HoverCraft: Stabilizing Drones with Smart Sensors

Jack Kernan
ROB 498
U of M Robotics
Ann Arbor, United States
jrkernan@umich.edu

Max West
ROB 498
U of M Robotics
Ann Arbor, United States
maxwest@umich.edu

Joseph Fedoronko
ROB 498
U of M Robotics
Ann Arbor, United States
fedoronj@umich.edu

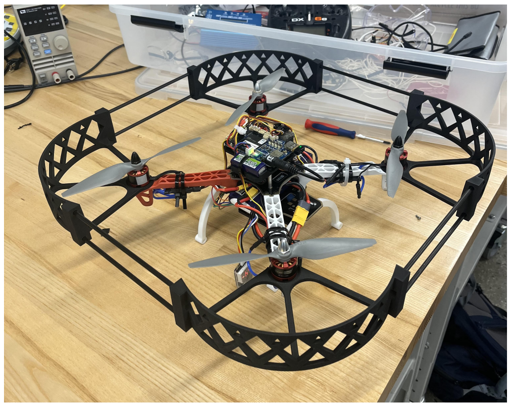

Fig. 1. Latest Version of Our Drone

I. Introduction

As technology advances and electrical components become increasingly compact and efficient, complex robotic systems are now accessible as “Do-It-Yourself” projects. Among these systems, drones have gained significant popularity due to their versatility and unique capabilities. With applications in agriculture, photography, mapping, and more, the drone industry is rapidly expanding, driving innovation and research.

Through ROB 498, we were able to enhance our engineering skills by building a drone using the kits in the lab. We applied software skills like programming, debugging and developing in linux, hardware skills like 3d modeling, designing and building and electrical skills like soldering, wiring and fastening through this class. This report details the steps we took to improve our drone, after we had it functioning properly. Figure 1 details our drone at the time of this report.

The goal of our final project was to enhance a basic drone platform by implementing a position sensing system capable of setpoint control. For this project, we chose 2 sensors: the VL53L1X time-of-flight (ToF) sensor and the PMW3901 optical flow sensor. The ToF sensor can sense the drone’s distance from the ground. This process works by the sensor emitting infrared laser signals, which travel to the target—in this case, the ground—and reflect back to the sensor. The time taken for the signal to return is then used in conjunction with known constants to calculate the distance to the target. The optical flow sensor, on the other hand, measures motion relative to the ground by analyzing patterns of light. It captures consecutive images of the ground surface and calculates the shift in these images using optical flow algorithms. By comparing the pixel movement between frames, the sensor determines the velocity of the drone along the x and y axes. This velocity data, combined with the ToF sensor’s altitude measurements, allows our drone to accurately estimate its position with an altitude range of up to 4 meters.

With these position readings, our goal was to enhance our existing flight controller and additionally add a DRONE_LOCK flight mode that keeps it stationary. We chose these specific sensors for their accuracy and increased range. We hope this builds a foundation for more advanced autonomous operations in the future.

II. Methodology

This section details the methods and procedures used to implement our EUAS final project. We start by describing the sensor hardware, followed by describing the development of the driver software for the sensors, and concluding with a description of our integration with the existing flight controller.

A. Hardware

1) Sensor Architecture: To enhance flight stability and improve the drone’s flight controller, we integrated both a Time-of-Flight (ToF) sensor and an optical flow sensor into the system. This section outlines the methodology for the hardware integration of these sensors.

Figure 2 shows the full communication pipeline once everything is wired.

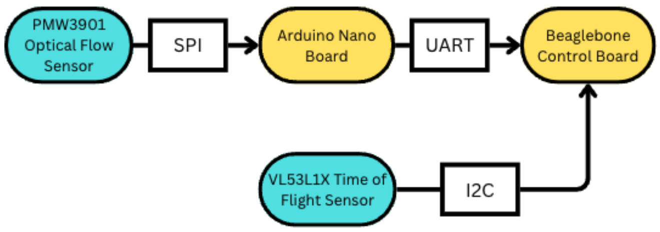

Fig. 2. Data Flow of Sensors

Starting with the ToF sensor, the first step was to solder and wire it to our beaglebone board. To communicate information from the sensor to our board, we had to make use of inter-integrated circuit (I2C) protocol. This involved connecting the serial data line (SDA), serial clock line (SCL), voltage common collector (VCC), and ground (GND) pinouts each to a pin that goes into a connector on the sensor. The beaglebone has a dedicated I2C port and we were able to plug in this connector, securing the connection.

The optical flow sensor, as seen in Fig. 3, uses serial peripheral interface (SPI) protocol. As will be discussed in our limitations section, we faced issues adding in SPI communication directly from our optical flow sensor to our board. This is why we used an Arduino Nano as a middle-man for processing the sensor data. To wire the optical flow sensor, we first connected the GND pinout of the sensor to our power distribution board to maintain a shared electrical reference point (ground in this case) among all the components in our drone. Then we connected our master-in-slave, or Controller In, Peripheral Out, pin (MISO) from the sensor to the arduino MISO pin (D12). The same process is conducted with our MOSI pin and chip select (CS) pins, connecting them to the corresponding pins on our sensor. Our last step was to connect our Vin voltage pin on the arduino to the PDB 5V pin, supplying it with power. This powered the sensor as well via the 5V pin on the arduino.

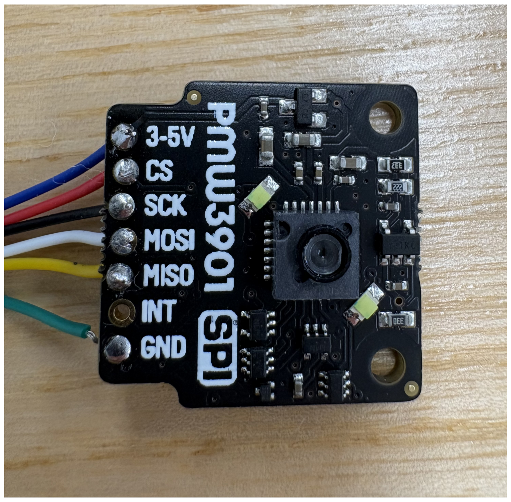

Fig. 3. PMW3901 optical flow sensor

Once the optical flow sensor was connected, the last step was to implement the communication from the Arduino to our beagle bone on the drone. We chose to use universal asynchronous receiver-transmitter (UART) communication which required connecting the transmit (TX) and receive (RX) pins on the Arduino to the corresponding opposite pins on the UART port on the beagle bone. The process allows the transmitter pin on the arduino to transfer data to the receive port on the beagle bone board, and the transmittor pin on the board to transfer data to the receive pin on the arduino. While UART communication contains 4 pins (VCC, GND, TX, and RX), the arduino already has a voltage and ground connection from the PDB which means that the only 2 pins that are required are the transmit and receive.

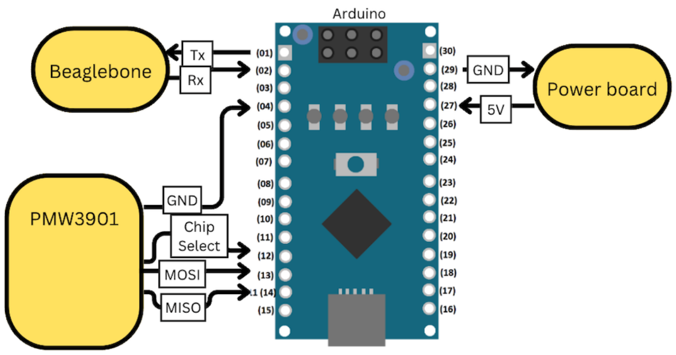

Fig. 4. Diagram of PMW3901 Sensor Pin-outs

2) Additive Manufacturing: Some additional hardware considerations for our final project were the two 3D printed components that we designed. The first component was a mount for our two sensors and the second was a landing gear. We used the Bambu A1 printers in the lab, utilizing PLA filament. As seen in Figure 5, the sensor mount has a 110x40mm mounting plate and pegs aligned to secure the corner holes of the sensors and Arduino. Each sensor position has a cutout to maintain an unobstructed view of the ground when the drone is upright. Additionally, we've incorporated heat-set inserts as a contingency; if any pegs were to break, screws can be used to fasten the sensors and Arduino securely. The mount also accommodates a slot for a second optical flow sensor, providing flexibility for various sensing configurations.

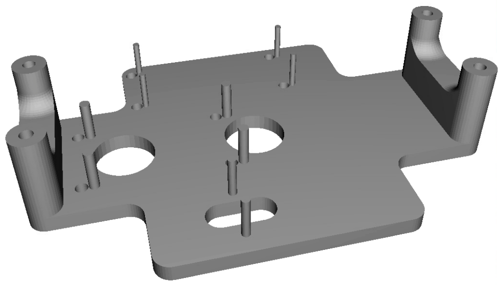

Fig. 5. Sensor Mount 3D Model

For the landing gear, we sourced a basic design online, ensuring that the legs do not interfere with the sensors' line of sight and are of sufficient length to prevent the mount from contacting the ground. This is shown in Figure 6

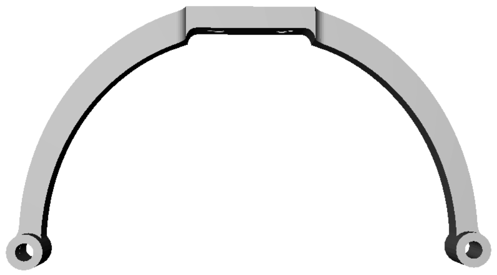

Fig. 6. Landing Gear 3D Model

B. Software

This final project presented a unique learning experience in regards to low-level sensor-software integration. In order to actually use these sensors we needed to develop drivers that read the sensor data and formatted it in a way we could use. That was easier said than done! Our methods ended up varying between the two sensors in our project so we have split this section up to describe each individually.

1) ToF Sensor:

To integrate the Time-of-Flight (ToF) sensor, our team utilized the existing documentation for the VL53L1X sensor inside of the provided EUAS Github repository. The documentation provided a header file, a C source file, and a test script for reading data from the sensor. By executing the test script, we successfully obtained accurate measurements when manually positioning the drone between 1 and 24 inches above the ground, as illustrated later in Figure 7.

Following the initial testing, modifications were required to integrate the ToF sensor code into our existing codebase. These changes included the creation of an initialization function, ToF__init, to properly configure the sensor and initialize the pulse rate settings used in our application.

The following functions were called in the initialization process:

```
VL53L1X_SensorInit(&Device);
VL53L1X_SetDistanceMode(&Device, 2);
```

These functions are called to properly set up the sensor to continuously read data later on. In addition, global variables such as the Device and rate needed to be added for use in later functions. Functions such as:

```
VL53L1X_CheckForDataReady(&Device, &tmp)
VL53L1X_GetDistance(&Device, &distance);
```

The CheckForDataReady function verifies whether new data is available from the sensor, while the GetDistance function retrieves the measured distance. The GetDistance function operates by instructing the sensor to emit infrared laser signals toward the ground to measure the drone's height. These functions are integrated into our state estimate march function, which executes within the main while loop. The march function continuously monitors and updates the state of the drone.

2) Optical Flow Sensor:

As described in the hardware section, the optical flow sensor proved to be more complicated and required an Arduino Nano for additional support. Our issues with integration and the motivation behind these design choices are outlined in the limitation section of the report, so that won't get covered here. Diving into the specific software details for the PMW3901 optical flow sensor, we'll start with the arduino.

The backbone of our driver comes from an open source library on Github by Bitcraze. They created an a7rduino driver for the PMW3901 that handles the SPI communication details. Using the SPI.h API, in addition to read and write commands that directly control the arduino pins, the provided code is able to decode the sensor information and output delta X and delta Y data directly from the sensor. At its core, the code handles low-level register interactions to extract the sensor's motion and image data for use in real-time applications like our drone.

When the sensor is initialized through the begin() function, the driver first sets up the SPI communication parameters and resets the sensor. It verifies the sensor's identity by checking the chip ID and its inverse, ensuring proper communication. The initialization also includes configuring various performance optimization registers specific to the PMW3901, ensuring that the sensor operates at optimal settings for motion tracking. The driver includes a function called readMotionCount() to extract motion data, returning the displacement in the X and Y directions. This is achieved by reading specific motion registers that store the accumulated changes in pixel position, which correspond to the sensor's perceived movement.

As described in the hardware section, the arduino uses serial communication over UART to send this motion data to the beaglebone. The final piece of the arduino code that allows the data transfer is the serial upload. In the main script that is uploaded to the board, the setup() function initializes a serial connection with a baud rate of 115200 (specified in the Bitcraze library) and the loop() function reads the motion every iteration and uploads it to the serial port. The delta X and delta Y are sent as comma-separated values for easy decoding once received by the beaglebone.

Once the data is sent to the beaglebone over the UART port, we needed code to interpret the data and save it to our drone software stack. To make it modular, we created a new C file called arduino_interface.c that handles all of the UART communication details with functions that we can call inside of state_estimator.c when requesting the delta X and delta Y data. The init function establishes connection with the UART port and opens a serial connection. The read_motion function reads the incoming data into a buffer and parses these values using the sscanf() function into integers. In state_estimator.c we created a new march function for the optical flow sensor. In this march function, the read_motion function from arduino_interface.c is called and the delta X and delta Y are stored into integers.

Importantly, these integers provide relative motion in units of pixel displacement per unit of time. To convert this into world-frame velocities we used the ToF distance to scale the pixel displacements. The following formula is what we used to calculate velocity estimates:

$$v_x = \Delta X \cdot h \quad \text{and} \quad v_y = \Delta Y \cdot h$$

where:

- v_x, v_y are the real-world velocities in the X and Y directions (m/s),
- $\Delta X, \Delta Y$ are the optical flow sensor's pixel displacements,
- h is the altitude measured by the ToF sensor (meters),

Once the velocities are calculated, these are then used to set members of the state_estimate object that we created called state_estimate.deltaX and state_estimate.deltaY. These values can now be used in controller.c to influence the flight controller.

C. Flight Controller Integration

Due to time constraints, we weren't able to fully integrate our sensor data into our flight controller. However, we have a detailed plan as to how we would improve our flight controller using sensor data from the ToF sensor and the optical flow sensor. Given the time, we would improve our ALT_HOLD mode using the ToF data and we would also add a new flight mode called DRONE_LOCK that keeps the drone stationary in all directions.

Improving the altitude mode using the ToF data wouldn't be too difficult. We would simply record the altitude of the drone when the mode is turned on and use that as our Z setpoint. Additionally, we would improve the setpoint_update_Z() function to use the ToF data as the Z position estimate. This would allow for fixed altitude flying.

The addition of a DRONE_LOCK mode would require data from both the ToF sensor and the optical flow sensor. It would function the exact same as ALT_HOLD, but additionally constrain the X and Y movement of the drone. The roll and pitch setpoints would be set to 0 and the optical flow sensor can be used for velocity state estimates. This functionality is similar to how the ball-bot from ROB 311 balances. All of the velocity setpoints are 0 and it's goal is to be perfectly still. The ToF sensor and the optical flow sensor are ideal for this kind of application.

III. Results

Due to challenges and time constraints we weren't able to integrate everything fully into the drone. The last testing that we were able to complete on our drone involved testing the drone's main feedback loop with our ToF sensor integration. When testing this, however, we noticed an interesting error that was occurring with the drone's motors. Once the drone was armed using the DSM controller, the motors would pulse on and off at a low throttle. This issue initially seemed to largely be related to our ToF sensor code using a sleep statement in order to delay the timing of the data collection. This sleep command, however, is needed because without this, the loop could execute too quickly, checking for data at a rate that is too high.

This can be seen as an example of ”busy waiting”, a common term in operating systems and threaded programming CPU resources are consumed inefficiently while waiting for an event to occur, rather than performing productive work. Our team aimed to solve this issue through the use of threads, while maintaining the sleep command to prevent CPU starvation for other parts of our program. This however, came with even more complications as the pulsing behavior continued, this time with the motors pulsing in an irregular pattern. In order to have solved this issue fully, our team would have needed to incrementally test our entire codebase as well as applying strategies to test the concurrent nature of our program to reduce the number of blocking operations executed per iteration of our loop.

Because we weren't able to complete the integration process, we admittedly don't have a lot of data to share. However, we ran a couple of tests with our sensors and have outlined some ToF data below in Figure 7 that we got from manually moving the sensor. To gather this data, we lifted the drone off a flat surface roughly to a height of 0.6m or 24in. This process was then repeated 3 times to create the graph in the figure. This data shows the accuracy of the sensor over a period of time and proves that we would be able to accurately measure height values to improve our flight controller.

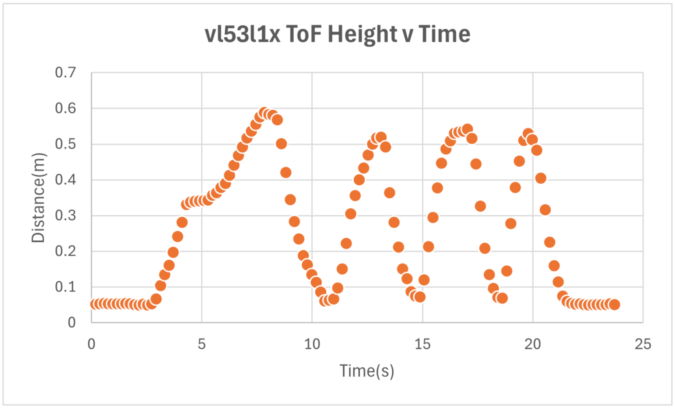

Fig. 7. ToF Test Data

We wanted to get a similar plot for our optical flow sensor, however we ran into some difficulty with reconfiguring the arduino. We spent hours the last two days before this was due trying to troubleshoot, but for some reason we weren't able to use the usb connection to upload new code to the board. However, we can confirm that the optical flow sensor captured accurate velocity data and Dr. Gaskell was a witness on the day we got it working!

IV. Limitations

As we have highlighted throughout the sensor integration process of our final project, we faced many challenges. Due to these challenges, we didn't end up making as much progress as we had hoped. We hope you understand and take that into consideration when assigning a final grade.

The main challenge that we faced was setting up SPI communication in order to read data from the optical flow sensor. The PMW3901 Optical Flow Sensor that Dr. Gaskell gave us used Serial Peripheral Interface (SPI) communication, a protocol with high clock speeds and the ability for data to be sent and received simultaneously with scalable integration. Instead of implementing a driver from scratch, we turned to Github repos.

After some searching we found a repo by Simon D. Levy that implements a driver for the PMW3901 in arduino. Our initial thought was to adapt this code into C so we could keep everything on the beaglebone. After a week or two of adapting, debugging, soldering and wiring we had an initial driver in C to test. Unfortuntately, it didn't seem to work. After hours of debugging, logic analyzer testing with Nick and Dr. Gaskell and replacing beaglebones it never seemed to work. In Simon's code, there is a check that verifies the Chip Id and the inverse of the Chip ID to make sure the driver is properly reading the registers from the sensor. This check was always failing and this issue was never resolved.

We decided to pivot and use an Arduino Nano, hoping that we could directly use the code in the repository. Even still, using Simon's code still proved cause the same issue with reading the registers that was present in our C code. It wasn't until we tried a different repository that was linked on Simon's Github that we made any progress. Bitcraze's PMW3901 arduino repo worked instantly when we copied it over to our Nano and we were finally able to capture delta X and delta Y measurements from the optical flow sensor.

It took weeks of troubleshooting until we were finally even able to get a reading from our sensor. Because of this, we were not able to make near as much progress as we would have hoped on our final project.

V. Conclusion

In this project, we worked to integrate a time-of-flight (ToF) sensor and an optical flow sensor to enhance drone position-sensing capabilities. The VL53L1X ToF sensor provided accurate altitude measurements, while the PMW3901 optical flow sensor enabled velocity tracking through delta X and delta Y values. With proper implementation, this system would drastically improve position sensing and position control. Although hardware and software integration proved challenging, particularly with SPI communication for the optical flow sensor, we overcame these hurdles by replacing our Beaglebone board and successfully acquiring position data. This achievement demonstrates the feasibility of our approach and highlights the potential for further improvements.

Moving forward, this work establishes the foundation for advancements in drone navigation and control systems using optical flow and ToF sensing. Applications of our sensing system include enhancing the altitude controller to utilize the ToF sensor when hovering below 4 meters, reducing altitude control noise. Position hold in the x and y axes is another viable application, leveraging the combined data from the ToF and optical flow sensors. Our system and mount can already accommodate a second optical flow sensor, which would enable the detection of drone rotation or translation, further improving positional accuracy.

Future steps that need to be performed include fully integrating and filtering the sensor data into the flight controller. In addition, improvements such as sensor fusion algorithms and extended testing under real-world conditions will refine the system's performance. Unfortunately, with our time constraint we were unable to test. Implementing these improvements helps achieve fully autonomous drone operations for applications such as mapping, search-and-rescue, and environmental monitoring.