

Alpha CubeSat Project

Cornell University, Sibley School of Mechanical and Aerospace Engineering,
MEng. Final Report

Armen Samurkashian

Advisor: Dr. Mason Peck

May 21, 2018

Abstract

This report discusses the design and integration of hardware and software components belonging to the attitude control system (ACS) of the Cornell University Space Systems Design Studio (SSDS) Alpha CubeSat. The Satellite's overall design and mission are introduced before covering the mission specific requirements of the ACS. The application to the NASA CubeSat Launch Initiative is briefly discussed. The report then covers the Simulink model which simulates the spacecraft dynamics and ACS. The Simulink ACS algorithm was built in C++ through Simulink's autocoding functionality, and a detailed autocoding guide, containing recommended best practices, is included. The relevant hardware elements of the ACS hardware are introduced before detailing the software written to integrate the system to the point where it can be tested at a flat sat level. The autocoded ACS functionality was tested as an embedded system. The report concludes with discussion of high-risk items and recommendations on future work in preparation for flight.

Introduction

Alpha is a 1U CubeSat, designed at Cornell's Space Systems Design Studio by a student team consisting of graduate, undergraduate, and high school students, under the supervision of Dr. Mason Peck. Alpha is a technology demonstration mission with the objective of successfully deploying a light sail for deep space exploration applications such as the Breakthrough Starshot Initiative.

The 1U satellite consists of the folded sail occupying the upper $\frac{1}{2}$ U, and the spacecraft avionics occupying the lower half. The sail is ideally expected to be a four meter by four meter square with four Sprite ChipSats on each corner. The Spacecraft bus, concentrated on the lower half, contains all of the avionics, including batteries, two flight computers, an IMU/magnetometer, a radio, and an attitude control system. Power is provided by solar panels on each of the CubeSat's six faces.

The payload is held in place by one of the solar panels, acting a door. The solar panel is secured to the CubeSat by a torsional hinge that springs open. The door is held secure on the opposite side from the hinge, where a spring-loaded latch is held in place by fishing line. When commanded, a piece of nichrome wire melts the fishing line. The latch springs free, opening the door and releasing the payload. A nitinol wire embedded in the sail assists in post-deployment unfolding.

The mission involves launch of the satellite as a secondary payload with the light sail folded into the satellite. The launch vehicle will then deploy the satellite in low Earth orbit (LEO). After deployment, the satellite's attitude control system is tasked with spinning the satellite about its body-centered z-axis, and then pointing that z-axis parallel to the Earth's magnetic North-South axis. The spin is to assist with the light sail on its deployment from the CubeSat. The satellite orientation gives the onboard antenna the best field of view of the Iridium constellation, which is used for transmitting commands to and receiving telemetry from the CubeSat. Upon establishing the desired spin rate and orientation, the payload is deployed. Verification is achieved through an onboard camera, whose boresight is aligned with the satellite z-axis to get the best view of the

deployment. Alternatively, deployment can be verified through successful reception of the sail's onboard ChipSats' signals.

This document describes work done during the semesters of fall 2017 and spring 2018. The project was handed off to the author and Liam Crotty during the fall of 2017. We were joined by Davide Carabellese, Jesse Gilmore, Joao Mesquita, and Saramoira Shields over the winter. Previous team members had completed much of the design work, including design of the spacecraft bus structure, and had made hardware selections for the spacecraft avionics.

Work remained in designing the attitude control system, fixing design issues and integrating and testing the design in preparation for flight. The objective was to get the satellite design to a “flat sat” level of integration, where all the avionics are integrated and tested in both hardware and software.

This document describes the work done during the semesters of fall 2017 and spring 2018 regarding the design and implementation of Alpha's attitude Control System (ACS). Specifically, the work was done to satisfy the following system level requirements, taken from the Alpha Verification Requirements Cross Matrix written mostly by Liam Crotty.

ALPHA-SRR-SYS-5	The CubeSat shall be capable of detumbling the satellite after initial deployment from the CubeSat dispenser.
ALPHA-SRR-SYS-6	The CubeSat shall orient its z-axis along magnetic north with a an angular velocity about its z-axis.

Figure 1: VCRM ACS Relevant System Requirements

The attitude control subsystem must also satisfy the additional requirements of the Attitude Control and Navigation Subsystem of the VCRM document.

ALPHA-SRR-AC&N-1	The CubeSat shall be capable of detumbling the CubeSat from **** angular velocity about any arbitrary axis.
ALPHA-SRR-AC&N-2	The CubeSat shall align its z-axis with magnetic north and have an angular motion about its z-axis during normal operations. ☐

ALPHA-SRR-AC&N-4	The CubeSat shall be capable of measuring its position and linear velocity.
ALPHA-SRR-AC&N-5	The attitude determination software shall process the current data from all sensors when determining orientation.
ALPHA-SRR-AC&N-6	The attitude determination software shall autonomously solve for all attitude adjustments.
ALPHA-SRR-AC&N-7	The AC&N subsystem shall have two control methods: detumbling and normal operation (pointing).
ALPHA-SRR-AC&N-8	The AC&N subsystem shall orient the CubeSat to within 10°.

Figure 2: VCRM ACS Subsystem Requirements

While this report is written to satisfy the requirements of the Cornell's MAE MEng program, it is also written as a reference and guide for current and future members of the Alpha project. Problems that were encountered and took months to solve are now at point where they could be fixed in under an hour if they ever rose again. This report is written to help convey that obtained knowledge. The Matlab/Simulink simulations and the C++ files are available on the project Google Drive folder under Fall2017/FlightCode/ACSCode. All referenced libraries are included as well.

As of the end of the spring semester of 2018, the attitude control subsystem is ready for flat sat level testing.

NASA CubeSat Launch Initiative

In fall of 2017, the design team, then consisting of the author and Liam Crotty, and SSDS staff member Lindsay Hayes, created a proposal to launch Alpha through the NASA CubeSat Launch Initiative (CSLI). The author's primary contributions were to the mission plan, description of the ADCS system and most importantly, description of the mission's benefit to the NASA Strategic Plan.

Alpha's contribution to two strategic goals was emphasized. First, Alpha's payload consisting of SpriteSats embedded in a sail was a new approach to space exploration missions that already in its existing form outperformed many vehicles in its class. Better yet, this would be achieved largely through commercial-off-the-shelf (COTS) components, minimizing the risk associated with developing new hardware.

The mission's contribution to the second strategic goal, of advancing "understanding of Earth and develop[ing] new technologies to improve the quality of life on our home planet."

Alpha was pitched as a model of balancing risk, by maximizing the use COTS components for avionics and upon successful operation, increasing confidence in their use for future missions, while advancing the state-of-the art in CubeSat deployments. Alpha's contribution to advancing

the Nation's STEM workforce was also emphasized, as the project had at that point engaged high-school, undergraduate, and graduate students. NASA selected Alpha for launch in March of 2018.

Simulink Dynamics Model

The simulation used to design the satellite's attitude control system was created in Simulink. This section will describe the dynamics portion of this model, and the necessary elements required to support it. A screenshot of the model is shown below.

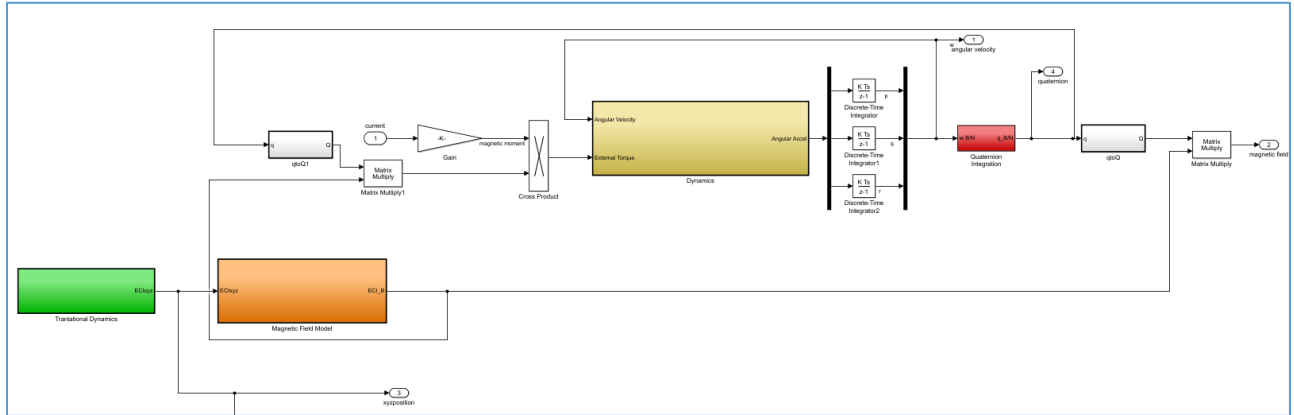


Figure 3: Simulink Dynamics Model

At a high level, the model takes in three currents that are generated from the attitude control system's torque commands to the spacecraft's magnetorquers. The model outputs spacecraft angular velocity, Earth's magnetic field expressed in spacecraft body coordinates, spacecraft position, and spacecraft orientation. It is important to note that the spacecraft's onboard instruments can only measure two of these outputs: magnetic field and angular velocity. The attitude control algorithm, which will be covered in its own section, was designed to stabilize the spacecraft as required using only these two outputs.

Translational dynamics

The spacecraft's kinematics are generated in the green subsystem block in figure 3. The spacecraft's initial position and velocity are specified by the user in the initialization script *starshot_DC_KD_setup.m*.

The dynamics are set up using assumptions from the restricted 2-body problem, with the Earth represented as a point mass and the spacecraft's translational dynamics only subject to the Earth's gravitational field. The spacecraft's position is described by the vector \vec{r} and is represented in the ECI coordinate system where:

$$xyz\vec{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix}$$

Position is computed by the numerical double integration of Newton's equation for gravitational acceleration. Acceleration for each component of \vec{r} is obtained by projecting the acceleration vector onto the component's respective axis such that:

$$\ddot{r}_i = \frac{\mu_{earth}}{|\vec{r}|^3} (\vec{r} \cdot \hat{i})$$

Where i is a dummy variable for the three components of the chosen components for the coordinate system components x, y, z .

Magnetic Field Model

Since the CubeSat's only method of attitude control is the use of magnetic torquers, the spacecraft's Simulink simulation contains a model of earth's magnetic field. The simulation uses the Earth magnetic dipole model. It is generated and expressed in the ECI reference frame in the orange colored subsystem block. The magnetic field takes the following form in a radial and tangential component.

$$\theta = \frac{\pi}{2} - \tan^{-1} \left(\frac{z}{\sqrt{x^2 + y^2}} \right)$$

$$B_r = -2B_0 \left(\frac{R_{Earth}}{r} \right)^3 \cos(\theta)$$

$$B_\theta = -B_0 \left(\frac{R_{Earth}}{r} \right)^3 \sin(\theta)$$

The two components are then rotated into and expressed in terms of the ECI coordinate system before being rotated again into the spacecraft body frame. The last rotation is done outside the orange subsystem block.

Magnetic Torquer Model

With the current input from the control algorithm and the magnetic field available in the spacecraft body coordinate system, the torque on the spacecraft can be computed. The magnetic moment vector can be expressed in terms of the three magnetorquer currents i_j , the number of coils in the magnetorquers (assumed identical for each magnetorquer) n , the cross-sectional areas of the magnetic torquers (assumed identical) A , and the unit normal vectors of the magnetorquer cross-sectional areas \hat{a}_j , expressed in the spacecraft body frame.

$$\vec{m} = \begin{bmatrix} i_1 n A \hat{a}_1 \\ i_2 n A \hat{a}_2 \\ i_3 n A \hat{a}_3 \end{bmatrix}$$

Because the torquers are aligned with the three axes of the spacecraft body-frame, we can rewrite the magnetic moment as such.

$$\vec{m} = \begin{bmatrix} i_1 n A \hat{b}_1 \\ i_2 n A \hat{b}_2 \\ i_3 n A \hat{b}_3 \end{bmatrix}$$

The torque exerted on the spacecraft body is the cross product between the magnetic moment and Earth's magnetic field at the spacecraft location.

$$\vec{\tau} = \vec{m} \times \vec{B}$$

With both magnetic moment and magnetic field expressed in the spacecraft body coordinate system, the matrix multiplication variant of the expression can be expressed as follows.

$$b_{\tau} = b_m \times b_B$$

The Simulink model carries this operation out outside the dynamics block, shown below outlined in a yellow box. The three currents are already in a three-by-one stack and aligned with the spacecraft body coordinate system. A gain containing the constants n and A turns the currents into the magnetic moment. The cross product is carried out and the torque is input into the dynamics block, which will be discussed in the next subsection.

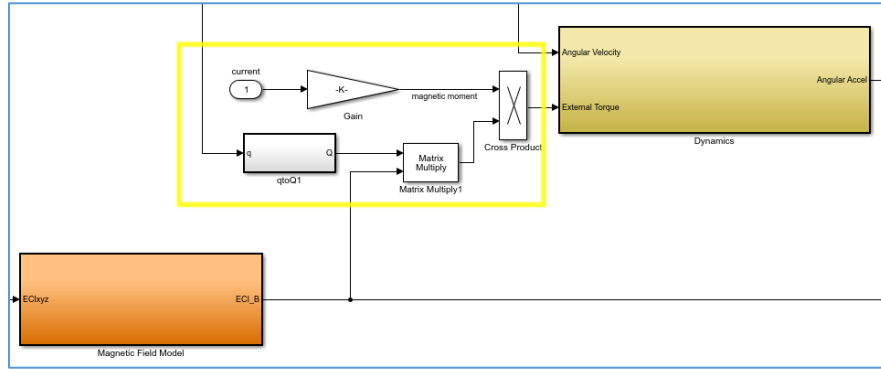


Figure 4: Cross Product to Obtain Magnetic Torque

Rotational Dynamics

The spacecraft rotational dynamics are computed using Euler's equation.

$$\vec{\tau} = I \dot{\vec{\omega}}^{B/N} + \vec{\omega}^{B/N} \times (I \vec{\omega}^{B/N})$$

$\vec{\omega}^{B/N}$ is the angular velocity vector of the spacecraft body reference frame with respect to the inertial reference frame. The torque is provided by the spacecraft's magnetic torquers as described in the previous section. Rearranging the equation, we compute angular acceleration $\dot{\vec{\omega}}$.

$$\dot{\vec{\omega}}^{B/N} = I^{-1}(\vec{\tau} - \vec{\omega}^{B/N} \times (I \vec{\omega}^{B/N}))$$

This operation is conducted in the Dynamics subsystem block, colored in gold in figure 4. The torque and angular velocity are used as inputs. The spacecraft inertia tensor matrix is retrieved from the Matlab initialization script. Angular acceleration is the output.

Integrating the equation numerically to solve for $\vec{\omega}^{B/N}$, we convert the angular rate vector into the quaternion form $\dot{\vec{q}}$ which we can then integrate once more to obtain spacecraft orientation expressed by the vector \vec{q} . The transformation between $\vec{\omega}^{B/N}$ and $\dot{\vec{q}}$ is the following.

$$\dot{\vec{q}} = \frac{1}{2} \begin{bmatrix} -\omega^\times & \omega \\ -\omega^T & 0 \end{bmatrix}$$

$$\omega^\times = \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix} \quad \omega = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad \omega^T = [p \quad q \quad r]$$

After the quaternion is obtained, it is used to compute the direction cosine matrix ${}^BQ^N$ that can translate vector expressions from the inertial to the body coordinate system. The matrix is computed using the quaternion elements.

$${}^BQ^N = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2q_3 + q_1q_4) \\ 2(q_3q_1 + q_2q_4) & 2(q_3q_2 - q_1q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix}$$

$$\vec{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}$$

The entire process can be seen in figure 5, shown below. The angular acceleration vector is taken from the Dynamics subsystem block, and integrated to obtain angular velocity. Angular velocity is fed back into the Dynamics block as an input. The initial condition for angular velocity at time $t = 0$ is obtained from the initialization script and is stored in the three integrator blocks. The angular velocity is transformed to a quaternion rate and integrated to obtain the quaternion vector in the red subsystem block labeled “Quaternion Integration”. It is then used to compute the direction cosine matrix in the white subsystem block labeled “qtoQ”.

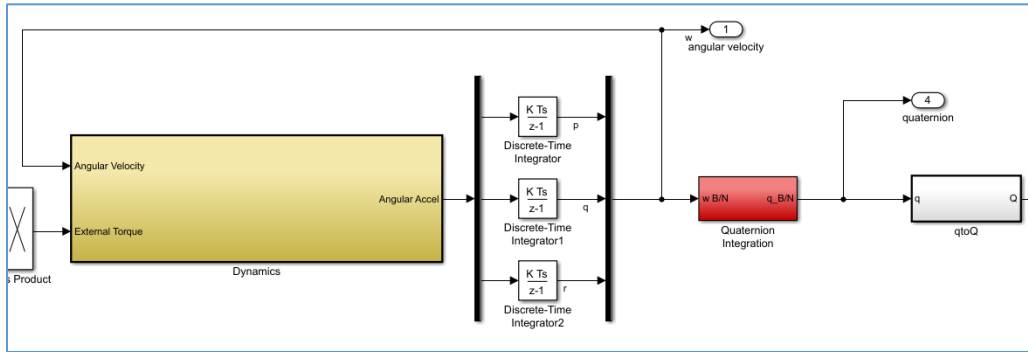


Figure 5: Rotational Kinematics

Control System

The mission concept of operations call for establishing a spin about the Earth’s magnetic North-South axis. Considering that half of the already small 1U internal volume is occupied by the

payload, space is extremely limited. A design choice was made to have the satellite control attitude only through magnetorquers. One advantage for doing so is simplicity. Magnetic torquers are simple devices from an electrical standpoint and contain no moving parts that can fail during flight.

The design choice presents a challenge however, considering that a magnetorquer cannot produce a torque about the axis with which it is aligned, which can be mathematically observed through the properties of the cross product through which the torque is generated.

$$\vec{\tau} = \vec{m} \times \vec{B}$$

Early on, a two part controller was identified as a solution for the problem. One controller would establish the spin at an attitude where there was sufficient control authority to do so. A second controller would point the spacecraft z-axis in the necessary direction once the spin had been established.

A two part controller was developed for this purpose by Davide Carabellese. The first controller is the detumbler. The detumbler is responsible for the initial stabilization of the spacecraft after it has been deployed from the launch vehicle. The detumbler is also responsible for establishing the desired spin about the spacecraft body z axis. After the stable spin has been established, the second controller assumes responsibility for controlling the spacecraft orientation by nudging the spacecraft's z axis until it is aligned with the local magnetic field vector.

Detumbler

The detumbler controller uses what's called a Kane Damper. The Kane Damper is a fictional device that can be thought of as a uniform sphere inside the spacecraft bus. The sphere's only interaction with the bus is through a (fictional) viscous fluid between the two objects. The torque exerted on the spacecraft by this damper would be proportional to the rotation of the spacecraft with respect to the damper. The torque is proportional to a damping constant c .

$$\vec{\tau} = c \cdot \vec{\omega}^{B/D}$$

The damper functions just like a dashpot in a second order damped spring-mass-dashpot system.

The damper can be used to bleed of energy in a control scheme as well. The desired torque can be modeled in the following way.

$$\vec{\tau}_{des} = c \cdot \vec{\omega}_{des}^{B/N}$$

$$\vec{\omega}_{des} = \int \left(\dot{\vec{\omega}} - (I_d^{-1} c \vec{\omega} + \vec{\omega} \times I_d^{-1} c \vec{\omega}) \right) dt$$

Because the magnetorquers cannot generate a torque parallel to the magnetic field, the desired torque must be projected on a plane perpendicular to the magnetic field in order to obtain the physically realizable desired torque. The magnetic moment command becomes.

$$\vec{m}_{cmd} = \frac{\vec{\tau}_{des} \times \vec{B}}{|\vec{B}|}$$

The moment command is then turned into a current command by dividing by dividing by the number of coils and the cross-sectional area of the magnetorquer.

$$\vec{i} = \frac{\vec{m}}{nA}$$

The current then is run through a saturation block to ensure that the system does not command a current that cannot be physically obtained.

Orientation/Pointing Controller

Once the desired angular velocity is obtained. The orientation controller is tasked with slewing the spacecraft z-axis parallel to the local magnetic field vector. The controller used to accomplish this is a variation on proportional-derivative (PD) control. The desired error that needs to be driven to zero is the angle between the spacecraft z axis and the magnetic field, θ_{BZ} . The cross product between the two vectors can be written in the following form.

$$\vec{z} \times \vec{B} = |\vec{z}||\vec{B}| \sin(\theta_{BZ})$$

The equation can be inverted to solve for θ_{BZ} . Next, consider the angle between the angular velocity vector and the magnetic field. The angle between the two can be solved for using properties of the dot product.

$$\cos(\theta_{\omega B}) = \frac{\vec{\omega} \cdot \vec{B}}{|\vec{\omega}||\vec{B}|}$$

The function $sign(x)$ is used to extract the sign of the argument.

$$sign(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

The magnetic dipole moment command becomes:

$$\vec{m}_{cmd} = \frac{\left(\sin^{-1} \left(\frac{\hat{z} \times \vec{B}}{|\vec{B}|} \right) K_p + \frac{d}{dt} \sin^{-1} \left(\frac{\hat{z} \times \vec{B}}{|\vec{B}|} \right) K_d \right)}{|\vec{B}|} sign(\cos(\theta_{\omega B})) sign(\omega_3) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The magnetic dipole moment command is then scaled to obtain the current command, just as in the case of the detumbling controller.

Controller Architecture

The figure below shows the controller architecture. Both controllers always run at the same time. Depending on the spacecraft state, the outputs of one controller or another are used. This can be seen in the “manual switch” that has the orientation controller output, labeled “point”, selected. The switch has to be toggled by the user in the Simulink model. The automated switching function is handled at the embedded system level on the actual flight computer.

The controller directly interfaces with the spacecraft dynamics model that was covered in the previous section. The entire dynamics simulation shown in figure 3 is contained in the subsystem block labeled “Plantv5” in figure 6. The controller takes the magnetic field and angular velocity as inputs and outputs a three currents (shown as a three-by-one vector) back into the plant. The inside of the StarshotACS block is shown in Figure 5, with both controllers.

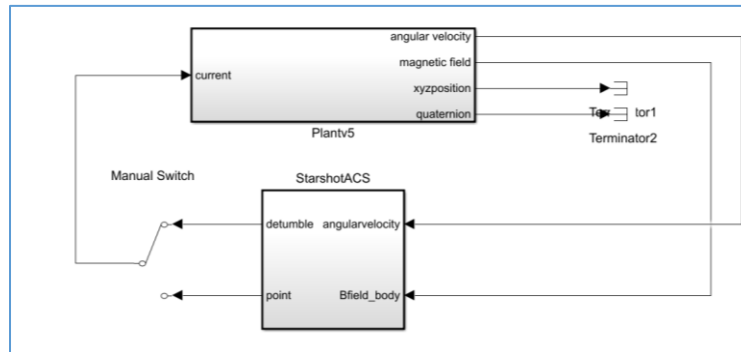


Figure 6: Interface Between Controller and Plant

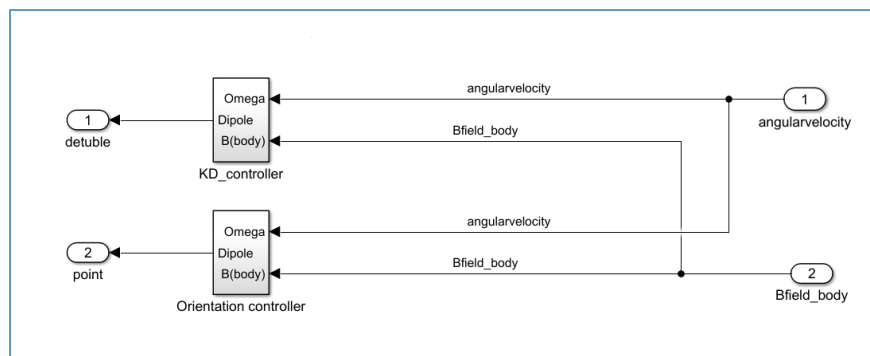


Figure 7: Detumbling and Orientation Controller Blocks

Autocoding

The flight software for Alpha is written in Arduino, essentially a superset of C++. The attitude control algorithm had to be in a form that could be implemented on the Adafruit M0 Adalogger, which functions as the vehicle’s attitude control system flight computer. Simulink has an autocode function that can translate Simulink blocks into C/C++ code. This section is a walkthrough of how to successfully autocode a Simulink block. By adhering to the following steps, the process shouldn’t take longer than ten minutes.

First, ensure that the Simulink model is working. During autocoding, errors are often vague and give little clue as to what the problem is. Ensuring that the Simulink model is debugged and all variables that are initialized through an external Matlab script are in the workspace can save a lot of time.

Open the model configuration parameters by clicking on the gear icon on the toolbar. It takes a few seconds to initialize and load but the following page should come up as shown in figure 8.

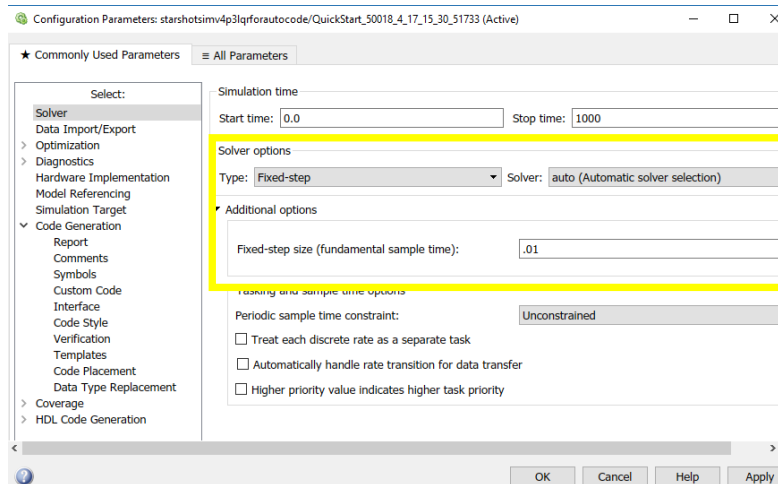


Figure 8: Autocoding Solver Configuration

Select the “Solver” option on the left side. Simulink cannot generate a variable step solver for autocoded systems. A fixed-step solver is required. Select the Fixed-step option, highlighted in yellow. Another parameter, “Fixed-step size”- the sampling time in seconds will appear. Enter the appropriate value. The figure above shows a sample time of .01seconds, or 100Hz. The start time should be set to zero. The stop time can be set to any value (preferably non-zero). It is up to the user to edit the autocoded system files in order for the stop time to actually be enforced.

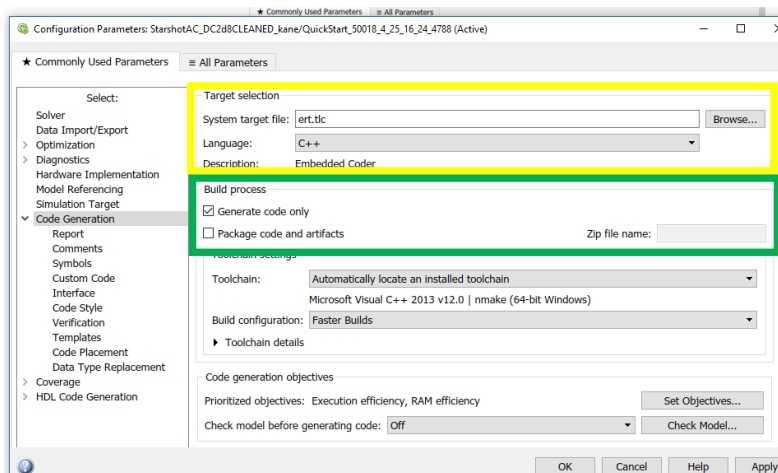


Figure 9: Autocoding Code Generation Configuration

Now select the “Code Generation” option on the left hand side. Select “ert.tlc” as the system target file. Select the appropriate language option, C or C++. All autocoding done for this project was done in C++. The subtleties associated with this choice will be discussed shortly.

Check the “Generate Code Only” box in the section of the page labeled “build process” (boxed in green). When finished, click “Apply” and close the window.

Next, right click on the desired block that is being autocoded and click the box labeled “Treat as atomic unit”, as shown below. When this box is not checked, the subsystem block hierarchy is

little more than a visual convenience to help declutter the Simulink model. When a block is treated as an atomic subsystem, it essentially becomes an isolated function of its own.

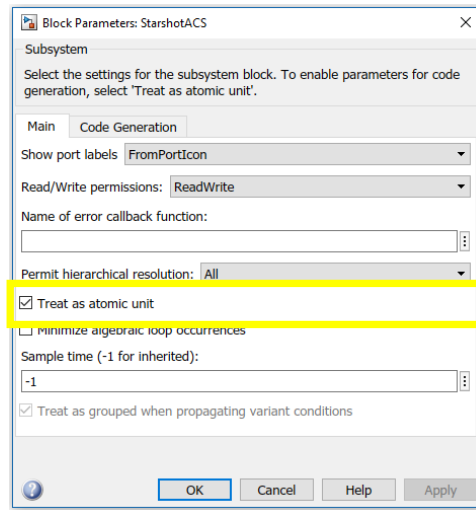


Figure 10: Subsystem Atomic Unit Designation

Close the window, and right click on the desired system block again, and select C/C++ Code. Then the Embedded Coder Quickstart option. A new window will pop up. Make sure the appropriate subsystem is highlighted i.e. “StarshotACS”.

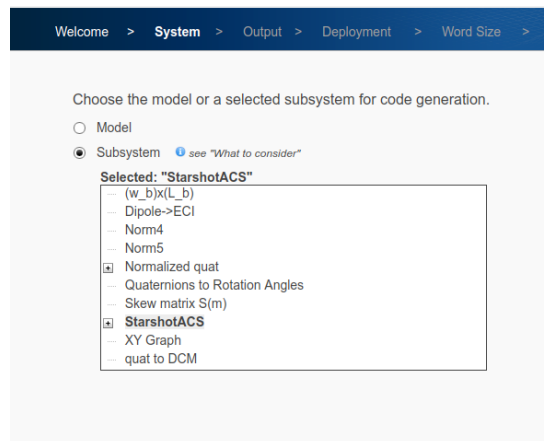


Figure 11: Subsystem Selection Verification

Click “Next” until the following window appears, showing the different options available for the code output. In order to output in C++, the integrator blocks in the system, if any exist, must be discrete time integrators. Simulink is able to autocode in C with continuous time integrator blocks in the model but unable to do so for C++. Click “Next” until the page asking about target hardware appears. Enter the following information for the Adafruit Feather as shown below.

Select your target hardware processor type. If your hardware processor is not listed, select "Custom Processor" to define your data type sizes.

Device Vendor:

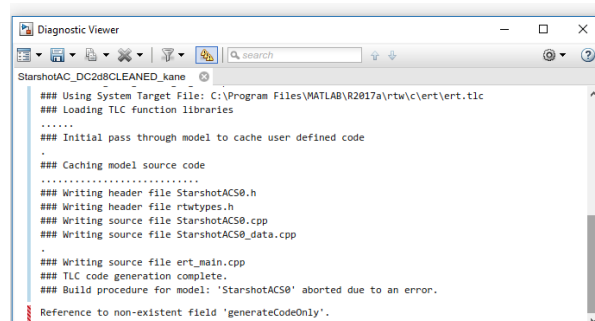
Device Type:

Number of bits

char:	<input type="text" value="8"/>	short:	<input type="text" value="16"/>	int:	<input type="text" value="32"/>
long:	<input type="text" value="32"/>	long long:	<input type="text" value="64"/>	native:	<input type="text" value="32"/>
pointer:	<input type="text" value="32"/>	size_t:	<input type="text" value="32"/>	ptrdiff_t:	<input type="text" value="32"/>

Figure 12: Autocoding Target Hardware Selection

The next page prompts the user to select to optimize between execution efficiency and RAM efficiency. At this point the difference between the two does not appear to be noticeable and either one should work. Execution efficiency has been selected thus far. The code generation should take a few minutes. An error will pop up claiming that there is a non-existent field “generateCodeOnly”. This is actually not a real problem. Notice that above the error, it is written that the code generation is complete.



```

Diagnostic Viewer
StarshotAC_DC208CLEANED_kane

### Using System Target File: C:\Program Files\MATLAB\R2017a\rtw\clert\ert.tlc
### Loading TLC function libraries
.....
### Initial pass through model to cache user defined code
.
### Caching model source code
.....
### Writing header file StarshotACS0.h
### Writing header file rtwtypes.h
### Writing source file StarshotACS0.cpp
### Writing source file StarshotACS0_data.cpp
.
### Writing source file ert_main.cpp
### TLC code generation complete.
### Build procedure for model: 'StarshotACS0' aborted due to an error.
Reference to non-existent field 'generateCodeOnly'.

```

Figure 13: Autocoding Error Reading

Navigate to the current working folder. There should be a folder named, in this case, *StarshotACS0_ert_rtw*. All the required code is located here. Place the folder’s contents in the Arduino library folder. This is only required of the C++ and H files. The text and .mat files are not needed.

It is recommended that the C++ files be edited with an appropriate IDE. Atmel Studio was used for this purpose. Sometimes, when C++ or H files are edited and saved in the Arduino IDE, they are recognized as Arduino files and if the user attempts to refer to them in code, the IDE will generate an error claiming that the file does not exist.

Once the autocoded files are in their own library folder, open the file “ert_main.cpp”. This is the file that is intended to run the entire simulation and it has a main function to do so. However, the flight code has its own main function and only one can be loaded on the flight computer. For that reason, everything in the ert_main.cpp file should be commented out. If this is not done, when the Arduino IDE compiles the flight code and reads the line to include the autocoded files as a

library, the `ert_main.cpp` main function will override the Arduino code and will be uploaded to the ACS flight computer instead.

File Overview

The autocode process generates several files. Only five are actual C++ files that are needed to run the simulation. The `ert_main` file has already been covered.

The `rtwtypes.h` file contains all of the data types that are constructed to execute the simulation. These are all common data types, many of which are simply renamed as “real time” data types. The `<systemname>_data.cpp` file stores constants such as the identity matrix. The `<systemname>.h` file is the most important file from an interfacing perspective. This file stores the C++ class attributes of the autcoded system including its inputs and outputs. Last is the file `<systemname>.cpp`. The guts of system are stored here in the system class’ step function. Calling this function will run the algorithm for one time step.

This completes the steps necessary to go from Simulink model to Arduino Library. The next section will cover the integration between the autcoded ACS software and the Arduino-written software that enables the ACS to take data from sensor inputs and generate current commands to the magnetorquers in real time.

ACS Software-Hardware Integration

The overall attitude control system consists of five unique hardware components. These are the Adafruit Feather M0 Adalogger, the Adafruit Feather M0 Feather Proto, the LSM9DS1 IMU/Magnetometer, the h bridge and the magnetic torquers (built in house). This section will cover four of the hardware components and the relevant software that has been written to integrate it into the overall ACS architecture. The magnetorquers will not be covered since the relevant software is used to communicate with the H-bridge and the material to construct the magnetorquers was unavailable at the time of writing.

Adafruit Feather M0 Adalogger

The heart of the attitude control system is the Adafruit Feather Adalogger. This microcontroller is responsible for communicating with all other ACS components, and all of the ACS software resides on this device. The Adalogger also is tasked with photographing the sail on deployment, storing the data from the photograph and sending the photograph data, along with ACS data, back to the main flight computer for transmission.

Real Time Operations

The attitude control system is expected to control the spacecraft in real time. Recall that when the parameters for the Simulink controller autocode were configured, one was the sample time. In order for the system to run in real time, two things need to happen. First, the attitude control algorithm must be able to run at least as quickly as the sampling rate specified in the autocode parameters. Even then, there is still nothing tying the algorithm’s sample time to real time on board the spacecraft. If the algorithm were put in a while loop, it would simply run as fast as it possibly could and would almost definitely not match real time.

The solution to this is a timer interrupt. Timer interrupts are a functionality of microcontrollers that allow functions to be run at a given time interval. Timer interrupts are triggered at a given interval that the programmer can vary. They are usually tied to an interrupt service routine- a function that runs outside the main code every time the timer interrupt is triggered. Timer interrupts are also advantageous because they are independent of the microcontroller's main function. This is extremely powerful as it allows critical tasks to be executed even if the main function is hung up.

If the control algorithm is set up as the interrupt service routine, and the control algorithm sampling time matches the time intervals at which the timer interrupt is triggered, then the control algorithm essentially becomes a real-time algorithm.

Setting up a timer interrupt involves combing through a microcontroller data sheet and configuring individual registers. The process is very well documented for most Arduino boards, but less so for Adafruit boards. A special third party library specifically designed to configure timer interrupts in one line was obtained for the task.

Setting up the interrupt and the ISR is simple. Excerpts from the file *acs.ino* are used to illustrate. First, the necessary library is referenced with an include line.

```
#include <avdweb SAMDtimer.h>
```

Afterwards, the interrupt service routine, named *myISR* in this example, is defined. Usually, the function that is to be run directly goes into the ISR. However, the attitude control algorithm wouldn't function correctly when this was attempted and a work-around approach was adopted. The primary purpose of the ISR is to switch the integer called *permission* to one every time the interrupt is triggered. Every time *permission* is set to one, the ACS algorithm is allowed to run in a separate function. After the algorithm runs, *permission* is set to zero and the ACS algorithm cannot run for another time interval until the interrupt is triggered again and *permission* is set to one once more.

Setting the variable *pinset* to high or low toggles the Adafruit Feather's LED to on or off. This is simply a debugging tool to help ensure the ISR is functioning as intended. It must be removed for the flight build as the physical pin is allocated to a hardware connection.

```
void myISR(struct tc_module *const module_inst){
    if(pinset==HIGH){
        pinset=LOW;
    }
    else{
        pinset=HIGH;
    }
    permission=1;
}

SAMDtimer mytimer=SAMDtimer(4,myISR,1e6);
```

Figure 14: Interrupt Service Routine

Below the ISR, the timer interrupt is set up with the *SAMDtimer* function, which has three arguments. The first argument is the timer that is being referenced. The second argument is the interrupt service routine, defined above. The last argument is the time interval in microseconds. In the example above, the time interval is set to one second for debugging purposes.

Autocoded Attitude Control Algorithm Integration.

The autocoded ACS algorithm can be used as a library and referred to with the following line.

```
#include <StarshotACS0.h>
```

The *ert_main* file that was entirely commented out is a useful reference for integrating the autocoded system into the Arduino IDE. Simulink's autocoder creates classes out of autocoded blocks when the C++ option is selected. First the class instance must be instantiated with the following line.

```
static StarshotACS0ModelClass rtObj;
```

The initialization function *rtObj.initialize* is included in the Arduino setup function, but the function itself does not contain anything and is just included for the sake of thoroughness.

For interfacing with the ACS algorithm's inputs and outputs, it is useful to consult the H file that is referenced in the include line, in this case *StarshotACS0.h*. The most important eight lines of code are shown below in Figure 15.

```
61 // External inputs (root inport signals with auto storage)
62 typedef struct {
63     real_T angularvelocity[3];           // '<Root>/angularvelocity'
64     real_T Bfield_body[3];              // '<Root>/Bfield_body'
65 } ExtU;
66
67 // External outputs (root outports fed by signals with auto storage)
68 typedef struct {
69     real_T detumble[3];                  // '<Root>/detumble'
70     real_T point[3];                    // '<Root>/point'
71 } ExtY;
```

Figure 15: System H File Input and Output Structs

The structs *ExtU* and *ExtY* are the inputs to and the outputs of the ACS algorithm, respectively. To provide the algorithm with sensor data, *ExtU* must be tied to the sensor readings from the IMU/magnetometer. Similarly, the currents that must run through the magnetorquers to achieve the desired torque on the spacecraft must be set to the output of the ACS algorithm. A screenshot of the *runsim* function is provided below to illustrate the process.

```

void runsim(){
    if(permission==1){

        sensors_event_t accel, mag, gyro, temp;
        lsm.getEvent(&accel, &mag, &gyro, &temp);

        rtObj.rtU.angularvelocity[0]=gyro.gyro.x*3.14159/180.0;
        rtObj.rtU.angularvelocity[1]=gyro.gyro.y*3.14159/180.0;
        rtObj.rtU.angularvelocity[2]=gyro.gyro.z*3.14159/180.0;
        rtObj.rtU.Bfield_body[0]=mag.magnetic.x;
        rtObj.rtU.Bfield_body[1]=mag.magnetic.y;
        rtObj.rtU.Bfield_body[2]=mag.magnetic.z;
        rtObj.step();
        count=count+1;
        //rtObj.rtY.detumble;
        //rtObj.rtY.point;

        if(ACSmode==0){
            current1=rtObj.rtY.detumble[0];
            current2=rtObj.rtY.detumble[1];
            current3=rtObj.rtY.detumble[2];
        }
        if(ACSmode==1){
            current1=rtObj.rtY.point[0];
            current2=rtObj.rtY.point[1];
            current3=rtObj.rtY.point[2];
        }

        permission=0;
    }
}

```

Figure 16: Autocode I/O Integration in Flight Code

If the interrupt service routine gives the algorithm permission to run, the ACS algorithm is executed in the following way. First, the algorithm input rtU is populated with sensor data. This portion is boxed in yellow. Recall that the algorithm's structs and functions are attributes of the algorithm's class. Therefore they must be called using dot notation. Once the system inputs are configured, the ACS algorithm is run for one time step using the step function, boxed in green.

The algorithm runs both a detumbling controller and the orientation/pointing controller. Depending on the state of the spacecraft, one of two ACS modes is selected. The currents are then set to the output of one of the two controllers. The method by which the current command is sent to the magnetorquers will be covered next.

TB6612FNG H-Bridge Breakout Board

Once the desired current commands are obtained, they must be somehow sent to the magnetorquers. Most microcontroller pins are not designed to do this. They can generally only output zero or five volts, corresponding to low and high logic. This is inadequate for achieving the desired current magnitude and the desired current direction. To solve the problem of current magnitude, a method called pulse-width-modulation is used, where a digital signal oscillates between zero and (in this case) five volts. By controlling the amount of time spent at the maximum voltage over once cycle, called the duty cycle, the digital pin can create a signal that is effectively between zero and five volts.

To solve the problem of current direction, a device called an H-bridge is used. Without going into too much detail, an H-bridge is able to route a current in one of two directions through an actuator. The direction is decided by two specific signals that are sent from the microcontroller.

Each TB6612FNG contains two H-bridges. Two are used for three magnetorquers. An image of the board breakout is shown below.

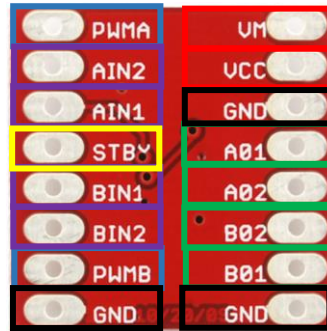


Figure 17: H-Bridge Pinouts

The pins labeled PWMA and PWMB are the two signals that set the magnitude of the current going through each magnetorquer. These pins are boxed in blue. The pins AIN1, AIN2, BIN1, and BIN2 are for the signals that are used to set the direction of the current going through each of the two magnetorquers connected to the H-bridge. The pins that are boxed in green, AO1, AO2, BO1, and BO1 are directly connected to the leads of the two magnetorquers. The standby pin, boxed in yellow, is used to turn the system on or off without cutting power to the board. In order for the system to be on, the standby pin logic should be high (5V). The pins boxed in red, VM and VCC, are power inputs. The ground pins should be connected to the overall ground voltage of the entire electrical system.

The spacecraft electrical has been designed with specific pinouts on the Adafruit Adalogger assigned to specific pins on the 2 H-bridge breakout boards. The specific setup is shown below in figures 18 and 19.

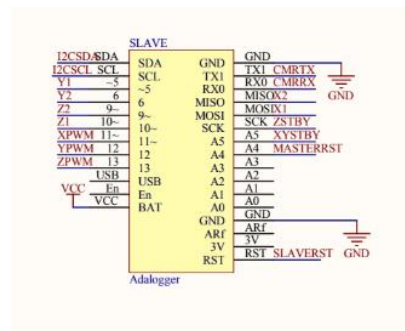


Figure 18: Adafruit Feather Adalogger Assigned Pinouts

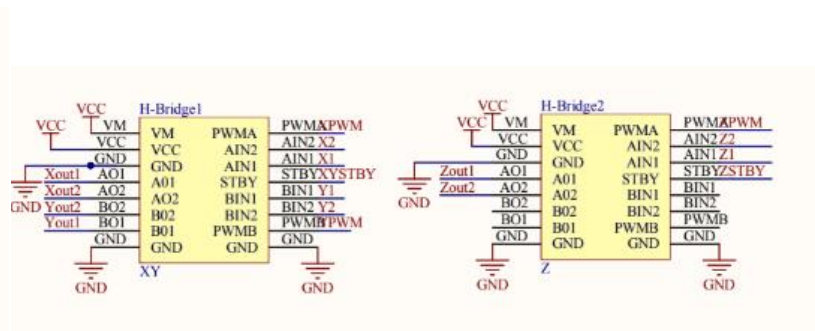


Figure 19: H-Bridge Assigned Pinouts

In order to simplify the process of relaying the current commands to the H-bridge, a special function was created to abstract the entire process into one line of code. The function is called *ACSwrite* and contains five arguments. An example is provided below for the magnetorquer that is aligned with the spacecraft body x axis.

ACSwrite(Xtorqorder,current1,xout1,xout2,xPWMpin)

The third and fourth arguments of the function correspond to the pins that command the current direction, as specified in the electrical design. Current1 is the current command set by the ACS algorithm. *xPWMpin* corresponds to the pin on the Adafruit feather that transmits the PWM signal.

The first argument, *Xtorqorder*, addresses the default polarity of the magnetorquer meaning, i.e. which way is positive and which way is negative. By default, the argument is set to one for each of the three magnetorquers. This corresponds to the signal output pin xout1 being set to 5V and the signal output pin xout2 being set to zero volts for positive current commands. For negative commands, the order is reversed, with xout2 set to 5V and xout1 set to 0V.

If the magnetorquer is installed incorrectly, switching *Xtorqorder* from one to zero will reverse the polarity, and change the direction that is default positive to negative and vice versa.

To make matters easier, the pin variables are set to their designated pin numbers at the beginning of the *acs.ino* script: e.g. *xout1=23*, *xout2=22* etc. This means that the function call seen above can simply be run as is without consulting the system level electrical design diagrams and manually entering pin numbers.

At the time of writing, magnetorquers were not available due to supplier delays. The function was tested for each of the three axes using LED's to verify that the default and backup polarities worked as intended for both positive and negative commands. When the materials for magnetorquers are available and the magnetorquers are built and ready, ACSwrite should be tested to ensure that the current through the magnetorquers matches the commanded current.

Unlike the ACS algorithm, which is run at a prescribed time interval, the ACSwrite function calls are placed in the while loop of the main function, meaning that they run continuously in between the timer interrupts as well.

LSM9DS0

The LSM9DS0 is a nine-degree-of-freedom sensor consisting of an accelerometer, gyroscope and magnetometer. The device's on board gyroscope and magnetometer are used as the inputs to the ACS algorithm. The sensor communicates with the Adafruit Adalogger using the I2C communication protocol. The communication can either be set up manually or by using two libraries that are built for the device. The latter was chosen to save time and because the library functions output data in SI units. The libraries are first included at the beginning of the *acs.ino* script.

```
#include <StarshotACS0.h>
#include <Adafruit_LSM9DS0.h>
```

The specific functions are called when the timer interrupt is triggered. The two lines that are boxed in yellow create a “sensor event” and store the sensor data. The data is then taken and used as inputs to the ACS algorithm as previously described.

```
sensors_event_t accel, mag, gyro, temp;
lsm.getEvent(&accel, &mag, &gyro, &temp);

rtObj.rtU.angularvelocity[0]=gyro.gyro.x*3.14159/180.0;
rtObj.rtU.angularvelocity[1]=gyro.gyro.y*3.14159/180.0;
rtObj.rtU.angularvelocity[2]=gyro.gyro.z*3.14159/180.0;
rtObj.rtU.Bfield_body[0]=mag.magnetic.x;
rtObj.rtU.Bfield_body[1]=mag.magnetic.y;
rtObj.rtU.Bfield_body[2]=mag.magnetic.z;
```

Figure 20: Gyro/Magnetometer Library Code

Adafruit Feather M0 Proto

The Feather M0 is the master flight computer, connected to the ACS flight computer, power, system and radios. It does not run any ACS code, but it does relay commands to the Adalogger. The Adalogger is also responsible for sending requested data pertaining to the spacecraft attitude. It must additionally send the photographic data, stored on an SD card, to the Feather M0 Proto, to be transmitted to the ground station to verify mission success. Much of this data can be classified as mission-critical or flight critical.

I2C connection between Adafruit Feather Adalogger and M0 Proto

In preparation for flat-sat testing, the I2C connection between the two computers was identified to be faulty. The fault was traced to two root causes, one software and one hardware.

The hardware issue was the lack of pull-up resistors connected to the I2C clock and data lines between the two microcontrollers. The default logic in I2C for both lines is high, because the I2C drivers can only pull the signal line low, not drive it high. Pullup resistors connected to a voltage source of 3-5 volts are required. Some microcontrollers have this set up internally. The Adafruit Feather family does not.

The fix was first implemented on a breadboard. Two Adaloggers were set up with two external 2.7k Ω resistors connected to a 3.3V voltage source on one of the Adaloggers. Lower resistivity corresponds to higher data rates and 2.2k Ω -10k Ω is recommended for the Adafruit Feather. This was sufficient to establish basic communication between the two microcontrollers. The breakout board design on which both microcontrollers sit was modified with the fix.

The hardware fix was only one part of the issue. The second issue was the versatility, or rather the lack thereof, of the Arduino I2C library. The functions to read the incoming I2C data could only read 1 byte at a time. A new function that could read and concatenate several bytes was necessary.

Several methods were attempted in-house, but could not successfully satisfy successful data transfer of small numbers with floating point precision, which would be lost to round-off error during the casting and recasting process. A third party library was obtained for this purpose, conveniently called I2C_Anything, and was successfully able to transmit and receive floating point precision quantities in their entirety.

While the library's initial performance is promising, it should be tested with every data type that can be expected to be transmitted over the I2C bus, considering the importance of this communication interface.

Embedded System Level ACS Simulation

To ensure that the autocoder has in fact built the ACS algorithm correctly, the autocoded ACS algorithm must be tested with simulated dynamics and sensor data. Ideally, the autocoded ACS algorithm, implemented on the Adafruit Feather, would receive simulated IMU and Magnetometer data, send magnetorquer commands to the actual magnetorquers and a data acquisition board would complete the loop by feeding current data to the Simulink Dynamics model over a serial connection.

Towards the end of the spring 2018 semester, the materials for the magnetorquers had not arrived and it was clear that development of a Simulink-Arduino interface via serial connection was a project that would likely require months of work.

A new simulation setup was conceived that would work around these issues. The ACS algorithm was implemented on the Adafruit feather just as it would be for flight in a script called *acssimv2*. The spacecraft dynamics model was autocoded as well and incorporated into the file *plantsimv2*, which was implemented on a second Adalogger. An I2C connection was established between the two microcontrollers. The Adalogger with the ACS software was set up as the master, and the Adalogger with the dynamics simulation was set up as the slave. The ACS Adalogger would request gyroscope and magnetometer measurements, which the dynamics equipped Adalogger would send over the I2C bus. The ACS would then send current commands back over the I2C line to the dynamics simulation, completing the loop.

A few very important subtleties complicated the implementation. The first was run time. The ACS algorithm and the Spacecraft Dynamics, exist on the same Simulink model as can be seen in figure 6. This means that the two are run with the same sampling time. In order for the Simulink simulation to converge, the model has to be run at with a minimum sampling rate of 100Hz. This is true for the simulations written both by the author, and by Davide Carabellese.

The 100Hz limit meant that the entire embedded system loop cycle time, starting from sending the initial data request, receiving I2C data, running the ACS algorithm and sending commands back out, had to be less than .01 seconds in duration. The cycle time for the ACS algorithm was well within this, averaging at about 0.002 seconds. The run time for the supporting functions however, pushed the overall system run time over the 0.01 second limit, meaning that the simulation could not be run in real time. The simulation was run at 1/3 real time. A simulation for identical initial conditions was run in Simulink. The Simulink simulation detumbled the spacecraft in about 5200 seconds. The autocoded ACS algorithm was able to match this performance. The final angular velocity state was verified through the serial monitor.

Conclusions and Recommendations

At the time of writing Alpha's ACS had been developed to a point where it was ready for testing at the flat-sat level. Hardware design issues have been rectified, and the software had been developed to a point where all of the various ACS elements could integrate with one another. The ACS embedded system level test had demonstrated that the autocode process had successfully duplicated the Simulink model of the attitude control algorithm. Based on the author's experiences during the hardware integration process, a few items contain the highest risk.

First is the magnetorquers, simply because all of the relevant software to use them was developed without the magnetorquers available in house. When the appropriate materials are received, and the magnetorquers are built, they should be tested with the ACSwrite function. Additionally, the current Simulink simulations use estimates for the magnetorquer parameters. Once the magnetorquers built and tested, these parameters should be updated in the simulation and the simulation should be re-run to ensure that the controller can still stabilize the CubeSat about the desired state.

Second, the I2C communication proved to be extremely troublesome. The embedded system level ACS simulation revealed the deficiencies in the Arduino I2C library and valuable knowledge was gained in rectifying these deficiencies through the simulation. However, the library has not been tested for several of the data types that will need to be transferred between the two Adafruit Feathers. Ensuring that the fixes are applicable to all the necessary data types should be a high priority.

Third, while the embedded system level ACS simulation was able to match the performance of its Simulink counterpart, the setup is not practical in any way. A concerted effort should be made to develop a serial interface between the autocoded ACS algorithm and the dynamics model in Simulink. Doing so will create the possibility of adding many more features with ease. The autocoded ACS algorithm could run at a separate and more manageable sampling time. The simulation can be run with a more accurate magnetic field model, instead of the Earth magnetic dipole model that is being used now. Sensor noise can be introduced to the magnetometer and gyroscope measurements.

Alpha is not the only SSDS project that would benefit from this. PAN has already made progress in creating a serial-Simulink interface with similar purpose. A basic interface could be developed to service both teams. Each team could then tailor the interface to their own specific needs.

References

F Landis Markley, John L Crassidis, *Fundamentals of Spacecraft Attitude Determination and Control*, pages 39, 45, ISBN: 978-1-4939-0801-1

D.M. Torczynski, R. Amini, *Magnetorquer Based Attitude Control for a Nanosatellite Testplatform*, AIAA 2010-3511, presented 20-22 April 2010

E. Siani, M. Lovera, *Magnetic Spacecraft Attitude Control: A Survey and Some New Results*, Control Engineering Practice, Volume 13, Issue 3, ISSN 0967-0661

Appendix 1. Linear Quadratic Regulator

A linear quadratic regulator was the original candidate for second controller, responsible for achieving the final pointing. The error that needed to be driven to zero stemmed from two sources. The first was alignment error between the spacecraft z-axis and the local magnetic field. The second was the difference between the desired angular velocity and the current angular velocity. The alignment error was quantified through an error vector \vec{e} taken to be the cross product between the angular velocity vector and the magnetic field. If the two are parallel, then the error vector becomes zero.

$$\vec{e} = \vec{\omega} \times \vec{B}$$

Taken in matrix form, the equation becomes.

$$\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} B_3\omega_2 - B_2\omega_3 \\ B_1\omega_3 - B_3\omega_1 \\ B_2\omega_1 - B_1\omega_2 \end{bmatrix}$$

If the derivative of the magnetic field $\dot{\vec{B}}$ is assumed to be near zero, then the error rate can be approximated to be the following

$$\begin{bmatrix} \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \begin{bmatrix} B_3\dot{\omega}_2 - B_2\dot{\omega}_3 \\ B_1\dot{\omega}_3 - B_3\dot{\omega}_1 \\ B_2\dot{\omega}_1 - B_1\dot{\omega}_2 \end{bmatrix}$$

The approximation allows the error vector to be expressed in terms of the state derivatives of angular velocity, which is a nonlinear equation in the following form.

$$\dot{\vec{\omega}}^{B/N} = \mathbf{I}^{-1}(\vec{\tau} - \vec{\omega}^{B/N} \times (\mathbf{I}\vec{\omega}^{B/N}))$$

The system can be linearized by taking the Jacobian of the function producing a matrix A . The analytical form is long and will instead be used in the abbreviated form.

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

The Jacobian can be used to compute the derivative of the error angle by substituting it into the matrix equation for error angle rate.

$$\dot{e}_1 = (B_3A_{21} - B_2A_{31})\omega_1 + (B_3A_{22} - B_2A_{32})\omega_2 + (B_3A_{23} - B_2A_{33})\omega_3$$

$$\dot{e}_2 = (B_1A_{31} - B_3A_{11})\omega_1 + (B_1A_{32} - B_3A_{12})\omega_2 + (B_1A_{33} - B_3A_{13})\omega_3$$

$$\dot{e}_3 = (B_3A_{21} - B_2A_{31})\omega_1 + (B_3A_{22} - B_2A_{32})\omega_2 + (B_3A_{23} - B_2A_{33})\omega_3$$

The Jacobian matrix for the entire system can be stated in terms of A and the matrix coefficients of the linearized system for $\dot{\vec{e}}$, written as the matrix A' .

$$A_{sys} = \begin{bmatrix} 0_{3 \times 3} & A' \\ 0_{3 \times 3} & A \end{bmatrix}$$

The B_{sys} matrix for the system can be computed to be inverse moment of inertia multiplied by the magnetic field B^\times matrix, stacked underneath a 3x3 matrix of zeros, as the torque does not directly affect the matrix.

$$B_{sys} = \begin{bmatrix} 0_{3 \times 3} & & \\ I^{-1} \begin{bmatrix} 0 & B_3 & -B_2 \\ -B_3 & 0 & B_2 \\ B_2 & B_{-1} & 0 \end{bmatrix} & & \end{bmatrix}$$

The input magnetic dipole moment can be written as:

$$m = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix}$$

The linearized system can be written in state-space form:

$$\begin{bmatrix} \dot{e} \\ \omega \end{bmatrix} = \begin{bmatrix} 0_{3 \times 3} & A' \\ 0_{3 \times 3} & A \end{bmatrix} \begin{bmatrix} e \\ \omega \end{bmatrix} + \begin{bmatrix} 0 \\ I^{-1} B^\times \end{bmatrix} m$$

At this point the optimal gain matrix can be obtained with Matlab's built in LQR function. The controller was tested with the nonlinear dynamic Simulink model. The controller could stabilize the model at times but not without unacceptable steady state error in angular velocity. The root cause was never determined, since the other controller design had made much more progress. One possible explanation is that the linearization occurs for a single magnetic measurement. During the pointing phase, the spacecraft is rotating and the magnetic field measurement in the body reference frame would change as well.

A possible solution would be to continuously update the optimal gain matrix using the updated magnetic field measurements. This would however, have been computationally expensive. Matlab's LQR solver would have to be run at every time step. Autocoding this functionality would likely have been extremely challenging. The concept might not be applicable to this spacecraft but a more demanding optimal control application might be able to make use of this work.