Open-Source CubeSat Flight Software and Simulation

A Senior Project Report

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Engineering

By

James

Gruber

June 13, 2025

Introduction

In the past two decades, the cost of launching mass to Earth orbit has decreased dramatically [1]. Furthermore, CubeSats have gained popularity due to their small and light form factor, which limits integration and launch costs [2]. Therefore, it is easier than in previous decades for budget-limited groups to launch satellites, especially CubeSats, into orbit.

However, obstacles remain, especially for groups with minimal satellite development experience and extremely tight budgets, such as high schools and undergraduate university student groups. Existing satellite hardware and software solutions are often prohibitively expensive and complex. Therefore, a fellow student and I set out to develop a cheap, relatively simple, open-source CubeSat stack. The other student, an electrical engineering major, designed avionics and related hardware. Meanwhile, I developed the software and simulation for the satellite, which is the focus of my senior project.

Having concluded the development of my senior project, there now exists an open-source flight software framework that allows for the development of complex flight software functionality in a simplistic and reliable design pattern. Furthermore, a sample flight software package was developed, along with a simulation to verify behavior, both of which serve as a strong jumping-off point for satellite development teams. I anticipate that these tools can greatly aid budding satellite engineering groups and curious students in building robust, real-time flight software. The tools will allow them to conceptualize their flight software design choices via a realistic simulation.

Background

Flight Software

This senior project involves the development of a complete CubeSat stack. The CubeSat hardware development was completed by a different student, who conducted this development before the start of my senior project [3]. As such, the flight computer architecture was predetermined, which drove software tooling choices. Specifically, a microcontroller, the STM32F7 [4], was chosen as the flight computer, meaning that the use of Linux onboard the satellite was not possible.

I initially considered using an open-source off-the-shelf flight software framework to develop the sample flight software package. Flight software frameworks allow for the rapid development of flight software packages and associated software infrastructure by providing abstractions to allow a developer to focus on high-level software systems design. Two obvious candidates for such frameworks stand out, both developed by NASA. The first is the core Flight Software system, or cFS, developed by NASA Goddard [5]. The second is F` (pronounced “F Prime”), developed by JPL [6]. I have some experience developing with F`, which made it especially attractive.

Unfortunately, both of these frameworks suffer from a common issue, which is a lack of good microcontroller support. F` advertises the ability to run on a bare metal platform, but this would be highly limited, as there would be no threading capability. Both F` and cFS documentation specify that for full functionality, they require an OS layer. However, extensive documentation and examples only exist for Linux systems, and official support of a microcontroller-capable OS (e.g., FreeRTOS [7]) as the underlying OS layer for both F` and cFS is nonexistent. As such, I made the decision to build a simple flight software framework that explicitly uses an RTOS and

runs on a microcontroller, and to develop a sample flight software application using this framework. The most widely used RTOS is FreeRTOS, which is open-source, so this is what I chose to develop the framework on top of.

Simulation

To verify the behavior and performance of my flight software framework and the associated example application, it was necessary to develop a simulation.

There are many simulation frameworks available to simulate satellites in orbit. Examples include FreeFlyer [8], Matlab [9], and Nyx [10]. FreeFlyer and Matlab are closed-source, and as such were not viable given the open-source nature of this project. Nyx is open-source, however, its API is for Rust. From previous Rust development experience, I know that it is difficult to quickly prototype software with it.

One of the few frameworks that satisfied my needs was Basilisk, developed by CU Boulder's Autonomous Vehicle Systems Lab [11]. This framework is relatively simple to use, contains excellent documentation, is open-source, and has support for Python scripting, allowing for quick development and verification.

The role of the simulation developed using Basilisk was to simulate the dynamics that the spacecraft would experience on orbit while having my flight software control the simulated spacecraft. Therefore, I needed to understand orbital mechanics to the extent that I could specify correct orbital parameters of the simulated spacecraft. Furthermore, I needed to understand control theory such that I could develop and implement an algorithm that could accurately apply torques in order to control the spacecraft's attitude and velocity. To develop these skills, I referenced *Spacecraft Attitude Dynamics and Control: An Introduction* [12], which is used by Cal Poly Aerospace Engineering students to learn about these mentioned topics.

Design

Flight Software

Overview

As mentioned in previous sections, the flight computer that was chosen for the satellite stack is the STM32F7 [4], meaning that Linux could easily be used. As such, the choices for underlying software were either to go bare metal or use an RTOS. I chose to use an RTOS as I wanted to write software that could utilize threading functionality. Specifically, I chose FreeRTOS due to its popularity, rich documentation, and the fact that it is open-source.

In terms of programming languages, I chose to use C++. I wanted to develop the flight software in an object-oriented manner, and I also value C++'s memory safety features (at least compared to C).

Under the flight software framework, functionality is split among so-called "workers," which are defined as C++ classes. Workers can communicate via a dedicated interprocess communication layer which is built on top of FreeRTOS queues [7].

Workers

Workers encapsulate an isolated piece of functionality in a flight software package. The functionality is defined using a C++ class. Each worker owns a FreeRTOS thread on which it executes isolated from the rest of the system.

To illustrate the design of the worker paradigm, consider an arbitrary worker that implements the functionality of a watchdog. A watchdog works by resetting the system to its "default" state if it has not been "tapped" in under a certain amount of time.

In the case of the example that I developed, the watchdog worker keeps track of the last time it was notified by every other worker. If every worker notifies the watchdog within a certain amount of time, the watchdog worker will tap the hardware watchdog onboard the satellite or development board. The hardware watchdog resets the microcontroller if it is not tapped in under a certain amount of time.

From this description, a few things about the nature of workers become clear. First of all, the watchdog worker is executing in a tight loop and should not exit, since its functionality is critical throughout the lifetime of the mission. It is most often the case that workers should never exit. As such, there is a function, called *Run()*, which serves as a de-facto “entry point” for execution of the worker, which will nominally never exit. Since this function is common to all workers, it is defined virtually in the base class, *Worker,* from which all other workers inherit.

As can be seen in Figure 1, *Run()* is not directly entered into by the FreeRTOS scheduler. *xTaskCreate()* is responsible for defining the entry point of a FreeRTOS task, which is analogous to a thread on a UNIX system. One of the arguments supplied to *xTaskCreate()* is a function pointer of the form *void (*)(void *)*, which references a function with return type void and a void pointer argument. This function pointer argument to *xTaskCreate()* serves as the entry point for the task. We cannot pass a function pointer of *Run()* to FreeRTOS’s *xTaskCreate()*, as the *Run()* function is declared as an instance member function. In C++, instance member functions have a hidden argument which is a reference to the object to which it belongs, meaning that the function signature of *Run()* does not match that which is required for a function to be supplied as the task entry point in *xTaskCreate()*. Rather, we need a static member function that would not have any hidden arguments.

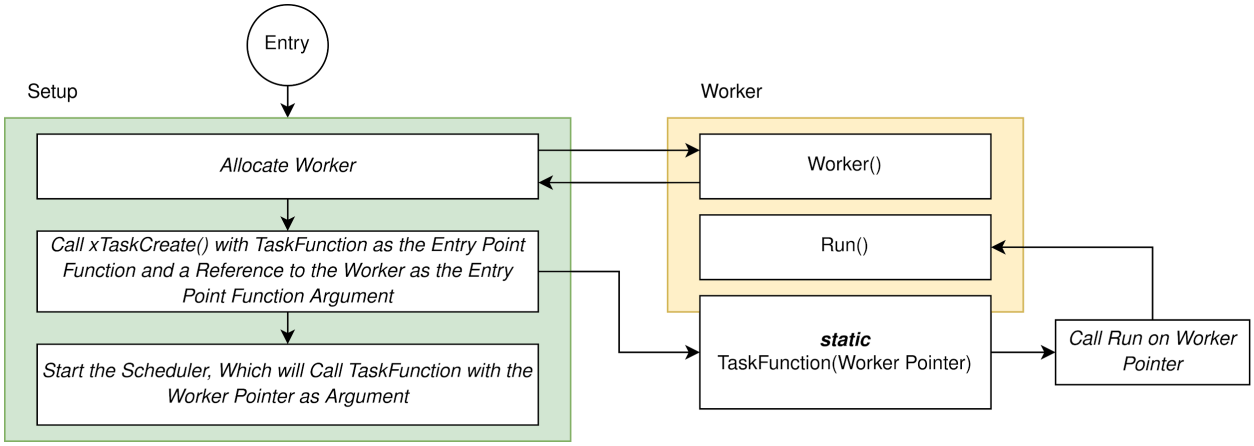

Figure 1: Worker Setup Architecture

It would not make sense to rewrite *Run()* to be static, since its implementation varies from worker to worker. Furthermore, specific worker instances may vary based on constructor arguments, such as different watchdog workers having different timeout values. Therefore, a static function was developed, called *TaskFunction()*, which belongs to the worker base class. This function takes a pointer to a worker object, from which *Run()* is called. The *xTaskCreate()* function is called in a system setup function, where the specific worker object is heap-allocated and passed to the *xTaskCreate()* function. After all workers have been created and *xTaskCreate()* has been called for each one, the FreeRTOS scheduler is started using the *vTaskStartScheduler()* function. As I discovered after lots of struggle, it is important to dynamically allocate memory on the heap for objects created prior to starting the FreeRTOS scheduler, as once the scheduler is started, FreeRTOS obliterates the original call stack.

Returning to the watchdog worker example, it is important to highlight how object-oriented programming is used to enhance the functionality of the flight software. In the case of the watchdog worker, high-level watchdog functionality is defined in a class, called *WatchogWk*, which inherits from the base worker class. While most of the functionality of the watchdog is specified here, one function, *Tap()*, is declared virtual. A subclass inherits from *WatchdogWk*,

and defines the *Tap()* function for a specific hardware watchdog. In this way, there can be multiple watchdogs with the same software functionality, but different hardware watchdog mechanisms. For example, on the final satellite flight board, there will be a hardware watchdog that is external to the STM package and placed on the board, as well as one that is internal to the STM package. Each requires a specific definition of the *Tap()* function.

Inter-Worker Communication

To facilitate communication between context-isolated workers, an inter-worker communication (IWC) system was developed that utilizes FreeRTOS queuing functionality. While FreeRTOS queues are useful in isolation, the IWC system provides additional functionality to ensure safety and consistency. The IWC system is built around a class called *FswIpc*, which stands for “Flight software Interprocess communication.”

The *FswIpc* class contains functionality for sending and receiving messages between workers. Every worker takes a smart pointer to an *FswIpc* instance as a constructor argument. Workers are then able to call *Send()* to send a message to another worker. *Send()* contains arguments that specify a destination, message contents, and time to block on sending. Similarly, workers can call *Recv()* to receive a message from another worker. However, this function only contains an argument to specify the time to block on receiving; a source worker cannot be specified. While this functionality is somewhat limiting, it helps to enforce deterministic designs by forcing workers to handle all received messages.

The *FswIpc* class uses a class called *Channel.* The number of workers on the system is defined at compile time, and so the *FswIpc* instantiates an array of *Channels* such that each worker has a *Channel* for receiving messages from other workers. When a worker calls *FswIpc.Recv()*, this function calls *Channel.Recv()* on the specific channel assigned to the calling worker. This is determined using a specific identification number which is set at runtime prior to starting the

scheduler. Similarly for *FswIpc.Send()*, *Channel.Send()* is called for the *Channel* corresponding to the destination worker.

Simulation

Overview

I chose to use the Basilisk simulation framework to develop a simulation to verify the behavior of my flight software. The goal of the simulation was to accompany a sample flight software application and demonstrate closed-loop attitude control of a simulated spacecraft, with the flight computer running the flight software “in the loop” with the simulation. To reach this goal, I learned the Basilisk framework, basic orbital mechanics, and basic control theory.

Basilisk

Basilisk is a well-featured, well documented simulation framework that is simple to set up and use. I used an official example as the base for my simulation. This involved stripping out unnecessary features, as well as developing a custom module in Python to demonstrate the correctness of the control algorithm for the satellite. Figure 2 illustrates the simulation architecture. After writing and simulating the control algorithm in pure Python, I rewrote it in C++, using the pybind tool to allow for interaction between the C++ and Python layers [13]. After verifying the correctness of the C++ algorithm, I implemented functionality to send and receive simulation information over my laptop’s serial interface to communicate with the microcontroller flight computer.

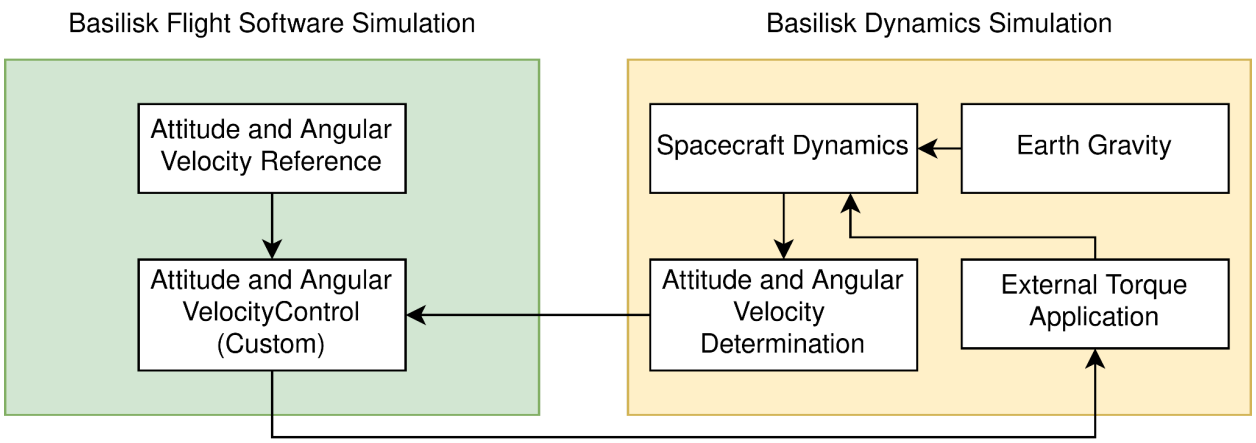

Figure 2: Simulation Architecture

Orbital Mechanics

To develop an accurate simulation of the satellite's environment, I learned basic orbital mechanics. My goal was to understand and simulate the orbit on which our satellite would likely reside. In recent years, SpaceX has launched a dedicated rideshare program for small satellites, called Transporter. For the foreseeable future, these missions will be launching into a sun-synchronous orbit [14]. This orbit utilizes a natural perturbation in the circumference of the Earth near the equator to cause the satellite's orbit to process at a specific rate, such that the orbit processes 360 degrees over the year, as seen in Figure 3. This results in consistent sun exposure at all times of the year; the satellite will always have a line of sight to the sun. For small satellites with a limited ability to generate power, this is a crucial advantage.

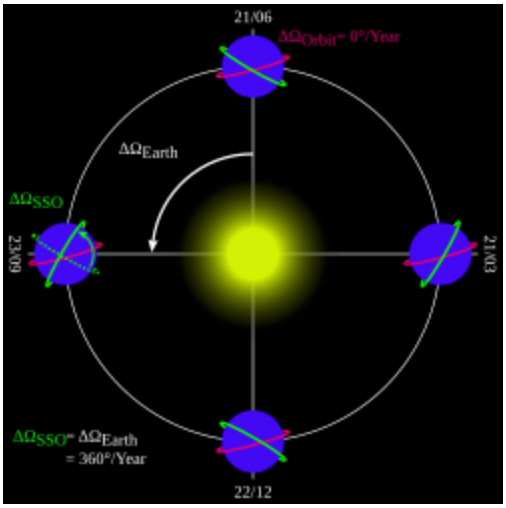

Figure 3: Visualization of the Nodal Precession of a Sun Synchronous Orbit. Credit: [Sun Synchronous Orbit - Wikipedia]

An accurate simulation of the likely orbit was instantiated in the Basilisk simulation by specifying an altitude of 600km, eccentricity of 0, and inclination of 98.2 degrees. While the accurate simulation of a sun-synchronous orbit is valuable for precise analysis, this was not necessary for the final proof of concept of my simulation. Nonetheless, simulating orbits proved a valuable exercise, and having the ability to simulate a sun-synchronous orbit for future simulation work is very valuable.

Control Theory

To demonstrate the ability of the flight software to reliably and deterministically control the spacecraft, it was necessary to understand the basics of control theory.

In my case, I chose to utilize a PD control loop. In PD control, the control input is determined based on a proportional scaling of the error signal, and a scaling of the derivative of the error signal, as in Equation 1.

$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt} \qquad (1)$$

In this case, $u(t)$ is the control input, $e(t)$ is the error, $\frac{de(t)}{dt}$ is the derivative of the error, and K_p and K_d are the proportional and derivative gain terms, respectively.

In the case of my demonstration, I wanted to show the satellite's ability to point itself in a given direction. Thus, the proportional error term would be representative of the error from the direction in which the spacecraft is pointing to the desired direction, and the derivative error term would be representative of the angular velocity error of the spacecraft with respect to the desired angular velocity. Furthermore, the control input that we are feeding back into the system is a torque to apply to adjust the spacecraft's orientation and angular velocity. As such, we can more specifically define the equation for the control of the spacecraft with Equation 2.

$$\tau = K_p q_{err,v} + K_d \omega_{err} \qquad (2)$$

On the left side of Equation 2, we see the term for torque applied to the spacecraft. The application of this torque completes the control loop, as it affects the spacecraft orientation and angular velocity, thus changing the orientation error and angular velocity error components, which feed back into the torque to be applied.

Note that $q_{err,v}$ is the vector component of a quaternion, which is a four-dimensional mathematical object - three dimensions for a three-dimensional vector, and one for a scalar. For a thorough description of the theory of quaternions and their applicability to three-dimensional orientation transformations, see *Spacecraft Dynamics and Control: An Introduction, Section 1.3.4: Quaternions* [12].

To conceptualize how Equation 2 fits into the broader spacecraft attitude management architecture, Figure 5 should be referenced.

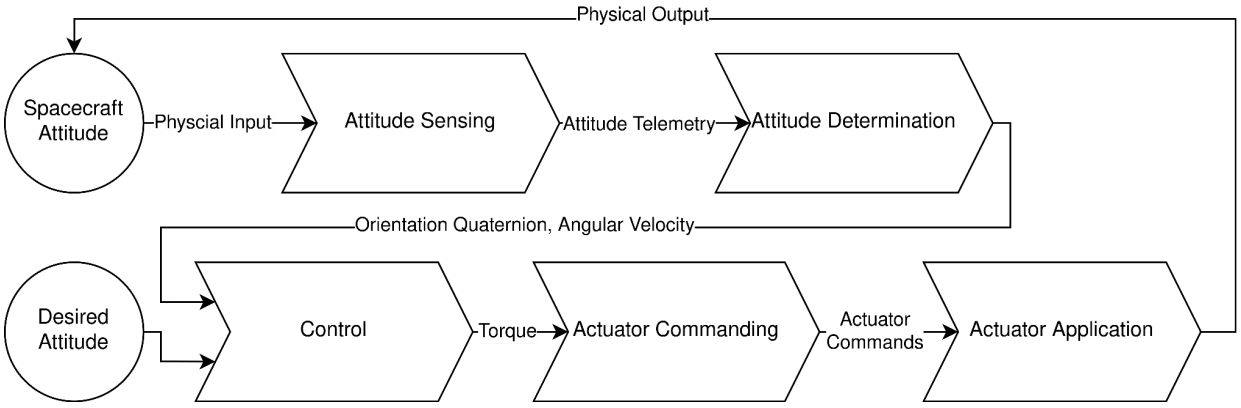

Figure 5: Ideal Spacecraft Attitude Management Control Flow

As can be seen, the control step is only part of the loop required to accurately manage the spacecraft's attitude. We also need to determine the attitude, which is a non-trivial problem. This is because we need to design a sensor system to accurately gather spacecraft attitude telemetry, and design an algorithm to synthesize this telemetry into orientation and angular velocity information for use by our control algorithm. Furthermore, we need to design an actuator system that can be used to efficiently modify the spacecraft's attitude, as well as an algorithm to take the torque value which is output by our control algorithm, and convert it into a set of commands for specific actuators to cause the torque to be applied to the spacecraft.

In the case of my senior project, I only focused on the control algorithm. As can be seen in Figure 2, Basilisk was responsible for attitude determination and torque application. It should be noted that in this case, Basilisk is not simulating specific sensors, rather it is taking the truth state of the spacecraft attitude and angular velocity and feeding it to my control algorithm. Specific actuators are not simulated either, rather the 3-axis torque output from the control algorithm is directly applied to the spacecraft dynamics. The flow of the control algorithm is shown in Figure 6.

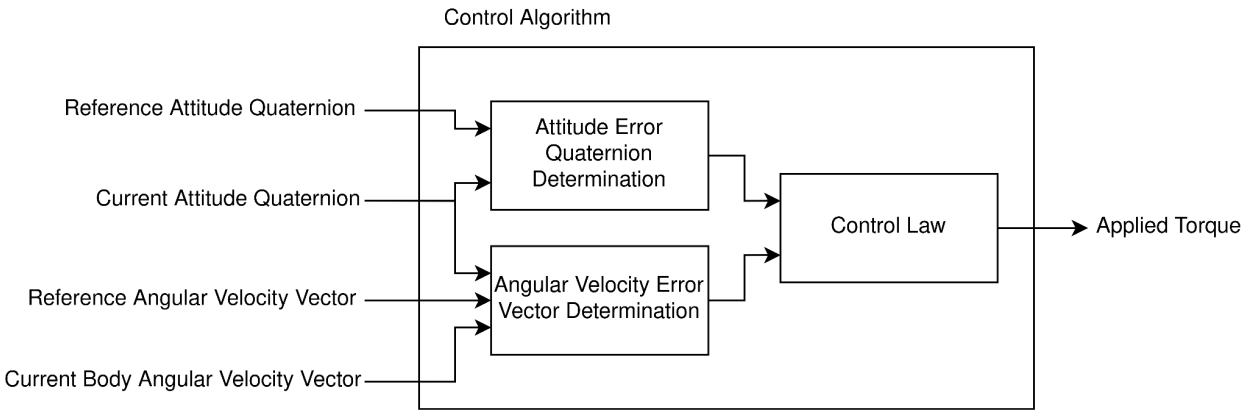

Figure 6: Control Algorithm Diagram

Let the reference attitude quaternion be denoted q_{ref}, the current attitude quaternion q_b, the reference angular velocity vector ω_{ref}, the current angular velocity vector ω_b, the attitude error quaternion q_{err}, the angular velocity error vector ω_{err}, and the applied torque τ. τ is determined via the control law in Equation 2. It is simply a matter of vector algebra between $q_{err,v}$, ω_{err}, and the control gains. There are many ways to go about tuning these gains, as this affects the responsiveness and speed of the control, which may differ depending on mission requirements. In my case, I chose an inertia matrix which would be typical of a small CubeSat, and backed out valid control gains from that.

Determining q_{err} can be done using Equation 3.

$$q_{err} = q_{ref}^{-1} \otimes q_b \qquad (3)$$

q_{ref}^{-1} is the conjugate of q_{ref}, and $\otimes$ is the quaternion multiplication operator.

Determining the angular velocity error is a little more complicated. Because the current angular velocity is expressed in the body frame (the frame of the spacecraft), but the reference angular velocity is expressed in the inertial frame (the frame of the Earth), we have to do some extra

math to convert the reference angular velocity in the body frame. This can be done using Equation 4.

$$q_{\omega ref,\, body} = q_b \otimes q_{\omega ref} \otimes q_b^{-1} \qquad (4)$$

Note that while $q_{\omega ref,\, body}$ and $q_{\omega ref}$ are quaternions, their vector components are $\omega_{ref,\, body}$ and ω_{ref}, respectively, and their scalar components are 0. Now that we have $\omega_{ref,\, body}$, which is expressed in the body frame, we simply need to subtract it from ω_b to get ω_{err}.

Having completed the math, I developed a Basilisk module that computes the applied torque, as shown in Figure 4.

Integration

To integrate the simulation and flight software, I first rewrote the control algorithm in C++ and tested it on my laptop. Thus, I could verify and debug the algorithm before running it on the microcontroller. I used the Python serial package to output attitude and angular velocity telemetry over serial to the microcontroller, and to receive torque commands back from the microcontroller. I used the Python struct package to serialize and deserialize the necessary data.

I used a Basilisk feature which forces the simulation to run in real-time, thus giving my algorithm on the microcontroller a “realistic” amount of computation time, as opposed to forcing faster than real-time operation.

On the flight computer/microcontroller side, I used the STM’s Hardware Abstraction Layer to receive and transmit data using a USART peripheral [15]. From an architectural perspective, I developed two workers, *NavWk* and *CtrlWk.*

NavWk is responsible for attitude determination for the simulated satellite. For now, this is just a matter of receiving attitude and angular velocity telemetry over USART and sending the telemetry to the *CtrlWk* via the inter-worker communication layer, but I anticipate that this could be extended to encompass the explicit attitude determination step as shown in Figure 5 in the case of a higher fidelity simulation involving specific sensors.

CtrlWk is responsible for computing the torque to be applied to the spacecraft. This involves receiving telemetry from *NavWk* via the inter-worker communication layer, running the control algorithm on the telemetry, and finally sending the applied torque over USART back to the simulation running on my laptop. This could be extended to encompass the actuator commanding step in Figure 5 for a simulation that involves specific actuators.

Finally, I viewed the angular velocity of the simulated spacecraft over time. The spacecraft starts pointed in an arbitrary direction with an arbitrary angular velocity. The reference attitude is a given arbitrary quaternion that is constant, and the reference angular velocity is zero. As such, a successful simulation entails the error angular velocity component on all axes approaching zero asymptotically, as seen in Figure 7.

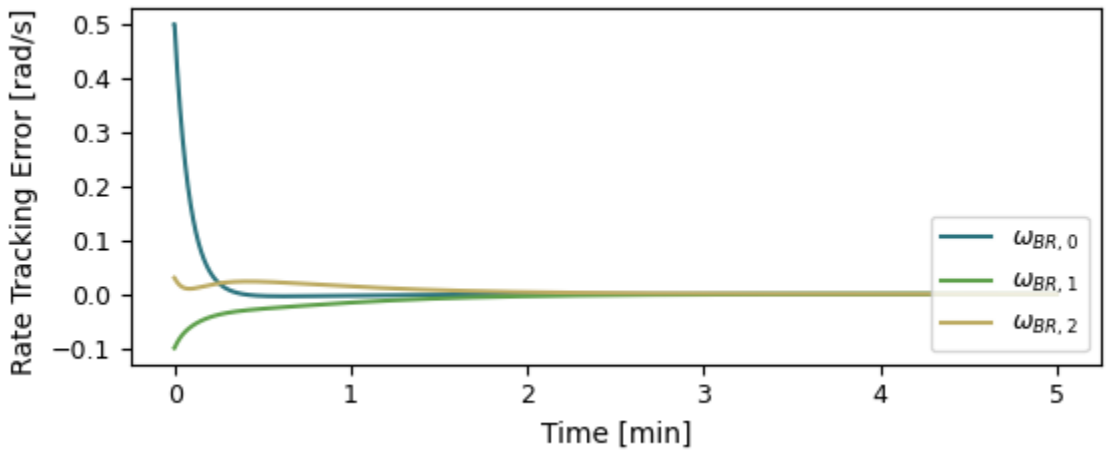

Figure 7: Angular Velocity Error Over Time

Conclusion

I am happy with the final state of my senior project, though work remains to be done to take it to its logical conclusion.

I was able to create a flight software framework, and a simulation to verify that it behaved accurately and deterministically in real-time, running on a flight-like microcontroller. This was the primary goal and is a minimum viable product for this project.

Nonetheless, there are a few things I would have liked to tackle given more time. would have liked to run the flight software on the actual flight hardware. I received flight hardware a few weeks before the end of this quarter but did not have time to integrate it. Another thing I would have liked to do was integrate sensor and actuator inputs and outputs into the simulation, which would have allowed for more accurate mission simulations. Furthermore, I would have liked to develop a longer mission test that could have utilized my newfound knowledge of orbital mechanics more rigorously.

Despite all of this, I am proud of the product I have created. I anticipate that, given a little more time, it can become a useful tool for students and spacecraft designers.

Bibliography

[1] H. Jones, “The Recent Reduction in Space Launch Cost.” *48th International Conference on Environmental Systems*, July 2018

[2] S. Malisuwan and B. Kanchanarat, “Small Satellites for Low-Cost Space Access: Launch, Deployment, Integration, and In-Space Logistics.” *American Journal of Industrial and Business Management*, 12, 1480-1497, Oct. 2022

[3] B. Burkhardt, “Open-Source CubeSat Flight Board.” *Cal Poly Digital Commons,* December 2024.

[4] STMicroelectronics, *STM32F75xxx and STM32F74xxx Advanced Arm®-Based 32-Bit MCUs Reference Manual*, RM0385, rev. 6, Nov. 2020. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0385-stm32f75xxx-and-stm32f74xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

[5] Goddard Engineering and Technology Directorate, “The Core Flight System”. [Online]. Available: https://github.com/nasa/cFS

[6] Jet Propulsion Laboratory, “F Prime: A Flight-Proven, Multi-Platform, Open-Source Flight Software Framework”. [Online]. Available: https://github.com/nasa/fprime

[7] R. Barry, *Mastering the FreeRTOS™ Real Time Kernel: A Hands-On Tutorial Guide*, v1.1.0 Real Time Engineers Ltd., 2023. [Online]. Available: https://github.com/FreeRTOS/FreeRTOS-Kernel-Book/releases/download/V1.1.0/Mastering-the-FreeRTOS-Real-Time-Kernel.v1.1.0.pdf

[8] a.i. solutions, “FreeFlyer Astrodynamics Software”. [Online]. Available: https://ai-solutions.com/

[9] MathWorks, “MATLAB”. [Online]. Available: https://www.mathworks.com/products/matlab.html

[10] Nyx Space, “Nyx: Comprehensive Spaceflight Dynamics”. [Online]. Available: https://github.com/nyx-space/nyx

[11] CU Boulder Autonomous Vehicle Systems Laboratory, "Basilisk Simulation Framework Documentation". [Online]. Available: https://avslab.github.io/basilisk/

[12] A. H. de Ruiter, C. J. Damaren, and J. R. Forbes, *Spacecraft Dynamics and Control: An Introduction*. Hoboken, NJ, USA: Wiley, 2013.

[13] pybind, “pybind11”. [Online]. Available: https://github.com/pybind/pybind11

[14] SpaceX, “Rideshare Program Available Flights.” [Online]. Available: https://rideshare.spacex.com/search?orbitClassification=2&launchDate=2025-10-11

[15] STMicroelectronics, *Description of STM32F7 HAL and low-layer drivers*, UM1905. [Online]. Available: https://www.st.com/resource/en/user_manual/um1905-description-of-stm32f7-hal-and-lowlayer-drivers-stmicroelectronics.pdf

Appendix

Flight software source code can be found at github.com/Burkhardt-Gruber/helmsman-fsw

Simulation source code can be found at github.com/Burkhardt-Gruber/fsw-sim