance of the system. I considered just using input 1 (error) to determine the FAM rules, and just matching the output with that input, but when I wrote FAM rules that considered both inputs, the response was much quicker, and it took much fewer iterations for the error to drop to zero.

I based G1’s FAM rule table on the principle that if the error is zero or going in the right direction, I make no correction, but if it is nonzero and going in either the wrong direction or staying constant, then I make a correction proportional to the current error and the rate of change of the error.

I based G2’s FAM rules on the principle of smoothing the rate of change of the error so convergence happens faster. If the error is large, but quickly approaching zero, I don’t make any change to the sum of G2, since its sum will stay the same the next iteration. I wait for the error to become medium or small or zero before I give G2 a value to slow down its approach toward zero. If the error is slowly approaching zero and far away from zero, I speed it up by making G2 larger in magnitude. Only when both error and the rate of change of error are zero does the system stop changing the sum of G2 for good.

**c) Performance**

**Test 1: fc = 0, theta = 1 Test 2: fc = 0, theta = 5**

** **

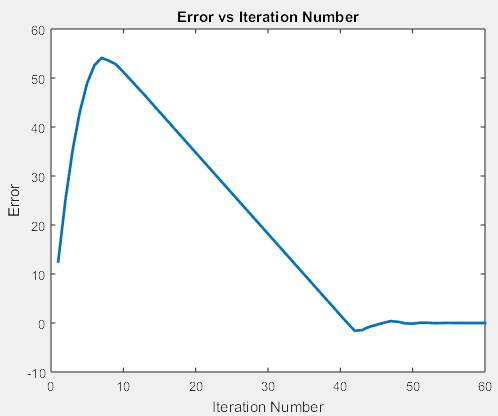
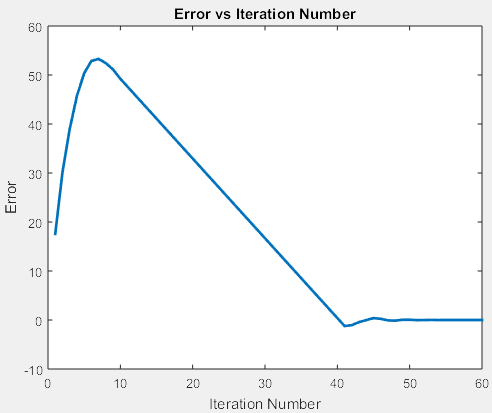
The system takes **10 iterations** to The system takes **14 iterations** to permanently permanently drop below 0.1 error: drop below 0.1 error:

** **

In steady state, the error is 0 (55 iterations): In steady state, the error is 0 (55 iterations)

** **

**Test 3: fc = 2, theta = 0 Test 4: fc = 2, theta = 5**

** **

The system takes **51 iterations** to The system takes **49 iterations** to permanently

permanently drop below 0.1 error: drop below 0.1 error:

** **

The system appears never to reach a truly steady In steady state, the system oscillates with miniscule

state, but instead oscillates with miniscule errors: error:

** **

**d) Sample Test and Program Output**

To show how the simulation operates, I will run it at fc = 2, theta = 5, and show some intermediate outputs:

Input 1: error Input 2: derivative of error

Crisp k Crisp k  

Fuzzy k Fuzzy k

LN:  LN: 

MN: MN:

SN: SN:

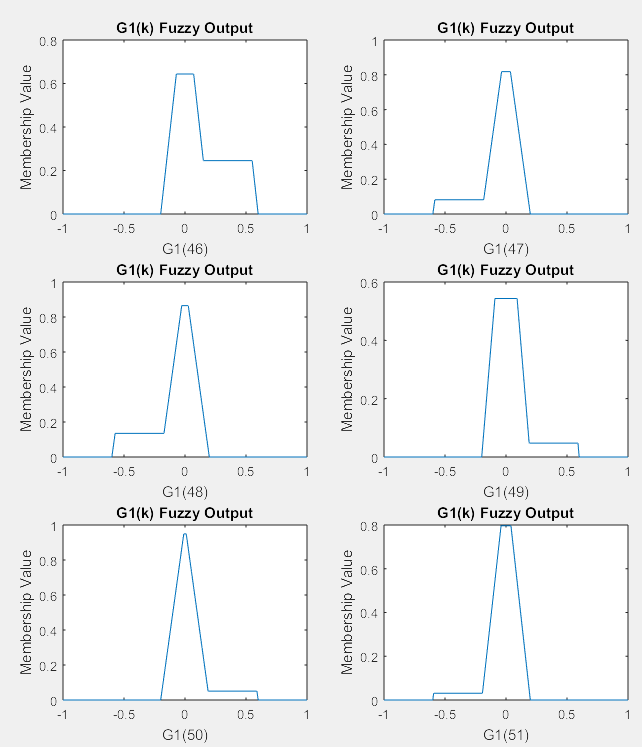
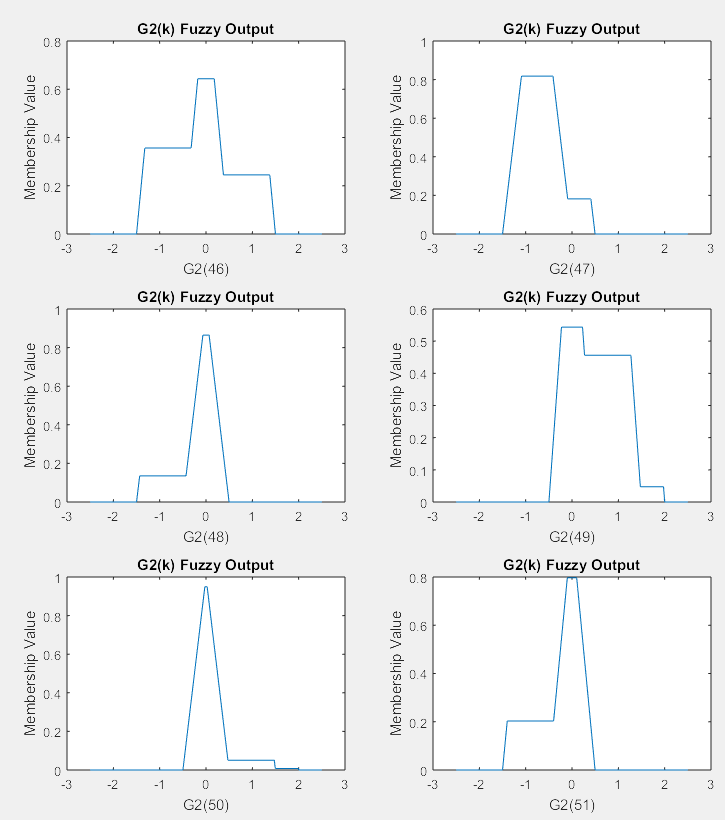
Z:  Z: 

SP: SP:

MP: MP:

LP: LP:

Fuzzy G1: (k= 46 - 51) Fuzzy G2 (k= 46 - 51)

** **

The shapes of the fuzzy outputs are due to the membership functions and the FAM rules. These outputs are defuzzified using the centroid method.

Defuzzified values:

G1(46) = 0.1353 G2(46) = 0.2220

G1(47) = -0.0569 G2(47) = -0.0817

G1(48) = -0.0836 G2(48) = -0.2091

G1(49) = 0.0424 G2(49) = 0.0535

G1(50) = 0.0364 G2(50) = 0.0903

G1(51) = -0.0241 G2(51) = -0.0449

Other Program Calculations: y(k) = G1(k) + sum(G2(1:k))

sum(G2(1:46)) = 12.7582 y(46) = 12.8935

sum(G2(1:47)) = 12.6766 y(47) = 12.6197

sum(G2(1:48)) = 12.4675 y(48) = 12.3838

sum(G2(1:49)) = 12.5210 y(49) = 12.5634

sum(G2(1:50)) = 12.6113 y(50) = 12.6476

sum(G2(1:51)) = 12.5663 y(51) = 12.5422

phaseIn(k) - phaseIn(k-1) = 12.5664

VCO Output:

phaseIn(k) = 2\*pi\*fc\*k + theta phaseOut(k+1) = phaseOut(k) + y(k)

phaseIn(46) = 2\*pi\*2\*46 + 5 = 583.0530

phaseIn(47) = 2\*pi\*2\*47 + 5 = 595.6194 phaseOut(47) = phaseOut(46) + y(46) = 595.7013

phaseIn(48) = 2\*pi\*2\*48 + 5 = 608.1858 phaseOut(48) = phaseOut(47) + y(47) = 608.3209

phaseIn(49) = 2\*pi\*2\*49 + 5 = 620.7522 phaseOut(49) = phaseOut(48) + y(48) = 620.7048

phaseIn(50) = 2\*pi\*2\*50 + 5 = 633.3185 phaseOut(50) = phaseOut(49) + y(49) = 633.2681

phaseIn(51) = 2\*pi\*2\*51 + 5 = 645.8849 phaseOut(51) = phaseOut(50) + y(50) = 645.9158

As demonstrated, the fuzzy logic-based PLL successfully locks the output phase to the input phase.

**e) Possible Improvements**

The fuzzy logic-based PLL may be improved by fine tuning. The number of membership functions can be increased, the universes of discourse can be changed for each function, and the FAM rules may be tuned via trial and error. I took careful consideration when creating my fam rules, so I think the program is tuned quite well as is, but there may be a better implementation yet. The system I created works very well, as it can successfully lock to any input, although high-frequency inputs take longer to lock to than low-frequency inputs. There may be some way of minimizing the number of iterations needed to lock on to high-frequency inputs, although this may result in more steady state error since that would require a larger universe of discourse.

**f) Program Listing**

MATLAB Code: