Ben Church - 10006197

ben.church@queensu.ca

Fuzzy control of rod balancing robot

CISC 870 Fuzzy Logic Implementation

# Overview

The program contained in this deliverable simulates a mobile robot which attempts to keep a freely rotating rod balanced on top of itself by moving back and forth in one dimension. The fuzzy logic is implemented in FuzzyLogic.py ‘from scratch’. The file amounts to a simple fuzzy logic library which demonstrates implementation of the subject matter on a mathematical level, if at the expense of the extensive functionality offered by an open-source library. The implemented fuzzy logic is described in greater detail in the Fuzzy Logic section of this document. While fuzzy logic is the intended focus of this project, BalanceBot.py contains the most lines of code as it implements the desired behavior using tools from FuzzyLogic.py. The simulated robot behavior from BalanceBot.py is described in the Simulated Robot section.

The simulation is iterated from Main.py, which is also used to call functions which produce various plots of interest. Fuzzy membership sets, and a visualization of the robot can be printed to png images at each iteration of the simulation. The resulting image sequence can be scrolled through for the effect of animation, or loaded into ImageJ[[1]](#footnote-1) as an image sequence and saved as a .gif for actual animation. The histories of physical quantities through time can be plotted at the end of the simulation. Plotting utilities and general program use are described in the Program Usage section of this document. All plotting was done with the matplotlib[[2]](#footnote-2) open-source plotting library for Python. Numpy[[3]](#footnote-3) was also used throughout the program.

# Fuzzy Logic

The FuzzyLogic class in FuzzyLogic.py contains one type of fuzzy membership set, a trapezoid defined by four geometric parameters. These parameters are the locations of the so-called: “Left foot”, “Left shoulder”, “Right shoulder”, and “Right foot”. These parameters are represented as 2D points, the first dimension is the continuous variable domain over which the sets are defined, and the second is the membership value of the sets though the domain. The parameters are illustrated in Figure 1 on an example plot of the tilt value of the rod at one moment in some simulation. Implemented this way, trapezoids reduce to triangles when Left shoulder = Right shoulder. Initializing a fuzzy set with these parameters saves them for later membership value computation and plotting.

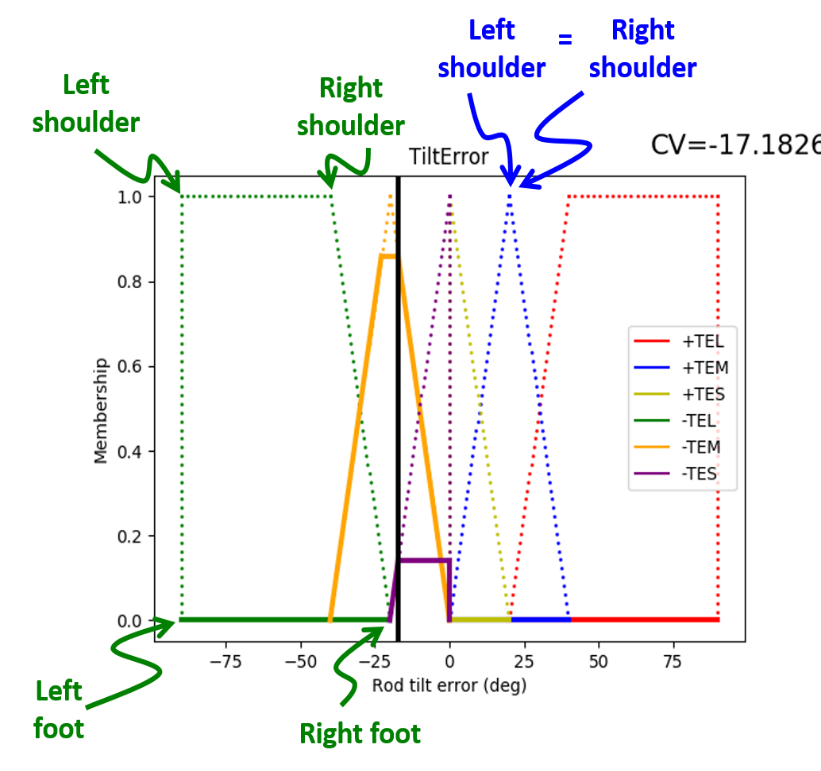


Figure : Sample fuzzy membership plot illustrating the geometric parameters used to construct fuzzy set membership functions.

When a set’s ComputeMembership() function is called with a crisp value argument, it returns a updates the set’s current membership value by considering the geometric context of the set’s feet and shoulders. The ComputeMembership() function calls the set’s (class’) ComputeArea() function, which in turn calls the ComputeCenterOfArea() function. These update the geometric representation of the fuzzy sets given their current membership values. The set’s area, used in defuzzification and plotting, is represented and stored as the sum of “Left triangle”, “Center rectangle”, and “Right triangle” areas, illustrated in Figure 2. The calculation of the set’s center of area is straightforward once the total area is computed.

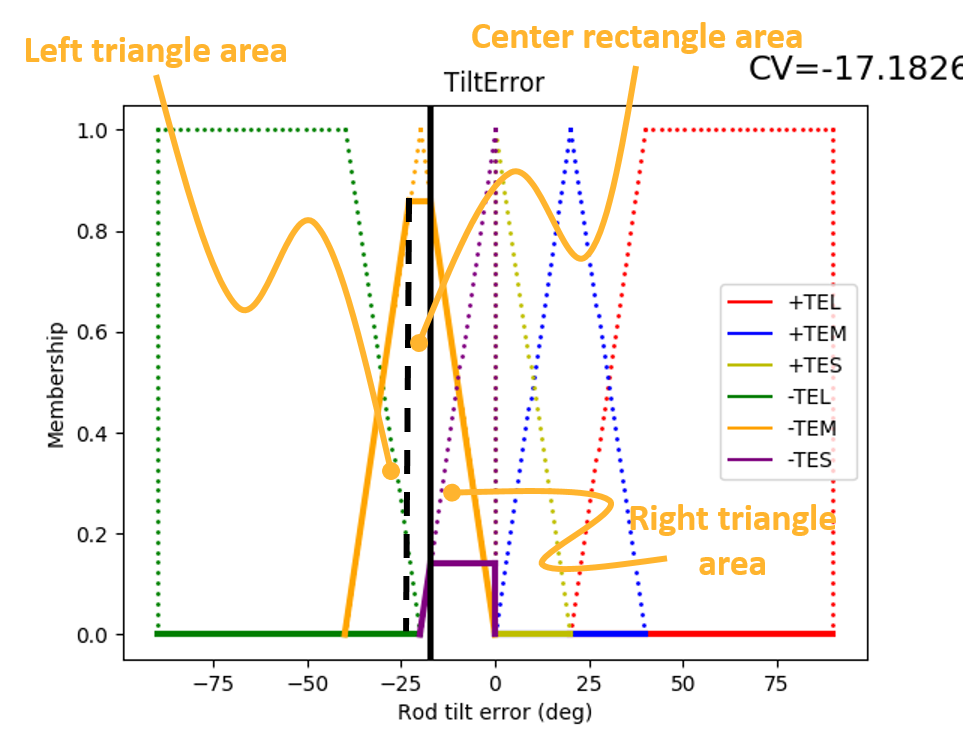


Figure : Sample fuzzy membership plot illustrating the geometric areas combined to compute the total set area.

FuzzyLogic.py also contains conjunction, disjunction, and implication operators used to implement fuzzy rules in BalanceBot.py (and plotting tools described later). The conjunction and disjunction operators are recursively defined on lists of fuzzy set membership values, easing the use of multiple rules to govern a single output fuzzy set which use multiple predicates in their antecedents. The implication operator function is not used in the control of the robot. When fuzzy rules were used to determine the membership of the output set, the implication rules themselves were considered to have truth values of 1, which was used with the antecedent truth value to infer the truth of the consequent (motor output). This occurs in FuzzyLogic.py’s ImplicationAntecedant2Consequent() function.

# Simulated Robot

The BalanceBot class is initialized with many attributes. There are paths to the directories where plots and sequences of plots are saved. The sequences being plots of, for example, membership functions or the robot visualization at each iteration of the simulation. Environment and robot physical properties, and the variables used to store their histories, are instantiated. The user defines the robot’s mass, maximum motor force, rod length, and rod mass, as well as the maximum wind speed. Fuzzy sets are also instantiated upon initialization. Fuzzy sets for, both positive and negative, rod tilt angle, wind speed (currently not used in any rules), rod tilt integral, and motor response, are (tediously) defined.

The purpose of the tilt angle sets is so the robot can respond as the rod has begun to fall, by chasing it, standing it back up. This alone is insufficient to flip the rod past 0 degrees, back in the other direction; the fuzzy sets compelling the robot to chase the tilt lose membership as the tilt diminishes. Therefore the integral of the rod’s tilt with respect to time is computed throughout the simulation and used as a crisp input to the tilt integral fuzzy sets membership functions. The integral allows the robot to provide an increasing response the longer the rod remains tilted to one side, and drive past the tipping point, (potentially) achieving balance. The motor response fuzzy sets define the crisp force value, output by the robot’s motor, in response to the memberships of the various sets.

Once the robot is instantiated, the simulation is iterated by calling the IterateSimulation() function. First, this updates the mechanical state of the robot given the previous values for all the relevant physical variables by calling the UpdateMechanicalState() function. Next, the fuzzy set memberships, areas, etc… are updated with a call to the UpdateFuzzySets() function. Finally, the motor response sets are defuzzified to decide value to output to the motor in the UpdateControls() function.

Updating the robot’s mechanical state consists of 1D linear kinematics and dynamics for robot, and 1D angular kinematics and dynamics for the rod. The robot kinematics and dynamics are simply the acceleration computed from motor response force and mass, and the resulting position and velocity, which are currently used only for plotting. The rod kinematics and dynamics are the angular acceleration of the rod computed from the total torque resulting from gravity, acceleration, and wind (not currently used in rules) given the rod’s moment of inertia. The resulting tilt, tilt integral, and angular velocity (the former two of which are used in fuzzy set functions) are then computed.

UpdateFuzzySets() takes an sNorm, a tNorm, and an ImplicationMethod as strings. The implemented varieties can be found in their corresponding functions in FuzzyLogic.py. First, the tilt angle, tilt integral, and wind speed sets are updated. They are represented as truncated forms, rather than scaled ones, upon consideration of their membership values. Then the motor response sets’ membership values are determined based on the following rules:

* PosTiltErrorSmall 🡪 PosMotorResponseLow
* PosTiltErrorMed 🡪 PosMotorResponseMed
* NegTiltIntegralLow 🡪 PosMotorResponseMed
* NegTiltIntegralMed 🡪 PosResponseMedHigh
* PosTiltErrorLarge 🡪 PosMotorResponseHigh
* NegTiltIntegralHigh 🡪 PosMotorResponseHigh

Where the ‘Pos’ and ‘Neg’ prefixes denote the physical direction which pertains to the rule. The ‘Neg’ side has symmetric motor response rules. Multiple rules governing one motor response set are combined using the fuzzy disjunction, however, no rules using the conjunction are currently implemented.

Finally, the total masses of the positive and negative response rules are computed and compared to determine whether to resolve to move forwards or backwards in UpdateControls(). Positive and negative are not all combined. If they were, the tilt angle and tilt integral could cancel out, immobilizing the robot. The rules for the chosen direction of motion are defuzzified using the center of mass method, producing a single, crisp motor response force value. Having computed the robot’s response, the simulation is ready to iterate again.

# Program Usage

The program is run, start to finish, with the console command (and with matplotlib and Numpy installed): python Main.py. The existing directory structure is necessary for successful plot capture. There are four groups of user-defined parameters which can be used to modify the output of the program. They are the fuzzy operations for robot control rule evaluation, the visualization reference frame (addressed in the Implementation Limitations section), and both fuzzy set names and physical quantities for plotting. The allowed values for these parameters are listed at the top of Main.py. The SimulationIterations variable in Main.py can be used to change how many times the simulation is iterated. The time-resolution of the simulation is defined near the top of the BalanceBot’s initialization function. Main.py currently implements all four program functions: iterating the simulation, drawing the sequence of robot visualizations to files, drawing the sequences of fuzzy membership functions to files, and plotting physical quantity histories. It should serve as an illustrative template should you wish to try different inputs. Any plotted figures must be deleted between program runs for overwriting.

# Implementation Limitations

While the robot control scheme can maintain the rod’s balance for several oscillations, the system ultimately remains unstable, usually trying several attempts to observe any oscillation. Stability might be achieved by tuning the geometric parameters, or introducing a better combination of fuzzy rules. Additional fuzzy rule sets could be used as well. Integral control, despite straying away from the linguistic spirit of fuzzy logic into more abstract control theory, was required to have the rod oscillate about equilibrium at least once. However proportional-integral control necessitates a balance between the two, the proportions of which are difficult to express with a few fuzzy rules rather than two continuous gain variables.

Additional position control keeping the robot at the origin of the arena would be useful to keep the robot in the plot window. Currently, no control rules take into account the position of the robot relative to the origin. Such rules could be keep the robot near the origin. The robot’s tendency to wander away interfered with plotting the animation at a reasonable scale when the view’s reference frame was at the arena’s origin. Given the difficulty in achieving the primary function of rod balancing with a combination of fuzzy rules, the option to plot with a view fixed on the robot was implemented. This option is used as the default, defined in Main.py’s call to Bot.DrawBotToFile().

The robot was meant to use the conjunction operator to combine measurements like the tilt, and wind speed, or tilt and tilt integral. However the inability to achieve stability with elementary rules indicated that more sophisticated functionality would not solve the problem. More sophisticated rules are more difficult to justify on a physical basis, as well.

While most of the code is in BalanceBot.py, the detail of the simulation is rudimentary, being implemented ‘from scratch’. The physics simulation does not take into account system properties normally considered in control theory. The motor force is simply set to the desired value. Real motors have inertia, causing a transient response in its actual force output. This can be modelled with transfer equations based on 2nd order differential equations relating displacement (error), and first two time-derivatives, to force, through a restorative force constant (a spring constant, for example), a damping force constant, and the system inertia. Although such sub-system transient response must be taken into account in actual implementation tuning, I chose devote time and effort to other aspects of the project, as the effects from this can be ignored to illustrate proof of concept. Furthermore,

# Appendix A – Main.py

1. <https://imagej.nih.gov/ij/> [↑](#footnote-ref-1)
2. <http://matplotlib.org/> [↑](#footnote-ref-2)
3. <http://www.numpy.org/> [↑](#footnote-ref-3)