Sensor Networks for Automated Water Depth Mapping in AWD-based Rice Field Irrigation Systems

EGR 250/251: Final Project Report

Authors:

Oleg A. Golev

Shadman Jahangir

Sumanth Maddirala

Under the guidance of:

Professor Jay B. Benziger

Emmanuel A. Osorno

Community Project Studios

Keller Center, Princeton University

Sept. 2019 - May 2020

Contents

1 Introduction **2**

1.1 Background Context 2

1.2 AWD-Based Irrigation System 3

1.3 Benefits of AWD-Based Irrigation 4

1.4 Concerns, Purpose and Goals 5

2 Proposed Product **6**

2.1 The Sensing Device 6

2.2 The Central Controller Station 9

3 The Prototype **11**

3.1 Resources and Cost 11

3.2 Data Flow and Circuit Diagrams 12

3.3 Setup and Operation 13

3.3.1 Preparing the Experimental Space 13

3.3.2 General Procedure 14

4 Analysis of Results **15**

4.1 Imperfections in the Readings 15

4.2 Error Correction 16

5 Conclusion and Future Work **18**

6 References **20**

Appendices **22**

A Appendix: Pin-out Details for Circuit Schemas **22**

B Appendix: Receiver Arduino Code **24**

C Appendix: Sensor Arduino Code **26**

D Appendix: Receiver Log Example **29**

E Appendix: Sensor Log Example **30**

F Appendix: Data Analysis **31**

1 Introduction

1.1 Background Context

In the status quo, rice (*Oryza sativa*) is the staple food for more than half of the world's population, with more than 3.5 billion people relying on rice for at least 20% of their daily caloric intake (GRiSP, 2013). While Asia has historically accounted for 90% of the global rice consumption, economic growth and increased urbanization in Sub-Saharan Africa has resulted in an increase of per capita rice consumption (Nigata et al., 2017). Increased household incomes and urbanization in the region have resulted in a shift in food preferences from the more traditional staples of corn, sorghum, root and tuber products, toward rice, wheat and meat. This comes at a time where rice has shifted from a largely luxury food to a staple for the growing middle-class across Sub-Saharan Africa. Presently, rice is the second most important source of calories after corn, replacing roots, tubers, millet and sorghum in many countries of the region. It is projected that by 2026, the total rice consumption in Sub Saharan-Africa will grow from 25-26 million to 36 million tons (Nigata et al., 2017).

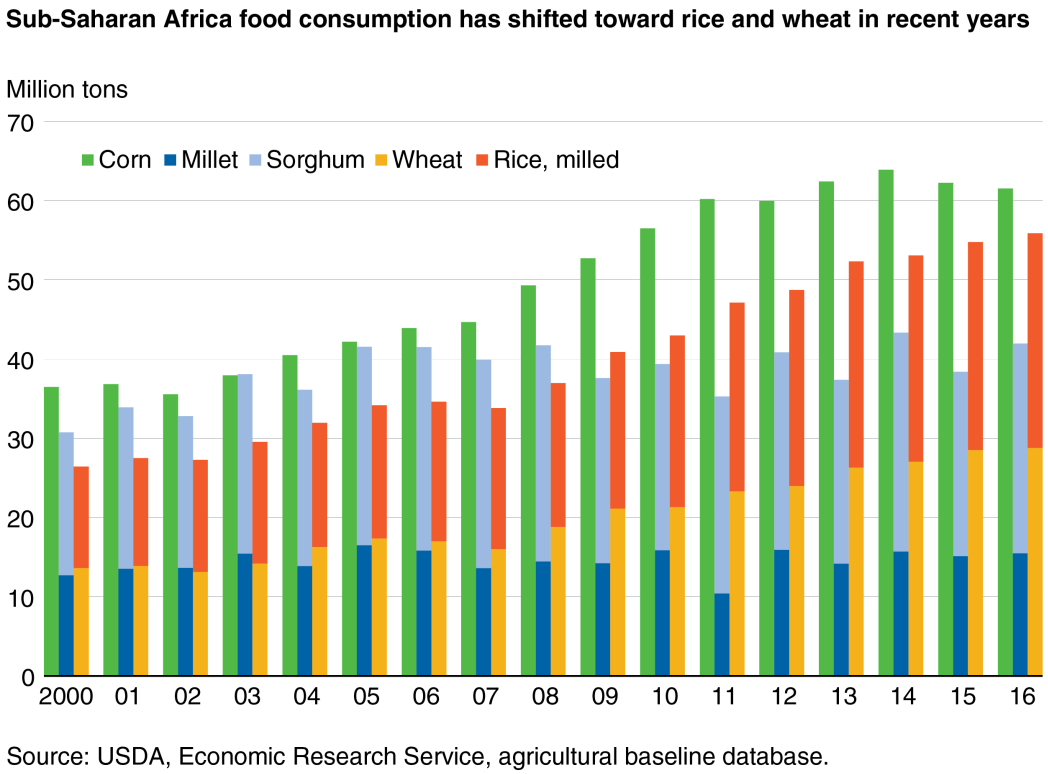


Figure 1: Food consumption in Sub-Saharan Africa has shifted toward rice and wheat in recent years. Graph adapted from Nigata et al., 2017.

This represents a unique concern because while rice is a vital staple for an increasingly large number of people around the world and may be key to ensuring global food security, traditional rice cultivation, especially with the use of flooding rice paddies, demands higher water investment

than that of other cereal crops (Pimetentel et al., 2004), with a 1 kilogram bag of rice requiring over 2,500 liters (Bouman, 2009). With over 4 billion people potentially at risk from water scarcity (Mekonnen and Hoekstra, 2016), finding agronomic processes that both reduce water use and result in higher crop yield is crucial to keep up with demand and an increasing population.

1.2 AWD-Based Irrigation System

One practice that has been shown to reduce the amount of water used for rice cultivation is an irrigation management system known as "Alternate Wetting and Drying" (AWD). As compared to more traditional methods, AWD relies on intermittent rather than continuous flooding of fields to achieve a sufficient water level for rice growth. Only once a low enough soil moisture level is reached, the field is reflooded (Linquist et al., 2014). In comparison to continuously flooded rice systems, AWD has been reported to reduce total water input by up to 23% (Bouman and Tuong, 2001), with a meta-study confirming the reduction under mild AWD-conditions, i.e when soil water potential was ≥ -20 kPa (Carrijo et al., 2017).

Current implementation of AWD involves the use of a PVC water tube/pipe (can also be called "pani pipe"), that is used to monitor the water level. The pipe, usually 7-10 cm in diameter and 30 cm long, has perforations in the bottom 20 cm. These tubes will often have larger diameters to ensure that the water is easily visible and soil in the pipe is easy to remove. The perforations along the bottom of the tube remain below the soil and allow water to travel from the soil into the tube. The top 10 cm remains above the surface. While the water that travels inside the tube is typically measured by a scale, the PVC pipe set-up does have variations. For example, some farmers will have the bottom 15 cm of the tube perforated, and may use more manual methods such as a measuring tape to measure the depth of the water (Lampayan et al., 2014).

While the use of pipes or tubes is standard to measure water depth in an AWD-system, the actual process of irrigating and re-irrigating fields is not so clear cut. After a field is initially flooded, the water level gradually decreases due to evapotranspiration, seepage and percolation. The tubes can be used to measure water depth down to 15-20 cm below the soil's surface; once the water goes below that level, the fields should be re-flooded such that the water level is 5 cm above the surface. The depth and extent of flooding, however, are based around the rice's stage of development. For example, if the rice is flowering, the field should always be kept flooded; on the other hand, in later ripening states, the water level is allowed to drop 15 cm below the soil surface before re-flooding is needed. To ensure that the growth of weeds is suppressed, AWD can be started 1-2 weeks after transplantation, or up to 3 weeks after in the event that there are many weeds present. Common local fertilizers are adequate for use in AWD-systems (Lampayan et al., 2014).

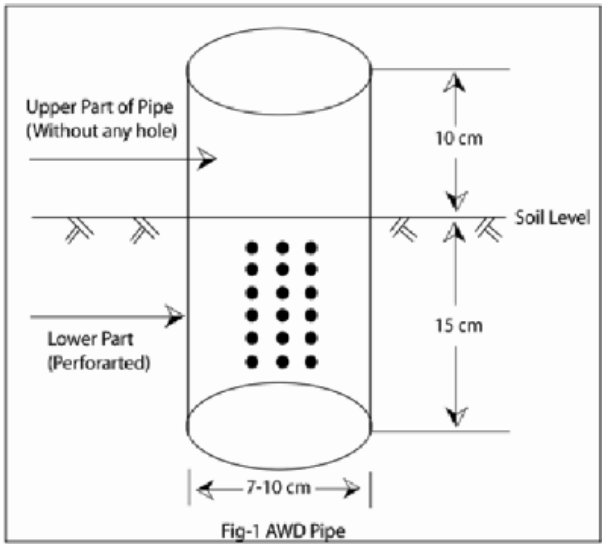


Figure 2: Diagram for a tube construction designed to be placed into dirt fields for water depth monitoring in AWD setups. Figure adapted from Neogi et al., 2018.

Figure 3: Practical implementation of a PVC pipe in the current model. Figure adapted from Lampayan et al., 2014.

1.3 Benefits of AWD-Based Irrigation

While AWD-based irrigation has been shown to reduce the water input required to grow rice (Linquist et al., 2014; Carrijo et al., 2017), AWD has also shown numerous other benefits in regards to its impact on production, yield, human health, environment and the socio-economics of local communities.

First, AWD has been shown to reduce greenhouse gas (GHG) emissions, especially that of methane, by 45% compared to continuously flooded systems. A combination of AWD with nitrogen-use efficiency and better management of organic inputs can further reduce GHG emissions (Richards and Sander, 2014). Additional environmental benefits include reduction of arsenic accumulation in grain (Linquist et al., 2014), as well as a reduction in methlymercury concentration in soil (Rothernberg et al., 2016). Furthermore, as AWD requires less flooding than more traditional methods, it also has the potential to decrease mosquito and water-borne diseases (Celeridad, 2019).

AWD also reduces the frequency of flooding needed in fields, which contributes to better soil structures and allows for the inter-cropping of rice with other crops, i.e. crop diversification. Furthermore, for communities that suffer from lack of water, this method is a cheap but practical alternative to other irrigation methods, especially when considering additional water scarcity issues that may arise in the future.

It's clear that the AWD-based irrigation system has numerous benefits, but it is important to

remember that many of the benefits are centered around the careful monitoring of water levels at different stages of the rice crops' growth. For example, Carrijo et al. (2017) states that mild conditions are most suitable for ensuring that water input is reduced without sacrifice to the yield. The implementation of a system to monitor the depth of water in rice fields is something that we will aim to address in this project.

1.4 Concerns, Purpose and Goals

While there are numerous benefits that come from the use of an AWD-based irrigation system, it is not without concerns. One prominent issue is the fact that the timing, frequency and extent of wetting and drying cycles depend heavily on the rice's stage of development, as well as other external factors such as prevailing weather and field conditions. Furthermore, one must be able to accurately determine the level to which that field has already been flooded and determine the best time to re-flood. As shown in Figure 3, this process can often be cumbersome and labor-intensive, which would also detract the farmer's time from other important tasks. Additionally, manual methods to measure the depth of water in the field may not necessarily be the most accurate, which would also impact rice growth under the AWD framework.

In order to resolve these issues, we propose the use of a water sensor that would be able to help farmers better manage flooding and re-flooding of rice fields in an automated fashion. The sensor can be inserted via a pole/pipe below the soil to determine water depth. Also connected to the sensor would be a transceiver system that would be able to send information of the current height of water in the field to a central computing unit, which could then be observed by the farmer without having to measure the depth of the water manually. The benefits of such a system are two-fold:

1. Farmers will be able to measure the height of water present in the field accurately, which will aid in making decisions of when to re-flood the field

2. The automation of the process will ensure that farmers are able to spend their time doing other more important tasks, resulting in an increase in productivity and efficiency.

There are several more factors to be considered in regards to the management of such a sensor system. First, given the benefit of the AWD-system in reducing water scarcity, the use of such a sensor system would see most import in the developing world. To ensure that the sensor can actually be implemented on a wider scale in the developing world, the sensor would need to be cheap. Furthermore, as long term use of the sensor would also be beneficial to local communities, we must ensure that the sensor is robust enough to withstand various environmental conditions and exhibit longevity. Furthermore, in the event that sensor malfunctions, the sensor should be able to send a failure signal that would alert the farmer for replacement. Finally, the accuracy of the sensor should ideally be $\pm$ 1-cm and could be coupled with/attached to the depth piping construct. This is important as different stages of the rice crops' development will require different re-irrigation schedules, and these differences are based on variations in water depth.

Thus, the goals of this project are two-fold:

1. Creation of a cheap, but robust, water sensor system that is able to successfully automate water-depth tracking

2. Confirming the accuracy of the sensor in order to determine real-world viability

In the developing world, this sensor would provide convenience and scalability for rice cultivation given reduced direct labor on upkeep of the rice fields. This increase of productivity and efficiency in farming is coupled with additional benefits of AWD-irrigation (Section 1.3).

2 Proposed Product

A technological rehaul of an AWD-system calls for a deployed depth-sensing system with a cascade of remote sensor devices and a central base system that can request information from the devices with a simple push of a button. The sensor devices would be installed at regular intervals over the area of the entire rice field and send water-depth measurements to the central controlling device at the user's request from the controlling device's interface. The central device must then display the data as a 3-dimensional map of water depth of the entire field. We will now discuss these two integral pieces in more detail: an instance of a desired sensor module itself and the receiver station for data retrieval.

2.1 The Sensing Device

The sensing device has the primary objective to measure water depth at the location of its deployment. Physically, the device must resemble a small lag-pole sized appropriately to capture data in the desired depth-measurement interval. The device will have four primary parts: the head, the body, the solar panel and a hollow, perforated casing connected by the body.

The head must house the main electronics board containing the processing chip and a wireless transceiver magnified with an internal antenna. Thus, the head must be water-proof and attached securely to the body. The body also houses the depth sensor, which can be capacitance-based, resistance-based, or a laser sensor. To ensure that the depth sensor is able to get the most accurate measurements, it is placed within the sensor casing, which is perforated along the bottom to allow water to enter. The sensor casing must also be of the desired length, fit to the height over which the measurements are desired to be taken. For instance, if the customer wants to make sure that the water level on the field remains between 10 centimeters above and 5 centimeters below the ground level, the length of the sensor casing must be at least 15 centimeters. For illustration purposes, the below diagram approximates the desired final product of this proposal:

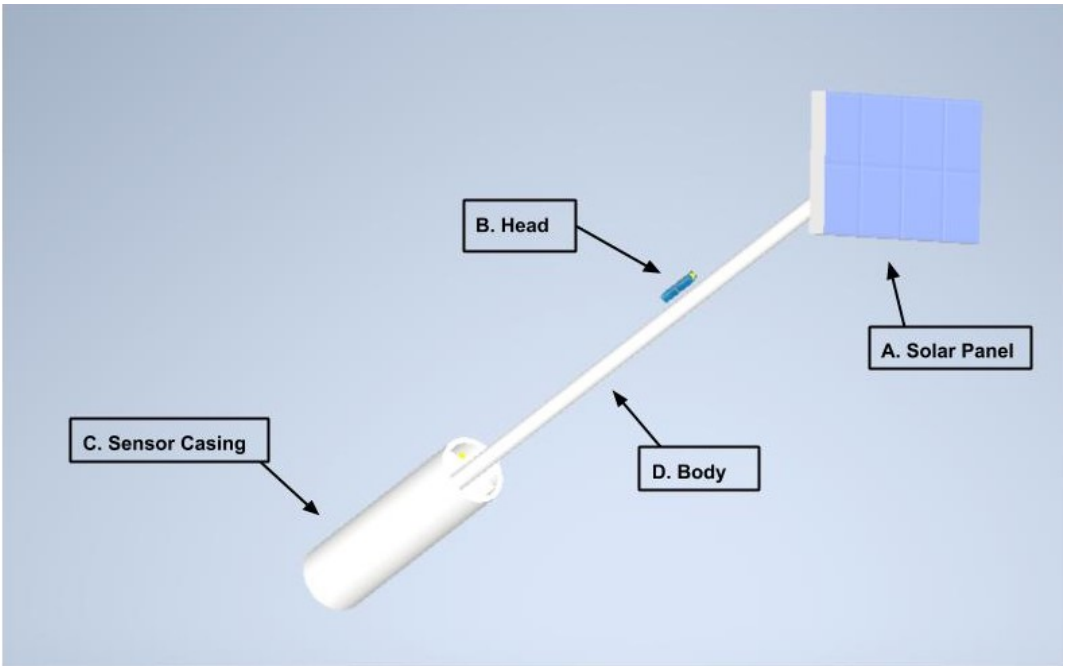


Figure 4: General assembly diagram for the sensing device. **A.** Solar panel used to charge on-board batteries. This ensures that the water system's microprocessor always has sufficient energy to operate. **B.** The microprocessor unit that holds the transceiver unit. This allows the water sensor to connect to the receiving hub and communicate with other water sensors to ensure more accurate measurements are taken across the field. **C.** Sensor casing holds the Milone-tech water sensor. The casing is perforated along the bottom to allow water to enter the casing (shown in more detail in Figure 5), where the depth is measured by the Milone eTape. **D.** Body connects the three main components of the sensor apparatus: the solar panel, the microprocessor unit and the sensor casing.

Note that the proposed product would be powered by a solar panel. Since the device would be in standby for most of its operation time and thus would not require a continuous source of power, the solar panel would be ideal for providing charging capability for a small internal battery to power the microcontroller. Functionally, a large metal pole or branch could be dug into the ground at the desired location of the sensor. The sensing device in Figure 4 can then be attached to this pole with the head above the highest expected water level the field may be flooded to. The zeroing process for this product would be done using a special-purpose button in two steps: assuming that the sensor is resistance-based, pressing the button once to record the resistance when no water is present, and once to measure the resistance when the sensor casing is fully submerged in water. The code will then take care of the rest. Our prototype (which we describe in detail later) follows a similar procedure, yet the resistance is measured and then inputted into the code manually.

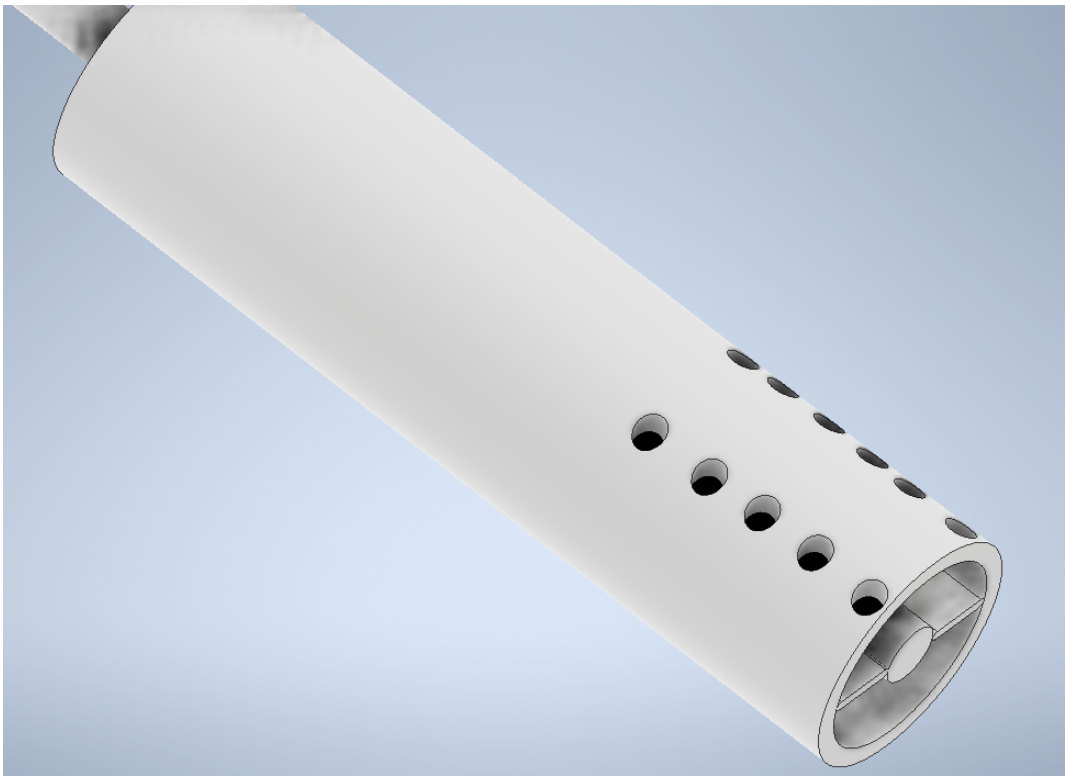


Figure 5: Close-up of sensor casing. The bottom 15 centimeters are perforated to allow water to flow into the chamber; this exposes the Milone eTape to water and allows for water depth measurements to be made. The perforations are filtered to block most dirt from entering into the casing.

In designing the sensing device, it must have multiple essential characteristics to guarantee its successful deployment in developing countries:

1. **Accuracy**: every time water depth data is requested, the sensor must deliver an accurate reading promptly and consistently to the user, independently of the properties of the water and the earth conditions.
2. **Robustness**: the device must be responsive in all conditions. In constant contact with water, dirt, and harsh environmental conditions, the sensor assembly must be robust and secure to last a long time with little maintenance and no issues with electronic communication. In the case it does not stand the test of time, the sensor must be easily replaceable or repairable, with easy access to the modularized internals.
3. **Reliability of Use**: the sensor must retrieve data reliably. When the user requests data, the sensor must deliver. If delivery does not occur, it must be certain that the sensor malfunctioned due to a fatal hardware error rather than a communication error or a software error, while the chance of a fatal hardware error must be extremely low. Likewise, the sensor should always have sufficient power to perform its function, so a solar panel and a small battery can prove optimal for this purpose given field conditions.
4. **Price**: for use in developing countries, the assembly must be affordable for mass deployment.

That is, a single field may have multiple sensors installed to give an accurate water-depth map. Thus, the cost of these sensors must be low and scalable.

5. **Range**: water-depth sensors may be installed on kilometer-long and equally wide fields. Thus, the system must incorporate sensors capable of long range data transmission over open surface, preferably of at least 500 meters. Likewise, if the sensor is located too far away from the central controller station (for instance, at the other side of the field), the sensor must be able to send measurement information and its unique location ID to its closest neighbour, which will then attempt to pass the information in a similar fashion to the base station. That is, the system must be able to cascade requests for data to sensors that may be out of reach of the base station as well as cascade responses with the water-depth data back to the base station that may be out of reach for some sensors. This is described in more detail in the next subsection.

In regards to device size, for most applications in systems using AWD, an effective measurement length to the height of thirty (30) centimeters should provide ample flexibility to the user for accurate depth-measurement.

2.2 The Central Controller Station

The function of the central controller station primarily lies in requesting, receiving, and then displaying data to the user. We will discuss each briefly:

1. The device must contain a button that, when pressed, sends a request for water-depth data to all sensors within its effective range
2. As the sensors receive the request, pass it on (if necessary), take the measurement, and cascade it back to the central processing station, the station must use the location-ID's to make sure no duplicate data is collected.
3. After all sensors have sent a response back to the base station (or until timeout), the device must render a predicted water-depth topographical map of the field and display it to the user on an LED screen. Thus the map of the water status of the field is the information delivered to the user from the extended network of small sensor nodes preinstalled on the field.

Similar systems with small sensing nodes and a central processing controller have been established for natural disaster prevention to monitor water levels in real-time. This system is only a natural translation of the more traditional application. For illustration purposes, the below diagram approximates the desired system logic for the product described in this proposal:

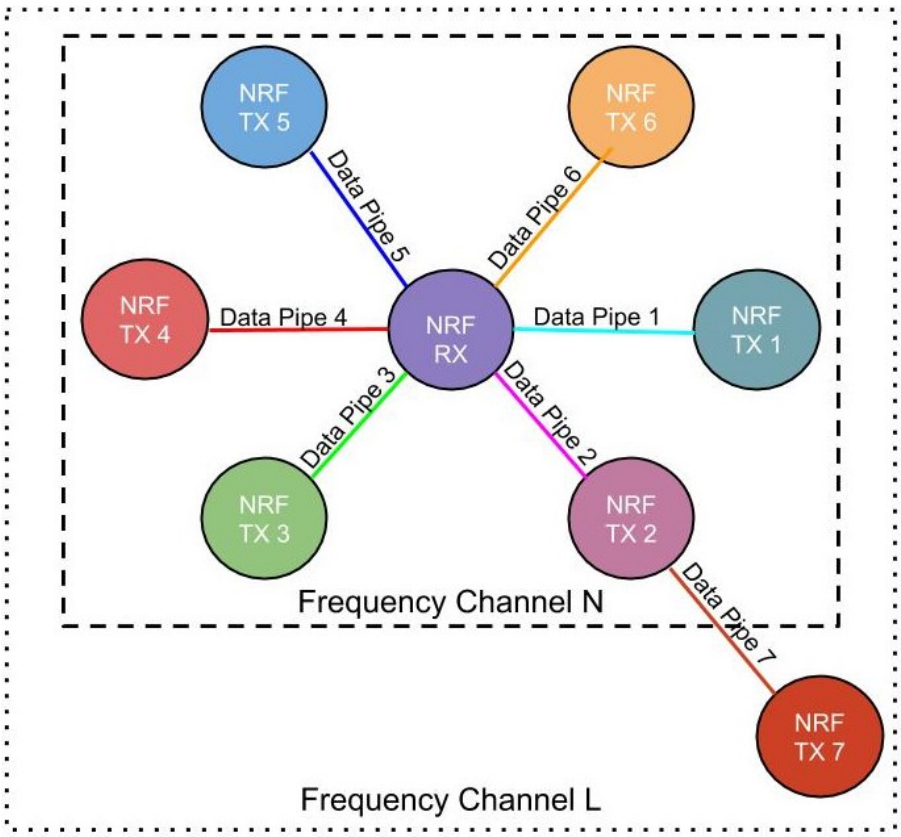


Figure 6: Data pipeline connections between receivers and transmitters. The central processing unit NRF RX is connected to the sensors NRF TX 1-6, and TX 2 sends along its own as well as NRF TX 7's data since NRF TX 7 is too far to directly communicate with the central station.

Information is sent between individual sensing devices and the central controller station by way of transceivers. Transceivers are devices that are both able to transmit and receive information, typically over a specific frequency (called a "channel"). To better justify the system logic shown in Figure 6, let us examine the nRF24L01+ transceiver, a commonly used low-powered transceiver which operates at 2.4 GHz worldwide ISM frequency band. The nRF24L01+ transceiver has a specific feature known as "multiceiver" in which a specific channel can be further divided into 6 parallel data channels, called data pipes. The data pipes allow a receiver hub, called NRF RX, to receive data from up to 6 transmitter units, called NRF TX, simultaneously. As a result, the receiver hub would be able to collect data from multiple transmitter units and process them; if the transmitter units were spread out over a field, they would be able to send data about water depth in their local region to the receiver hub. Since each of the transmitters can also act as a receiver, a cascading effect can be achieved, in which transceivers can also receive data from other transceivers, before sending its own and all collected information to the central hub. For example in Figure 6, the central hub (NRF RX) is connected to NRF TX 1-6 directly through the respective data pipes. Note however, since NRF TX 7 is far away from the NRF RX, NRF TX 2 can act as a "receiver" unit along data pipe 7, which in turn sends the data to the central receiving hub.

3 The Prototype

As a proof of concept, we will construct a prototype using the best-documented cheap commercially available microcontroller board: Arduino UNO. Specifically, we will simulate one-to-one communication between two Arduino boards, where one will serve the role of the controlling base station. This Arduino, which we will refer to as the receiver, must have the functionality to request data at the will of the user and then display the data on the computer screen. The other board must effectively act as the water-depth sensor, being able to take a water-depth measurement and send it back to the receiver when the data is requested. We will refer to this Arduino as the sensor.

3.1 Resources and Cost

The following were used for assembly of the receiver Arduino:

- Elegoo UNO R3 Board: $16.99 / 1pc
- Push Button, a 2000 Ohm resistor, and male-to-male jumper cables from the genetic Elegoo Electronics Kit: $17.98 / kit
- Makerfire NRF24L01+ 2.4GHz Transceiver: $14.98 / 10pc

The following were used for assembly of the sensor Arduino:

- Elegoo UNO R3 Board: $16.99 / 1pc
- Milone Tech 12-inch eTape: $29.99 / 1pc
- A 2000 Ohm resistor and female-to-male jumper cables from the genetic Elegoo Electronics Kit: $17.98 / kit
- Makerfire Arduino NRF24L01+ 2.4GHz Transceiver: $14.98 / 10pc

It is clear that the prototype could be scalable to a higher number of sensors connected to the same receiver Arduino. For a receiver and nine sensors, the total cost would be:

- 10x Elegoo UNO R3 Board → $169.90
- 9x Milone Tech 12-inch eTape → $269.91
- 2x DEYUE Breadboard Set / 6 PCS → $17.98
- 1x GenBasic 4-inch Female-to-Male Jumper Wires / 4pc → $4.99
- 1x GenBasic 8-inch Male-to-Male Jumper Wires / 40pc → $4.99
- 1x Makerfire NRF24L01+ 2.4GHz Transceiver / 10pc → $14.98
- 1x ELEGOO Components Basic Starter Kit → $9.86

The total cost for a nine-sensor and one central receiver system would be $492.61, or $49.26 for each device. Most of the cost is hoarded by the boards and the eTape sensors. The cost could be

brought down significantly by replacing the UNO R3 boards with microcontrollers embedded into a PCB board ($5 per board) and replacing the eTape with a cheaper laser sensor bought in bulk ($20 each). Thus, the total cost could be brought down to $282.80, or $28.28 for each device.

3.2 Data Flow and Circuit Diagrams

The two Arduinos are connected to different computers separated by several feet. Both computers are running the Arduino IDE software that supports data output to a Serial Monitor (a terminal equivalent for Arduino). The receiver sends a data request at a press of a button using the nRF24L01 transceiver. The circuit diagram for the receiver Arduino setup is shown below in Figure 7.

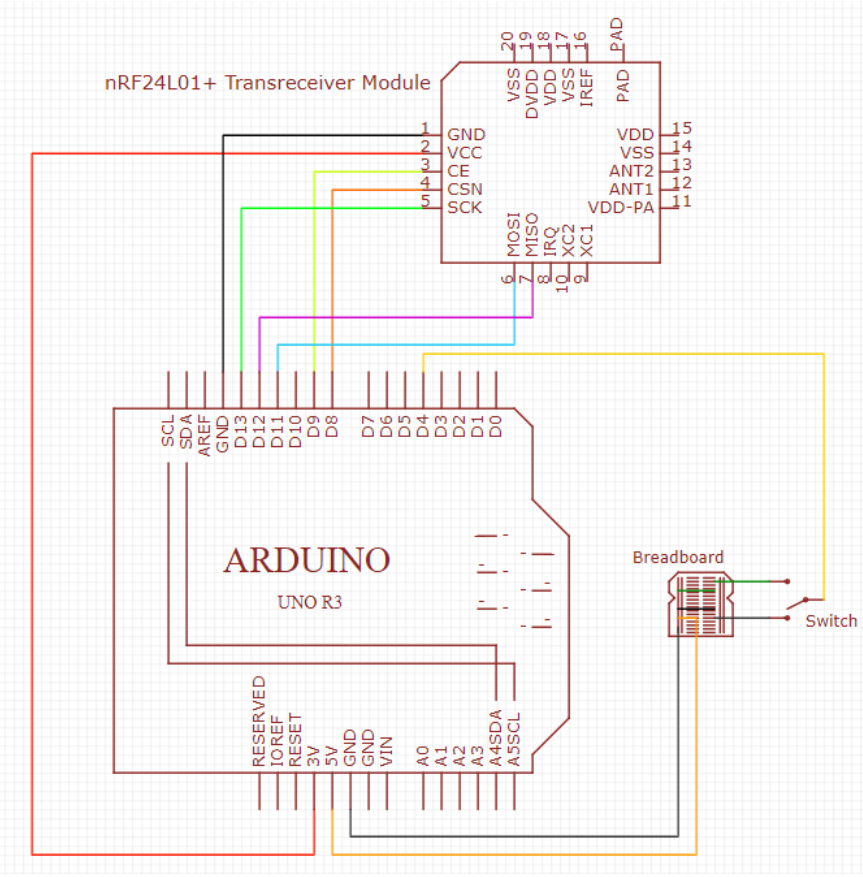


Figure 7: Detailed schematic of the receiver Arduino circuit board, the attached nRF24L01 transceiver module, and the button assembly used in this experimental study.

In turn, the sensor receives the data request, takes a measurement with the eTape, writes the data to its Serial Monitor log, and sends the data to the receiver. The circuit diagram for this sensor Arduino setup is shown below in Figure 8.

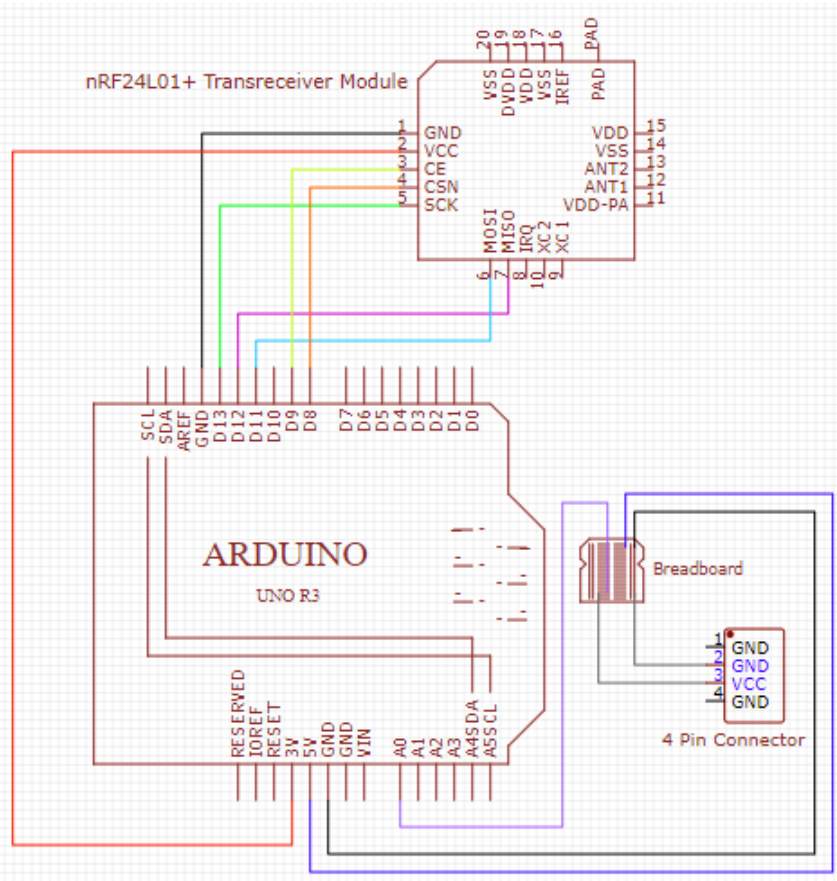


Figure 8: Detailed schematic of the sensor Arduino circuit board, the attached nRF24L01 transceiver module, and the eTape assembly used in this experimental study.

The receiver Arduino then accepts the data, writing it to its own Serial Monitor. If any error occurs in the communication protocol between the sensor and the receiver, both will print the appropriate error to their Serial Monitor log. For a more detailed description of the pin-out for Figures 7 and 8, see Appendix A.

3.3 Setup and Operation

3.3.1 Preparing the Experimental Space

1. Complete the circuit setup described in the diagrams above.
2. Plug the Arduino's into two separate computers, open the Arduino IDE on both.
3. Compile and upload the receiver Arduino program file using the Arduino IDE on the computer connected to your receiver Arduino. The receiver Arduino code is attached in Appendix B.

4. Compile and upload the sensor Arduino program file using the Arduino IDE on the computer connected to your sensor Arduino. The sensor Arduino code is attached in Appendix C.
5. Click the Serial Monitor buttons on both of the Arduino IDE's to view the data streams.
6. Upon pressing a button on the receiver Arduino, a token is sent to the sensor Arduino, which collects the data and sends it back. Both Arduinos print the transmitted data to the Serial Monitor and keep a detailed log of all data processing and transmission operations. Examples for proper log data for the receiver Arduino and the sensor Arduino are included in Appendices D and E respectively.
7. Calibrating the sensor Arduino and editing the code in EPICS_transmitter_code.ino:
   (a) Press the button on the receiver Arduino to initiate data collection when no water is present. In the code, record the measured resistance as the constant named ZERO_VOLUME_RESISTANCE
   (b) Fill a cylindrical container (preferably 12+ cm in height) with distilled water and lower the sensor to its maximum reading mark. Wait 2-3 minutes.
   (c) Press the button on the receiver Arduino to initiate data collection with the sensor fully submerged in water. Record the measured resistance as CALIBRATION_RESISTANCE in the code file.
   (d) Take a ruler and measure the radius of the cylindrical container. Record this value in the code at this line: "float radius = _._;"

3.3.2 General Procedure

1. Click the Serial Monitor button on the Arduino IDE's to view the data streams
2. Fill up a cylindrical container of water (preferably with a maximal capacity of 1L or higher) up until the 980mL mark
3. Dip the eTape some cm below the surface of the water. Using a ruler, measure the height of the water from the meniscus to the bottom of the eTape
4. Press the button on the receiver Arduino at regular intervals to collect data until the readings of volume and depth calibrate (wait for approximately 90 seconds). Record these values in a data sheet. Also, record the resistance value output by the program[1]
5. Repeat steps 3 and 4 by increasing the depth of the sensor in 2.5-cm increments (2.5, 5, 7.5...) until the 30-cm mark
6. Repeat steps 1-5 for different solution concentrations, each of 0.5 M, 1.0 M, and 2.0M respectively. Each of the solutions should be formed by using roughly 58 grams of salt per number of moles, and mixing the liter of water for about 1 minute.

[1] eTape directly measures resistance, from which we can determine the volume of water, and thereby its height.

4 Analysis of Results

4.1 Imperfections in the Readings

As mentioned earlier, we manually measured the height and depth of the water in our pipe. During data collection, we realized that there was a significant discrepancy between our measured data and what our Arduino program was outputting, for the volume and consequently the height data. For lower volumes and heights, we noticed that the numbers output by the program were significantly lower than what they should be; similarly we noticed that for the higher ranges, the output from the program was significantly higher than the target heights. The data for four different trials with different salt concentrations are presented in Figures 9, 10, 11 and 12 below.

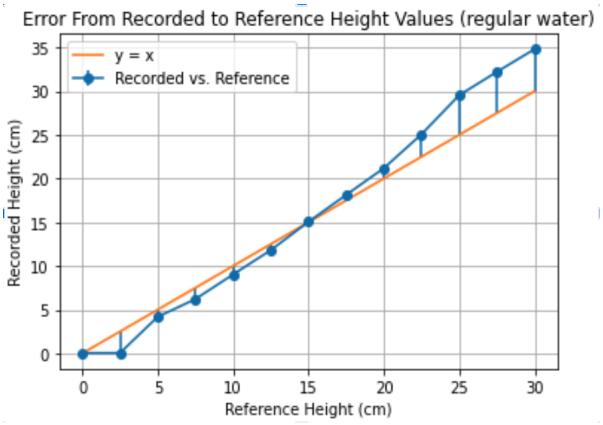


Figure 9: Experimental data (blue) and desired measurements (red) for measuring depth of regular tap water.

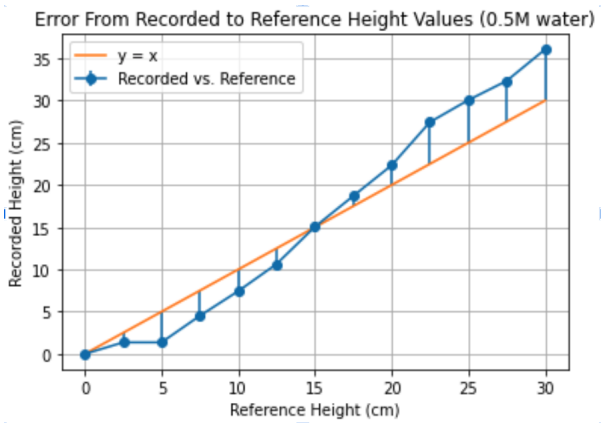


Figure 10: Experimental data (blue) and desired measurements (red) for measuring depth of water with 0.5 molar salt concentration.

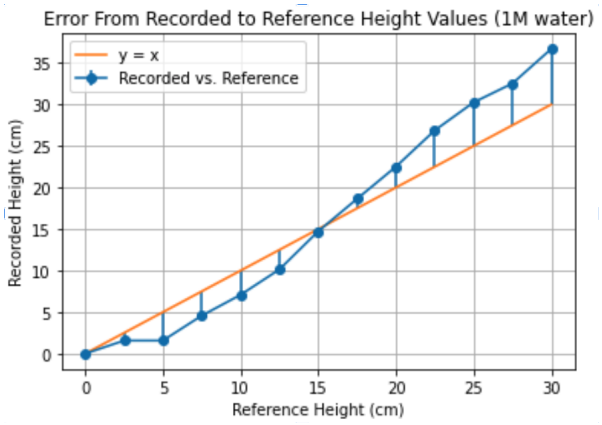


Figure 11: Experimental data (blue) and desired measurements (red) for measuring depth of water with 1.0 molar salt concentration.

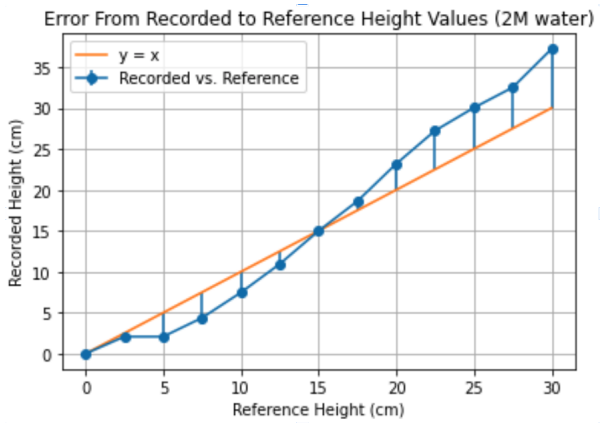


Figure 12: Experimental data (blue) and desired measurements (red) for measuring depth of water with 2.0 molar salt concentration.

Overall, the output data took an arch-shape (bowed-out) when mapped against the reference height. That is, the recorded values underestimate the reference values below 15 cm, and overestimate them above 15 cm. This shape was more pronounced for the salt concentration data sets (0.5, 1.0, and 2.0 M). For the regular water data (Figure 9), the recorded values were very close to the reference ones for lower heights, yet the measured values were heavily overestimated at higher heights. Meanwhile, for the salt concentration data sets (Figures 10, 11, and 12), the recorded values undershoot and overshoot the reference values symmetrically across the 15 cm mark.

The overall errors in the readings from the program are reflected in the Mean-Squared Error of the data for each experiment (Table 1), which is significantly higher for the salt concentration trials than the regular water trial. The original data set and the Python code used to process it is included in Appendix F.

Concentration (M)	0.0	0.5	1.0	2.0
Mean Squared Error (cm_2)	6.47	11.55	12.28	12.57

Table 1: Comparison of the salt concentration of the water used and the Mean Squared Error of the water depth measurements.

Based on similar MSE values for the salt data, we can conclude that the water level sensor operates irrespective of the specific salt concentration. Given that there was a significant difference between the no salt data trend and the salt data trend, the team finds that the presence of salt, though not the concentration of it (at least at these levels), significantly affects the readings.

It is also worthy to note why these trials were conducted at several different salt concentrations. Rice fields are known to have significant concentrations of salt that can vary at changing environmental conditions. Hence the inclusion of these data sets is meant to replicate the minerals' presence and help determine whether salt concentration has any influence on the water level readings.

4.2 Error Correction

In order to improve the accuracy of our data and thereby the accuracy of the sensor, we decided to map a corrective function for the height, which would correct the values outputted by the program, shifting them closer to the reference the data. Upon collecting our data, we input it into a program[2] to find a corrective function of largest significant polynomial power. After reviewing several possible polynomial approximations, across all concentrations, by inspection, we decided that the cubic polynomial, of the form...

$$f(x) = ax^3 + bx^2 + cx + d$$

[2]Credits: https://arachnoid.com/polysolve/

...most reasonably corrected the output data to the desired height data, given suitable coefficients. This decision was made to best capture the overall bowed out shape we described earlier. The data output of the corrective function appeared to roughly match the manually measured reference data. Since the data for all conducted experiments with the salt added to the water showed very similar shape and Mean Squared Error, we constructed two corrective functions: one for correcting depth measurements with regular water and one for correcting depth measurements with 0.5 - 2.0 molar salt water:

$$f_{\text{reg}}(x) = (4.671 \times 10^{-4})\,x^3 - (3.171 \times 10^{-2})\,x^2 + (1.402)\,x - (3.036 \times 10^{-1})$$
$$f_{\text{salt}}(x) = (3.194 \times 10^{-4})\,x^3 - (1.886 \times 10^{-2})\,x^2 + (9.934 \times 10^{-1})\,x + (3.471)$$

After passing the measured height data through the respective correcting function, we see a much closer fit to the desired reference (Figures 13 and 14) while the MSE has decreased to 0.625 and 1.514 respectively for the no-salt and salt mesurement data.

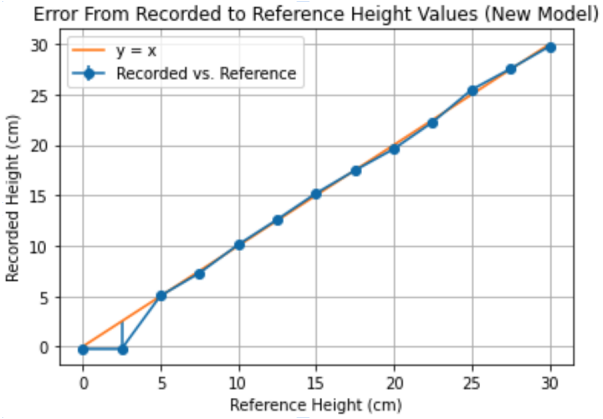


Figure 13: Corrected data (blue) and desired measurements (red) for experimental data of water depth with regular rap water.

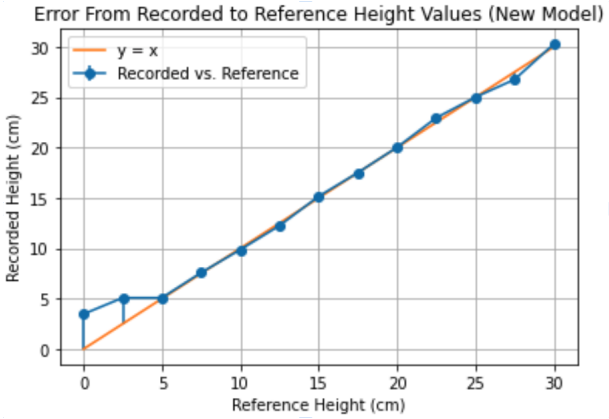


Figure 14: Corrected data (blue) and desired measurements (red) for experimental data of water depth with salt-added water.

As evident in all of the above graphs, we can see that after correction via the proposed functions, the height values output by the programs are much closer to what they should be. While there are other general discrepancies between the reference and recorded heights for other values, it is evident that in the corrected graphs most of the error is centered around the origin. This is attributed to a loss of functionality at the reference height of zero, which is a limitation of the eTape sensor itself. When the data is collected for the reference height of 0 or 2.5-cm, the sensor reports a height of 0-cm, unable to differentiate the two. Because of this and the inability of the loss function to fix the data, the team determined that the minimum height of the water level for tracking with this sensor should be at least 5 centimeters. For heights above 5 cm, the trend lines appear to fit well, with only very slight deviations in values at heights above 20 cm. A corrective function like the ones we provided above can be written into the sensor Arduino code directly to adjust the measured data before sending it to the receiver.

5 Conclusion and Future Work

Ultimately, the group was able to configure a functional water sensor using an Arduino transmitter-receiver pair, NRF24L01+ modules, and an eTape sensor. Using the correction measures as described in the last section, the team was able to synchronize the readings output by the sensor Arduino such that they matched the measurements conducted manually. Through this process, the team revised the Arduino program and was able to ensure that the output was sufficiently accurate.

As it stands now, the product is currently dependent on access to a nearby computer for both the receiver and the sensor. The only need to connect the Arduino units to a computer is to provide a power source and means of data output. In order for the sensor product to be usable in rice fields as was initially intended, compatibility with a remote source of power such as a battery, mini solar panel, or solar-charged battery is required, alongside a small LED monitor for displaying data. Because batteries need to be replaced, the solar energy methods are the more optimal alternatives, as they would be able to provide power for longer periods of time without maintenance. This solution also capitalizes on the high levels of sunlight in major parts of Africa. More generally, the team also believes that the product designed thus far is advantageous because many of its parts can be made to be easily replaceable or repairable; hence if something were to break, one would need not purchase the entire assembly, but rather replace a single part.

Some examples on how the team could implement a feasible form of solar energy would be to use a conventional garden solar cell with a rechargeable battery. There are many examples of this used in daily life which the team could replicate. One could use a standard lead-acid battery, and given the small size of the final product and the typical five year lifespan of the battery, pursuing these would be cost-effective and efficient. Other options would be to use a lithium-ion battery, which is often used in computer and car electronics, or flow batteries, which consist of zinc bromide but have a lower lifespan. Overall, all of these options are fairly clean, cost-efficient, and temperature independent, and should meet the necessary conditions for the team's water sensor.

In addition, another step to be taken in order for the product to be usable in rice fields as desired is to downsize the model. Instead of using the prototype-friendly Arduino board, the use of smaller-sized chips would allow for the product to be smaller and more integrable. We would be able to buy these chips and integrate them within custom PCB's for a much more convenient down-scaled testing model.

These are the two next steps the team would like to work towards in the future. Still, given that the cost of the materials needed for each device is around fifty dollars, which is cheaper than the water sensors used by most industrial rice fields, we believe that there has been satisfactory progress made thus far. We anticipate that even when including the cost of the solar-powered

battery, once built, the overall cost of manufacturing the sensor will be less the price of most water sensors used in rice fields. Even more, working with smaller units, via the downsizing process with smaller chips, would further reduce the cost. Upon producing multiple sensors, the team would complete preliminary testing with the sensors to ensure that the readings output by all the units are standardized and accurate. This would be done by comparing the readings with the manually collected data via the process outlined in the procedure section.

Overall, we believe that in conjunction with practicing AWD, the use of the final product will allow for farmers to conserve significant volumes water and reduce their costs of production, manual labor, and would be feasible for most rice farmers in low-income and low-resource nations.

6 References

[1] Bouman, B. (2009). How much water does rice use? *Rice Today, 8(2),* 28-29.

[2] Bouman B and Tuong To. (2001). Field water management to save water and increase its productivity in irrigated lowland rice. *Agricultural Water Management, 49(1),* 11-30.

[3] Carrijo D., Lundy M.E. and Bruce Linquist. (2017). Rice yields and water use under alternate wetting and drying irrigation: A meta-analysis. *Field Crop Research, 203,* 173-180.

[4] Celeridad, RL. (2019). Five non-mitigation benefits of alternate wetting and drying. Research Program on Climate Change, Agriculture and Food Security. https://ccafs.cgiar.org/news/five-non-mitigation-benefits-alternate-wetting-and-drying#.Xq-4dYKhKhJQ

[5] GRiSP (Global Rice Science Partnership). (2013). Rice almanac, 4th edition. Los Baños (Philippines): International Rice Research Institute.

[6] Lampayan R.M., Yadav S. and Elizabeth Humphreys. (2014). Saving Water with Alternate Wetting Drying (AWD). Rice Knowledge Bank. http://www.knowledgebank.irri.org/training/fact-sheets/water-management/saving-water-alternate-wetting-drying-awd

[7] Linquist B., Anders M.M., Adviento-Borbe M.A.A., Chaney R.L., Nalley L.L., Da Roda E.F.F. and Chris van Kessel. Reducing greenhouse gas emissions, water use, and grain arsenic levels in rice systems. *Glob. Change Biol., 21 (1),* 407-417.

[8] Mekonnen M.M. and Arjen Y., H. (2016). Four billion people facing severe water scarcity.*Sci. Adv., 2 (1),* 1-6.

[9] Neogi M.G., Uddin A.K., Uddin M.T. and Muhammad Hamid. (2018). Alternate wetting and drying (AWD) technology: A way to reduce irrigation cost and ensure higher yields of Boro rice. *J Bangladesh Agril Univ 16(1),* 1-4.

[10] Nigatu G., Hansen J., Childs N. and Ralph Seeley. (2017). Sub-Saharan Africa Is Projected To Be the Leader in Global Rice Imports. United States Department of Agriculture Economic Research Service. https://www.ers.usda.gov/amber-waves/2017/october/sub-saharan-africa-is-projected-to-be-the-leader-in-global-rice-imports/

[11] Pimentel D., Berger D., Filiberto D., Newton M., Wolfe B., Karabinakis E., Clark S., Poon E., Abbett E. and Sudha N. (2004).Water resources: agricultural and environmental issues. *Bioscience, 54 (1),* 909-918.

[12] Richards M. and Bjoern Sanders. (2014). Alternate wetting and drying in irrigated rice: Implementation guidance for policymakers and investors. International Rice Research Institute. https://pdfs.semanticscholar.org/786b/a47ddf6894fcf7c5728546128740916260ba.pdf

[13] Rothenberg S.E., Anders M., Ajami N.J., Petrosino J.F. and Erika Balogh. (2016).Water management impacts rice methylmercury and the soil microbiome. *Sci. Total Environ., 572,* 608-617.

[14] Solar Alliance. (2018). Solar Rechargeable Batteries: How Do They Work? https://www.solaralliance.org/solar-rechargeable-batteries/

Appendices

A Appendix: Pin-out Details for Circuit Schemas

Figure 7:

Here we describe the pin-out for the nRF24L01+ Transceiver module. Note that SPI refers to "Serial Peripheral Interface" and is a protocol used for two-way communication between two devices. In general, it is classified by use of four signals: MOSI (Master Out Slave In), MISO (Master In Slave Out), SCK (The Clock), and SS (Slave Select). Note that for pin assignments in this project, the SS and IRQ pins were not used.

- **GND:** (Ground) is connected to the ground of the system.
- **VCC:** (Power) supplies power to the nRF24L01+ module. In our prototype, it was connected to the 3.3V output of the Arduino.
- **CE:** (Chip Enabled) is an active-HIGH pin that enables SPI communication. It is connected to the D9 PIN on the Arduino unit.
- **CSN:** (Chip Select Not) is an active-LOW pin that is normally kept HIGH to ensure that the SPI is not disabled. It is connected to the D8 PIN on the Arduino unit.
- **SCK:** (Serial Clock) accepts clock pulses from the SPI bus Master, which allows SPI communication to function. It is connected to the D13 PIN on the Arduino unit.
- **MOSI:** (Master Out Slave In) is the SPI input to the nRF24L01+ transceiver module. It is connected to the D11 PIN on the Arduino unit.
- **MISO:** (Master Out Slave In) is the SPI output from the nRF24L01+ transceiver module. It is connected to the D12 PIN on the Arduino unit.
- **IRQ:** (Interrupt) is used when an interrupt is required and can be used to alert users to new data.

Note that when it came to the connection between the nRF24L01+ transceiver module and the Arduino unit, the V_IN PIN on the Arduino unit was not used since the Arduino unit was connected via a USB-A power cord to a computer for power.

For the receiver Arduino, there is a connection between the breadboard and the 5V of the Arduino unit and a connection between the breadboard and a switch. The function of the switch is to control when requests are sent from the receiver hub to the sensor(s) unit(s).

Figure 8:

The pin-out for the nRF24L01+ module to the Arduino is identical to that of Figure 7. Differences arise when considering the breadboard connection to the water sensor. VCC and GND PIN on the water sensor are connected to the breadboard such that resistance can be measured between the two. Namely, the GND pin is connected directly to ground on the breadboard while the VCC pin connects to a 5V-powered line with the A0 cable on that same line. The resistance is then measured by reading the A0 PIN on the Arduino unit. The line is powered by a connection to the 5V breadboard rail by a 2 $k\Omega$ resistor.

B Appendix: Receiver Arduino Code

```
//-------------------------------------------------------------------------------------------------
// EPICS_receiver_code.ino
// Authors: Oleg Golev, Shadman Jahangir, Sumanth Maddirala
//-------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
#include <Button2.h>
//-------------------------------------------------------------------------------

// define the digital pins used
#define CE_PIN   9    // controlling standby mode of the transceiver module
#define CSN_PIN  8    // SPI communication of the transceiver module
#define BUTTON_PIN 4  // button, which when pressed, sends a request for data

// button connected to pin 4
Button2 button = Button2(BUTTON_PIN);

// communication address channels
const byte slaveAddress[5] = {'R','x','A','A','A'};   // for reading sent data
const byte masterAddress[5] = {'T','X','a','a','a'};  // for sending data requests

// create a radio
RF24 radio(CE_PIN, CSN_PIN);

// used to receive the volume, then height data from the transmitter
char dataReceived[20]; // this must match dataToSend in the TX

// array with filler data to send as a "token" to request data
int requestData[2] = {109, -4000};
bool newData = false;

int count = 0;

//-------------------------------------------------------------------------------
//
//  Set up the transceiver.
//
//-------------------------------------------------------------------------------

void setup() {

    // initialize serial communication port and set the baud rate (same on both devices)
    Serial.begin(9600);
    Serial.println("-----------------------------------------------------------------");
    Serial.println("Receiver Starting");
    Serial.println("-----------------------------------------------------------------");

    // define the action handler of the button
    button.setPressedHandler(pressed);

    // set data rate
    radio.begin();
    radio.setDataRate(RF24_250KBPS);
    radio.setRetries(1, 15);  // delay, count

    // open two pipes for bi-directional communication
    radio.openWritingPipe(masterAddress);
    radio.openReadingPipe(1, slaveAddress);
}

//-------------------------------------------------------------------------------
//
//  Continuously check whether the button was pressed.
//
//-------------------------------------------------------------------------------

void loop() {
    button.loop();
}
```

```
//--------------------------------------------------------------------------------
//
//  When the button is pressed, send a request for data, wait for a response,
//  and print the response to Serial Monitor.
//
//--------------------------------------------------------------------------------

void pressed(Button2 &btn) {

  // handles the delayed response issue (requires one press of the button for setup)
  if (count <= 1) {
    count++;
  }

  // otherwise, execute the normal sequence.
  else {
    send();
    delay(5000);
    getData();
    getData();
    Serial.println("------------------------------------------------------------");
  }
}

//--------------------------------------------------------------------------------
//
//  Send a request for data to the receiver, then start listening for a response.
//
//--------------------------------------------------------------------------------

void send() {

    radio.stopListening();
    bool rslt = radio.write(&requestData, sizeof(requestData));

    if (rslt) {
      Serial.println("Request for Data Sent");
    }
    else {
      Serial.println("FAILED: Request for Data");
    }

    radio.startListening();
}

//--------------------------------------------------------------------------------
//
//  Get available data from the reading pipe. This can be either volume or
//  height data passed by the sensor module.
//
//--------------------------------------------------------------------------------

void getData() {
  if ( radio.available() ) {
      radio.read( &dataReceived, sizeof(dataReceived) );
      Serial.print("Data received ");
      Serial.println(dataReceived);
  }
}

//--------------------------------------------------------------------------------
```

C Appendix: Sensor Arduino Code

```
//-------------------------------------------------------------------------------------------------
// EPICS_sensor_code.ino
// Authors: Oleg Golev, Shadman Jahangir, Sumanth Maddirala
//-------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
//-------------------------------------------------------------------------------------------------

// define the pins used for nRF24L01
#define CE_PIN   9    // controlling standy mode
#define CSN_PIN  8    // SPI communication pin

// sensor configuration:
#define SERIES_RESISTOR   2000  // value of the series resistor (ohms) in the circuit
#define SENSOR_PIN        0     // analog pin number to which the sensor is connected

// calibration values when using the 1000-mL 6-cm diameter graduated cylinder:
#define ZERO_VOLUME_RESISTANCE    2400    // resistance (ohms) when no liquid is present
#define CALIBRATION_RESISTANCE    460     // resistance (ohms) when filled
#define CALIBRATION_VOLUME        1000    // filled volume (mL)
float radius = 3.0;                       // radius of the graduated cylinder (cm)

// store the volume and height measurements to send to the receiver
char volumeDataToSend[20];
char heightDataToSend[20];

// blank placeholder array to accept a "token" as a request for data
int dataReceived[2]; // to accept the request for data

// communication address channels
const byte slaveAddress[5] = {'R','x','A','A','A'};   // when writing data
const byte masterAddress[5] = {'T','X','a','a','a'};  // when accepting a data request

// create a Radio
RF24 radio(CE_PIN, CSN_PIN);

//-------------------------------------------------------------------------------------------------
//
//  Set up the transceiver.
//
//-------------------------------------------------------------------------------------------------

void setup(void) {

    // initialize serial communication port and set the baud rate (same on both devices)
    Serial.begin(9600);
    Serial.println("-----------------------------------------------------------------");
    Serial.println("Transmitter Starting");
    Serial.println("-----------------------------------------------------------------");

    // set data rate
    radio.begin();
    radio.setDataRate(RF24_250KBPS);
    radio.setRetries(1, 15);   // delay, count

    // open two pipes for bi-directional communication
    radio.openWritingPipe(slaveAddress);
    radio.openReadingPipe(1, masterAddress);
    radio.startListening();
}

//-------------------------------------------------------------------------------------------------
//
//  Continuously check whether data was requested. If so, process the request by
//  measuring the resistance over the sensor, translate it to volume (mL) and height (cm)
//  data, and finally send it back to the requesting device. If the requesting device
//  attempts to request data too quickly, transmission will fail.
//
//-------------------------------------------------------------------------------------------------
```

```
void loop(void) {

    // continuously check whether there is a request
    if (radio.available()) {

        // read the transmitted message and stop listening
        radio.read( &dataReceived, sizeof(dataReceived) );
        Serial.println("Received a request for data");
        radio.stopListening();

        // measure sensor resistance by averaging 20,000 readings
        float resistance = 0;
        for (int i = 0; i < 20000; i++) {
          resistance += readResistance(SENSOR_PIN, SERIES_RESISTOR);
        }
        resistance = resistance / 20000.0;

        // map resistance to volume.
        float volume = resistanceToVolume(resistance, ZERO_VOLUME_RESISTANCE,
                                          CALIBRATION_RESISTANCE, CALIBRATION_VOLUME);
        volume = correctVolume(volume);

        // map volume to height given the radius of the cylinder
        float height = volume / (3.1415 * radius * radius);

        // format the volume and height data before sending
        String s = "Volume: " + String(volume) + " mL";
        s.toCharArray(volumeDataToSend, sizeof(volumeDataToSend));

        String s2 = "Height: " + String(height) + " cm";
        s2.toCharArray(heightDataToSend, sizeof(heightDataToSend));

        // send the requested data
        send();

        // log the data that is being sent
        Serial.print("Resistance: ");
        Serial.print(resistance, 2);
        Serial.println(" ohms");
        Serial.print("Calculated volume: ");
        Serial.println(volume, 5);
        Serial.print("Calculated height: ");
        Serial.println(height, 5);
        Serial.println("-------------------------------------------------------------");
      }
}

//-----------------------------------------------------------------------------------------
//
//  Send volume, then height data in two separate messages. Keep track of whether these
//  transmission failed and print the appropriate success / failure message to serial
//  monitor. After transmission is done, set the transceiver to accept requests again.
//
//-----------------------------------------------------------------------------------------

void send() {

    // boolean return value to keep track of transmission failures
    bool retVolume;
    bool retHeight;

    // sending volume and height
    retVolume = radio.write(&volumeDataToSend, sizeof(volumeDataToSend));
    retHeight = radio.write(&heightDataToSend, sizeof(heightDataToSend));

    // log results of the transmission of volume
    if (retVolume) {
        Serial.println("Acknowledge received: volume successfully sent");
    }
    else {
        Serial.println("Transmission FAILURE: volume");
    }

    // log results of the transmission of height
    if (retHeight) {
```

```
        Serial.println("Acknowledge received: height successfully sent");
    }
    else {
        Serial.println("Transmission FAILURE: height");
    }

    // start listening for requests again
    radio.startListening();
}

//-------------------------------------------------------------------------------------------
//
//  Take in the volume calculated from the resistance and apply a correcting function
//  (constructed based on prior empirical testing data) to the volume measurement:
//  y = 10.2 + 1.19 * x - 0.000455 * x^2 + 0.0000000569 * x^3
//
//-------------------------------------------------------------------------------------------

float correctVolume(float vol) {
  return (10.1619 + 1.1863 * vol - 0.00045455 * pow(vol, 2) + 0.00000005687 * pow(vol, 3));
}

//-------------------------------------------------------------------------------------------
//
//  Read resistance at the specified analog pin.
//
//  This code was purposed from a public GitHub resource, released under an MIT license:
//  https://github.com/tdicola/SmartMeasuringCup/blob/master/YunSmartMeasuringCupSketch/YunSmartMeasuringCupSketch.ino
//
//-------------------------------------------------------------------------------------------

float readResistance(int pin, int seriesResistance) {

    // get ADC value.
    float resistance = analogRead(pin);

    // convert ADC reading to resistance.
    resistance = (1023.0 / resistance) - 1.0;
    resistance = seriesResistance / resistance;
    return resistance;
}

//-------------------------------------------------------------------------------------------
//
//  Convert measured resistance across the 12" eTape sensor into volume.
//  Since the graduated cylinder has a consistent cross section, the change in water
//  height is directly proportional to the measured change in volume.
//
//  This code was purposed from a public GitHub resource, released under an MIT license:
//  https://github.com/tdicola/SmartMeasuringCup/blob/master/YunSmartMeasuringCupSketch/YunSmartMeasuringCupSketch.ino
//
//-------------------------------------------------------------------------------------------

float resistanceToVolume(float resistance, float zeroResistance, float calResistance, float calVolume) {

    if (resistance > zeroResistance || (zeroResistance - calResistance) == 0.0) {
    // Stop if the value is above the zero threshold, or no max resistance is set (would be divide by zero).
        return 0.0;
    }

    // Compute scale factor by mapping resistance to 0...1.0+ range relative to maxResistance value.
    float scale = (zeroResistance - resistance) / (zeroResistance - calResistance);

    // Scale maxVolume based on computed scale factor.
    return calVolume * scale;
}

//-------------------------------------------------------------------------------------------
```

D Appendix: Receiver Log Example

```
20:08:16.883 -> ------------------------------------------------------------
20:08:16.952 -> Receiver Starting
20:08:16.952 -> ------------------------------------------------------------
20:36:32.949 -> Request for Data Sent
20:36:37.950 -> Data received Volume: 41.55 mL
20:36:37.983 -> Data received Height: 1.47 cm
20:36:38.016 -> ------------------------------------------------------------
20:36:38.460 -> Request for Data Sent
20:36:43.454 -> Data received Volume: 35.90 mL
20:36:43.487 -> Data received Height: 1.27 cm
20:36:43.522 -> ------------------------------------------------------------
20:36:44.273 -> Request for Data Sent
20:36:49.265 -> Data received Volume: 27.83 mL
20:36:49.300 -> Data received Height: 0.98 cm
20:36:49.334 -> ------------------------------------------------------------
20:38:42.751 -> FAILED: Request for Data
20:38:56.070 -> ------------------------------------------------------------
20:38:58.159 -> Request for Data Sent
20:39:03.187 -> Data received Volume: 10.20 mL
20:39:03.221 -> Data received Height: 0.36 cm
20:39:03.221 -> ------------------------------------------------------------
20:39:07.092 -> Request for Data Sent
20:39:12.085 -> Data received Volume: 10.20 mL
20:39:12.119 -> Data received Height: 0.36 cm
20:39:12.153 -> ------------------------------------------------------------
20:39:14.623 -> Request for Data Sent
20:39:19.618 -> Data received Volume: 10.20 mL
20:39:19.653 -> Data received Height: 0.36 cm
20:39:19.687 -> ------------------------------------------------------------
20:39:39.048 -> Request for Data Sent
20:39:44.024 -> Data received Volume: 221.42 mL
20:39:44.058 -> Data received Height: 7.83 cm
20:39:44.093 -> ------------------------------------------------------------
20:39:45.939 -> Request for Data Sent
20:39:50.967 -> Data received Volume: 231.79 mL
20:39:51.000 -> Data received Height: 8.20 cm
20:39:51.035 -> ------------------------------------------------------------
20:39:55.306 -> Request for Data Sent
20:40:00.296 -> Data received Volume: 234.60 mL
20:40:00.330 -> Data received Height: 8.30 cm
20:40:00.365 -> ------------------------------------------------------------
20:40:03.957 -> Request for Data Sent
20:40:08.990 -> Data received Volume: 239.33 mL
20:40:09.023 -> Data received Height: 8.46 cm
20:40:09.056 -> ------------------------------------------------------------
20:40:13.234 -> Request for Data Sent
20:40:18.238 -> Data received Volume: 235.52 mL
20:40:18.274 -> Data received Height: 8.33 cm
20:40:18.307 -> ------------------------------------------------------------
20:40:19.500 -> Request for Data Sent
20:40:24.521 -> Data received Volume: 234.16 mL
20:40:24.556 -> Data received Height: 8.28 cm
20:40:24.590 -> ------------------------------------------------------------
20:40:26.336 -> Request for Data Sent
20:40:31.328 -> Data received Volume: 230.77 mL
20:40:31.363 -> Data received Height: 8.16 cm
20:40:31.396 -> ------------------------------------------------------------
20:40:44.344 -> Request for Data Sent
20:40:49.367 -> Data received Volume: 395.79 mL
20:40:49.400 -> Data received Height: 14.00 cm
20:40:49.433 -> ------------------------------------------------------------
20:40:50.599 -> Request for Data Sent
20:40:55.635 -> Data received Volume: 420.17 mL
20:40:55.635 -> Data received Height: 14.86 cm
20:40:55.670 -> ------------------------------------------------------------
20:40:57.112 -> Request for Data Sent
20:41:02.126 -> Data received Volume: 425.20 mL
20:41:02.159 -> Data received Height: 15.04 cm
20:41:02.194 -> ------------------------------------------------------------
20:41:20.383 -> Request for Data Sent
20:41:25.387 -> Data received Volume: 506.17 mL
20:41:25.420 -> Data received Height: 17.90 cm
```

E Appendix: Sensor Log Example

```
20:47:47.271 -> ------------------------------------------------------------
20:47:47.342 -> Transmitter Starting
20:48:17.180 -> ------------------------------------------------------------
20:48:31.099 -> Received a request for data
20:48:34.924 -> Acknowledge received: volume successfully sent
20:48:34.958 -> Acknowledge received: height successfully sent
20:48:35.030 -> Resistance: 2362.60 ohms
20:48:35.030 -> Calculated volume: 32.86572
20:48:35.068 -> Calculated height: 1.16242
20:48:35.104 -> ------------------------------------------------------------
20:52:50.956 -> Received a request for data
20:52:54.832 -> Acknowledge received: volume successfully sent
20:52:54.869 -> Acknowledge received: height successfully sent
20:52:54.906 -> Resistance: 2287.22 ohms
20:52:54.943 -> Calculated volume: 77.59830
20:52:54.978 -> Calculated height: 2.74456
20:52:55.011 -> ------------------------------------------------------------
20:53:18.395 -> Received a request for data
20:53:22.171 -> Acknowledge received: volume successfully sent
20:53:22.240 -> Acknowledge received: height successfully sent
20:53:22.274 -> Resistance: 2114.62 ohms
20:53:22.311 -> Calculated volume: 175.01324
20:53:22.349 -> Calculated height: 6.19001
20:53:22.383 -> ------------------------------------------------------------
20:57:35.871 -> Received a request for data
20:57:39.635 -> Acknowledge received: volume successfully sent
20:57:39.669 -> Acknowledge received: height successfully sent
20:57:39.704 -> Resistance: 1997.04 ohms
20:57:39.738 -> Calculated volume: 237.46777
20:57:39.771 -> Calculated height: 8.39895
20:57:39.807 -> ------------------------------------------------------------
20:58:48.989 -> Received a request for data
20:58:52.836 -> Acknowledge received: volume successfully sent
20:58:52.871 -> Acknowledge received: height successfully sent
20:58:52.946 -> Resistance: 1822.73 ohms
20:58:52.946 -> Calculated volume: 324.41024
20:58:52.984 -> Calculated height: 11.47400
20:58:53.021 -> ------------------------------------------------------------
20:59:45.047 -> Received a request for data
20:59:48.920 -> Acknowledge received: volume successfully sent
20:59:48.953 -> Acknowledge received: height successfully sent
20:59:49.023 -> Resistance: 1614.16 ohms
20:59:49.023 -> Calculated volume: 419.89508
20:59:49.058 -> Calculated height: 14.85119
20:59:49.093 -> ------------------------------------------------------------
21:02:50.338 -> Received a request for data
21:02:54.178 -> Acknowledge received: volume successfully sent
21:02:54.249 -> Acknowledge received: height successfully sent
21:02:54.286 -> Resistance: 1366.15 ohms
21:02:54.324 -> Calculated volume: 521.87310
21:02:54.360 -> Calculated height: 18.45803
21:02:54.360 -> ------------------------------------------------------------
21:03:34.986 -> Received a request for data
21:03:38.829 -> Acknowledge received: volume successfully sent
21:03:38.896 -> Acknowledge received: height successfully sent
21:03:38.934 -> Resistance: 1167.75 ohms
21:03:38.969 -> Calculated volume: 594.86187
21:03:39.003 -> Calculated height: 21.03956
21:03:39.041 -> ------------------------------------------------------------
21:05:30.252 -> Received a request for data
21:05:34.078 -> Acknowledge received: volume successfully sent
21:05:34.113 -> Acknowledge received: height successfully sent
21:05:34.183 -> Resistance: 915.16 ohms
21:05:34.218 -> Calculated volume: 677.35266
21:05:34.218 -> Calculated height: 23.95716
21:05:34.254 -> ------------------------------------------------------------
21:08:30.018 -> Received a request for data
21:08:33.764 -> Acknowledge received: volume successfully sent
21:08:33.835 -> Acknowledge received: height successfully sent
21:08:33.873 -> Resistance: 546.73 ohms
21:08:33.910 -> Calculated volume: 778.19030
21:08:33.945 -> Calculated height: 27.52367
```

F Appendix: Data Analysis

Below you will find the Python 3.7 script used to parse, clean, and visualize the following data (Figure 13) taken directly using the sensor-receiver Arduino setup described in the paper:

A	B	C	D	E	F	G
Volume (mL)	0	71.5	143	215	287	358
Height (cm)	0	2.5	5	7.5	10	12.5
Volume (mL)	2.07	2.07	118.18	175.69	255.32	333.96
Height (cm)	0.05	0.05	4.18	6.21	9.03	11.81
Resistance (ohm)	2316728	2316728	2117043	2018286	1881519	1746482
Volume (mL)	0.00	38.76532	38.76532	127.73089	210.25723	301.11737
Height (cm)	0.00	1.37108	1.37108	4.51769	7.43655	10.65016
Resistance (ohm)	2400000.50	253426.50 ohms	253426.50 ohms	100641.75 ohms	958915.37 ohms	802877.00 ohms
Volume (mL)	0	44.6459	44.6459	129.41851	200.26144	286.70401
Height (cm)	0	1.57907	1.57907	4.57738	7.08301	10.14038
Resistance (ohm)	401799.50 ohms	243327.50 ohms	243327.50 ohms	097743.50 ohms	976081.62 ohms	827629.75 ohms
Volume (mL)	0	58.92519	58.92519	123.79241	211.0787	309.49874
Height (cm)	0	2.08411	2.08411	4.37839	7.4656	10.9466
Resistance (ohm)	408750.00 ohms	218805.00 ohms	218805.00 ohms	107405.50 ohms	957504.62 ohms	788483.25 ohms
Volume (mL)	10.1619	10.1619	81.68025	170.78224	237.46777	324.5834
Height (cm)	0.35941	0.35941	2.88893	6.04036	8.39895	11.48013
Resistance (ohm)	2400.01	2400.29	2280.23	2122.38	1997.04	1822.37

H	I	J	K	L	M	N
430	501	573	644	716	787	860
15	17.5	20	22.5	25	27.5	30
425.44	512.64	599.16	708.72	835.14	910.8	984.21
15.05	18.13	21.19	25.07	29.54	32.21	34.81
1589368	1439623	1291026	1102879	885778	755845	629762
425.42401	528.80194	631.44995	776.04376	849.71313	913.98400	1019.20788
15.04674	18.70310	22.33363	27.44775	30.05334	32.32653	36.04817
1589399.37 ohms	411863.62 ohms	235581.37 ohms	987263.62 ohms	860747.68 ohms	750372.37 ohms	569666.43 ohms
416.49533	526.9746	636.42047	759.85876	855.24157	919.60437	1037.26464
14.73094	18.63846	22.50944	26.8753	30.24888	32.52531	36.68682
1604733.00 ohms	415001.75 ohms	227045.25 ohms	015058.87 ohms	851253.50 ohms	740720.18 ohms	538656.75 ohms
423.92236	526.56323	655.24853	770.01336	849.03967	919.08892	1051.9945
14.99363	18.62392	23.17536	27.23446	30.02952	32.50708	37.20779
1591978.25 ohms	415708.25 ohms	194711.00 ohms	997619.87 ohms	861904.37 ohms	741605.50 ohms	513360.56 ohms
427.25613	487.90667	522.1665	626.24047	676.5122	742.95343	778.1903
15.11154	17.25668	18.46841	22.14938	23.92743	26.27738	27.52367
1597.21	1452.12	1365.39	1075.71	917.92	684.94	546.73

Figure 13: Raw data taken by the water sensor Arduino. For each of the four trials (0.0 M, 0.5 M, 1.0 M, and 2.0 M salt concentrations), the data consists of the respective resistance (Ohm), volume (mL), and water depth (cm) measurements.

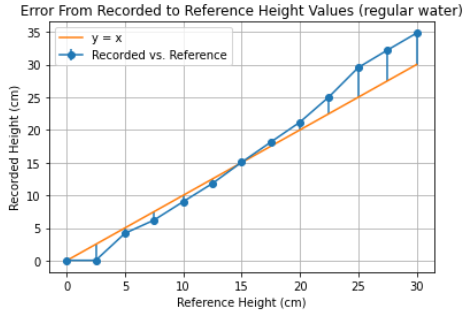
EPICS Data Analysis

```
import numpy as np
import csv
from collections import defaultdict

# rows of our data
KEYS = ["V_REF", "H_REF", "V_REG", "H_REG", "R_REG", "V_0.5", "H_0.5", "R_0.5",
        "V_1", "H_1", "R_1", "V_2", "H_2", "R_2", "V_COR", "H_COR", "R_COR", "R_COR"]

# parses resistance values into usable form
def parseRes(row):
  values_list = [float("{0:.4g}".format(float(val.split()[0]))) for val in row[1:]]
  standardized = [val//1000 if val > 5000 else val for val in values_list]
  return standardized

# storing data into our dictionary
data = defaultdict(list)
with open("/content/drive/My Drive/EPICS 2019-2020/Analysis/data_cleaned.csv") as file:
    csv_reader = csv.reader(file, delimiter=',')
    fill = 0
    for row in csv_reader:
      new_data = []
      if (fill > 1) and (fill - 1) % 3 == 0:
        new_data = parseRes(row)
      else:
        new_data = [float("{0:.4g}".format(float(val))) for val in row[1:]]
      data[KEYS[fill]] = new_data
      fill += 1


# print out all data
for key in data.keys():
  print(str(key) + ":\t", end="")
  print(data[key])
```

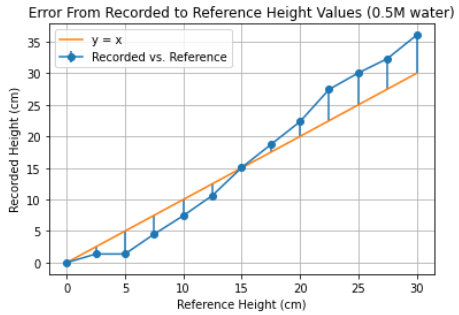
```
V_REF:  [0.0, 71.5, 143.0, 215.0, 287.0, 358.0, 430.0, 501.0, 573.0, 644.0, 716.0, 787.0, 860.0]
H_REF:  [0.0, 2.5, 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0, 22.5, 25.0, 27.5, 30.0]
V_REG:  [2.07, 2.07, 118.2, 175.7, 255.3, 334.0, 425.4, 512.6, 599.2, 708.7, 835.1, 910.8, 984.2]
H_REG:  [0.05, 0.05, 4.18, 6.21, 9.03, 11.81, 15.05, 18.13, 21.19, 25.07, 29.54, 32.21, 34.81]
R_REG:  [2317.0, 2317.0, 2117.0, 2018.0, 1882.0, 1746.0, 1589.0, 1440.0, 1291.0, 1103.0, 885.0, 755.0, 629.0]
V_0.5:  [0.0, 38.77, 38.77, 127.7, 210.3, 301.1, 425.4, 528.8, 631.4, 776.0, 849.7, 914.0, 1019.0]
H_0.5:  [0.0, 1.371, 1.371, 4.518, 7.437, 10.65, 15.05, 18.7, 22.33, 27.45, 30.05, 32.33, 36.05]
R_0.5:  [2400.0, 2253.0, 2253.0, 2101.0, 1959.0, 1803.0, 1589.0, 1412.0, 1236.0, 987.0, 860.0, 750.0, 569.0]
V_1:    [0.0, 44.65, 44.65, 129.4, 200.3, 286.7, 416.5, 527.0, 636.4, 759.9, 855.2, 919.6, 1037.0]
H_1:    [0.0, 1.579, 1.579, 4.577, 7.083, 10.14, 14.73, 18.64, 22.51, 26.88, 30.25, 32.53, 36.69]
R_1:    [2402.0, 2243.0, 2243.0, 2098.0, 1976.0, 1828.0, 1605.0, 1415.0, 1227.0, 1015.0, 851.0, 740.0, 538.0]
V_2:    [0.0, 58.93, 58.93, 123.8, 211.1, 309.5, 423.9, 526.6, 655.2, 770.0, 849.0, 919.1, 1052.0]
H_2:    [0.0, 2.084, 2.084, 4.378, 7.466, 10.95, 14.99, 18.62, 23.18, 27.23, 30.03, 32.51, 37.21]
R_2:    [2409.0, 2219.0, 2219.0, 2107.0, 1958.0, 1788.0, 1592.0, 1416.0, 1195.0, 997.0, 861.0, 741.0, 513.0]
V_COR:  [10.16, 10.16, 81.68, 170.8, 237.5, 324.6, 427.3, 487.9, 522.2, 626.2, 676.5, 743.0, 778.2]
H_COR:  [0.3594, 0.3594, 2.889, 6.04, 8.399, 11.48, 15.11, 17.26, 18.47, 22.15, 23.93, 26.28, 27.52]
R_COR:  [2400.0, 2400.0, 2280.0, 2122.0, 1997.0, 1822.0, 1597.0, 1452.0, 1365.0, 1076.0, 917.9, 684.9, 546.7]
```

```
from matplotlib import pyplot as plt
from sklearn.metrics import mean_squared_error

# plotting height data and errors
def customGraph(x, y, title):

    yerr = [x[i] - y[i] for i in range(0, len(x))]  # vertical errors
    zeros = np.zeros(len(x))                        # horizontaal errors

    # plot the data with error bars
    fig, ax = plt.subplots()
    ax.errorbar(x, y, xerr=None, yerr=(zeros, yerr), fmt='-o')
```

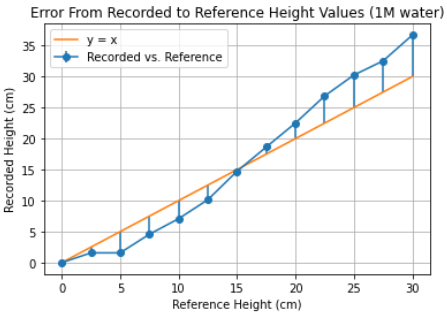
```
    # plot the reference y = x line
    plt.plot(x, x)

    # define the look of the graph
    ax.grid()
    ax.set_xlabel('Reference Height (cm)')
    ax.set_ylabel('Recorded Height (cm)')
    ax.set_title(title)
    plt.legend(['y = x', 'Recorded vs. Reference'], loc='upper left')

    # show the graph
    plt.show()

    # print the mean-squared error
    print("Mean Squared Error: " + "{0:.4g}".format(mean_squared_error(x, y)))

# graphing reference to record height for regular water
customGraph(data['H_REF'], data['H_REG'], 'Error From Recorded to Reference Height Values (regular water)')
```




```
Mean Squared Error: 6.47

# graphing reference to record height for 0.5 molar water
customGraph(data['H_REF'], data['H_0.5'], 'Error From Recorded to Reference Height Values (0.5M water)')
```

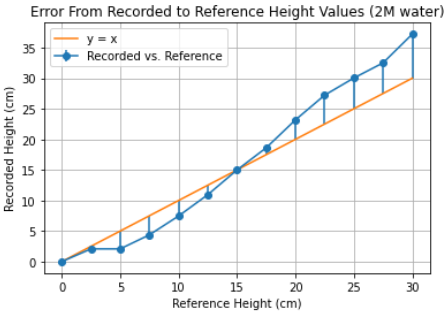



```
Mean Squared Error: 11.55

# graphing reference to record height for 1 molar water
customGraph(data['H_REF'], data['H_1'], 'Error From Recorded to Reference Height Values (1M water)')
```



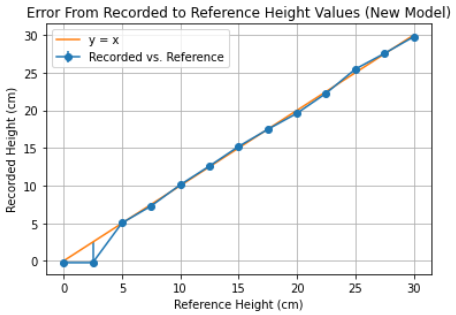

```
Mean Squared Error: 12.28
```

```
# graphing reference to record height for 2 molar water
customGraph(data['H_REF'], data['H_2'], 'Error From Recorded to Reference Height Values (2M water)')
```



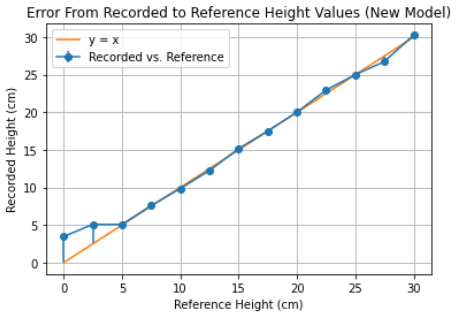

```
Mean Squared Error: 12.57
```

```
# TESTING OTHER MODELS TO MINIMIZE ERROR
# ---------------------------------------

# CORRECTING THE EXPERIMENTAL DATA WITH REGULAR WATER

# model written in lambda form as follows
f = lambda X: -3.036 * pow(10, -1) + X * 1.402 + -X**2 * 3.171 * pow(10, -2) + \
              X**3 * 4.671 * pow(10, -4)

# this vectorizes the model so that it can be applied to each element of a list
f = np.vectorize(f)

# this runs the regular height data through the model
corrected_data = f(data['H_REG'])

# this graphs the new corrected data against what it should be
customGraph(data['H_REF'], corrected_data, 'Error From Recorded to Reference Height Values (New Model)')
```




```
Mean Squared Error: 0.625
```

```
# CORRECTING ALL EXPERIMENTAL DATA WITH SALT ADDED

# model written in lambda form as follows
f = lambda X: 3.471 + X * 9.934 * pow(10, -1) - X**2 *1.886 * pow(10, -2) + X**3 * 3.194 * pow(10, -4)

# this vectorizes the model so that it can be applied to each element of a list
f = np.vectorize(f)

# average all height data points together into one list for data with salt added
salt_data = [(data['H_0.5'][i] + data['H_1'][i] + data['H_2'][i]) / 3.0 for i in range(len(data['H_0.5']))]

# this runs the regular height data through the model
corrected_data = f(salt_data)

# this graphs the new corrected data against what it should be
customGraph(data['H_REF'], corrected_data, 'Error From Recorded to Reference Height Values (New Model)')
```




```
Mean Squared Error: 1.514
```