Author: Anthony Iran Coipel

General

In a control system course, it is quite common to utilize a water tank height controller problem to discuss some facet of control system theory. This project is essentially going to be the real-life implementation of said textbook problems. It will start with fundamentals such as creating a physical structure, a way of sensing the water height, and a way to manipulate the water flow rate into the tank. It will then move into applying various control system theory techniques to experiment with various ways to create controllers and how to optimize them. As of the time of writing, the fundamentals of the project are complete and are described in this report. In addition, some control theory is used in the form of a proportional-integral controller, however, it is not optimized and instead “hand” tuned.

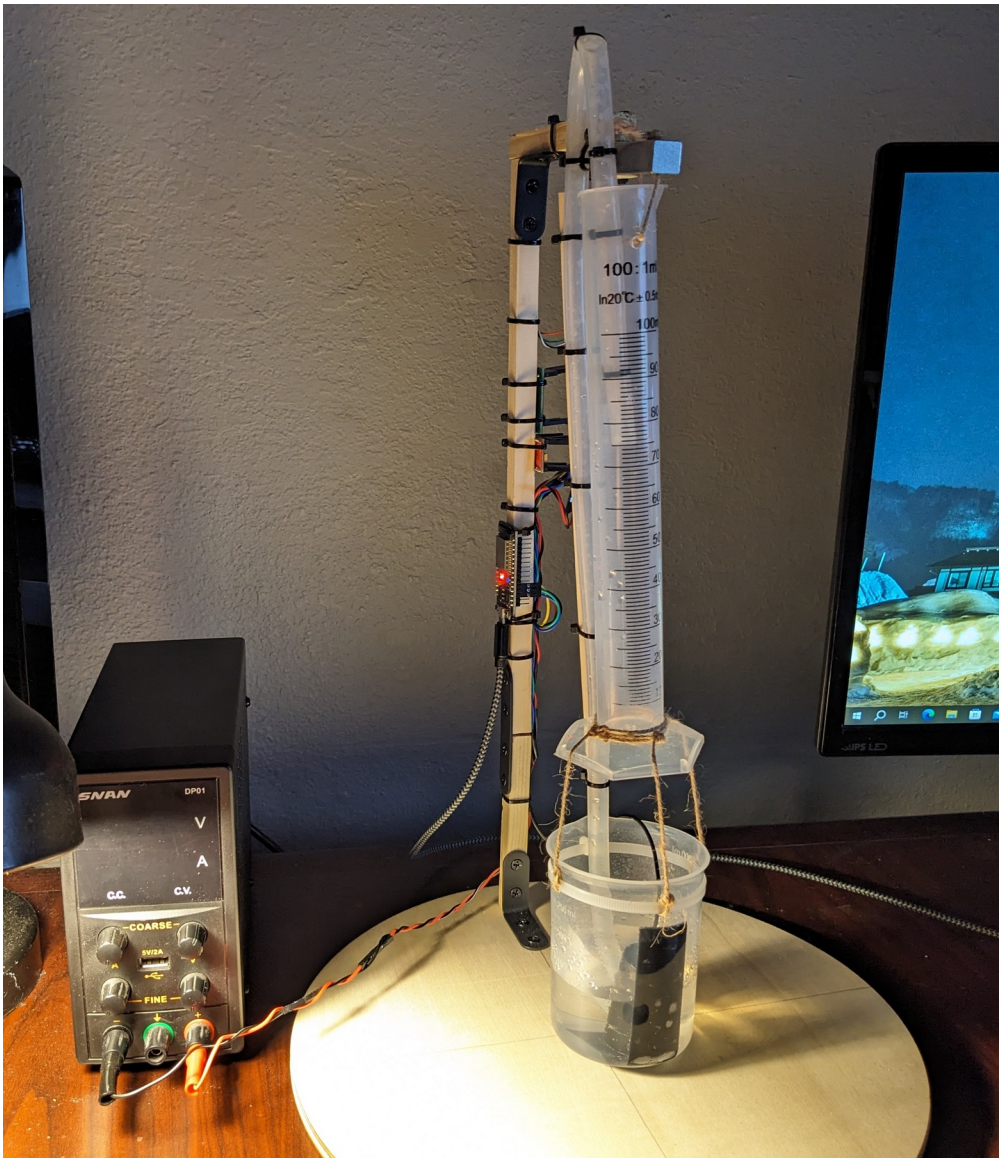

Fig. 1, Structure Full View

Hardware

Sensor: Strain Gauge

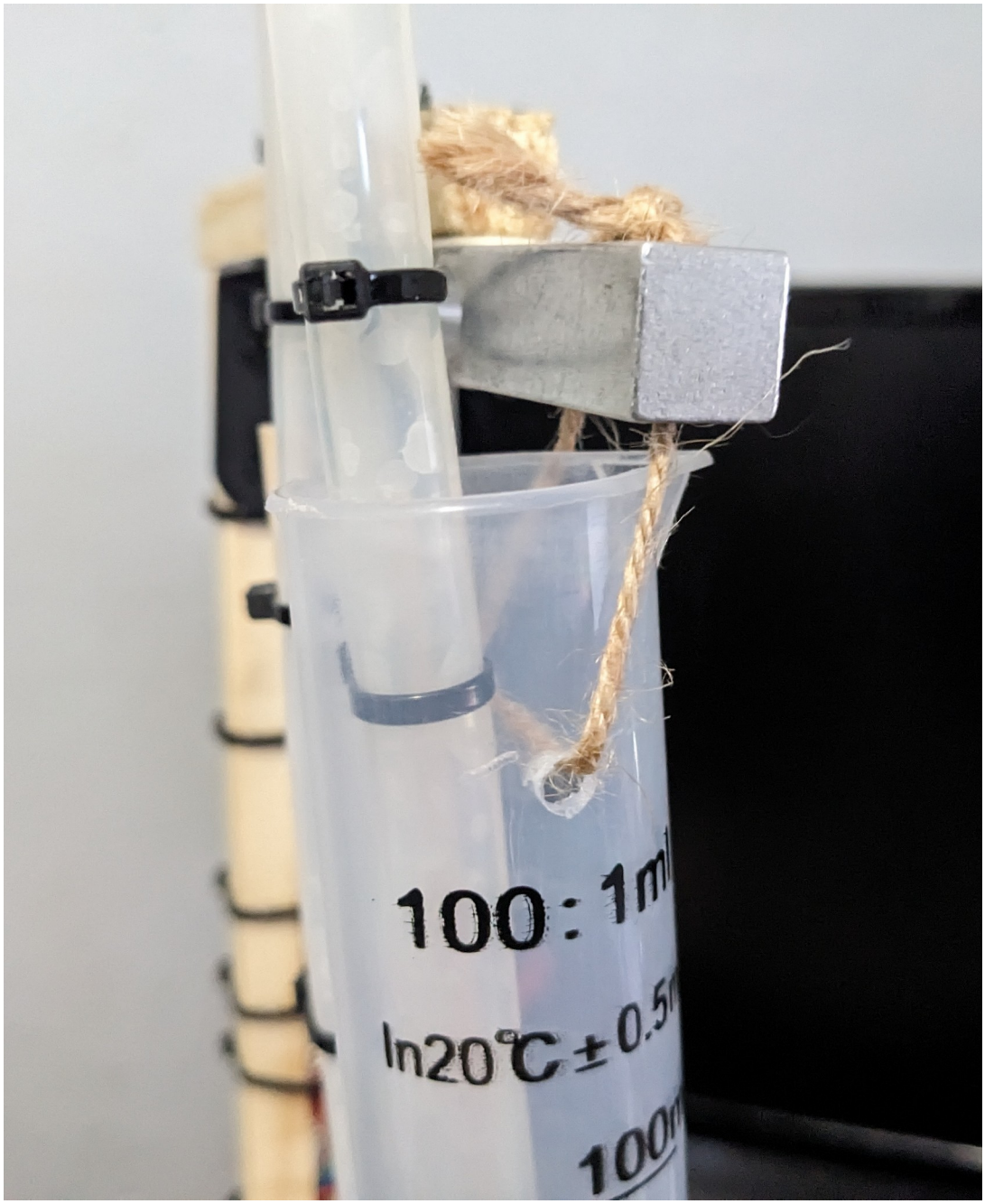

Fig. 2, Liquid Tank (Beaker) Tied to Strain Gauge

There exists a variety of ways to measure the height of water inside a tank. However, some of these sensors have weaknesses such as being too simple such as a bobbing sensor that can only when one water level is reached. More can be placed for more sample spots however this is cumbersome and still has poor resolution for my purposes. Ultrasonic sensors can get covered in a mist of water which requires maintenance in cleaning. Admittedly, in a large tank and with dedicated maintenance

ultrasonic sensors can be quite practical. As for those voltage dip-sticks for measuring water height, they generally are quite short and when they are longer they are incredibly expensive. Given how little water will actually be in the tank, it would be most practical to instead measure the weight of the tank of water and deduce the volume and height thereafter. This is where the use of a strain gauge comes in. A strain gauge reports weight in terms of a unit-less strain value that with empirical data can be converted into determining the weight of objects that are straining the sensing material. If you want to be scientific about it you can convert strain values to weight; remove the tanks weight from the whole weight calculation to get the weight of the water; using water density to deduce water volume; using beakers diameter obtain cross-sectional area to deduce height from volume. However, I opted to do all of this in one large, empirical, step converting strain values into beaker height directly instead of doing multi-step experiments and measurements. I recommend designing a control system that can reach a setpoint strain first and then perform experiments to determine what a given strain value means in beaker height. For instance, your control system has stabilized around a value of 1.23 strain unit-less value and with a visual inspection you observe a mL value of say 10.47 mL. This means a direct strain to mL conversion factor is 10.47 mL/1.23 strain = 8.51 mL/strain.

Complication

While using the strain gauge is great thus far, there does exist one critical issue that must be addressed. Strain gauges are vulnerable to electromagnetic interference (EMI). These devices rely on making voltage difference measurements as small as mV or μV in a Wheatstone bridge. EMI can throw off these measurements to be erroneous. The water pump used in this system has enough power running through it as well as being pulse width modulated (PWM) that generates enough noise to interfere with strain gauge operation. There are many ways to combat and overcome this EMI issue. I opted for the most straightforward and cheapest option which is simply to automatically shut off the water pump for a sufficiently small amount of time prior to taking a strain sample. This results in clean data. As a consequence, however, the sampling rate must be reduced so that there is enough time to perform a clean sample and for the water pump to run long enough after to manipulate the water height. Luckily, incredibly high sampling rate is not necessary since the system itself not incredibly fast in the first place. For instance, if a system takes 20~30 seconds to reach steady state increasing the sampling period (reducing the sampling rate) from 0.1 seconds (10 Hz) to 2 seconds (0.5 Hz) isn’t going to cause enough data loss to make designing a stable discrete controller impossible.

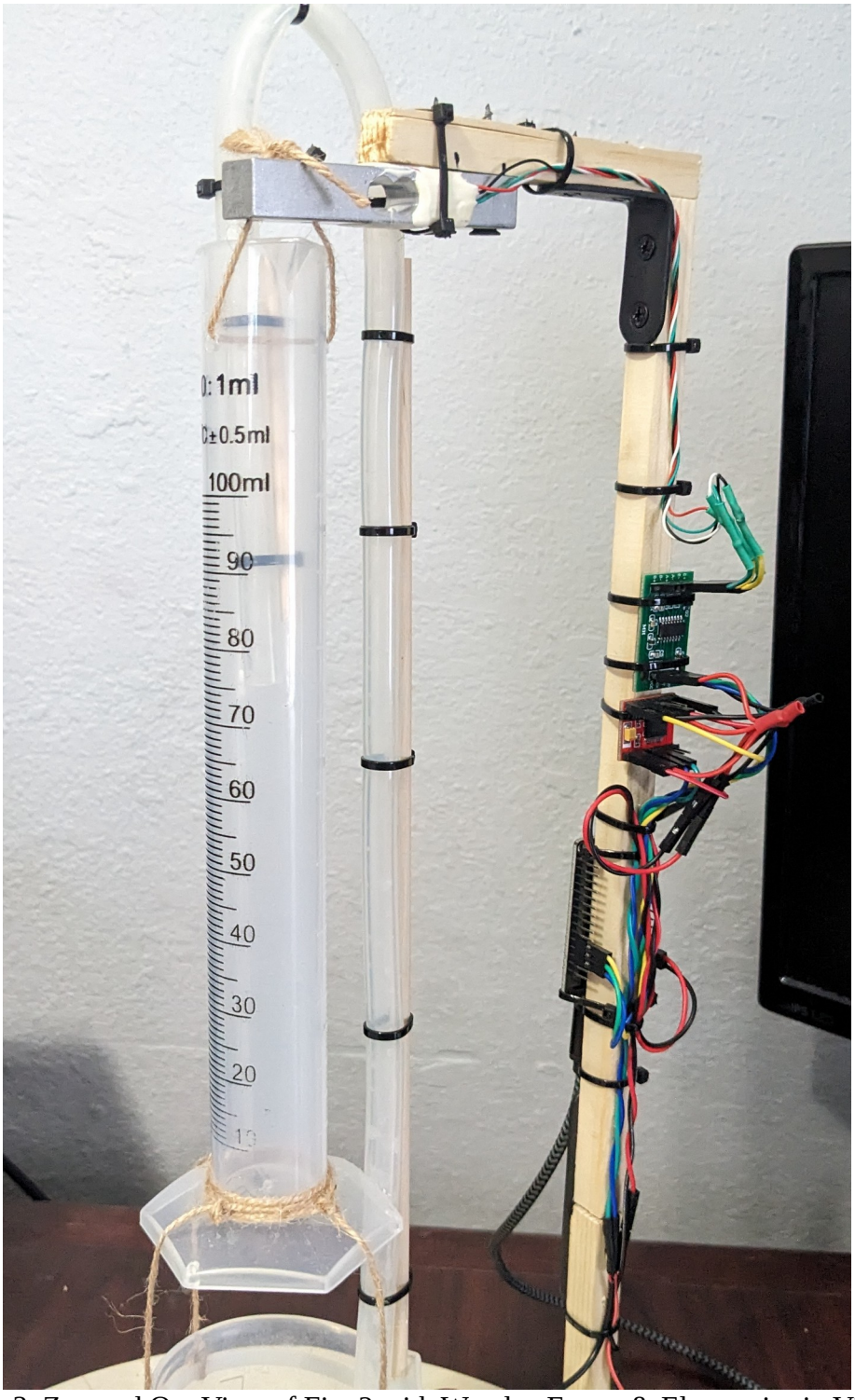

Fig. 3, Zoomed Out View of Fig. 2 with Wooden Frame & Electronics in View

Programming wise, the raw information of the strain gauge is not immediately useful and is instead connected to a circuit component called the HX711 combined with an Arduino library to conveniently return numerical data that can be used in programs. My Arduino library of choice is by

Bogde that can be found on GitHub here: https://github.com/bogde/HX711. However, I recommend installing the library through the Arduino IDE.

Complication

It has been reported that sometimes the HX711 board has a known defect. Luckily it can can be remedied by soldiering the GND and E- terminals together as shown. I did it as a precaution. Disregard the shoddy soldering.

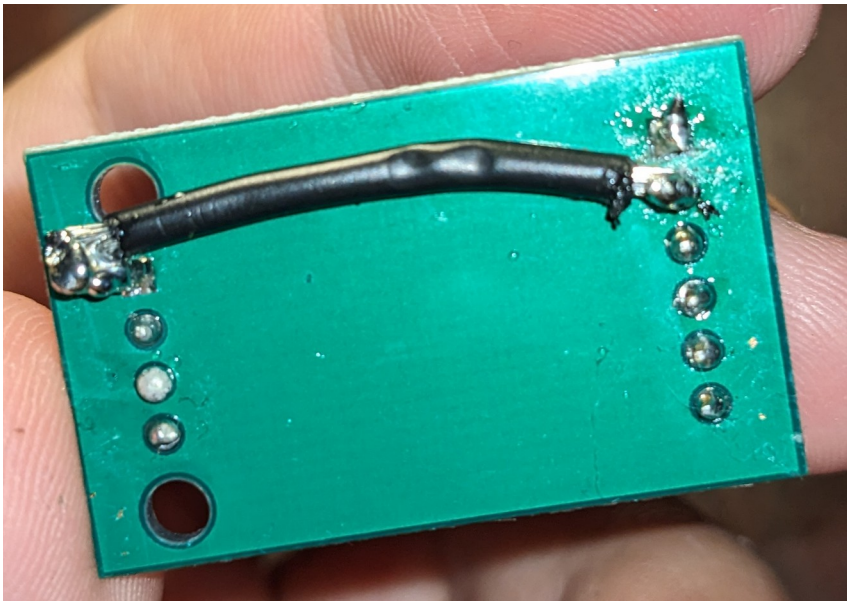

Fig. 4, HX711 Grounding Issue Fix

Actuator: DC Liquid Pump

Fig. 5, Liquid Pump Inside Bottom Tank

As to not waste water, a second tank is placed under the first to catch the used water. A water pump is attached to the wall of the bottom tank and recirculated to the top tank via silicone tubing. Before system startup 150 mL is loaded to the bottom tank. This is more water than the top tank can hold, but is done as to ensure smooth running by preventing the pump from sucking in air. The control and powering of the liquid pump is handled via a TB6612FNG motor driver. The TB6612FNG is a compact motor driver that can handle the current demands of the water pump. The motor driver has pins for providing electrically isolated relatively high amperage power necessary to drive a motor. VM and a GND pin to supply the motor DC power. There are pins for a digital processor, ESP32 in this case, such as a PWM signal pin to control the water pump power via duty cycle percentage control. Signal references VCC and GND pins are present. The rotation direction and other details are controlled with others pins on the motor driver that I recommend hard wiring (in other words, don't connect to the processor). Reason being that these pins will not change values. The hard wiring was AIN1 & STBY to VCC; AIN2 to GND. As for motor outputs AO1 to motor positive, with AO1 to motor negative. Feel free to use the same ESP32 pins described in the code or assign your own (consult your processor's pin capabilities as not all pins are created equal). Note: Arduino uses different code for PWM signals than ESP32 does.

Structure: Wood, Metal Brackets, & Screws

This projects frame is nothing more than pieces of wood connected together metal brackets and screws to a circular wooden base. The frame is constructed large enough that the primary top tank can be connected to the strain gauge and suspended over the bottom tank. Rope is used to suspend the primary tank and connect it to the end of the strain gauge and to have the primary tank aligned to the bottom tank to prevent it from spilling the water. Zip-Ties are used extensively to connect electronics to the skinny wooden posts as well as to force the silicone tubing to hold its shape with the assistance of wooden sticks. Velcro is used to secure the bottom tank at approximately the center of the base and under the suspended primary tank and to secure the water pump to the wall of the bottom tank. Velcro does in fact work just as well underwater to my surprise. Below the circular base of the structure is a anti-slip silicone padding to keep the structure stationary but more importantly to prevent the structure from scratching itself or what it rests on.

Complication

This should seem an obvious point, but use the correct length of screw. This especially goes for the bottom of the structure where a protruding screw can damage what it rests on as well as allow the system to wobble. It is very unlikely that the metal bracket kits will contain small enough screws. Also the attachment of the strain gauge to the wood will require longer screws. I used what I could find and hence the screws are protruding. This is more a cosmetic issue than a safety issue to me however since you should grab the structure from the circular base not the top of the frame. If you desire to also use wood that is as thin as mine purchase shorter screws. Pilot holes (drilling a hole smaller than the threads of the screw in question) are mandatory as otherwise you will more than not crack the wood.

Software

Embedded System: ESP32

The main program compiled on the ESP32 is quite bare bones and simple. The pins needed for the communication with the HX711 are specified as the “LOADCELL” variables and the motor driver PWM pin is specified. A VCC pin is also specified due to a complication that connecting the VCC to the 3.3V of the ESP32 doesn’t permit for uploading code to the ESP32. The work around is to manually power the strain gauge and motor driver VCC reference in code with a pin. The power shuts off during program upload automatically. After variable initialization, the code powers and calibrates the strain gauge and sets up the motor PWM pin. Following, the code simply loops listening for the messages “power:<data>” or “arduino:strain”. Sending a message with “power:<data>” where <data> is a numerical value between 0~255 (uint8) will have the ESP32 change the motor power as requested. Example: “power:215” will set the motor power to 215. Sending “arduino:strain” will have the ESP32 send a 3 decimal point precision value of what the strain gauge senses as a unit-less strain value. The majority of the heavy lifting in terms of processing data and implementing a control system occurs within a personal computer running MATLAB. Tip: Arduino IDE serial monitor can also send the messages for testing/debugging purposes.

Personal Computer: MATLAB

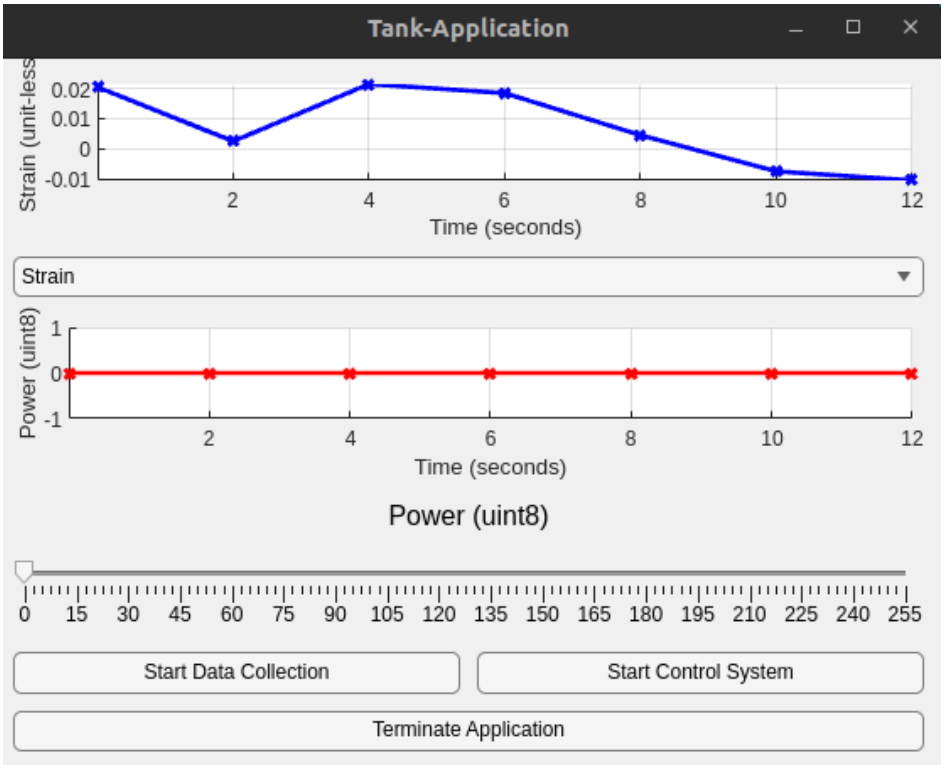

Fig. 6, Example Image of Tank-Application GUI/HMI (On Ubuntu Linux)

Application: tank_app

The heart of the program resides with MATLAB code. The MATLAB code is responsible with obtaining and storing the time, power, and strain data of when the system is active. It is also responsible

for implementing a control system that will send commands to the ESP32 to execute. To best accomplish this, a collection of classes for an application were created to cleanly implement the program goals. Primarily the program “tank_app.m” serves as the “main” where the application code resides that will serve as a human machine interface (HMI). It has a variety of UI elements such as a plot for strain/volume and power. The ability to change the sensor units from strain or milliliters. A slider to manually change the motor power or to specify a strain/volume setpoint depending on the mode, and some buttons to change the mode to data collection, control system mode, or to terminate the program. Some of the settings such as serial_port, baudrate, samples, etc can be modified via hard-coded changes within the “tank_app.m” near the top of the program commented with “--- User Configurations ---”. Perhaps in the future I’ll implement a more elegant system such as a configuration file and a GUI for convenience. The application leverages the classes “ShifterArray.m”, “Tank.m”, “SamplingRegulator.m”, “PI_Controller.m” to compartmentalize the program with their functionally explained below.

Class: ShifterArray

“ShifterArray.m” is a custom made array class. The array is of finite size specified by the user. When the user requests to insert data with the **insert(entry)** method the contents of the array are all shifted once to the left, the data placed at the end of the array, and the oldest entry at the start of the array is deleted out of the array. This array comes in useful as to temporarily store and display live data as well as for filtering relevant data. There probably exists a data structure already programmed into MATLAB to conduct the following operations, but I did not find it particularly difficult to implement my own. The “Tank.m” properties and methods make use of the “ShifterArray.m” class to hold data. ShifterArray has a few other methods to reinitialize the array, check if its full, and the **array** and **recent_value** properties.

Table 1, ShifterArray Properties Documentation

Properties	Details
array	Stores the data provided in an array.
recent_value	Essentially the end of the array and the last inserted piece of data. Mainly provided for convenience.

Table 2, ShifterArray Private Properties Documentation

Private Properties	Details
size	Used to keep track of the size (number of elements) of the array

Table 3, ShifterArray Methods Documentation

Methods	Details
ShifterArray(size)	Constructor where the size of the desired array to be made is provided.
initialize(value)	If you do not desire your array property to be

	filled with “nan” on creation you can instead use this method to change what all the entries start as.
insert(entry)	Will push out the leftmost (starting) value in the array, shift the contents one to the left, and add the provides entry to the rightmost (end) of the array.
isfull()	Will check if there is a single “nan” in the array property. If there is a “nan” it will return false for not full and if there is no “nan” it will return true to signify that the array is full. If you utilized the initialize(value) method than it the array will be considered full if the value is anything but “nan”.

Class: SamplingRegulator

When constructing or when calling the **reset()** method the sampling regulator object will get a new reference time at that moment. The **get_run_time()** method will take that reference time that was recorded and will compare the current time to the reference time to return how much time has passed in seconds. The final and most important method is the **hold().** This method job is to take the sampling_period property determined at construction, find the next valid target time which is a multiple of the sampling_period and halt the program in a while loop doing nothing until enough time has elapsed. The purpose of this class is to as best as possible enforce the desired sampling rate by slowing down the rate at which code is executed. This class is intended to be utilized in a while loop that performs sampling where it is constructed before the loop begins and **hold()** as the last statement in the while loop. The hold() method takes into account redefining a better new target time if the sampling loop ever hangs. Make sure that the **hold()** method is the final command executed in the while loop so that the computation time of the while loop is part of the holding rather than an additional delay in time after the **hold()**. In other words, utilize the computation time of your sampling loop code execution in your favor.

Table 4, SamplingRegulator Properties Documentation

Properties	Details
sampling_period	The sampling_period is used to calculate what the next target_time should be aimed for. For instance, if sampling_period is 0.1, then ideally the target times will be 0, 0.1, 0.2, … etc. If there is a hiccup/hanging in the program runtime then the hold() method will adapt to the next best target.
time_reference	In order for programs to tell the time elapsed two time samples are needed. The time_reference serves as the time sample taken when the user constructs the SamplingRegulator object. It is then used in the sample regulation process

Table 5, SamplingRegulator Private Properties Documentation

Private Properties	Details
prior_target_time	Used speed up finding/calculating the next best target time. It avoids having to avoid recalculating values that have already been calculated.

Table 6, SamplingRegulator Methods Documentation

Methods	Details
SamplingRegulator(sampling_period)	Constructor where the desired sampling_period to regulate to is provided. Construction also takes a time sample to fill the time_reference property.
get_run_time()	Utilizes time_reference property to compare the current time sample to return the elapsed time in seconds.
reset()	Reinitialize the time_reference to resample the moment the reset() method is called as well as sets the memory of prior calculated values in prior_target_time back to 0.
hold()	The meat of the class. This function will find a suitable target_time and stall the program until the run_time approximately equals target_time.

Class: PI_Controller

This class is quite simple. It mainly consists of initializing the constants for a proportional-integral discrete controller with some added controls such as signal clipping and clamping. Kp, proportional constant, Ki integral constant, sampling_period used for calculation purposes, bounds an array containing the lowerbound and upperbound for clipping purposes, offset for clamping the signal (give a DC offset). The **calculate(error)** method uses the error between the setpoint – output (SP-PV) to return an actuation signal. In this project there was a large nonlinear region around motor power 0~180 (as in values between 0~180 don't have enough power to flow any water and hence break additivity and scalability) so to circumvent this issue I set the bounds to [180, 255] with a offset of 180. Most systems have some nonlinearity to them so its great to create controller code that allows to manipulate the signal to avoid nonlinear regions of operation and to focus on the linear region.

Table 7, PI_Controller Properties Documentation

Properties	Details
Kp	Proportionality constant
Ki	Integration constant
sampling_period	The sampling period between samples. Needed since this PI controller is discrete not continuous.

Table 8, PI_Controller Private Properties Documentation

Private Properties	Details
prior_error	Used to track the prior error sample needed for calculation purposes
prior_x	Used to track the prior raw (without offsetting or bounds) PI controller output for calculation purposes
offset	Value used to take the raw PI controller value and add or subtract a value to it. This offset is useful for when you want to redefine what should be considered the “zero” mark regarding actuation. What a PI controller considers zero may not be where you want zero to really be.
lowerbound	The lowest actuation value permitted to be outputted. The internal raw PI controller calculations will not be affected by this.
upperbound	The highest actuation value permitted to be outputted. The internal raw PI controller calculations will not be affected by this.

Table 9, PI_Controller Methods Documentation

Methods	Details
PI_Controller(Kp, Ki, sampling_period, bounds, offset)	Constructor where all fundamental constants are defined for PI controller calculation purposes. The argument bounds is an array [lowerbound, upperbound]
calculate(error)	Returns a value that has gone through a discrete PI controller calculation that then has modifications after the fact such as offsetting the value (signal clamping) and not permitting values to exceed the upperbound or going under the lowerbound (signal clipping).

Those reads curious on why the **calculate(error)** method calculates the PI discrete controller the way it does I present the following below. Note: x[n] & X[z] is actuation signal and u[n] & U[z] is error signal. Ts is the sampling period. Also this is the raw Proportional Integral signal which has clipping and clamping applied after unbeknownst to the raw PI code. In other words, the PI code is unaffected by our introduction of clipping and clamping post processing.

Table 10, PI_Controller Discrete Equation Derivation

$C_{pi}[z]=K_p+K_i\frac{T_s}{z-1}$	Standard Textbook Transfer-Function for Discrete PI Controller
$X[z]=K_pU[z]+K_i\frac{T_s}{z-1}U[z]$	Transfer-Function Output-Input Separated
$(z-1)X[z]=K_p(z-1)U[z]+K_iT_sU[z]$	Multiply Both Sides by $(z-1)$
$(1-z^{-1})X[z]=K_p(1-z^{-1})U[z]+K_iT_sz^{-1}U[z]$	Multiply Both Sides by z^{-1}
$X[z]-z^{-1}X[z]=K_pU[z]-K_pz^{-1}U[z]+K_iT_sz^{-1}U[z]$	Distributive Property
$x[n]-x[n-1]=K_pu[n]-K_pu[n-1]+K_iT_su[n-1]$	Inverse Z-Transform via Time-Shifting Property
$x[n]=x[n-1]+K_pu[n]-(K_iTs-K_p)u[n-1]$	Simplify and Rearrange

Class: Tank

This class has a variety of functionality. Primarily its **open_connection(serial_port, baudrate)** method opens a serial connection with the ESP32 so that data can flow back and forth. **close_connection()** terminate this communication. **close_connection()** is more of a formality as clearing the object spawned with Tank class also closes the connection. With an open connection the method **get_strain()** returns a reading from the strain sensor with no additional processing in other words, raw data. The method **get_strain_w_pause()** is a wrapper over **get_strain()** except where the motor is powered off for a motor_pause_s amount of time. Note: the method does not turn the motor back on that is left to the programmer to supply power after calling the method. The methods brother, **set_power(value)** allows for a value between 0~255 integer and is sent to the ESP32 to be executed to set the motors power level via PWM duty cycle. If you desire to cleanly store finite relevant time, power, strain, and setpoint data the **set_samples(samples)** method allows you to specify the size so that a ShifterArray for “time_record”, “strain_record”, “power_record”, “setpoint_record” are initialized within the object spawned with **Tank()**. All the properties and methods of a shifter array can be used. A helper method in Tank is **clear()** which will revert all those shifter arrays back to all nan and to calibrate the strain gauge. The final use of the Tank class is to generate processed sensor data. Since the strain gauge is susceptible to noise or in need of recalibration various methods exists for these purposes. The **calibrate_strain_gauge()** method will re-zero the strain gauge data by polling for data and averaging it. By default it will be polled 10 times with 100 millisecond pauses and will be averaged. This is a 1 second delay for calibration purposes not including the time of shutting down the motor for the motor_pause_s amount of time. To change how many samples and the sampling rate use **set_calibration_settings(calibration_samples, sampling_period_s)** method. An internal property called strain_offset is updated as a result from its default of 0. Note, calling the calibration method will cause **get_strain()** method to also return calibrated data and not truly raw data. Note, that **calibrate_strain_gauge()** is called automatically in multiple methods such as when calling **open_connection(serial_port, baudrate)** and **clear()**. The function **get_strain_filtered()** will return a strain value that has undergone clipping and moving-average filters. Note: the moving-average filter is not active by default since I had since then opted to the shutting off the motor momentarily technique to prevent noise instead rendering the moving-average filter not needed. However, I leave the code in case it becomes relevant and desired to be activated in the future. To control how **get_strain_filtered()**

operates utilize **set_bounds(bounds)** to control clipping and **set_moving_average_samples(samples)** to control how many points to use for the moving-average calculations.

Table 11, Tank Properties Documentation

Properties	Details
time_record	A ShifterArray for time data. Has all the properties and methods of a ShifterArray for convenient data bookkeeping.
strain_record_raw	A ShifterArray for raw strain data. Has all the properties and methods of a ShifterArray for convenient data bookkeeping. Useful for performing moving average filtering.
strain_record	A ShifterArray for strain data. Has all the properties and methods of a ShifterArray for convenient data bookkeeping.
power_record	A ShifterArray for power data. Has all the properties and methods of a ShifterArray for convenient data bookkeeping.
setpoint_record	A ShifterArray for setpoint data. Has all the properties and methods of a ShifterArray for convenient data bookkeeping.
strain_offset	Holds the offset value on how much to add or subtract from raw strain data to get an accurate reading. It effectively stores the raw strain data that should be considered the “zero” point.

Table 12, Tank Private Properties Documentation

Private Properties	Details
connection	A MATLAB object with functionality pertaining to transmitting data back and forward via a serial port or that which resembles a serial port in software such as UART communication utilized by an ESP32.
calibration_samples	An integer value of how many samples should be taken to be averaged to determine a strain_offset value
sampling_period_s	Another parameter regarding calibration which is the minimum amount of time that should pass before taking another sample used for calibration.
motor_pause_s	How long the motor should be powered off before a sample is taken. This value should be as small as

	possible while still preventing the presence of EMI for the strain gauge.
lowerbound	A parameter for a method to return filtered strain gauge data. This determines the lowerbound of the signal that should be clipped.
upperbound	A parameter for a method to return filtered strain gauge data. This determines the upperbound of the signal that should be clipped.

Table 13, Tank Methods Documentation

Methods	Details
Tank()	Constructor for Tank class which sets the moving average filter sample size to 1. In other words, by default the moving average filter does nothing to data.
set_motor_pause_s(motor_pause_s)	Allows change to the motor_pause_s property.
set_samples(samples)	Initializes the creation of various bookkeeping ShifterArrays. Specifically, time_record, strain_record, power_record, and setpoint_record properties.
set_moving_average_samples(samples)	Allows changes to the amount of samples the moving average filter uses and initializes the array that stores the raw strain data of that size.
set_bounds(bounds)	Used to specify the bounds used in clipping data when requesting for filtered strain data. The argument bounds is an array [lowerbound, upperbound]
open_connection(serial_port, baudrate)	Creates a serial connection with the ESP32. It also prompts the calibration of the strain gauge for use.
close_connection()	Destroys the serial connection so that it may be used again.
get_strain()	Requests the ESP32 to send unprocessed strain data. Function return the value.
get_strain_w_pause()	Requests the ESP32 to send unprocessed strain data. Function return the value. It also shuts off the motor for motor_pause_s amount of time before sampling to prevent the effects of EMI.
get_strain_filtered()	Essentially a wrapper over the get_strain_w_pause() method except it incorporates the clipper(strain) and the entirety of

	the strain_record_raw array to perform moving average filtering.
set_calibration_settings(calibration_samples, sampling_period_s)	Allows changes to calibration_samples and sampling_period_s properties pertinent to performing calibration of the strain gauge data.
calibrate_strain_gauge()	Initiates the collection of data to then average to fill the strain_offset property.
set_power(value)	Transmit a message to the ESP32 to change the power of the motor to the value specified to the method. Value must be uint8 (in other words 0~255 integer).
clear()	A convenience method which calls the clear method on the time_record, strain_record, power_record, setpoint_record, and then recalibrate the strain gauge.
clipper()	The method in charge of clipping the strain data (enforcing lowerbound and upperbound properties) when requesting for filtered strain data.