Developing a Raspberry Pi based IoT Sensor for Flood Sensing

Paul Oscai

*#Computer Science ,College of Engineering, Tennessee Technological University  
110 University Drive Cookeville. TN USA*

1poscai42@tntech.edu

## Introduction

The project aims to implement a flood detection system that uses a camera connected to a Raspberry Pi. The project designers ensured that the system is cost-effective, scalable, easy to maintain, and innovative.

This report will discuss each section of the project. It starts with a problem statement that highlights the significance of the project and explains the design decisions made. It also explains how the components were selected and how they work together.

During the course of the project, the team faced several challenges that needed to be overcome. One of the biggest challenges was the lack of battery capacity, which affected the device's performance. This report explains the causes of this issue and presents the possible solutions that were implemented to improve the system's performance.

Next, the report discusses the integration of the LTE module with the raspberry PI.

Next, the report discusses some of the shortcomings and major issues encountered during the project, such as the impact of sunlight on the solar panel and how the team overcame these issues using the hardware and software setup developed.

Finally, the report discusses how the image data was uploaded to the cloud for storage and statistical measurements. It also explains how the system can access data and respond to a flood in almost real-time, provided that a mechanism is set up for alerts.

In conclusion, this report provides comprehensive insights into a flood detection project that utilizes a camera connected to a Raspberry Pi. The hope is that this report sparks an interest in this important area and provides valuable information about this project.

## System Design

This paper aims to provide an overview of the system's component connections without delving into detailed schematic diagrams. Instead, it focuses on the essential inputs to the system, with power supplied to the peripherals. While there are unused inputs and outputs, particularly on the PiJuice module, the included pictures illustrate the basic overall components without detailing the assembly of the modules.

It's important to note that the HATs connect through the GPIO pins, where the Raspberry Pi receives both input and output power. In our project, we supplied input power to the MicroUSB input. The PiJuice Hat diagram excludes the J6 EEPROM programming connector and the J7 firmware programming connector.

This paper covers the device's power capabilities, battery life, wireless connectivity through the cellular HAT, and initial tests with a solar panel. We further integrated the hardware and software, and developed autonomous functions, but these tasks were completed after assembling the device.

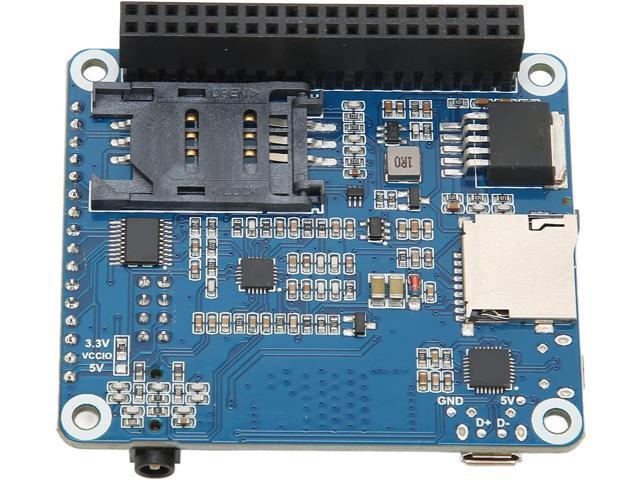
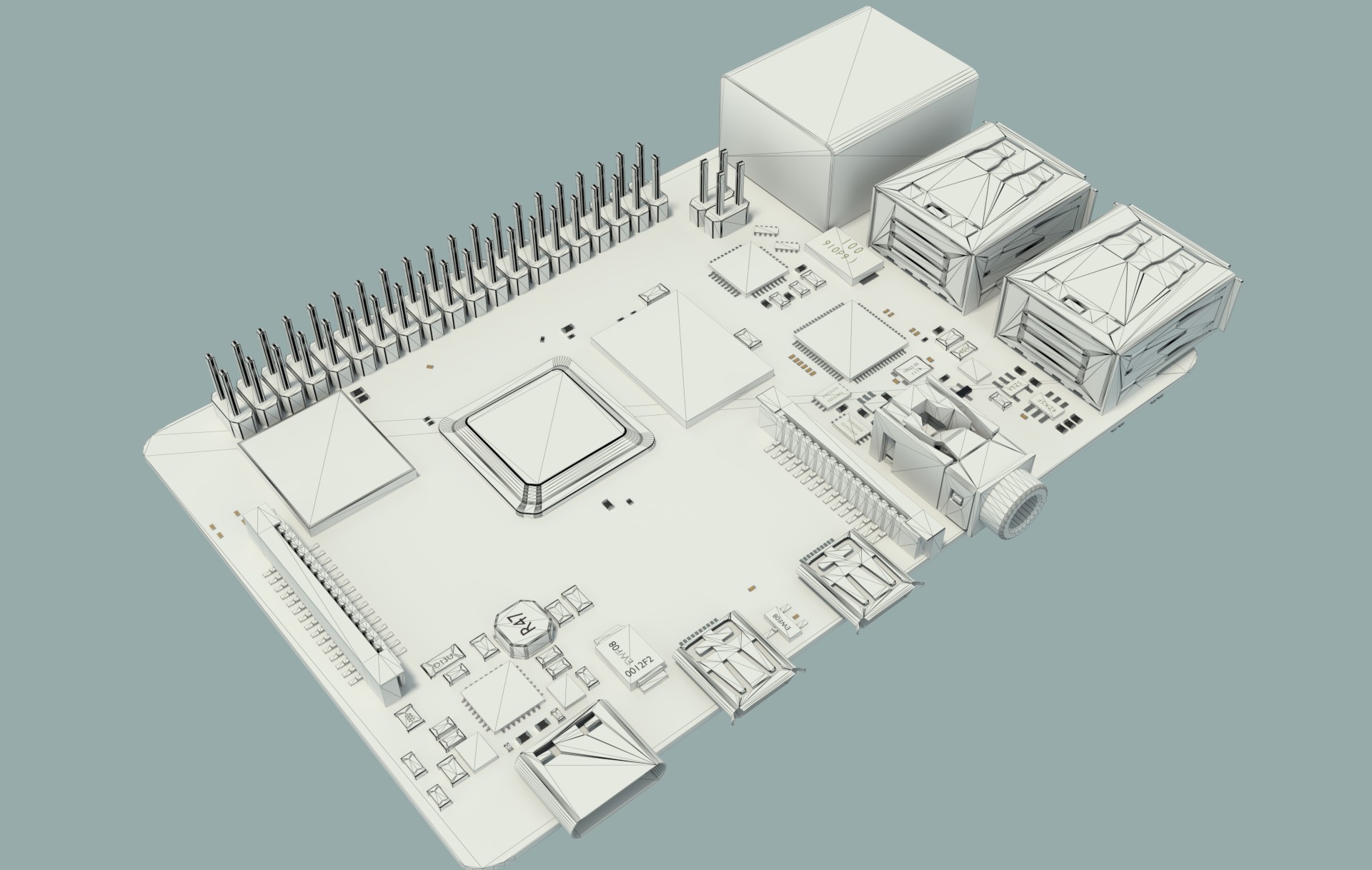


Figure 1 left: Raspberry Pi 4 board showing USB ports, power, camera attachment,

Figure 2 right: Cellular LTE HAT used in this project.

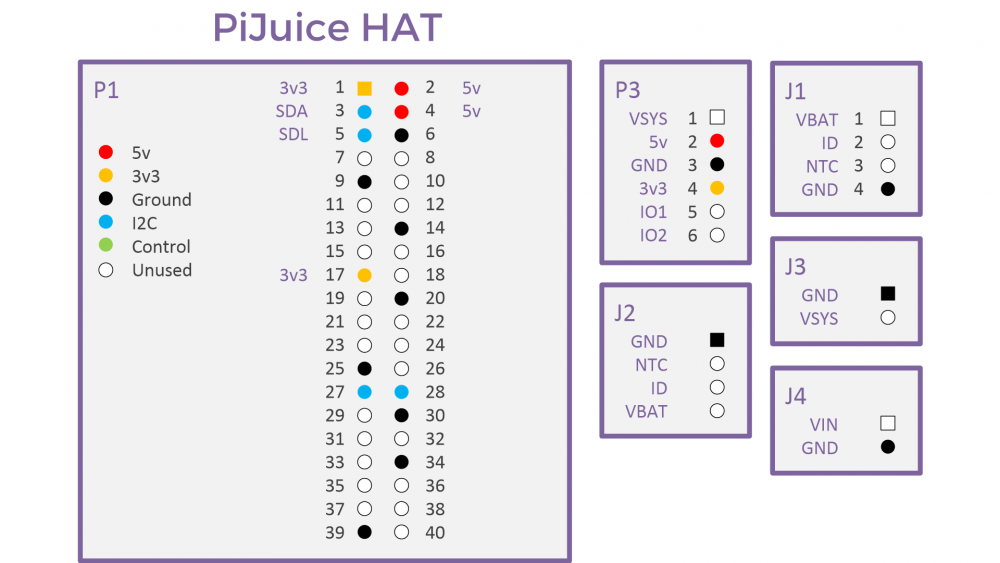


Figure 3: PiJuice Hat pinout for connectors and main board.

## Integrating the LTE module in the System

This section provides details about one of the components used in the project and the order of tasks completed, including the developments and the direction of the project. My goal was to create a reliable, low-cost, and high-throughput device that could perform the required function when placed at a location.

To achieve this goal, I conducted technical information gathering to determine the necessary modules to bring together with the Raspberry Pi 4 device. I considered various factors, such as cost, timeliness, and educational usefulness. For instance, a low budget and cellular connection in the area were crucial considerations. Although a slightly higher cost cellular module could have offered us unseen benefits due to its added frequency range and availability of carriers, I chose a cellular HAT for the device.

After researching the network availability of popular cellular modules, I discovered that only the SIM7600G-H 4G LTE Cat-5 HAT could provide us with the choice of a network carrier and compatibility with global frequency bands from major carriers. I did not want to use a module that would limit our choice of networks. I wanted a better response should we have to change network carriers or modules. Therefore, I chose the more lenient global frequency option as I was confident that we would have a greater choice when it came to buying the SIM card.

After ordering the module, I tested it in the laboratory with a T-Mobile SIM card by installing the necessary software to install a ppp network interface. I also had to fit the module with a SIM adapter for the current SIM card we were testing. Initially, I gathered evidence that T-Mobile connections would give us a working ppp connection.

The next step was to configure the network to automatically run the connection script and create the Linux interface autonomously. Making the device run the program while connected to the network required a few more tasks and some testing to be completed. First, I had to automatically connect the cellular device through a Linux script, which turned out to be an interesting development for a few reasons.

One, the pon (cellular modem) can fail and retry on its own without any help, but it works best when only called once. Two, the task of creating the route can be done automatically when using the command “pon provider.” Three, the connection can be established during the boot and run a program using cellular connection.

Once the device is booted and a monitor is detected, the connection will drop, requiring a user to re-establish the connection. The automatic connection can be established in as little time as a normal user interface such as WiFi, but with a terminal, the connection takes roughly 10 times longer or 30 seconds. I did not extensively test this because we will not deploy the device with a monitor.

## Integrating the PiJuice and Solar Panel in the system

In this project, I will now discuss the testing I carried out on the PiJuice module before utilizing it as an automatic sensor. The PiJuice module is an uninterruptible power supply that integrates a charge controller to convert voltage levels to the input used by the Raspberry Pi. The module has several important features, such as powering on and off based on time, responding to hardware readings, saving settings status, and providing essential information.

Initially, I conducted tests without the cellular HAT assembled, and the device would turn on, run the flood program, and shut down, recording its count and time for every five runs. The device kept time and recorded its last count with 15% battery remaining. I discovered that the performance difference between an idle device and one continuously running the flood program was not significant. However, I noticed that the device could not reboot with four USB devices plugged in, and the maximum was two. Sometimes the device could work when all USB devices were plugged in while running, but it would automatically shut down when trying to run the flood program, which uses the camera.

Further tests were carried out on the charging ability of the battery and solar panel when I received the 6V 3.5W solar panel. While the solar panel was somewhat effective, it did not match the current drawn from the RPI with all the attached peripherals. Hence, I had to power down the Pi to charge the battery with it.

To connect remotely to the device and carry out initial tests, I installed an SSH server and VNC server, as shown in the images. It is crucial to ensure that the connection is working correctly before running other programs and scripts. A connection test can help in that respect. Finally, it is essential to note that some programs, such as a VNC server, work incorrectly when run more than once.

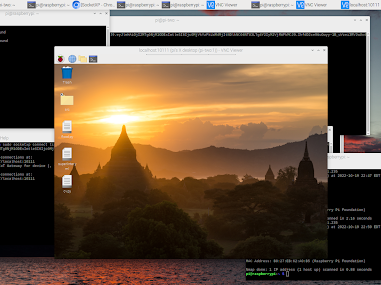


Figure 4 left: Showing what charging device looks like: LEDS can be disabled for power reasons.

Figure 5 right: The Pi can be accessed through a VNC server.

## Uploading Data to Google Cloud

FEWS is designed to upload information to the cloud for storage to be able to visually verify data and use for record keeping. This would require a program to run, eventually automatically, to process image data gathered at the device and be able to upload the information to the cloud.

The Python client library sends a request to the Google Oauth Authorization Server. The service account key creates a signed JWT request for an access token which is returned from the server. The program then accesses Google Drive to upload and access files [10].

Some choices of Cloud IoT are very organized with convenient features like APIs, device monitoring and webhooks. Using the particle API had a limitation that our project would not be able to work with - that being the data limit for a publish event. We had to do the task using Google Drive or Cloud Storage API. Cloud storage API can respond to upload events by triggering a Publish/Subscribe event. Cloud Functions can be activated by accessing a URL. We can use Functions to implement other services. Google Drive uploads responses cannot be as easily configured except with a Google Workspace Admin Account. We can still use Google Drive and Cloud Subscriptions as two separate services for this project.

The ideas for Cloud Events to use sensor information is a topic discussed later in the paper. We can use a simple notification service to alert cell phones using a Twilio webhook.

To describe a way to improve the performance of the device when operating autonomously, I will give an example of programming with Python. Programs can be run in wrapper scripts with the Python subprocess import. Output and errors from our programs can be read with return codes. The wrapper script can keep a file open and append it with error messages so we know when the device is malfunctioning. An instance of a hard to find error is an SSL error that only happens when the device initially boots after losing power and is not synced with the system clock.

To understand performance of interaction between PiJuice and the OS, we use the PiJuice system task to respond to minimum voltage and low charge errors. We seemed to have learned more with the device deployed with a stronger solar panel to understand why the device would have issues.

## Lessons learned

Results obtained from this project will be discussed in two broad categories. One major topic of this section is the ability of the project to gather data steadily. The other topic is about the findings based on the status of flood presence, what I can improve on, and the topics of future work.

To start with, I performed simple battery life tests as described before. Once I was sure that the device was in a testable state, we placed the device near the water source at the Shipley Farm. Initial data with a stronger solar panel was collected and organized into the following chart:

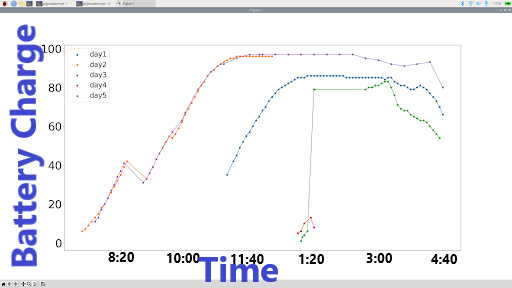


Figure 6: Battery charging percentage and times of day. Each graph is a different day. The device would lose power once capacity got lower.

The chart shows that the battery is able to fully charge in about 2 hours. There are a number of useful features with the PiJuice that help take readings and enable the device to work. Some of these features become cleared when power is completely lost in the device. Some settings still unexpectedly reset. With an automatic ssh connection in the operating system, it is possible to set some of these settings again when the device is powered on. The device has to be shut down through the command layer, removing power to the device and shutting down the operating system. The forced shutdown will cause the battery to die faster. The features that are useful for me are the power settings for BatteryCurrent and BatteryVoltage, ChargingEnabled, ChargeLevel. Some settings proved to be less useful in my scenario, namely WakeupOnCharge and MinimumVoltage and MinCharge.

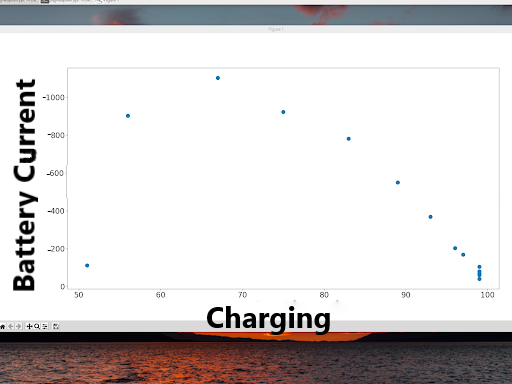


Figure 6: Battery current at different charges.

In the initial tests, I saw signs that the device would require careful attention to be able to operate autonomously. Not being able to reboot from battery power when battery capacity was very low (around 10%) seemed to be some kind of safety mechanism from the PiJuice. I realized that the device could operate with a level of voltage on battery power, but still not be able to reboot. This indicated that there must either be power or a sufficient battery level in order to reboot the device.

Moreover, initial solar power and remote deployment tests were done without using the cellular HAT which was in the idea of saving time and energy to practice the integration of using a solar panel and charge controller. There were small signs of insufficiency when tests were done with wireless HAT. The indications were that the device would reboot on battery power when the battery was very full, but was sensitive to even the slightest touch or movement. I did not put all of the facts together to realize that the current draw was very high for the capabilities of this battery.

With a number of problems, I gained even more information once deploying the device in the field with the more powerful 40W solar panel. Figure 6 shows how the current to the battery steadily goes to zero as it is charging. From this data, the charging status to the battery indicated “CHARGING\_FROM\_IN” meaning power is coming to the device through USB MICRO input and the battery is in a charging state. This is a slightly different result from when the device is charging at the USB-A input. Since the time the device was implemented at the site, we have learned that the charging and system power circuits are separate. This is another way of saying the battery cannot be charging and powering the device at the same time. This means that there is some undefined behavior when the solar panel is charging because it is sending varying voltage levels to the charge controller. To make this more complicated, the device readings are only estimates, so I cannot get accurate instantaneous battery current and system current readings. An idea I used to get more information about the charging of the battery is to reduce current draw from MICRO-USB. This will help me understand more about the charging status and why the device reads ‘PRESENT”, “BAD” or “WEAK”. The possible performance gain is that with the same level of sunlight, a “WEAK” power input might display a ”PRESENT” power input without any effects in the battery life.

Right now, most functions work properly with the solar panel even when the ChargingEnabled switch is set to disabled. In fact, the solar panel still seems to charge the battery when charging is disabled unless the readings I am getting from the device happen to be inaccurate. Data taken of the battery voltage when the charging status changes from “PRESENT” to “BAD” are shown in Figure 7. This is the charging state as read from the PiJuice. We can see that the voltage in the battery lowers as the charge diminishes, which would point to a reason for losing power. We can see an increase in performance as the time intervals were increased and there are more dots when the charge begins to diminish as compared to the earlier charging information. We have turned off charging when the battery becomes full.

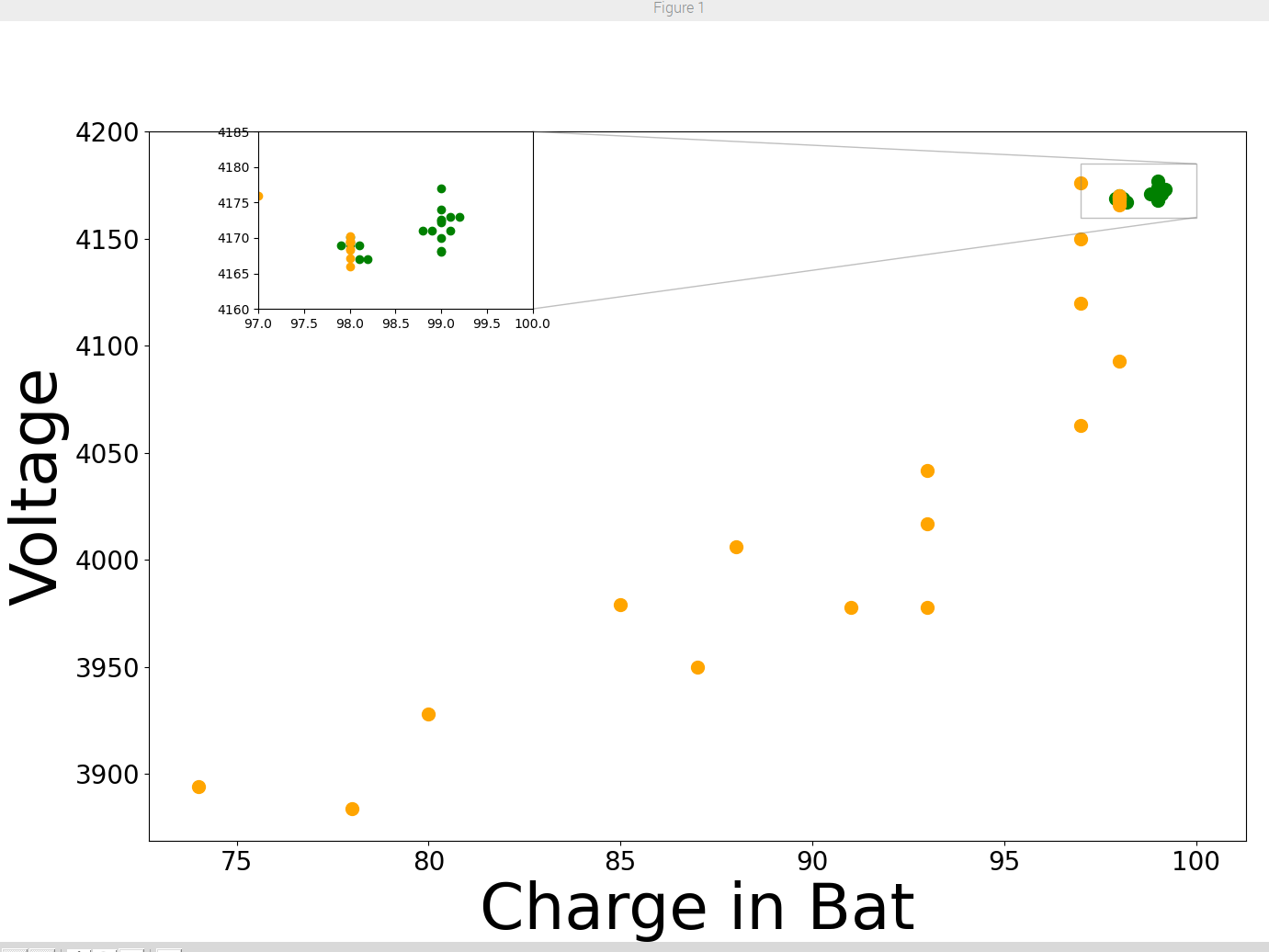


Figure 7: Battery voltage related to charge in battery.

## 

## Conclusions

This project has made great strides in creating a functioning flood sensor. The main obstacles are to reduce power consumption and find a way to effectively power the system to bring it to a functioning level. Future work includes increasing efficiency to save battery life once the issues of high current draw are solved. Evidence already shows that by disabling charge, problems with weak charging status can be avoided. We can make changes to organize the drive or otherwise use less data while ensuring that we keep needed data. Cloud possibilities entail functions based on webhooks, publish and subscribe, and immediate flood response. Data can be classified based on the time it is received, the perceived flood status, and the perceived water conditions. For now, I recommend a higher capacity battery to be sure that we can gather data on days where there is not enough sunlight. We can look for ways to compare our data with other sensors, which would give FEWS a unique perspective to compare with readings from different sensor types. After addressing the two issues of keeping the battery voltage sufficient in times of operation and keeping the alarm on schedule due to main power, we can save energy and improve battery efficiency. This can be done by modifying OS hardware devices and settings or by modifying our code to be more efficient, such as using less processing power and lowering the runtime.

The delivered items for this project include a Github page with code. These help sort and manage the data on the Drive account. There are programs to parse data and I have an example of a parsed file that is a csv file. Furthermore, it should be important to know what the data is used for to be able to do this correctly. For instance, the parsing programs will be different depending on the fields I am using from the raw data, or if the data is formatted differently. Data I used for matplot lib were text files uploaded to the drive. Then with separate programs, I remove whitespaces and extra added lines to get lists representing each upload, find the last upload for that day, use list items to create a csv. Another program can go through the lists to add 12 hours to times that happen after noon because system clock readings on the text file use 12 hour format. To create graphs, you can use a program to create a num array and a plot. You can also split data points based on parameters such as increasing charge, decreasing voltage etc. The most important lesson was to ask reasonable questions to understand the data. Sometimes, a new upload file ( as there is one file being appended to constantly on the device) is created and uploaded. Then, this file will not be the largest of the day, so the data from the rest of the day can be missed. Other times, the data is appended to the file with incorrect system times, so it looks like one day misses some points. If you are careful to detect these small changes, then it is easier to perform tasks such as finding the day’s last file or to add 12 hours to some times. The way I downloaded data was to use the latest upload from that day considering Google Drive records times in UTC. That would be from 5:00 AM to 5:AM the following day to get the full day in local time. Practically, we only need data local time starting at around 7AM.

There is a need for a new power solution. My suggestion is to find an estimate for our current draw in case we run into the same problem with a different battery. The higher capacity battery will allow the device to be alive longer. However, without modifying the power from the device, we cannot know if this solves the problem of rebooting. Another idea is to use a 3.8V nominal voltage battery to connect to the J4 connector. This is an upgrade to our device and should be done with extreme caution, possibly even bringing the device to the lab to test for use with PiJuice functions.

In conclusion, our approach to detecting floods requires us to have a steady power supply. The autonomous sensor still requires some work to improve its reliability but provides an innovative and effective approach.

## References

Arshad, Bilal, et al. "Computer vision and IoT-based sensors in flood monitoring and mapping: A systematic review." *Sensors* 19.22 (2019): 5012. Accessed 5-05-2023.

Lo, Shi-Wei, et al. "Cyber surveillance for flood disasters." *Sensors* 15.2 (2015): 2369-2387. Last Accessed 5-05-2023.

Sabbatini, Luisiana, et al. "A computer vision system for staff gauge in river flood monitoring." *Inventions* 6.4 (2021): 79. Last Accessed 5-05-2023.

Yu, Jaehyoung, and Hernsoo Hahn. "Remote Detection and Monitoring of a Water Level Using Narrow Band Channel." J. Inf. Sci. Eng. 26.1 (2010): 71-82.Last Accessed 5-05-2023.

Thekkil, Tibin Mathew, and N. Prabakaran. "Real-time WSN based early flood detection and control monitoring system." *2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*. IEEE, 2017. Last Accessed 5-05-2023.

Hiroi, Kei, and Nobuo Kawaguchi. "FloodEye: Real-time flash flood prediction system for urban complex water flow." *2016 IEEE SENSORS*. IEEE, 2016. Last Accessed 5-05-2023.

Pratama, Ari, Rendy Munadi, and Ratna Mayasari. "Design and implementation of flood detector using wireless sensor network with mamdani's fuzzy logic method." *2017 2nd International conferences on Information Technology, Information Systems and Electrical Engineering (ICITISEE)*. IEEE, 2017. Last Accessed 5-05-2023.

Ogie, Robert Ighodaro, et al. "Optimal placement of water-level sensors to facilitate data-driven management of hydrological infrastructure assets in coastal mega-cities of developing nations." *Sustainable cities and society* 35 (2017): 385-395. Last Accessed 5-05-2023.

Zhang, Zhen, et al. "In-situ water level measurement using NIR-imaging video camera." *Flow Measurement and Instrumentation* 67 (2019): 95-106. Last Accessed 5-05-2023.

<https://developers.google.com/identity/protocols/oauth2/service-account>. Last Accessed 5-08-2023.