TEAM CELESTIAL BLUE FINAL REPORT

by

Team Celestial Blue

Senior Design Project

Tufts University

Medford, Massachusetts

May 8, 2023

Abstract

In this report, we will discuss Team Celestial Blue's senior project for the *EE97/98–Senior Design Project* class taught by Prof. Ron Lasser at Tufts University. The team's project was a high fidelity flight computer for amateur rocketry. In the report, we will first explain our project's scope and define the problem to be solved. We will then discuss the theory of our flight algorithms as well as other background information. We will go over our project design and discuss how we re-scoped our project by cutting out certain elements which we did not have time to complete. We will explain the full method of our solution and include a parts list. We will provide results of our final testing, and analysis of those results. Last, we will provide a conclusion of our findings and learnings, as well as recommendations for any group of people who wish to develop their own flight computer.

Table of Contents

1.0 Introduction

This report was developed in the *EE97/98–Senior Design Project* class taught by Prof. Ron Lasser at Tufts University. The purpose of this report is to inform a general audience of Team Celestial Blue's senior design project–in the report we discuss the problem to be solved, the team's product design, the implementation, and our final results and analysis. Our senior project is a High Fidelity Flight Computer For Amateur Rocketry. We are sponsored by two individuals who work at Draper Labs in Cambridge, MA: Ian Fletcher and Tyler Klein. The sponsorship is appropriate because Draper Labs does research on aerospace navigation. Notably, they helped with the Apollo Moon Landings of the 1960's and 1970's.

Amateur rocketry is a hobby where everyday individuals build, test, and launch their own miniature rockets into the sky. Typically, the rocket deploys some parachutes during its descent, and the hobbyist can use GPS tracking to locate the rocket once it has landed. A critical safety feature of one of these amateur rockets is that it must deploy parachutes at the right time—you don't want to lose the rocket, or worse, have it crash land onto a valuable piece of property. Parachute deployment is critical, but most amateur rocketry hobbyists do not do it correctly.

Typically, hobbyists implement a cheap flight computer that deploys the parachutes based on timing—that is, hobbyists program the rocket to release the parachutes after a certain number of minutes have passed. But this can lead to catastrophic errors for many users. The timer may deploy the parachutes too early, like when the rocket is still moving upwards. Or the timer may deploy the parachutes too late, like when the rocket has already hit the ground.

Our solution is to build our own flight computer which deploys the parachutes as a function of rocket height and orientation. Our final product uses hardware sensors to take measurements about the surrounding environment, and then runs complex software algorithms to determine the rocket's height, speed, and orientation (in navigation terminology, these are described as *position, velocity, and attitude,* or *PVA*). Our flight computer uses position, velocity, and attitude (PVA) readings to determine the optimal time for parachute deployment. Hobbyists will be rest-assured knowing that our flight computer can keep their rocket safe.

We started designing this product in September of 2022, and finished implementing the product in April of 2023. This was all the time we had, given that the class lasts just two semesters.

We are asked by course staff to assign the results of our project to one of four cases: (1) project success, (2) project re-scope, (3) project change, and (4) project failure. The team has decided that we fit under the second category, project re-scope. This is because we needed to re-scope our project in order to meet our success criteria.

2.0 Theory / Problem Background

Our flight computer involves some hardware sensors and is very heavy on software algorithms. In fact, all of the theory behind our project is software-related. There are two software algorithms that our flight computer needs to run: (1) an IMU Strapdown, and (2) an Extended Kalman Filter (EKF). Both of these algorithms are quite complicated, so we will discuss them below. This section of the report gets particularly heavy on math and physics, so feel free to skip. Just know that the Extended Kalman Filter (EKF) is used to determine the best estimate of PVA, and the IMU Strapdown helps the EKF obtain this best estimate.

2.1 IMU Strapdown

The IMU strapdown is an algorithm that converts from IMU data to PVA (position, velocity, attitude). The strapdown uses the IMU acceleration a_{IMU} and IMU angular rotation $\omega_{IMU}$to compute PVA. The first step in doing so is to calculate acceleration in the Earth frame at an arbitrary timestep, a_j:

$$a_j = T_E^B a_{IMU} + g(r_j) - \Omega \times (\Omega \times v) - 2(\Omega \times v)$$

Next, the strapdown computes PVA (where P is r, V is v, and A is q) using the following three equations:

$$v_j = v_{j-1} + a_j \Delta t$$

$$r_j = r_{j-1} + v_j \Delta t$$

$$\overline{q_I^B} = \frac{1}{2}\left[0;\ \omega_{IMU}\right] \times q_I^B \qquad \text{Where } \left[0;\ \omega_{IMU}\right] \text{ is a vertical vector.}$$

The q_I^B is called a quaternion, which is a common aerospace representation of rocket orientation. Our flight computer uses this quaternion representation. Note that the terms “attitude” and “quaternion” will be used interchangeably for the remainder of this report.

The equations also depend on previous outputs (e.g., v_j depends on v_{j-1}), so the strapdown can run indefinitely as long as you give it an initial condition. For our initial conditions, velocity can be initialized to zero (assuming the rocket starts at rest), while position and quaternion can be initialized via GPS. Furthermore, Δt is the time step, which is determined by the speed of a single execution loop in our code.

2.2 Extended Kalman Filter

The Kalman Filtering algorithm is a method of reducing error in the sensor readings to get the most accurate result. Our flight computer uses an instance of Kalman Filtering to reduce error in the IMU Strapdown by considering GPS position measurements.

The basic principle behind the EKF is the following: if you have two sensors with standard deviation σ_1 and σ_2, then an EKF can let you combine the results in such a way that the new standard deviation is less than both σ_1 and σ_2.

The first step of an EKF is to design a state vector x. The state vector should contain all the data you care about predicting. For instance, one of our state vectors will contain 3 position values, 3 velocity values, and 3 attitude values—9 values in total.

In an EKF, you have (1) a "predict step," and (2) an "update step." These steps are application-specific–that is, they are different depending on what you are using Kalman filtering for. For our implementation, our predict step runs one iteration of the IMU strapdown, and our update step takes a single GPS position measurement and updates the state vector accordingly.

We saw what prediction looks like in the IMU Strapdown section. For the update step, the equation looks like this:

$$x_k = A_k x_{k-1} + w_{k-1}$$

A_k is known as the state propagation matrix, and it actually gets computed in the predict step after running the IMU Strapdown. Note that w_k is the difference between estimated measurement and actual measurement, and it gets computed in the update step by subtracting the GPS measurement from the position estimate in the state vector.

To summarize this section, we have the IMU Strapdown which uses acceleration and rotation data from the IMU to compute PVA. We have the Extended Kalman Filter which combines the IMU Strapdown prediction with GPS updates to statistically determine the best estimate of PVA.

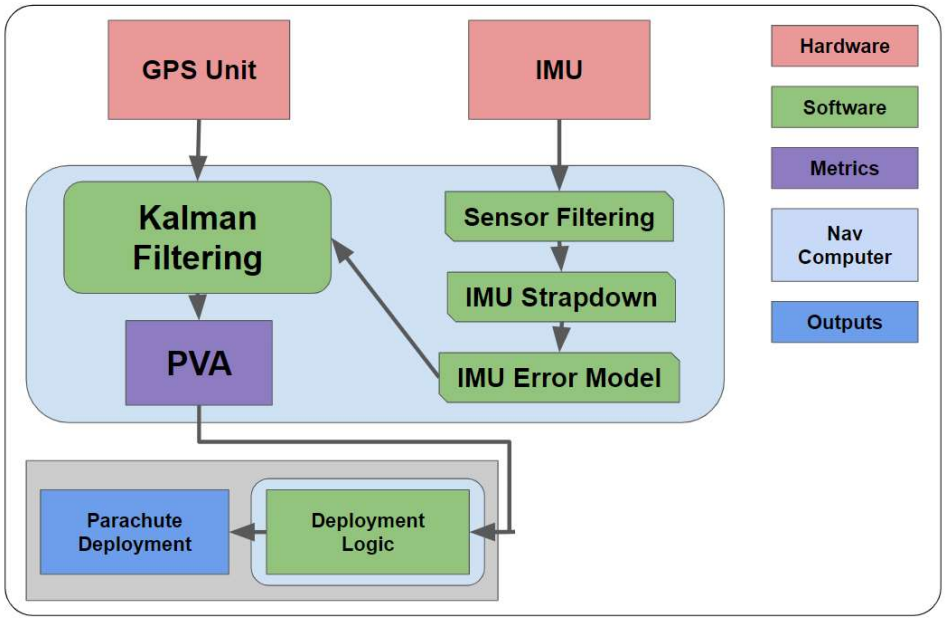
3.0 Product Design

In this section we talk about the design of our flight computer. We go over the system diagram (see Figure 1), as well as the additional components which we cut out of the final design.

3.1 Final System Diagram

The group came up with the following system diagram:

Figure 1: System Diagram of Team Celestial Blue's Flight Computer



Our flight computer requires a GPS and an IMU. The device also requires a microprocessor itself–that's the light blue box. For software, the computer passes filtered IMU data into the IMU Strapdown. The output of the IMU strapdown gets passed into the Kalman Filter, along with GPS data. The output of the Kalman Filter is a continuously-updated PVA estimate. The PVA estimate can be used to deploy the parachutes via the deployment logic.

3.2 Project Rescope

Originally, our flight computer was also intended to use a barometer sensor. Unfortunately, we ran out of time and could not integrate the barometer sensor into our Kalman filter. Our flight computer was also going to transmit PVA data to the user via an RF transmitter/receiver pair. Again, we ran out of time and could not complete this. Since we had to cut these two elements out of our final design, this is why we decided that our project falls under the project re-scope category. Note that we were still able to complete everything else, so the re-scope was a successful one.

4.0 Method of Solution

After coming up with a project plan and a system diagram, our team broke off into two smaller groups: hardware and software. In this section we will discuss (1), our hardware solution, (2) our software solution, and (3) the integration of hardware and software.

4.1 Hardware Solution

4.1.1 Sensors

We began with selecting the sensors, as these are the easiest to test and integrate with the software. Our search began on Digikey, since it has a large catalog. We settled on Adafruit's Ultimate GPS and BNO055 IMU, since they are small, low-cost, and Adafruit provides extensive documentation and software libraries, making them easier to work with. Originally, we also wanted to have a transmitter/receiver pair to remotely transmit data from the flight computer to a ground computer, however this proved to be difficult to implement and was de-scoped due to time constraints. We also selected barometers, but since our prototype was not going to be placed on a rocket, they were de-scoped very late in the design process.

The IMU supplied the Kalman Filter with linear acceleration and gyroscope data. The BNO055 provides internal filtering, so we did not have to implement sensor filtering in software. While writing scripts to collect and format the IMU data appropriately, occasionally the sensor returned an error and crashed the program. Handling the exception resolved the issue, but resulted in no data for the algorithm iteration. First, we tried simply returning the previous iteration's data, but we learned that this strategy did not work; if there were a large number of errors handled, the positional error increased rapidly. To rectify this, we implemented a moving average of the 10 previous data points.

4.1.2 Navigation Computer

On the software side, the algorithms used perform many heavy matrix computations, requiring a reasonably powerful computer with floating point and matrix acceleration capabilities. This computer also needs to be compatible with Adafruit's sensor libraries. Our first choice was the BeagleBone AI-64, since it achieves high FLOPS and Adafruit's website specifies their libraries are compatible with BeagleBone. However, upon receiving the computer and attempting to install the libraries, many errors were encountered. It turns out the AI-64 is the only BeagleBone computer that is incompatible with Adafruit's libraries, so we migrated to the BeagleBone Black Rev C, which is less powerful but has floating-point acceleration, an on-board GPU, and a significantly smaller footprint making it simpler to package.

4.1.3 Power System

The power system requirements were partially informed by the project requirements and partially by the requirements of the flight computer. Originally, it was designed around the AI-64. The project requires a battery with a relatively long life, and must be capable of outputting 3A to satisfy the AI-64. Then, a 5V, 3A voltage regulator must be placed in between to provide stable power to the computer.

After migrating to the BeagleBone Black, the requirement dropped to 1A, which is compatible with the same system.

4.1.4 Parts List

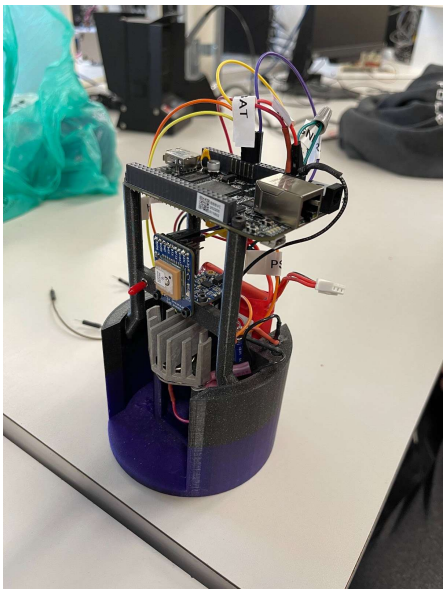
Listed below are the parts used and their costs:

BeagleBone Black Rev C……	$65.00
Adafruit BNO055 IMU ……..	$34.95
Adafruit Ultimate GPS ……...	$29.95
7.6V LiPo battery ……...	$24.99
LM323K 5V, 3A voltage regulator……..$60.00	
PLA ……	$10.00
TOTAL COST……..**$224.89**	

4.1.5 Device Housing

After selecting all parts, it was important to wire them together into a single package so they could be transported easily. One of our required specifications is that the package must fit within a 4in diameter rocket tube (inner diameter 3.9in). Below is an image of our device housing (see Figure 2).

Figure 2: Team Celestial Blue's Final Product



The prototype of the housing was 3-D printed and serves as proof the vital components can fit within the required dimensions. The wires were soldered after being trimmed to appropriate sizes to limit cluttering. Since the housing was 3-D printed, it is not sturdy enough to survive a launch.

4.2 Software Solution

We needed to come up with a good way to implement the system diagram mentioned in the Product Design section. Specifically, we first needed to implement an IMU Strapdown and an Extended Kalman Filter. We also had to make this code easily testable. Second, we also needed to write code that provides input data to these algorithms. Third, we needed to write code that saves the output data of these algorithms to a file for later analysis. In the following section, we describe the architecture of the software solution which we came up with.

The entire software repository can be found online at
https://github.com/barrycoder123/flightCPUforAmatRocketry

All of our code was written in Python. Our board, the BeagleBone Black Rev C, has the Python interpreter installed, so it made sense to use Python instead of C/C++ because Python is faster to develop and debug. The Python code was mostly developed using the Spyder IDE. Github allowed us to transfer the code between our PCs and the BeagleBoard. The board was command-line only, so we had to use Vim if we wanted to make local changes to the code.

4.2.1 Software System Architecture

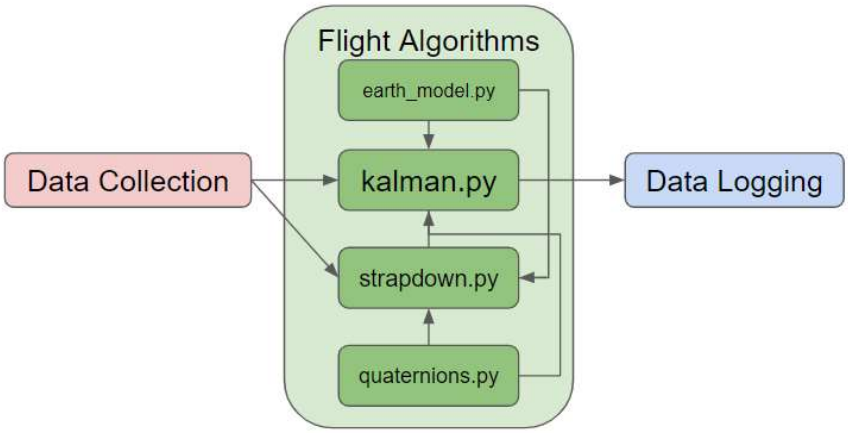
The software system is complex but easy to think about if you break it down into three main components: Data Collection, Flight Algorithms, and Data Logging (see Figure 3). These components each exist in their own folder for organization.

Figure 3: Simple Diagram of our Software Architecture

Data Collection comes first in the lineup, but it is by far the hardest component to understand. So we'll explain it after we explain the other two.

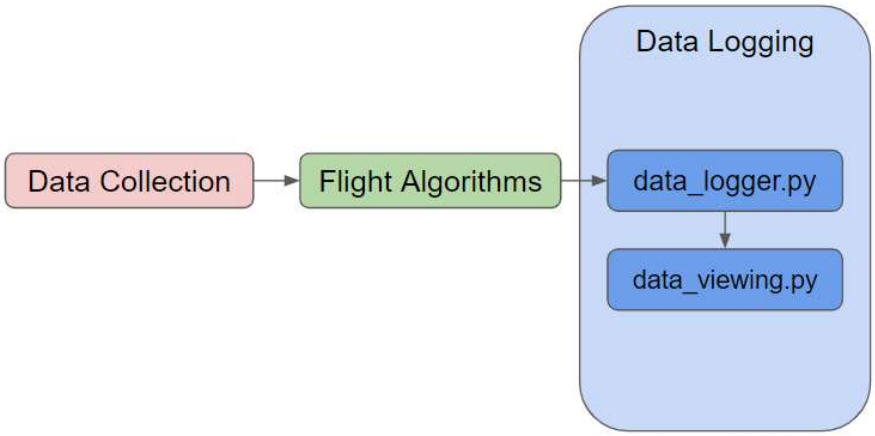
First, the Flight Algorithms component (see Figure 4) does a few things. First, it implements our IMU Strapdown and EKF in the files strapdown.py and kalman.py. Both of these files need quaternion math and Earth reference frame conversions, so we added two files, quaternions.py and earth_model.py which do exactly that.

Figure 4: Expansion of the Flight Algorithms Component



Next, the Data Logging component (see Figure 5) deals with saving EKF results and viewing them at a later time. For saving, you can use data_logger.py, which lets you write the results of the EKF to a .csv file. (The file name is based on date and time). For viewing, you can use data_viewing.py, which lets you open the .csv file and plot the contents on top of a map. The data viewer file has a main function, so the idea is that you'd run this on your own PC after transferring the .csv file from the flight computer.

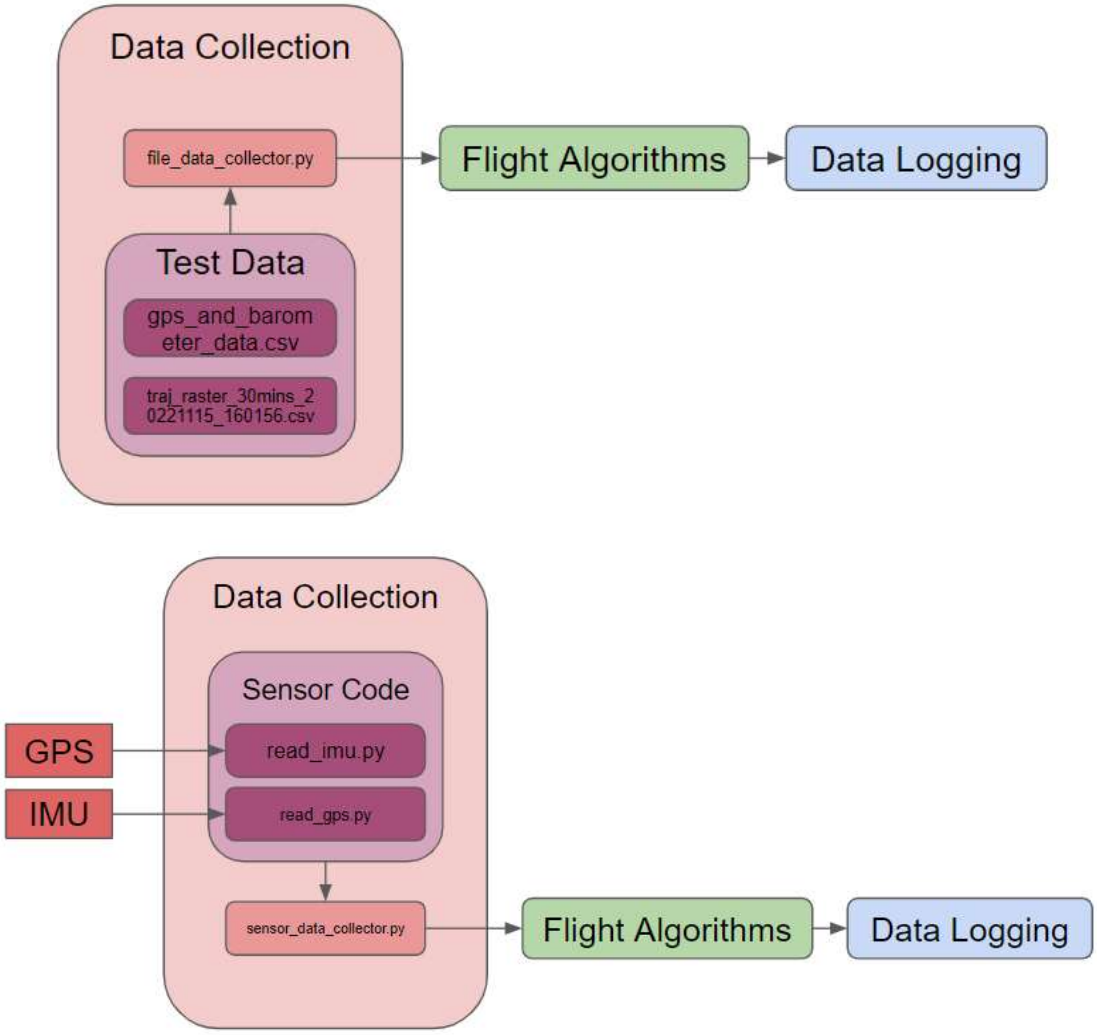
Figure 5: Expansion of the Data Logging Component



Now comes the explanation of the Data Collection component. This folder deals with providing input data to the IMU Strapdown and EKF algorithms. There are two configurations that you can use–you can either test the algorithms using our test data (related files are in the Test Data sub-folder), or you can run the code using real data from the hardware sensors (related code is in the Sensor Code sub-folder).

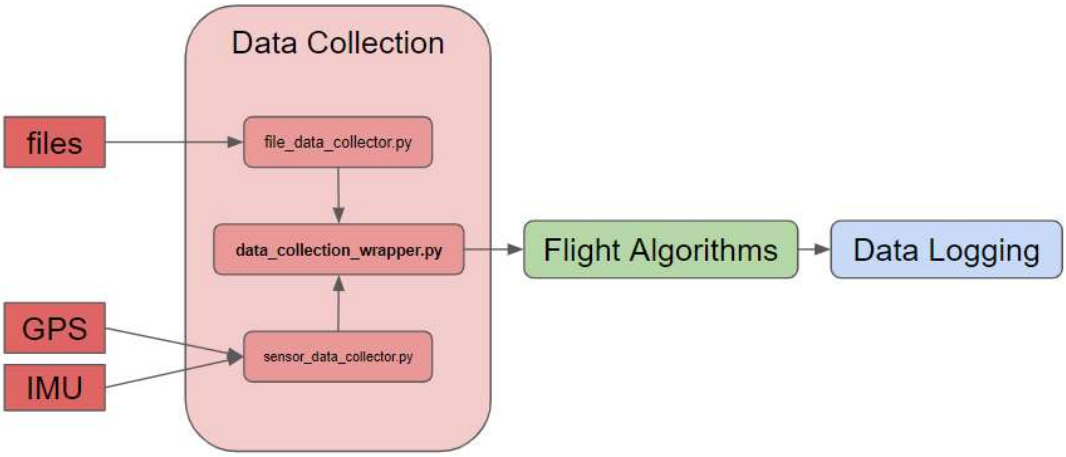
For getting test data from .csv files, we have a Python class called file_data_collector.py, which can read our .csv files and pass the readings into the strapdown and EKF. For getting real-time data from sensors, we also wrote a Python class called sensor_data_collector.py, which can read from the hardware sensors and pass those readings into the strapdown and EKF. See Figure 6 for these configurations.

Figure 6: Expansions of the Data Collection Component



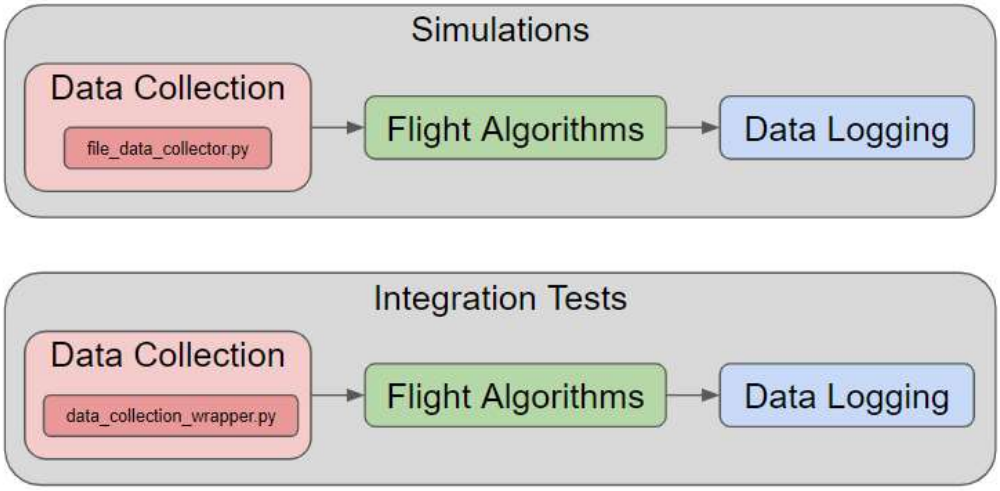
Third, we wrote a Python class called data_collector_wrapper.py. This class prompts the user to choose between the file_data_collector and the sensor_data_collector classes. This was helpful at the beginning stages of hardware integration. It allowed us to write Python scripts and test them on both our PC and on the physical flight computer. See Figure 7.

Figure 7: Full Expansion of the Data Collection Component



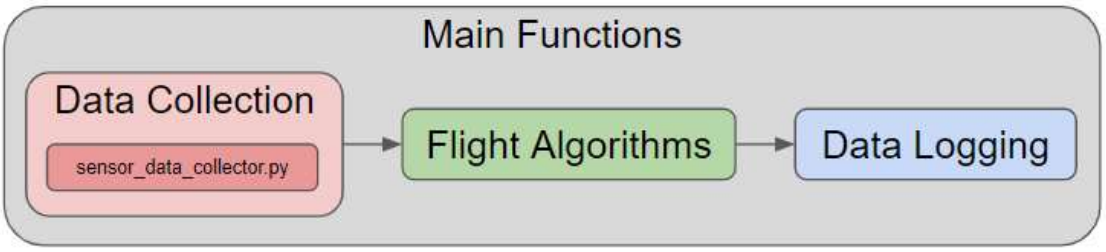
If you want to run any of this code, you'll have to head to two other folders called "Testing & Validation" and "Main Functions." They let you run the software in various configurations. First, Testing & Validation contains some simulations which use the file_data_collector, and some integration tests which use the data_collector_wrapper (see Figure 8).

Figure 8: Simulations and Integration Tests



Second, Main Functions contains code which strictly uses the sensor_data_collector. The Main Functions folder contains the scripts that should be run when the user desires to use the computer for navigation purposes (see Figure 9). There is a script that is meant for running the rocket on the ground, and a script that is meant for launching the rocket into the sky.

Figure 9: Main Functions



Seeing that “Simulations” uses the file_data_collector and “Main Functions” uses the sensor_data_collector, you might find it redundant that “Integration Tests” uses the data_collection_wrapper. There actually is a purpose for the Integration Tests–they were helpful for us once we started integrating our software with the hardware. After verifying the Simulations with file data, we needed to seamlessly test the code using sensor data. This is where the Integration Tests came in–they allowed us to toggle between file data and sensor data. Once these tests passed, we developed the Main Functions, which use sensor data only.

4.2.2 Testing the Flight Algorithms

During the implementation of the software described above, we had to run thorough tests to verify that our flight algorithms were implemented correctly. We used a set of test data generated by our sponsors which simulates the flight path of a rocket over Cambridge, MA (the test data can be found in two .csv files in the Test Data folder). We generated some sensor data as inputs to our algorithms, and generated the corresponding PVA data to compare with the outputs of our algorithms. For example, we passed the generated acceleration and rotation data into our IMU strapdown and compared its output with the expected PVA test data. We ran a similar scenario for our Kalman filter. We also unit-tested the quaternions.py and earth_model.py files. You can see the results in section **5.0 Results**. Theoretically, once the flight algorithms had been verified, we would be able to wire up our hardware sensors and run our software on the real flight computer.

4.3 Hardware/Software Integration

Integrating the hardware and software together was not as easy as we thought it would be. First, the IMU data looked particularly messy, and it took us a while to figure out that the solution was to write a function that calibrates the IMU. We also had to implement a moving average for when the IMU failed to return any data. Second, the GPS data was initially in the wrong format; the device was providing data in “degrees, decimal minutes,” and needed to be configured to provide data in “decimal degrees,” since that is what our Kalman filter expected. Third, we had to compute initial conditions for the strapdown and Kalman filter. While position and velocity initialization was trivial, attitude initialization proved to be quite complicated. According to our sponsors at Draper, this would have required its own Extended Kalman Filter to do. So for this reason, our code borrows the initial attitude directly from our test data file instead of computing an initial attitude. Our hypothesis was that the initial attitude in Medford, MA should be nearly identical to that of one in Cambridge. After implementing these three tasks, integration was complete, and we began our final testing.

5.0 Results

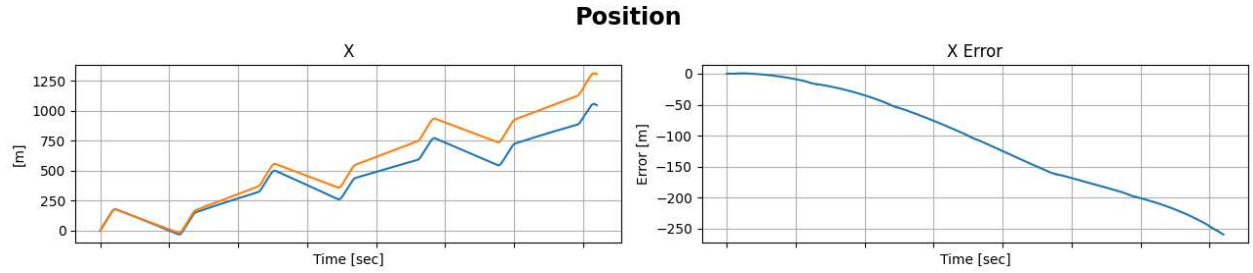
Within our time frame of September 2022 to April 2023, we were able to complete the flight computer. Along the way, we had to re-scope our design so we could meet the minimum viable product. We were not able to launch the flight computer on a rocket in time, but we were able to test the device on the surface of the Earth. In this section, we discuss (1) the results of verifying our software with the generated .csv file data, and (2) the results of our final testing performed around the Tufts campus.

5.1 Verifying our Algorithms with the Test Data

Recall that we generated some test data which is stored in .csv files. We tested both our strapdown and our EKF with this test data. Below are the results. For simplicity, we are including just X position.

5.1.1 IMU Strapdown Verification

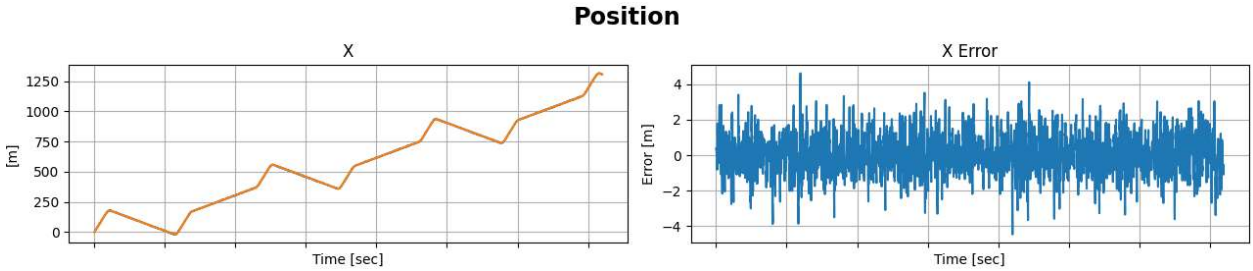
We have verified that the strapdown is correct by comparing the results (blue) to our test data (orange):



An unfortunate side effect of the IMU strapdown, even when implemented correctly, is that there is a bit of drift between the test data position and IMU strapdown. The error exceeds 250 meters after some time. This is expected. We can fix this by using Kalman filtering.

5.1.2 Extended Kalman Verification

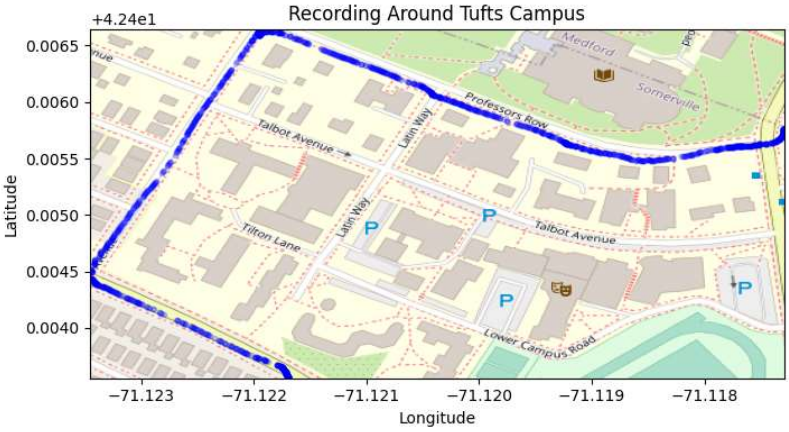
We have verified that the Kalman filter is correct by comparing the results (blue) to our test data (orange):

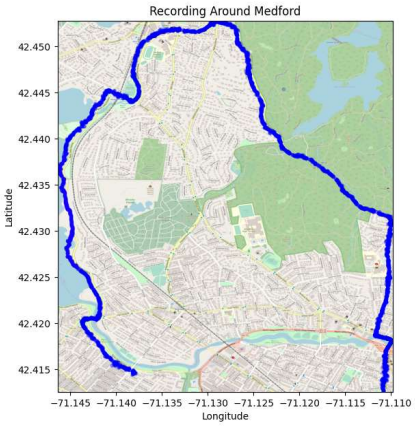


As you can see, there is much less of a drift between the expected output and the achieved output; in fact, the outputs are so close that the orange line appears to be stacked on top of the fuzzy blue line. Additionally, the error is within 2 meters in most cases.

5.2 Testing our Flight Computer in the Car

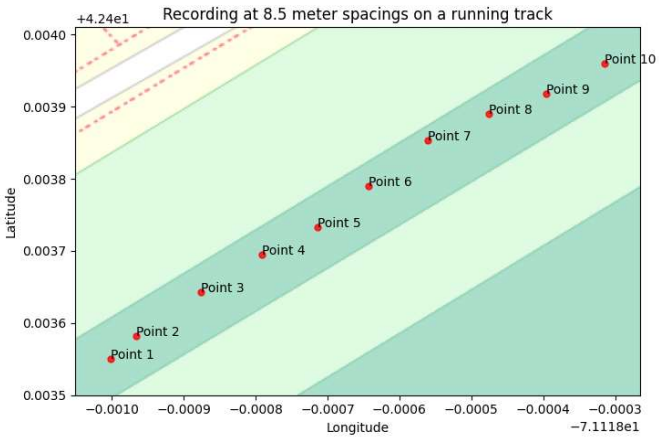
After integrating everything together and building our final flight computer, we then drove the computer around in a car. We drove around campus and around the town of Medford. Below are the results of those tests. Qualitatively, the results look quite good; it is clear that the computer was able to track the location of the car along the road.





5.3 Testing our Flight Computer on the Tufts Running Track

We also walked the flight computer around campus. We walked down the straight section of the Tufts running track. We wanted to get exact measurements, so we made sure to cross over the 8.5 meter hashes at exactly 10 seconds. Below are those results.



Point	Computer Distance From Start (m)	True Distance From Start (m)	Error in Distance (m)
1	0.0	0.0	0.0
2	4.59	8.5	3.91
3	14.58	17.0	2.42
4	23.61	25.5	1.89
5	31.17	34.0	2.83
6	39.78	42.5	2.72
7	49.51	51.0	1.49
8	57.39	59.5	2.11
9	64.45	68.0	3.55
10	72.55	76.5	3.95

You can see that at each spacing, the flight computer is off by between 1 and 4 meters. On average, the flight computer was off by 2.487 meters.

6.0 Acceptance

The following acceptance report was written by our sponsors at Draper Labs:

"Team Celestial Blue's final product met Draper's expectations for the project. The team successfully demonstrated several critical functionalities, including reading sensor data from barometers, designing a to-scale mockup, and ultimately successfully implementing a GPS+IMU Kalman filter. The team also presented their material in a way that clearly showed the results they achieved. The team did not deliver a fully integrated, fully functional prototype, but that was beyond the scope of what Draper believed to be realistic for the given timeframe.

Draper was impressed by the technical progress and overall growth of the team throughout the year. The team frequently asked questions, digested the responses, and returned with intelligent follow-up questions that clearly demonstrated their understanding of the material. The team also improved on their communication skills as the year went on, which enabled Draper to better assist as needed. Additionally, as the project developed, the students demonstrated improved leadership and organizational skills, as the sub-teams worked well together and clearly understood the big-picture goals.

Overall, the team met our expectations and should be proud of what they achieved during the project."

– Tyler Klein and Ian Fletcher, May 8th, 2023

7.0 Analysis

As seen in section **5.0 Results**, our flight computer can successfully compute PVA. Final testing showed us that our computer has a positional accuracy of 2.487 meters. This is relatively high, considering that we were hoping for an accuracy of 0.5 meters or less. This error means that our computer is not yet ready for active flight. There are improvements that need to be made.

There are two reasons for this error. First, the IMU we used was pretty cheap ($20) and therefore it was not accurate. Consequently, the output of the IMU strapdown is also not accurate, and the Kalman Filter can only do so much to correct this poor accuracy. Second, the GPS sensor is not meant to be used around large buildings, as they can interfere with the GPS trying to ping satellites. Incidentally, the Tufts running track, where we took the 2.487 meter measurement, is next to a large set of bleachers as well as the Granoff Music Center and Aidekman Arts Center, which probably all interfered with GPS readings. We are confident that these reasons are why our flight computer had a low accuracy.

Thankfully, there are a few fixes. Number one is obvious–use a better IMU sensor. Our component cost was well below our price target, so there is plenty of room for a higher quality IMU. Second would be to perform testing at a location that is far away from large buildings or metal structures. This is difficult to do in the suburbs of Boston, but if we had more time and resources we could have tested at a farther location. The third way to improve our system would be to add a barometer sensor. The GPS is often inaccurate in measuring altitude alone, so adding a barometer to the Kalman filter could improve the altitude measurement. Fourth, in addition to adding a barometer, we'd also want to fine-tune the sensor variances of our Kalman filter. The sensor variances let the Kalman filter know how accurate each sensor is, and therefore lets it know how much to factor a sensor measurement into the PVA estimate.

8.0 Conclusions

Since we have a few ideas on why the flight computer is not accurate enough, we are optimistic that the computer would be ready for active flight if we kept working for another month or two. After fine-tuning our Kalman filter and getting results to within 0.5 meters, we'd need to add parachute deployment logic which would allow the rocket to trigger parachutes at a certain altitude. We'd then want to test our computer on a rocket in tandem with a pre-existing commercial flight computer as a backup safety measure.

We still learned a few important tips while working on this project. The first was that you want to start early. The hardware team learned that ordering parts must be done as soon as possible because shipping often gets delayed. The software team learned that we should have started writing the code at an earlier time. Looking back, if we had completed the Kalman filter a month sooner, then we would have had time for more testing and possibly could have lowered our positional accuracy to within 1.0 meters or less. The second tip we learned is that communication is key. There were a few instances where the hardware and software teams did not communicate their needs effectively, and this led to delays.

9.0 Recommendations

The Celestial Blue team makes the following recommendations to any group of people who wishes to develop their own flight computer for amateur rocketry:

- Define your problem, and plan out all tasks that need to be completed.
- Generate a parts list. Choose sensors that are high accuracy–don't cheap out!
- Start early. Order your parts as soon as you can, and don't underestimate the time it takes to develop your software.
- Once you configure your sensors, make sure the data coming from them is accurate.
- For the Kalman filter, don't rely on GPS to do the heavy lifting–adding a barometer is crucial in getting an accurate altitude measurement.
- Plan for hardware/software integration to take longer than expected.
- Perform testing far away from large buildings or obstacles. It doesn't make sense to test the device in a city, because you'd ideally be launching the rocket from a rural area.

With these recommendations in place, you will be well on your way to completing a high fidelity flight computer similar to the one that team Celestial Blue developed.

List of References

1. J. Zhang, E. Edwan, J. Zhou, W. Chai and O. Loffeld, "Performance investigation of barometer aided GPS/MEMS-IMU integration," Proceedings of the 2012 IEEE/ION Position, Location and Navigation Symposium, 2012, pp. 598-604, doi: 10.1109/PLANS.2012.6236933.

   This paper explains that you can use Kalman filtering to fuse GPS with IMU. Their experimentation indicates that results are more accurate when adding a barometer, something we wanted to do but unfortunately did not have time to.

2. Steffes, Stephen. Development and analysis of SHEFEX-2 hybrid navigation system experiment. Diss. Bremen, Universität Bremen, Diss., 2013, 2013.

   This paper is rather dense but includes the 9x9 state propagation matrix for our Kalman filtering of PVA. It was sent to us by our sponsors Ian and Tyler.

3. T. Klein, I. Fletcher; “Intro to Kalman Filters,” Presentation. 9 Nov 2022

   Ian and Tyler gave us this presentation on Kalman filtering and the IMU Strapdown; their presentation slides were super helpful for our understanding of these topics because they are light and include just the right amount of information and math equations needed