Cislunar Explorers Semester Report

Tyler King

May 22, 2022

Contents

1 Introduction

This semester I moved from the Communications subteam to the Optival Navigation (OpNav) subteam, working closely with the software side and unscented Kalman filters (UKFs) in particular.

A large part of the semester revolved around setting up documentation, which would be used to help build up my understanding of the various components of OpNav along with allowing me to set up formal reports for students in future semesters. This would segue into a focus on working on UKF unit tests, which began with initial trivial tests that were dependent on input values and then expanding upward and comparing it with existing data from other sources (like `traj2.csv` from Sean and the Orekit simulation data from Dr. Muhlberger).

I worked closely with Emerald Liu, Tanya Zhou, and Rohit Valiveti, first working with them on analyzing and documenting UKFs and then moving on to setting up the `test_ukf_functions.py` method that would contain our code for trivial tests and then more advanced tests to existing data.

2 Comms Work

2.1 Flatsat Setup

Early on in the semester, me and Toby were tasked with setting up the flatsat and moving it out of B40 and into B30 (our new lab room). Although there was already an ESD mat and grounding tools in the area, they hadn't been used in a little while and thus we were worried about their quality.

We started by dusting down everything and emptying/cleaning down a box that we would use to store our flatsat. Working with Josh and members of Alpha, we were able to enter B40 and stay grounded while moving the flatsat to a moving table, which we then rolled into B30, grounded ourselves again, and set up flatsat. As a part of the setup, we also took some grounding materials from B40, along with some ESD cream for future use.

This setup was then used by Emerald and Yolanda to fix the crossing over of the graphs in Heroku (as a continuation of Ka-Hyun's work from FA21). This was successfully fixed and these changes were then updated to the repository.

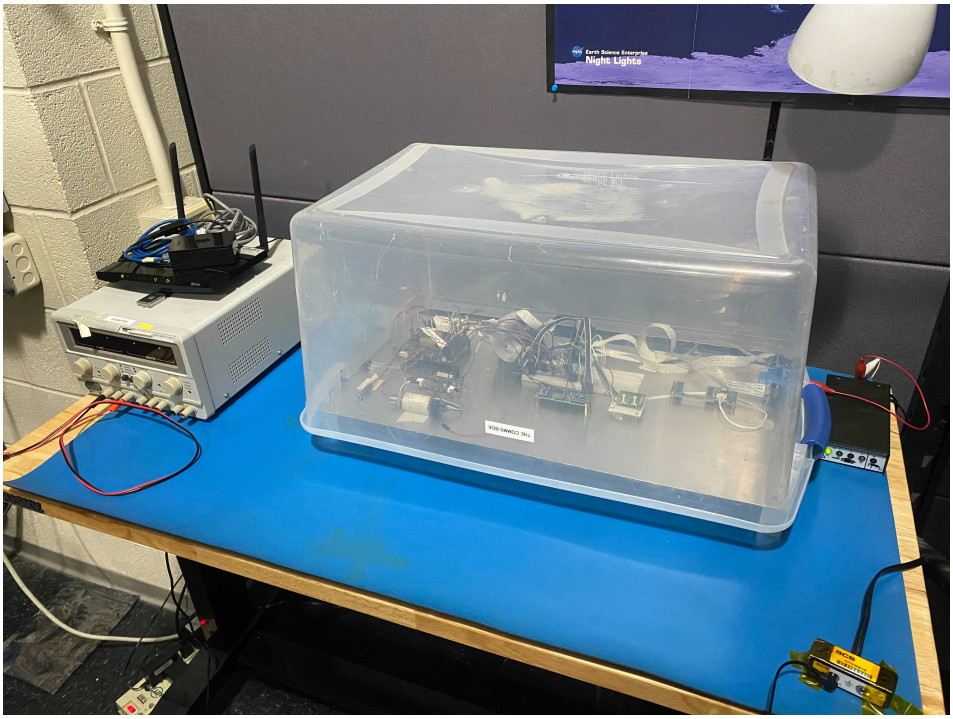


Figure 1: Image of the flatsat setup in the the back left corner of B30. We covered it in a box to make sure that no dust got into the flatsat components and grounded everything as a safety precaution.

2.2 Comms Problem Statement

To help finalize the work I did last semester for future students who may work on the communications subteam, I worked with Paul, Toby, and other members who were associated with comms last semester to formulate a list of all past and current problems, along with listing their importance and whether or not the problem would be tacking.

3 Unscented Kalman Filter Documentation

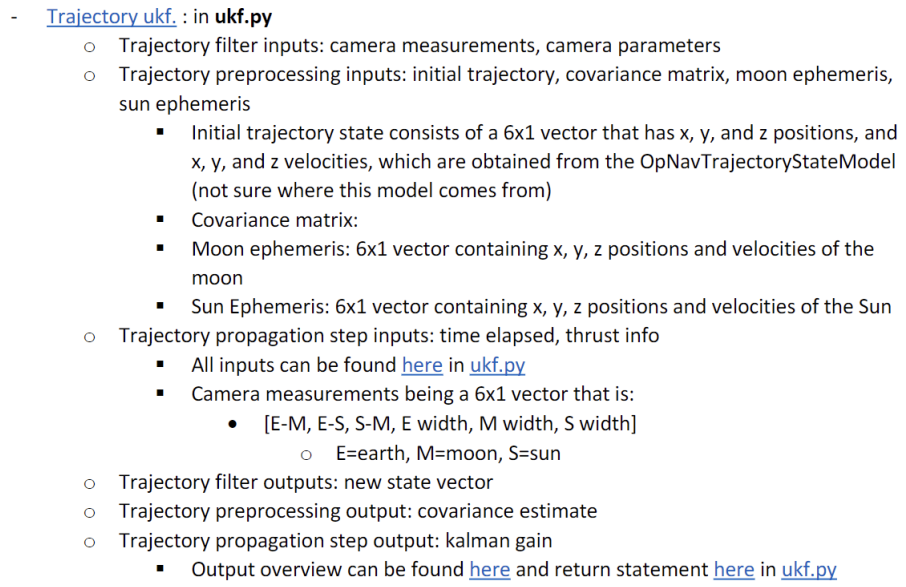
The first half of my semester was characterized by a lot of work related to setting/writing up documentation.

3.1 OpNav Subsystem Documentation

The very first task I did this semester was conduct a small write-up for the importance of UKFs and their connection to the overall OpNav system in this document. It would go on to serve as a general onboarding for OpNav for future members.

In particular, I was tasked with gathering relevant materials (including end of semester reports and other core documents) that discussed content about the trajectory and attitude UKFs. I also helped trace how this system evolved over time (i.e. swapping from an MEKF to a UKF due to the nonlinear dynamics of our system) and track the (original) testing suite for UKFs designed by Sean.

3.2 UKF-Documentation



- Trajectory ukf. : in **ukf.py**
  - Trajectory filter inputs: camera measurements, camera parameters
  - Trajectory preprocessing inputs: initial trajectory, covariance matrix, moon ephemeris, sun ephemeris
    - Initial trajectory state consists of a 6x1 vector that has x, y, and z positions, and x, y, and z velocities, which are obtained from the OpNavTrajectoryStateModel (not sure where this model comes from)
    - Covariance matrix:
    - Moon ephemeris: 6x1 vector containing x, y, z positions and velocities of the moon
    - Sun Ephemeris: 6x1 vector containing x, y, z positions and velocities of the Sun
  - Trajectory propagation step inputs: time elapsed, thrust info
    - All inputs can be found here in ukf.py
    - Camera measurements being a 6x1 vector that is:
      - [E-M, E-S, S-M, E width, M width, S width]
        - E=earth, M=moon, S=sun
  - Trajectory filter outputs: new state vector
  - Trajectory preprocessing output: covariance estimate
  - Trajectory propagation step output: kalman gain
    - Output overview can be found here and return statement here in ukf.py

Figure 2: Excerpt from UKF-Documentation that covers my preliminary analysis of `ukf.py`

This was the first in-depth UKF review I did. A large part of this documentation involved writing up the various parameters and outputs of the trajectory and attitude UKFs and using these to give a high level overview of each step of the algorithm.

In particular, I discussed the contents of the relevant inputs from `const.py` and the importance of their particular data structures. This information

would flow into the work I did a few weeks later when I built up the Lucid flowchart models for attitude and trajectory UKFs.

3.3 Lucid Flowchart UKF Models

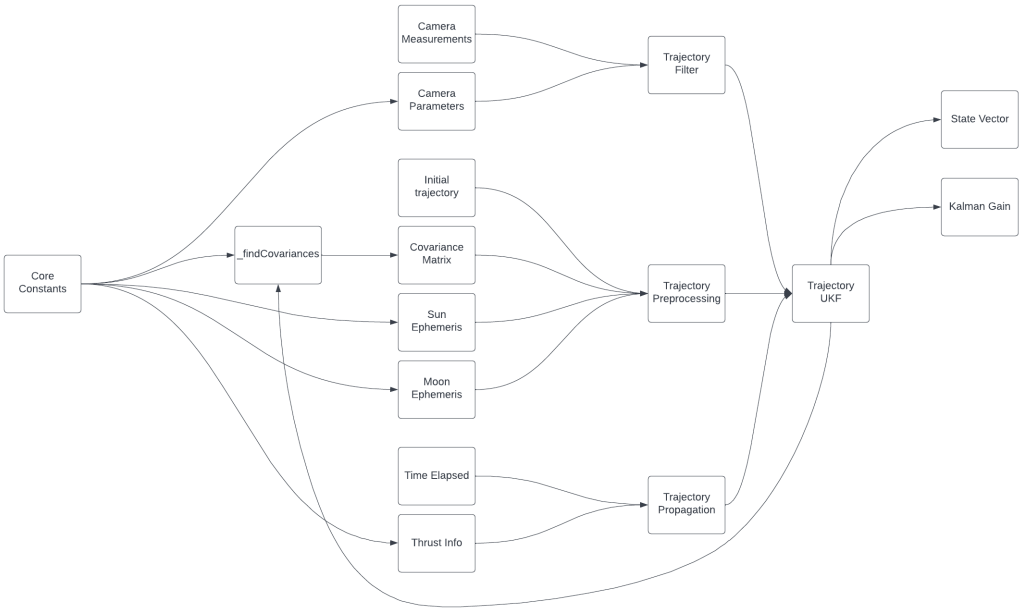


Figure 3: Breakdown of trajectory UKF inputs, outputs, and individual component relationships.

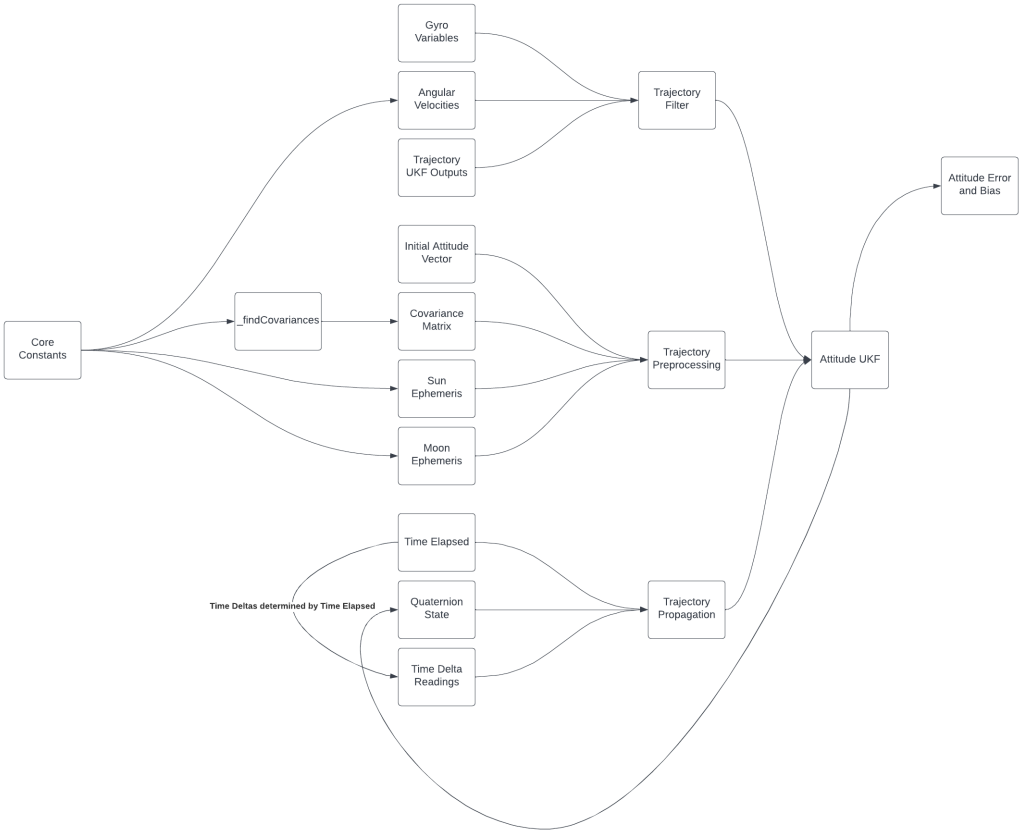


Figure 4: Breakdown of attitude UKF inputs, outputs, and individual component relationships.

4 Code Cleanup

Another major focus this semester was cleaning up past code to make it more readable and easier to operate on, both for this semester and in future semesters. Most of these edits were made in `const.py` and `ukf.py`, including small issues such as cleaning up excess variables (`cameraParams`) and removing them to allow for easier readability of `ukf.py` along with dealing with pathing conflicts in `const.py`. This was something I talked about with Adam and Stephen, where we went through and fixed every single path to allow for this file to run on all devices.

It is worth noting that we are currently looking at the importance of Kalman gain, since it seems like, while this value is computed, it is not necessarily used is any significant way (only to test the dynamics model) in

`ukf.py`.

5 General Overview of Trajectory UKF

Note that this entire section is more thoroughly outlined in this document.

Due to the nonlinear environment that occurs in space, Unscented Kalman filters (UKFs) were chosen by Paul Salazar [4] and implemented by both Paul and Sean Kumar [3] to model trajectory/attitude.

Because of the distinctly nonlinear environment surrounding the measurements and dynamics of the trajectory vector for the spacecraft, UKFs are implemented, which rely on creating an aggregate of points that are built via probabilistic models of noise. Note that we use the Wan/Der Merwe notation instead of the Julia/Uhllman notation for our trajectory UKF [5].

In the case of the trajectory UKF, we have 6 degrees of freedom (3 velocity, 3 position) [4, 1, 2].

5.1 Initial Arguments

To initialize the UKF, an initial state estimate of the trajectory `initState` is needed along with an initial covariance matrix P for error. Other measurements include `moonEph`, `sunEph`, and `measurements`, which all can be used to help determine the initial trajectory vector. Other arguments include `main_thrust_info` and `dynamicsOnly` which are used to vary the ordinary differential equation (ODE) that all sigma points are propagated through.

5.2 Cholesky Factorization

To begin the computation for the UKF, we utilize a Cholesky factorization model that decomposes $P(k)$ (the covariance matrix) and $Q(k)$ where $Q(k)$ is the process error covariance matrix. For Cislunar's trajectory UKF, this is a constant.

The columns of the Cholesky decomposition of these two matrices are used to determine the spread of the sigma points in the initial state.

We take the Cholensky decomposition matrices and redefine them as 1×6 matrices. These newly defined column vectors are then used to generate sigma points.

5.3 Generating Sigma Points

By definition of a UKF, $2(n_x+n_v)+1$ sigma points are generated (the function used for this is `__makeSigmas`). For the trajectory UKF, n_x is defined as 6, and the same is true for n_v. This gives us $2(6+6)+1 = 25$ total sigma points to work with, each of which have dimensionality 6. As a result, all our sigma points are stored in two 6×25 matrices. The first of which contains a matrix of all our sigma points and the second of which consists of a noise matrix by which our initial sigma points vary by in the propagation step.

It is worth noting that there is a high degree of symmetry that the sigma points experience surrounding the initial estimate. Due to the non-random selection of these sigma points, this degree of symmetry will always exist, although it will typically be embedded in higher dimensions. For our trajectory UKF, the space of our sigma points will be in six dimensions, with sigma points falling along one of six lines that intersect at the initial estimate.

The position and noise of these sigma points are determined by the set of equations outlined in Paul Salazar's end of semester report [4]. Note that this selection is standard in UKF literature.

When selecting the particular sigma points, note that there are scaling parameters that are based on the selection of sigma points such as n_x and α, which are then used to compute new variables λ and κ which are then used to tune the sigma points. In the case of the trajectory UKF, note that $n_x = 6$ and $\alpha = 0.001$.

5.4 Propagation Step

The sigma points are then propagated forward by some time step `dt` through some ODE. For the trajectory UKF, we use a gravitational ODE for our dynamics function that is based on the Runge-Kutta (RK) model:

$$\chi^i(k+1) = f(k, \chi^i(k), v^i(k)).$$

As of the writing of this document, we are looking into increasing the sub-stepping of the RK4 (4th order Runge-Kutta method) to increase the accuracy of the final trajectory UKF output.

Furthermore, the measurements associated with each sigma point are also propagated, giving us new values for the process error of each point:

$$\zeta^i = h(k+1, \chi^i(k+1)).$$

In the repository, our measurement model is defined by `__measModel`.

5.5 New Estimate

After all sigma points have been propagated, the initial estimate point is given a higher rating while all other sigma points are given a lower weighting. These weights are then used to compute the expected estimate and expected mean measurement and can be found in `__newEstimate`.

New computations for weights are also used to compute updates covariance estimates. `__findCovariances` is used to compute these changes

Again, the zeroth point has a higher weight which includes certain tunable values α and β. As mentioned earlier, α is fixed to 0.001, while β is a constant fixed to 2.

This allows us to compute the covariance matrices for expected residual covariance, state covariance, and cross covariance respectively (the latter two of which are used to compute the Kalman Gain):

To account for errors in the pixel conversion, the sensor error R is added to the cross covariance matrix.

The Kalman gain is then computed. This Kalman gain is used to determine whether more trust should be placed in the measurements obtained or in the dynamics model. This value also allows us to compute updated state and covariance estimates.

6 UKF Trivial Tests

6.1 Trivial Unit Test

We began building up this test case by simply testing the trajectory UKF against itself to make sure that we were obtaining somewhat logical results for one time step. We began by passing in no camera measurements as an argument and ran tests with an initial state vector velocity of 0. We then confirmed that the error bounds for the diagonal values of the error covariance matrix ($\pm$ 1 standard deviation) would contain the values after the single time step.

A major step I focused on was correcting the implementation of `test_ukf_functions.py` to fit into the unit test framework that had become commonplace over the past few semesters. While the original trajectory ukf testing script (`test_ukf.py`) did not have this implemented, Stephen was adamant about utilizing this package for all future tests.

6.2 Trivial Bounded Test

Due to the limitations of such a trivial test (results become irrelevant past a single time step), we began looking at alternatives methods to test the outputs of our trajectory UKF. One proposed example was having realistic upper bounds for the state vector positions and velocities that were dependent on the initial values (i.e. checking that the values did not drift far beyond hard set bounds). This allowed for much greater scaling, which meant we could run tests with higher time-steps along with incorporating sub-stepping into our computations.

6.3 `parameterized` Implementation

Around this time, we also introduced the `parameterized` package to deal with the difficulty of scaling up creation of unit tests. More specifically, it allowed us to easily stack 1×6 vectors for testing our expected UKF output with the outputs obtained from Orekit or `traj2.csv`. Although this would be an additional package that would be a mandated install for any person interested in running OpNav in the future, it greatly speeds up the design of various state vectors that we conduct tests on in the unit test class of `test_ukf_functions.py` (and has been fully integrated for any future work done on this script).

7 UKF Truth Data Validation

To create a less trivial test that implements more versatile tests, we (me, Tanya, Emerald, Rohit) decided to utilize past truth data that was developed by Dr. Muhlberger using the Orekit simulator (`traj2.csv`). This truth data allowed us to leverage known simulated results for relative position, relative velocity, relative distance to sun, and relative distance to moon to serve as a baseline comparison to the outputs of our trajectory ukf. Since the data was spaced out by a deviation of 3600 seconds (i.e. 1 hour intervals), we used these 1 hour checks as a benchmark for the trajectory UKF, setting `dt` to be the equivalent 3600 seconds and comparing the ukf output after every `dt` (until the upper bound time of 208800 seconds was reached) to the expected results.

8 Error Computations

To check the accuracy of the aforementioned computations, certain comparison algorithms were needed. Although initially I proposed using a method like mean-squared error (MSE), we determined to stick with the approach Sean used in the past that involved working with absolute error.

8.1 Mean Squared Error

```
# Data to test mean_squared_error
MSE_Test_State_vector_1 = TrajectoryStateVector(
    1.433e05, 5.3541e05, 1.9355e05, -0.93198, -0.077729, 0.049492
)
MSE_Test_State_vector_2 = TrajectoryStateVector(
    1.433e05, 5.3541e05, 1.9355e05, -0.93198, -0.077729, 0.049492
)

"""###################### end test data ################################"""

def MSE_test(self, MSE_Test_State_vector_1, MSE_Test_State_vector_2):
    # Check that MSE for position is 0
    assert (
        MSE(MSE_Test_State_vector_1, MSE_Test_State_vector_2)[0] == 0
    ), "Incorrect Mean Squared Error Computation for position"

    # Check that MSE for velocity is 0
    assert (
        MSE(MSE_Test_State_vector_1, MSE_Test_State_vector_2)[1] == 0
    ), "Incorrect Mean Squared Error Computation for velocity"
```

Figure 5: Trivial tests for velocity and position for MSE with the unit test package.

To compute the MSE, we utilize the equation

$$\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

where n is the number of data points, Y_i is the observed values (i.e. the outputs of our trajectory UKF), and $\hat{Y}_i$ is the expected values (i.e. the values obtained from the Orekit simulation data set). Note that n ranges from 1 to 3, and is utilized twice: once for the velocity vector and once for the position vector. The units for the first vector are m/s^2, while the units for the second vector are m^2.

This algorithm was implemented into `test_ukf_functions.py` with the `MSE` function, which takes 2 state vectors (first one observed, second one expected) and computes the MSE for velocity and position independently (since these two sets tend to differ by colossal margins, so doing a total MSE would not yield interesting results as one would overpower the other).

I also wrote 2 unit tests that confirmed the logical (trivial) outputs of the `MSE` function and added these to the unit test in case of future interest in working with them. However, since Sean had already worked with absolute error in the past, we felt that it was more fitting to continue operating with that instead.

8.2 Absolute Error

It is worth noting that the computations involved in absolute error and mean squared error are relatively similar (this can be visually confirmed from the equations). However, instead of the units being squared, the error for these values are instead to the first power (i.e. the error for the velocity vector is m/s and the error for the position vector is m.

Again separating the position and velocity errors to make the computations more independent, we (me, Tanya, Emerald, Rohit) utilize two sets of equations to compute their respective errors that were taken from Sean's past work.

For position error, we use the equation:

$$\sqrt{(x_i - x_i')^2 + (y_i - y_i')^2 + (z_i - z_i')^2}.$$

Note that x_i, y_i, z_i represent the results of the trajectory UKF's x, y, and z position outputs respectively, while the prime versions represent the expected results (i.e. the results generated by the Orekit simulator). For the velocity error, we use the similar equation:

$$\sqrt{(v_{i,x} - v_{i,x}')^2 + (v_{i,y} - v_{i,y}')^2 + (v_{i,z} - v_{i,z}')^2}.$$

In a similar way, $v_{i,x}, v_{i,y}, v_{i,z}$ represent the trajectory UKF's x, y, and z velocity outputs respectively, while the prime versions of those represent the expected results (again generated from the Orekit simulator).

To confirm that the results are reasonable, we again determine boundings for both position and velocity (this operation was mainly spearheaded by Tanya be worked on by everyone on the UKF subteam). We initialized these to be 1000km for position and 5km/s for velocity, which failed for the first two iterations of the algorithm (t=0 and t=3600). It is worth noting that the

velocity error was always well below the 5km/s bounding, but the position error did fail at t=0/t=3600.

When the position bounding was reduced to 100km for position, the first 15 (of 58) iterations failed.

As a result of these relatively large inaccuracies given the small initial position sizes, we looked into the various internals of the code. Toby mentioned that we were likely using a Runge-Kutta method with dimensionality 4 (RK4) for our `__dynamics_model`, which is a common iterative method that serves as a numerical analysis for ODEs. However, due to the massive (1 hour) time steps we were taking, the low dimensionality of the RK4 model was not sufficient to obtain highly accurate data as an approximation scheme for our gravitational ODEs. Something that was proposed was utilizing scipy's integrator (`scipy.integrate`) as a replacement for RK4, but no final decision has been made yet.

Knowing that RK4 tends to produce poor results for 1 hour time-steps, we then attempted to decrease the time-step down to smaller intervals (60 seconds, 20 minutes, etc) as a way to confirm whether or not the RK4 model would give reasonable values for these instances (something everyone on the UKF subteam worked on). Even in these scenarios, there were still multiple failures (again almost exclusively related to position), which indicated that there were other issues at play.

One proposed solution was processing camera measurements to improve accuracy of the UKF. This is something that is more thoroughly addressed in the future work section (section 10).

9 Status

Thanks to a heavy focus on Kalman filters and the work of Tanya, Rohit, and Emerald, we were able to make a lot of very solid documentation on UKFs, Kalman filters, particle filters, and a variety of other relevant algorithms that may prove useful to people in future semesters. This is closely tied to the success we had in setting up a variety of testing options for the trajectory UKF, eventually culminating in a PR that merged all relevant work we had done with testing up to that point (for reference, the PR can be found here). Although there was only 1 main PR across the semester for the UKF subteam, it merged a lot of key information, and backed with the relevant documentation, allows for a much more fluid understanding of UKFs for other individuals who may have an interest in OpNav.

Other less minor statuses include the setup of the flatsat (albeit with the slight caveat that the radio board is still somewhat dysfunctional) and

cleaning up pre-existing code to make future operations easier.

10 Next Steps

One large goal we hope to do is integrate the results of our UKF tests with the `CislunarSim` (our in-house flight simulator) to use that data as our truth information. Although we are still waiting for this repository to finish being set up, we hope to have this as a future step for upcoming semesters.

Another major task is being able to thoroughly test camera measurements and integrate them into `test_ukf_functions.py`. This should allow for a solid reduction of error since the assigned weightings of sigma points should be more accurate.

Another solution to lowering observed error is looking at alternative approaches to RK4 for the `__dynamics_model`. While packages such as `scipy.integrate` have been proposed, there are certain trade offs that come with replacing RK4 that will have to be considered if such a solution is decided upon.

Finally, we hope to eventually expand beyond just testing the trajectory UKF and begin progress on the attitude UKF. Much of the groundwork and documentation is already set out (sections 3.2, 3.3), so the key task is to begin analyzing what has already done in `test_attitude.py` and begin building up unit tests for this UKF. The code from `attitude.py` is not as thoroughly documented, however, and may need to be more thoroughly analyzed to understand the differences that come with utilizing quaternions.

11 Development Process Reflection

One thing that differentiated this semester from the last semester was the heavy use of Jira, which meant it was extremely easy to pick up any extra tasks (especially for our UKF subteam group) and divy up other larger tasks. Making Jira a requirement for all subteams was a step in the right direction for Cislunar and clears a lot of the confusion on what to do during down time (not in meetings, over break, etc). Personally, I did not find the sprints to be as effective as Jira tickets just because the issues tended to flow over very frequently and were rarely contained into a single sprint. A large part of this was due to the mathematical difficulty of understanding UKFs, which served as a blocker for a lot of early sprints (and meant even simple issues got kicked down multiple weeks).

Given my minimal exposure to version control and branch management

last semester (mainly when I was working with the AX5043 radio chip), the experiences I had with Git this semester felt like a breath of fresh air. It made it much easier to try out small-scale changes without decimating the master branch, and also made it easy to compartmentalize different subgroups.

I felt that the code reviews were particularly useful, and the ease with which Stephen, Toby, and Adam were accessible makes the team dynamic much better than at a more hierarchical system. This also ties to PR submissions, where it is always nice having a more experienced CS person look over everything and certify that it is correct.

Although there was a multitude of relevant resources provided by various members of Cislunar, I personally felt that the best resources were Paul Salazar's end of semester MEng report and Dr. Adam's introduction to Kalman filters. Furthermore, there were a compilation of useful videos that were produced by Emerald that gave good visual intuition to these tools.

Given my more hardware-oriented background last semester, I am familiar with a lot of the specialized assets that we use in the lab. This includes various tools like flatsat and CAD which have allowed me to work seamlessly with I&T in the past, along with getting cleanroom certified which allows me to do core hardware work (not particularly relevant this semester, but was important last semester and likely will be in the future).

12 CS Learning Reflection

The deep dive into Git this semester gave me key skills that I've been using in personal projects and hope to continue using in future professional settings. Similarly, with the massive size of the repository, I learned numerous good techniques related to large-scale repository contribution, including pull request submissions, pair programming (which was particularly relevant for our group given the mathematical maturity required to work with UKFs), and code reviews.

I also felt that I've become much more comfortable typing out large amounts of code, since my only prior exposure was CS 2110, and the projects there did not compare to the work done at Cislunar. This closely ties to the importance of working with a team, as I felt that over the semester I learned a lot of good techniques that relate to group coding (VS liveshare, peer programming, working on separate branches, etc).

Overall it was a satisfying semester where I made a great deal of inter- and intrapersonal progress in the field of computer science.

References

[1] A. Hunter. Estimation: Introduction by example. "https://vanhunteradams.com/Estimation/Estimation.html".

[2] S. Julier and J. Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, 2004.

[3] E. Kumar. End of semester report. "https://cornell.app.box.com/file/582013089083", Dec. 2019.

[4] P. Salazar. End of semester report meng. "https://cornell.app.box.com/file/461100064300", May 2019.

[5] E. Wan and R. Van Der Merwe. The unscented kalman filter for nonlinear estimation. In *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*, pages 153–158, 2000.