MATLAB Environment for Satellite Simulation: Development Guide

Burlion Research Lab
Undergraduate Research
Manav Jadeja, Sunny Cheng

Preface

This will likely be written as a type of story, making it easy for new people to follow along and understand the thought process that went into making this. At the same time, I will be using a top-down approach when explaining this product: starting with the overall structure of this program, going into the independent functions, and finally delving into the scripts that comprise it. In doing so, I will be making mention of how one would go about adding personalized edits and functionality dependent on mission goals. That being said, I will not be teaching you how to code each individual line. That is something that can only be learned through a head first dive, something I would recommend doing only if you have excessive time and a strong desire to learn more about satellite system simulation and the related areas of research. One final detail I want to make clear is that this is not a tutorial. I do not have the time nor patience to sit here teaching people all of the little details. You will have to take for granted some of the choices I made, though I will leave simple reasoning and maybe some sources to look at should one desire to learn more on their own. I do make tutorials on YouTube every once in a while as a sort of training but I would not recommend them to be your sole resource for learning this material (they are fun projects and I would recommend them if you are just getting started).

And that is about all I really need to say. This project was made possible by two main people. The first and foremost one is my advisor, Professor Burlion. It was only thanks to his research into slosh control for satellites that I was able to work on this amazing project and I am eternally grateful for the opportunity and continued guidance over the course of this project. The second one is Connie Liou, the enthusiastic and extremely energetic senior of mine who first helped me get involved. Being both a close friend and guide on my crazy adventure with nanosatellites, she has been an extremely crucial factor for keeping me going on this project despite the numerous shortcomings I had when starting out. Furthermore, in founding the Rutgers SPICESat mission, she was the one who suggested I go out and create tools to facilitate mission design and planning. I am eternally grateful to both of these people for giving me the opportunity to get involved and I cannot thank them enough for their support and guidance.

-Manav Jadeja

Table of Contents

Chapter 0: Overview

This is just going to be a quick overview of each chapter and what I plan on covering. Just something I feel is necessary if you only need specific bits of information regarding overall structure and the various components of this project. So the project name is "MATLAB Environment for Satellite Simulation". Taking the first letter of every word, we get the acronym MESS and that is what I will be referring to from now on. I would add a list of acronyms to this document, but I feel like that is a bit overkill considering the people reading this will likely be much smarter than me (in which case they know what I am saying without being told), or extremely new to this (in which case, just google the acronym with the work 'satellite' or 'space' at the end).

Anyways, the goal of MESS is to create an environment for satellite system-level simulation. Since a satellite consists of many interacting subsystems, each of which is its own area of research, creating a useful simulation with all of its components is a very computationally intensive task. As such, unless one happens to possess a supercomputer and the funds necessary to create optimal code for it, appropriate simplifications must be made to permit useful results whilst keeping computational time down as much as possible. And such is the goal of MESS: to be a *relatively simple* computational tool which encompasses the key features a *small team* would need to *quickly* perform mission design relevant analysis. Take note that the italicized words are not there to say the tool is simple, quick, or even usable for a small team. To put some references in here, the simulation, at its current state, runs a simulation with 12 state variables, soon to exceed 20, and takes roughly 5 minutes to simulate 3 hours (on my not-so-impressive desktop). The project was created by a team of me, with some help on the specialized areas by other people, though I hope to expand this in the future. As you would expect, this is not necessarily the norm when it comes to industry, where teams are larger, more state variables are used, and the computational time could be much longer. It is simply a lot in the eyes of someone new in the field (i.e. me). In regards to planning a nanosatellite mission, I would say MESS can offer at least some help when it comes to early stage advice, providing confirmation for proof of concept.

It is also meant to serve as a small bridge between the research team and the build team for this mission. The research team focuses on the attitude determination and control system (and its interactions with slosh) while the build team has to figure out the rest of the satellite that would meet the requirements set forth by the research team. MESS is designed to make it easy to communicate potential issues as well as areas of large margin effectively and with "concrete" simulation data, as opposed to a simple excel sheet containing a raw power budget which a member of the research team may not understand at first glance.

That being said, MESS has plenty of limitations and its results should not be trusted as truth. As with any simulation, it is only as accurate as its worst simplification and above all, it is neither tested nor a high-fidelity model that can simulate every aspect of a satellite. It is simply a guiding tool and should be used with caution and a grain of salt. But in terms of preliminary design, planning, and searching for potential issues, it should serve its purpose well enough to be used.

0.1 Structure

To create an accurate and easy to use simulation tool, the code of MESS is written in MATLAB, the default industry-standard engineering programming language. Being a relatively simple language, most engineering students (i.e. those with limited knowledge of programming) will have little to no difficulty in understanding the code. Furthermore, MESS takes heavy advantage of Systems Tool Kit (STK), the primary data source and visualization tool for space based systems. STK is a leading industry standard when it comes to multi-domain complex mission modeling, computation, and visualization. However, the educational version of this software provides more than enough tools and modules for us, which MESS takes full advantage of. In short, to run MESS, one would need MATLAB, STK, and the STK MATLAB Integration Module. This module permits us to use the data generation and visualization capabilities of STK with the computational freedom offered by MATLAB to create a satellite simulation environment[1]. The overall structure of MESS is as follows: Use the STK MATLAB Integration Module to create the STK environment and transfer the relevant environment data to MATLAB (Chapter 1), then run a simulation in MATLAB using the loaded data and load the results back into STK (Chapter 2). Finally, try some basic visualization (Chapter 3). At the very end, is a chapter about future work (Chapter 4).

0.2 Mission Specific Goals

Something I would like to mention here is that this project was done in conjunction with a Cubesat mission at Rutgers University. As such, MESS was made to aid the development of that mission. The mission is called SPICESat and revolves around deploying a nanosatellite with a large, unfilled tank of liquid inside. When the nanosatellite performs an aggressive attitude maneuver, the liquid in the tank will be excited and sloshing will occur. This sloshing fluid induces a disturbance torque, which will interfere with the attitude maneuver, potentially causing issues with the pointing of the satellite. The mission revolves around inducing this behavior, collecting data regarding it, and then deploying an active controller which accounts for the disturbance. The full mission consists of performing many such experiments and recording data in the tank via pressure sensors and video.

Since a large portion of our mission revolves around attitude determination and control system (ADCS) and command data handling (CDH), MESS was designed to provide an environment for modeling these aspects of the mission more accurately than the others. To be a bit more specific, the systems that MESS is primarily designed for are the ADCS and CDH, along with a basic model for an electrical power system (EPS) and communications system (COM). Granted these are the aspects of every mission, they have varying degrees of importance for us and MESS reflects this very clearly in its structure (as you will find out should you keep reading). Should the reader try to modify MESS (or even go out to create their own unique variation), it is highly recommended that they do so with a mission in mind as this provides an orderly path to follow for development. Regardless of the purpose, mission needs should be simulation needs.

[1] As a side note, it is entirely possible to use this combination of tools to model more than just space systems. One could very well use STK to model land-air-sea-space systems, however, this is beyond the scope of an undergraduate project (in my opinion at least) and will not be mentioned further.

0.3 Architecture

MATLAB Environment for Satellite Simulation
This tool simulates the satellite system dynamics by calculating dynamics for each time step. It uses a numerical solver, the 4th order Runge-Kutta, to compute the attitude subsystem. The command and the power subsystem have linear simulators. The tool closely integrates with STK to extract required input data for the simulation.

The command subsystem simulates the brain of the satellite. It has branching logic to determine the current flight mode of the satellite. Subsystems use the flight mode to determine what action to take. The command subsystem also tracks data capacity, simulating storage of data from the payload and downlinking by the communications subsystem. Each flight mode determines the data delta, (either increase or decrease) for each time step. The user sets the data deltas for each flight mode.

The power subsystem aims to simulate the battery's state of charge to verify safety margins are being met. It also provides granular voltage, current, and charge data for each solar array and the battery. Information about sun orientation is determined from STK, and the flight mode is used to determine whether or not to charge. The logic is adopted from a power simulator created by GSFC ETD, located here.

The attitude subsystem simulates the attitude dynamics of the satellite along with the main components of the attitude determination and control system: the magnetorquer, the reaction wheel system, and the star tracker.

The data regarding the ground station and sun orientation is extracted from STK. The magnetorquer returns the magnetic dipole moment given the magnetorquer magnetic dipole and the Earth's magnetic field. The reaction wheel system returns the control torque for the satellite to orient to a given attitude. The star tracker returns an estimate of the current attitude with custom noise. The attitude system has a method for simulating the attitude dynamics (given parameters such as inertia, current orientation, angular velocity, and the control torque) using the Runge-Kutta 4th Order Solver. All equations are taken from *Fundamentals of Spacecraft Attitude Determination and Control* by F. Landis Markley and John L. Crassidis.

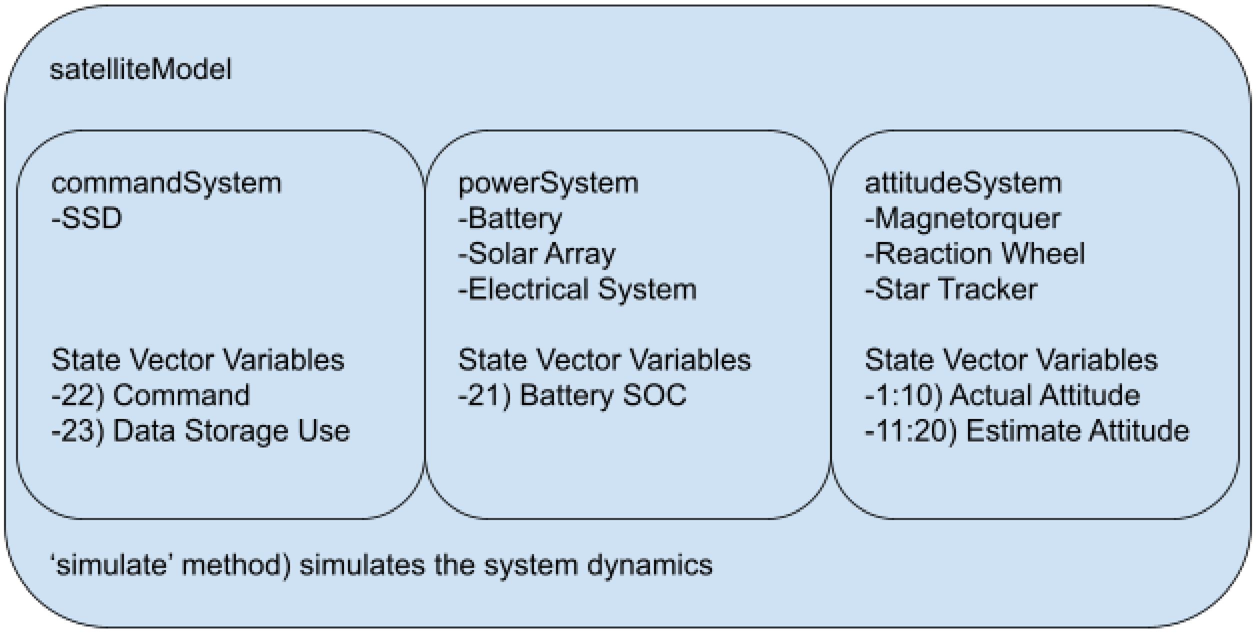

Figure: MATLAB objects hierarchy.

For information on how to run the simulation itself, the introductory paragraph in Chapter 4: Running Simulations and Future Work provides a VERY quick rundown of how to do this. However, it should be noted that this tool, despite my best efforts at keeping it simple, is anything but that. To delve into the specific components will not be easy though I hope this documentation provides the information necessary to make sure edits should one desire to.

Chapter 1: STK Data Extraction

In this chapter, I will be going over the simulation parameters defined in MATLAB. So the parameters used here fall into one of several categories: scenario parameters and system parameters. Scenario parameters are related to the parameters necessary for setting up the scenario, including things like scenario start and stop time, and even ground station information. This is generally for parameters that need to be defined for STK to create a full scenario. The system parameters are for MATLAB and will be used by MATLAB to run simulations for the system dynamics. While we will not be using the system parameters until Chapter 4, it is important to define them now as they are part of the input. Ok, that's enough nonsense, let's just get right into the code!

1.1 Preliminary Setup

Ok, first thing's first, we need to add all the relevant paths and folders to our main area. So let's do just that! The following lines add the folders 'lib', 'res', 'src', and 'tmp' to our environment and we can now use all the functions, files, and subfolders within them.

```matlab
% ADDING FOLDERS TO PATH (TEMPORARY)
addpath(genpath('lib'));
addpath(genpath('res'));
addpath(genpath('src'));
addpath(genpath('tmp'));
```

While this may seem like an obvious thing to do for a slightly experienced programmer, I did not know of this for quite a while and was the type of person to put everything into the main folder. So in the unlikely but plausible chance that there is someone out there undertaking a large project such as this without the expected experience, then something like this would help avoid a lot of organization hurdles.

While I'm here, I will also add another neat trick that helps quite a bit but may be glossed over in an introductory MATLAB for Engineering course: the extremely useful 'tic' and 'toc'. These are built-in timers and they work as follows: 'tic' will start a timer and 'toc' will report the time since the previous 'tic'. While this isn't something to run all the time, it does help with debugging and finding areas of code that should be optimized to increase efficiency. Below is the basic code for use of these functions, the second of which is my preferred method.

```matlab
% TIMING CODE
tic;
% INSERT CODE HERE
toc;
% disp(['Task Completed: ', num2str(toc), ' seconds'])
```

1.2 Scenario Setup

Ok, there is no need to put a long introduction paragraph here. Our goals are as follows: start STK, modify the scenario to our needs, add some objects, compute access, and then put them into MATLAB. This sounds like a lot of work and a long chapter so let's jump right into it. Below are some notes I want to add here for the reader before they get into this section. They aren't required but they can offer some help to those who are new.

First of all, all the functions for adding objects into STK use the MATLAB Integration Module. For the body of this documentation, I will only be putting the function line and comments regarding its use, the full function will be in the appendix. Furthermore, all functions that create objects within STK will be placed in the following location.

```
\src\system tool kit\
```

Secondly, these functions have a small section (usually towards the end), called "Graphics Stuff". This isn't really useful for the simulation itself, but rather just helps with fonts, colors, models, and persistent visibility in STK. It's not the most important to know how they work (in all honesty, I don't know this either), but they do make it easier to look at (in my opinion at least).

Third, I have colored my code to try and make it easier to read. The objects used by STK have the following format, `stkObject`. I also try to stick with the normal MATLAB format, with `% comments` and `'strings/char arrays'` (there might be some issues with the apostrophe if you try to copy paste code from this document so just take it off github or write it yourself).

1.2.1 STK Scenario

The majority of this code is taken from the official AGI help page for STK MATLAB Code Snippets Page. Of course, I will be defining some MATLAB functions to facilitate organization and this will come into use later on. Anyways, this is the code to start STK.

```
%%% STK LAUNCH
uiApplication = actxserver('STK12.application');
uiApplication.Visible = 1;
root = uiApplication.Personality2;
```

So to create the scenario in STK, I will be creating a function `scenarioInfo.m`, see Appendix A.1.1 for full function. This will help avoid creating single-use variables (i.e. startTime, stopTime, scenarioName, etc). Instead, it will return a single object (`scenario`) that will contain all the information we put in. The inputs and their respective data types are defined in the comments. The only thing to note is that `root` is an input, the object created earlier when starting STK. This is the object we are modifying (think of it as the object in MATLAB that will be modified when connecting to STK)[2] when we create and modify the scenario. As such, we need it to be an input to the function.

```
function [scenario, timeVector, dt] = scenarioInfo(root, scenName,
scenStartTime, scenStopTime, dt)
%%% SCENARIO INFORMATION
%   Information for Scenario (object) in Systems Tool Kit
%
%   Parameters
%     scenName                Scenario Name (char array: name)
%     scenStartTime           Scenario Start Time (char array: date)
%                                   % Format: 'dd MMM yyyy HH:mm:ss:SSS'
%     scenStopTime            Scenario Stop Time (char array: date)
%                                   % Format: 'dd MMM yyyy HH:mm:ss:SSS'
%     dt                      Time Step (double: seconds)
%
%   Definitions
%     scenStartTime           Analysis Start Time (datetime: date)
%     scenStopTime            Analysis Stop Time (datetime: date)
%     interval                Time Step (duration: sec)
%     timeVector              Time Vector (column list of datetimes)
%     scenario                Scenario (object)
%

end
```

One might also wonder what the purpose of the 'timeVector' is. This is a variable containing the full duration of the scenario as a list of datetimes. This isn't the most computationally efficient method to use, but it is easiest when trying to find the datetime for a single event (i.e. the start of an access window or eclipse), something that can be done quite a bit in this program. It is also to be noted that the time step 'dt' is both an input and an output. There is no real reason to do it this way, I just thought it was funny. An experienced programmer might wonder why I chose to do this way. To which I say, leave me alone, I didn't learn any better and it's too late to change it now. But if one were to send proper and thorough feedback, I wouldn't mind implementing it.

In the meantime, this function reduces down to a single line when executing, shown below.

```
[scenario, timeVector, dt] = scenarioInfo(root, scenName, scenStartTime,
scenStopTime, dt);
```

[2] Please note, I don't actually know how this works. I am a Mechanical Engineering Major, not a Computer Science Major. So I have no idea how this actually works, this is just the way I see it. For more information, reach out to the kindhearted folks on the AGI Support Team, they are more than ready to help.

1.2.2 STK Facility

The next goal is to add facilities to our scenario, using the `facilityInfo.m`, see Appendix A.1.2 for the full function. This is primarily intended to model ground stations in STK, so it would help to be able to create many facilities and their sensors (which can be limited based on their design) and modify them based on the specifications provided or tests conducted. Below is the function for adding a facility and sensor.

```
function [facility, fSensor] = facilityInfo(root, fName, fLocation, fColor,
fsName, fsCHA, fsRmin, fsRmax, fsElmin, fsElmax)
%%% FACILITY INFORMATION
%   Information for Facility (object) in Systems Tool Kit
%
%   PARAMETERS
%     fName              Facility Name (char array: name)
%     fLocation          Facility Location (3x1 double:
%                              longitude, latitude, altitude)
%     fColor             Facility Color (3x1 double: RGB)
%
%     fsName             Facility Sensor Name (char array: name)
%     fsCHA              Facility Sensor Cone Half Angle (double: deg)
%     fsRmin             Facility Sensor Range Min (double: km)
%     fsRmax             Facility Sensor Range Max (double: km)
%     fsElmin            Sensor Elevation Angle Min (double: deg)
%     fsElmax            Sensor Elevation Angle Max (double: deg)
%
%   DEFINITIONS
%     facility           Facility (object)
%     fSensor            Facility Sensor (object)
%

end
```

As you can see, there are a lot of parameters that we can add for a facility and its sensor. The majority of which fall under the sensor, as this is what we will be using to model the ground station (i.e. range, elevation angles, etc). While I tried my best to keep this as simple as possible, it is very difficult to do so without oversimplifying too much. On the other hand, it is also possible to add more complexity as per need (though I'm not sure how well or even useful that would turn out. Anyways, the parameter names and types are listed in the comments as before ('f{name} will be a facility parameter and 'fs{name} will be a facility sensor parameter). Similar to before, we need to add the input `root` and return the created objects `facility` and `fSensor`. The line of code that will be used in the final script will be the following.

```
[facility, fSensor] = facilityInfo(root, fName, fLocation, fColor, fsName,
fsCHA, fsRmin, fsRmax, fsElmin, fsElmax);
```

1.2.3 STK Satellite

Next up is the function for adding a satellite, this will be `satelliteInfo.m`, and the full function is in Appendix A.1.3. As you can probably guess, this is going to be a massive function. There are a lot of parameters that need to go into defining a satellite and its sensor. The main ones being the orbit parameters (here chosen to be standard keplerian elements) and the sensor parameters (similar to the facility sensor parameters from before).

```
function [satellite, sSensor] = satelliteInfo(root, sName, sSMA, sE, sI,
sAP, sAN, sL, sColor, sModel, ssName, ssCHA, ssRmin, ssRmax)
%%% SATELLITE INFORMATION
%   Information for Satellite (object) in Systems Tool Kit
%
%   PARAMETERS
%     sName                    Satellite Name (char array: name)
%     sSMA                     Semimajor Axis (double: km)
%     sE                       Eccentricity (double: unitless)
%     sI                       Angle of Inclination (double: deg)
%     sAP                      Argument of Perigee (double: deg)
%     sAN                      Ascending Node (double: deg)
%     sL                       Location in Orbit (double: deg)
%     sModel                   Satellite Model (char array: model path)
%
%     ssName                   Satellite Sensor Name (char array: name)
%     ssCHA                    Satellite Sensor Cone Half Angle
%                                    (double: deg)
%     ssRmin                   Satellite Sensor Range Min (double: km)
%     ssRmax                   Satellite Sensor Range Max (double: km)
%
%   DEFINITIONS
%     satellite                Satellite (object)
%     sSensor                  Satellite Sensor (object)
%

end
```

Similar to before, the function will return two objects, a `satellite` and `sSensor`. If the reader is wondering where to get all of these parameters, then the best answer I can give is ‘that's what the purpose of MESS is’ (one of them at least). To anyone still uncertain, ask whoever is incharge to point you towards books and resources on Spacecraft Mission Analysis and Design. So here is the function line for adding a `satellite` and `sSensor`.

```
function [satellite, sSensor] = satelliteInfo(root, sName, sSMA, sE, sI,
sAP, sAN, sL, sColor, sModel, ssName, ssCHA, ssRmin, ssRmax)
```

1.2.4 STK Access

The original purpose of STK (back when it was just Satellite Tool Kit) was to compute access windows (the times when two things can 'speak' to each other). Naturally, we would like to know when our spacecraft is within range of the ground station to receive instructions or send its data, but first we must do some setup. The first of which is to organize our ground stations together. If we have multiple ground stations, we would like to be able to put them together into a single array, compute their access one by one, and then create an array of access objects. The following code assumes we have 3 ground stations.

```
facilityArray = [facility1, facility2, facility3];
fSensorArray = [fSensor1, fSensor2, fSensor3];

for a = 1:length(facilityArray)
      % NEED TO AVOID MAKING IT LIKE THIS BUT HONESTLY IDK HOW TO
      accessArray(a) = satellite.GetAccessToObject(fSensorArray(a));
      accessArray(a).ComputeAccess();
end
```

The return of this is `accessArray`, which contains the access object between each `fSensor` and `satellite`. Note, the `accessArray` only contains information about the access window *times*, not particularly the *direction* in which the satellite would need to point. This is addressed in 1.3 when we need to find the quaternions telling us which way to orient the satellite to point at the ground station.

This is the first major area that I would improve upon if I could. First of all, there is definitely a way to run this 'for' loop in parallel (using 'parfor' or something similar), but I have yet to figure out how. Next up, there is probably a way to initialize the `accessArray` object. There should be a light warning from MATLAB telling you "consider preallocating for speed". While this area of the code isn't terribly long, this is an area to improve if you have a lot of ground stations to model and a lot of access to compute.

1.3 STK Data Extraction

This is likely going to be the longest chapter so it's going to take me several days to write this whole thing. Try and guess where I started and stopped writing if you are bored. Anyways, creating the satellite object in MATLAB is done with this one line.

```
satelliteModel = createSatelliteModel(root, scenario, satellite,
facilityArray, accessArray, timeVector, dt);
```

Basically, we will take all the STK objects created thus far, extract some more information from STK, and then use it to create the object in MATLAB. This slightly encroaches on the next section, where we do the MATLAB Simulation, but since we are still extracting data from STK, I will put it here. The next chapter will still focus on where that data is used and in which way. That being said, we will also define the following time variables.

```
% TIME
time = 0:dt:dt*(length(timeVector)-1);
t = 1:length(timeVector);
```

The 'time' variable is the time from start to finish, counting in steps of 'dt'. The 't' variable is just the indexing variable for 'time'. Once again, this is definitely not the most optimal method for going about solving this problem, it was the only thing I could think of at the time and so I am stuck with it unless I go back to redo everything (which I'm not doing anytime soon).

Since the functions in this section will all have the purpose of extracting data from STK, they will be located in the following location.

```
\lib\stk functions\
```

1.3.1 Access Quaternions and Booleans

Yes, I am using quaternions for everything. If you don't know what quaternions are, go read their wikipedia page and consult someone who did work in attitude control, they should be able to point you in the right direction. For the purpose of this simulation, we will only be using unit quaternions and they will be used as a means of representing rotations in 3D space. For this paper, the term 'access quaternion' represents the rotation needed to point to a given facility. Since the possibility of multiple facilities is present, there will also be 'access booleans', a boolean for each instant in time representing whether the satellite is capable of communication with the given facility. The initialization of these variables is shown below.

```
% BOOLS AND QUATERNIONS
accessQuaternions = zeros(length(t), 4, length(facilityArray));
accessBools = false(length(t), length(facilityArray));
```

The heart of this function is the function getAccessQuaternions.m, see appendix A.1.4 for the full function. This is a rather long and complex function so I will explain its inner workings before proceeding. The function can only calculate the access quaternions for one facility at a time, and so its inputs are the relevant objects from STK (`root`, `scenario`, `satellite`, `facility`, and its associated `access`). Another important thing to note here is that the quaternions provided here are the 'perfect values'. Which means they tell the spacecraft EXACTLY which way to orient to be pointing to the facility in question. This is something that we need to account for later on.

```
function [accessBools, accessQuaternions] = getAccessQuaternions(root,
scenario, satellite, facility, access, timeVector, dt)
%%% getAccessQuaternions
%     Get Quaternions associated with pointing towards an access window
%
%   INPUTS:
%     root                STK (object)
%     scenario            Scenario (object)
%     satellite           Satellite (object)
%     access              Access (object)
%     timeVector          Time Vector
%     dt                  Time Step
%
%   OUTPUTS:
%     accessBool          Boolean for Access Times
%                               0: Access Unavailable
%                               1: Access Available
%     accessQuaternions     Quaternion associated with Access Pointing
%                               Format: <qs, qx, qy, qz>
%

end
```

Quaternions work on the principle of creating an axis of rotation and angle of rotation about that axis. To do this, we need to create three vectors. The first is a vector representing the satellite antenna direction (by default, this is the body, fixed +X direction[3]). The second is a position vector from the center of the earth to the `satellite`. The third is a position vector from the center of the Earth to the `facility`. From the two position vectors, the relative position vector is found. The rotation axis is simply the cross product of the relative position vector and the satellite antenna vector and the rotation angle is angle between them, which can be done directly in STK. From here, we can directly convert these values into unit quaternions representing the rotation required to orient the satellite to the specified facility. Something we also need is the times when access is possible, this comes in the form of access booleans, the default is false and becomes true when the satellite is in range of the facility. And so the final function call is the following.

[3] So this vector isn't defined as an input to the function. Rather it is hard-coded into the function itself. I didn't have the patience to figure out all the cases so see the Appendix for this function if you want to change it.

```
[accessBools, accessQuaternions] = getAccessQuaternions(root, scenario,
satellite, facility, access, timeVector, dt)
```

Since we are not concerned with just a single set of access quaternions (recall that we could have multiple ground stations to account for), the function needs to be run in a loop and put together into a single variable that will contain all of the access quaternions. By iterating through the multiple ground stations, we can also add a small priority system to our access quaternions (i.e. if facility 1 and facility 2 have an overlapping access window, the program will check facility 1 first). In practice, we get the following loop.

```
for a = 1:length(facilityArray)
      [accessBools(:, a), accessQuaternions(:, :, a)] =
getAccessQuaternions(root, scenario, satellite, facilityArray(a),
accessArray(a), timeVector, dt);
end
```

1.3.2 Sun Quaternions and Booleans

So while the main purpose of this mission isn't to run a power simulation, I did feel it was necessary to add it because of how important power can be for a small satellite. As such, a function similar to the getAccessQuaternions.m will be defined for the sun quaternions. This function is called, unsurprisingly, getSunQuaternions, and its full code can be found in Appendix A.1.5.

```
function [sunBools, sunQuaternions] = getSunQuaternions(root, scenario,
satellite, timeVector, dt)
%%% getSunQuaternions
%     Get Quaternions associated with pointing towards the sun
%
%   INPUTS:
%     root                STK (object)
%     scenario            Scenario (object)
%     satellite           Satellite (object)
%     timeVector          Time Vector
%     dt                  Time Step
%
%   OUTPUTS:
%     sunBools            Boolean for Lighting Times
%                               0: Lighting Unavailable (sunlight)
%                               1: Lighting Available (sunlight)
%     sunQuaternions      Quaternion associated with Sun Pointing
%                               Format: <qs, qx, qy, qz>
```

```
%

end
```

Similar to the access quaternions, this function finds the quaternions between two vectors. In this case, it defines a satellite solar array vector, the normal vector to the solar array (by default, it is the satellite body +X direction[4]). The second vector is a position vector pointing to the sun (which comes built into STK). From there, we use the same procedure as before to find the sun rotation axis and angle, and eventually the sun quaternions. The sun booleans can be extracted directly from STK.

As a quick side note, this function was running slower than I expected so a modification was made to extract 1 in 10 data points and use a stepwise function to interpolate between them. If the time step of the simulation is 10 ms, then the sun values are collected every 100 ms. Considering the small difference this makes in the values, there is little reason for a small satellite team to account for it, however, should one require the accuracy of having each and every sun quaternion (in 10 ms intervals), a general path to revert the function will be put in the appendix. Here is the function call for getting the sun quaternions and sun booleans (along with the resizing function).

```
[sunBools, sunQuaternions] = getSunQuaternions(root, scenario, satellite,
timeVector, dt);
sunBools = imresize(sunBools, [length(timeVector) 1], 'nearest');
sunQuaternions = imresize(sunQuaternions, [length(timeVector) 4],
'nearest');
```

1.3.3 Magnetic Field

There is nothing special here, just extracting the Earth's magnetic field. There was no real reason to add this other than I wanted to. Anyways, the function is called getMagneticField.m and the full function can be found in Appendix A.1.6.

```
function [satBField] = getMagneticField(scenario, satellite, dt)
%%% getMagneticField
%     Get Full Magnetic Field
%
%   INPUTS:
%     scenario          Scenario (object)
%     satellite         Satellite (object)
%     dt                Time Step
%
%   OUTPUTS:
%     satBField         B Field at Satellite Location
```

[4] Once again, this is a hard coded value at the time of writing. I will leave a note in the respective appendix on what to modify to use a different solar array normal vector.

```
%                              ECF Frame (double: Tesla)
%

end
```

1.3.4 Final Setup

Ok so this is encroaching a little into the next chapter, where we set up the simulation environment and satellite system dynamics but I guess I'll put it here because it flows nicer with the rest of this section. So something to make note of is that we have many different quaternions for our satellite (i.e. access windows for each facility and a sun quaternion). And so we need a way to make the program decide which way to orient itself (if it even should orient itself). To do this, we need to create a simple command system, something that will take a look at the current state of the satellite and make a decision from the options available to it. This is done in what I call the 'Command System' (likely not the best name for it but it's too late to change it now). The modes of the command system follow directly from each possible set of desired quaternions.

```
%%% COMPONENTS AND SUBSYSTEMS
% Command System
      % 1: Nothing Mode
      % 2: Safety Mode
      % 3: Experiment Mode
      % 4: Charging Mode
      % 5: Access Location 1
      % 6: Access Location 2
      % n+4: Access Location n
qd = zeros(length(timeVector), 4, 4+length(facilityArray));
qd(:,1,1:3) = 1;
qd(:,:,4) = sunQuaternions;   % 4: Charging Mode
for a = 1:length(facilityArray)
      qd(:,:,a+4) = accessQuaternions(:,:,a);
end
```

I will leave the explanation of the modes for the later section when I delve into how the satellite model works in the later chapter, but I will explain the reasoning behind the 'qd' variable, as this is the variable that will hold all the quaternion data. The 'qd' is short for 'quaternionDesired' and it represents the desired quaternions for each mode. Since the 'Nothing Mode', 'Safety Mode', and 'Experiment Mode' have no pointing requirements, the quaternions in here are set to zero. For the 'Charging Mode', we simply input the sun quaternions collected earlier. The 'Access Location' is a bit trickier because we don't know how many ground stations we could have so we have to set it up with a for loop as we did earlier when computing access. The rest of the code and its nuances should be clear upon further examination or after I've explained the rest of the model.

Chapter 2: MATLAB Simulation

At the time of writing this documentation, there are three main subsystems modeled: the power system, the command system, and the attitude system. In summary, the command and data handling (CDH) system is responsible for deciding upon the current desired state, the power system runs a simple simulation of power generation and use, and the attitude determination and control system (ADCS) covers the attitude dynamics. Since the main goal of this simulation is attitude research, the ASDCS will encompass the majority of the computational resource use and have the most detail, however, the other two systems will be mentioned briefly here.

Each system has several components, modeled as objects in MATLAB, that will also be explored, however, the main functions of the system are built directly into the system itself. For example, the CDH system also models data generated by the experiment mode and so has a component called 'SSD' to model the solid state drive that would be responsible for storing generated data. However, the time evolution of the 'SSD' object occurs in the 'command system', which will be simulated in the overarching 'satelliteModel', the object that contains all of the aforementioned systems.

As a preparatory note, the final simulation is run in the 'satelliteModel' object with a loop and so there is a specified format to the state vector. Without going too far into detail yet, since we've yet to talk about the subsystems and their variables, I will simply introduce the format of the state vector and then extend upon it as we develop the systems more.

```
% State Vector Format
      % 1:4       SC Attitude Quaternion (Actual)
      % 5:7       SC Attitude Angular Velocity (Actual)
      % 8:10      Reaction Wheel Angular Velocity (Actual)

      % 11:14     SC Attitude Quaternion (Estimate)
      % 15:17     SC Attitude Angular Velocity (Estimate)
      % 18:20     Reaction Wheel Angular Velocity (Estimate)

      % 21        Battery State of Charge (SOC)
      % 22        Command
      % 23        Data Storage Use
```

I guess I really can't just show this to someone and not explain where it all comes from. The first 10 variables are the actual physics.. The next 10 variables are the estimated physics, the error is caused by error in parameters. The last three variables encompass the power system and command system: 21st variable is the battery state of charge, 22nd is the current command (provided by the command system), and the 23rd is the current data storage use (from 0-100%). As mentioned earlier, this state vector was written purely for attitude dynamics research and so constitutes the majority of variables. Should one try to add more variables, I wish them the best of luck and the small advice to invest in a powerful computer before doing so.

2.1 Command and Data Handling System

Being the simplest system, the CDH system has two main tasks: to decide what the satellite should do and manage data storage generation and use. Based on the command data handling system for a normal spacecraft, this system uses the booleans and state vector at a given time to generate the command for the next iteration and predict how much data would be generated (or sent) in the current time step. The information about data generation (and use) is loaded in the 'ssd' object while the command decision tree is left in the 'commandSystem' object.

2.1.1 SSD

The 'ssd' object is perhaps the most simplest one in this simulation. It has several variables for use in the object, but there are no methods inside of it. As mentioned before, the simulation itself occurs in the final 'satelliteModel' object which contains all the subsystems and the main simulation loop. The line of code for creating the 'ssd' object is shown below. The 'capacity' variable refers to the capacity of the SSD in megabytes (gigabytes felt like overkill). The next 5 variables are the various data generation rates for a given mode. For example, 'nothingDataGen' refers to the data generation rate when the satellite is in 'nothing mode' (some refer to this as standby). The other built-in modes are 'safety mode', 'experiment mode', 'charging mode', and 'communication mode'. The full code for the 'ssd' object can be found in A.2.1.

```
obj = ssd(capacity, nothingDataGen, safetyDataGen, experimentDataGen,
chargingDataGen, communicationDataGen, state0)
```

While it is possible to add more modes, one would have to modify a lot of code to do this. I'll mention more of this in the next section with the rest of the command system.

2.1.2 Command System

The 'commandSystem' object is much more complicated and has a lot more nuances to it. First and foremost, it is responsible for holding all the booleans as well as generating some on its own, for the experiment mode. To do this, the command system must have some other constraints placed on it to figure out when it is possible to conduct experiments. For the most generic of missions, this would be the 'socSafe', 'socUnsafe' (both in reference to the 'powerSystem' system), 'ssdSafe' (in reference to the 'ssd' object), 'dt', and 'expDuration' (the time step and experiment duration). The shortened reasoning for this is that the spacecraft must have enough resources to complete the experiment without risking operational functionality and these are the values that can be specified to constrain the system. Below is the code for creating the 'commandSystem' object, see A.3.1 for a more in depth exploration about the reasoning behind everything.

```
obj = commandSystem(socSafe, socUnsafe, ssdSafe, expDuration, dt, sunBools,
accessBools, ssd)
```

The ‘expBools’ (experiment booleans) are made as part of the object creation (more in A.3.1) and so there must be a function to get the current bools, combine this with the current state, and decide upon the next command. This is done by one of the methods, called ‘command’ (I know, very original and descriptive), and will return the current course of action from the given data. The format for the method and calling it is shown below (I am assuming the function will be called from the ‘satelliteModel’, referenced from here onwards as ‘obj’).

```
%%% Function Format
[command] = command(obj, batterySOC, ssdSOC, a)

%%% Function Call
command = obj.commandSystem.command(obj.stateS(a,21),...
        obj.stateS(a,22)/ssdCapacity, a);
```

It might be a bit confusing to get used to the ‘obj.stateS’ variable, but it would be best to reference the introduction to this chapter where I cover the state vector and its format. Once the state vector is brought into the picture, it should be fairly simple to see how the function will be used to generate a command for each time step in the simulation.

The other method of importance is the ‘dataGenerated’ function. This function returns the current data generation (or use) with the constraints of a real SSD. The format and calling for this method is below (similar to before and as will be the case for the rest of this paper, it is called from within the ‘satelliteModel’ object by ‘obj’).

```
%%% Function Format
[dataGen] = dataGenerated(obj, dt, state, command)

%%% Function Call
obj.stateS(a+1,23) = obj.stateS(a,23) +
        obj.commandSystem.dataGenerated(obj.dt, obj.stateS(a,23), command);
```

Similar to before, this function will take in the current state and command to return the expected data generation rate. For most modes, there is data generation and for the communication mode, the data generation would be negative. There is also the constraint of “we cannot exceed the capacity of the ‘ssd’ object or have negative data storage if we are done downlinking the data”. All of this will be more clear in A.3.1 where the raw code for the ‘commandSystem’ object is shown.

The command system as I’ve made it only has a handful of modes. The main modes are ‘nothing mode’, ‘safety mode’, ‘experiment mode’, ‘charging mode’, and ‘access mode’. The index for each of these modes is used for extracting data from certain variables in a clever way. As mentioned in section 1.3.4, the desired quaternions are stored in the format of the command system to make data extraction as easy as possible. One simply needs to get the current mode from the command system (using the ‘command’ method explained above) and use it for parsing through the quaternion data. As a quick example, if the command system outputs the command for ‘charging mode', the number associated with that command is '4'. And in the quaternion data, the "4th layer" corresponds to the sun-pointing quaternions. From there, the current time is used to get the current sun-pointing quaternion and the value is used.

```
% Command System
      % 1: Nothing Mode
      % 2: Safety Mode
      % 3: Experiment Mode
      % 4: Charging Mode
      % 5: Access Location 1
      % 6: Access Location 2
      % n+4: Access Location n
```

While this is definitely not the most ideal system for solving this problem, this is the most efficient one that my non-computer-science oriented mind could come up with that took the least amount of time and computational resources. Furthermore, these modes should be enough for the mission this simulation was designed to help. Should one decide to extend upon this, they should see the code in A.3.1, which details the command system, and start modifications from there.

2.2 Power System

The power system is based on a power simulation tool developed in Python by Connie Liou. Unfortunately, we reused the majority of her code after she permitted us to. We implemented the same code in MATLAB (with little modification). Since we are not the official developers of this code, we would rather redirect the user to the appropriate github page where one can learn the inner workings of this code from the source: https://github.com/connie-liou/powersimstore.

As for the implementation in our project, the main takeaway is the current battery capacity is a state variable and this evolves differently over time depending on the current command. Further explanation cannot be done at the moment as neither of us actually developed the code, however, going to the appropriate github page and working with the original project will help understand this a lot better. Below are some code snippets that we deem important, but we abstain from going into detail as it is likely to be wrong.

2.2.1 Battery

This is the model for a battery. The battery is the main component of the power system and its main purpose is to assist in the modeling of power consumption for a given maneuver and series of events. At the moment, the 'battery' object is based on the properties of a real battery and uses real charging and discharging data to model the charge and discharge cycle. There are also other parameters: capacity (in Ahr), R_charge and R_discharge (internal resistance for the charge and discharge respectively), maxV and max I (the maximum voltage and current of the battery), and the number of cells and linedrop. Below is the line of code for creating the 'battery' object, see A.2.5 for more details.

```
obj = battery(capacity, soc, R_charge, R_discharge,maxV, maxI, cells,...
        lineDrop)
```

The main function of the battery is to ‘step’ forward and determine the state of charge (SOC) at the next time step, however, since this requires the use of the other electrical components, this method and all the related methods have been reserved for the ‘powerSystem’ object.

2.2.2 Solar Array

The solar array is an object to model the power generation for the satellite. At the moment, this object is not being used to its maximum capability (see Chapter 4: Future Work, section 4.3 for more details), but has been set up for such a development in the future. The main properties of the ‘solar array’ object are the surface area of the panels (area), the normal vector to these panels (normalVector), and the charging efficiency given the solar constant for a given orbit (efficiency). The line below is used to create the object, see A.2.6 for more details.

```
obj = solarArray(area, normalVector, efficiency)
```

There is no function for determining the current power output of the panels over the course of the orbit. Should one desire to expand the simulation tool and include this function, it would require adding a math function with quaternion rotation as well as importing of solar flux data throughout the orbit from STK.

2.2.3 Electrical System

The electrical system is not designed to model the entire electrical system and all of its complex components, just incorporating the most common modes of the satellite and their current draw. Currently, there are 5 modes with their respective current draws supported: Nothing/Idle Mode (nothingLoadCurrent), Safety/Detumble Mode (safetyLoadCurrent), Experiment Mode (experimentLoadCurrent), Charging Mode (chargingLoadCurrent), and Communication Mode (communicationLoadCurrent). These are also the properties of the ‘electricalSystem’ object and the following is the line for creating this object.

```
obj = electricalSystem(nothingLoadCurrent, safetyLoadCurrent,
experimentLoadCurrent, chargingLoadCurrent, communicationLoadCurrent)
```

There are no methods within the ‘electricalSystem’ object.

2.2.4 Power System

The ‘power system’ object is used to model the power use of the satellite. While it is not inherent to the modeling of attitude dynamics and run simulations in ADCS development, it is helpful for sizing of systems and the power use of different ADCS subsystems can be compared and used for the mission planning phase. Once again, this isn't required and can be commented out (assuming it is done correctly) if speed is desired. Below is the line for creating the 'power system' object, full code is left to A.3.2.

```
obj = powerSystem(time, battery, batteryData, solarArray, electricalSystem)
```

Within the 'power system' object, there is one main method for stepping through the power simulation. This method is called 'step' and combines all the objects within the power system to iterate one time step forward and determine the battery SOC. This function takes in the current 'power system' (obj) and all of its components and state, the time step size (dt), and the current command (command) and returns the battery SOC on the next step.

```
%%% Function Format
[soc] = step(obj, dt, command)

%%% Function Call
obj.stateS(a+1,21) = obj.powerSystem.step(obj.dt, command);
```

2.3 Attitude Determination and Control System

This system is the main focus of this paper/simulation/mission. The ADCS is responsible for determining the orientation of the spacecraft in 3D space, receiving the desired orientation, and then going to that desired orientation (fancy way of saying GNC but for orientation only). To do this, there are three main components: the magnetorquer, the reaction wheel system, and the star tracker. These components fall under the ADCS, which will be modeled as the overall system containing the dynamics. All of this will be explored in this section.

2.3.1 Magnetorquer

Even though it is not used in this simulation very much, the magnetorquer is another simple object that should be explained. The magnetorquer creates a magnetic dipole which interacts with the Earth magnetic field and induces a small torque. This is often used for detumbling a spacecraft as the increased angular velocity makes it difficult to use active control, such as a reaction wheel, and the magnetorquer is a passive system that will gradually reduce the angular velocity before the active control system takes over. Wow, that's a long sentence to say 'it's better to only use a magnetorquer for detumbling'. As mentioned previously, the magnetorquer is not as useful for the main mission unless unstable attitude behavior has been brought forth and so will not be considered very heavily. The full code for the magnetorquer can be found in A.2.2.

```
obj = magnetorquer(dipoleMagnitude, direction, magneticField)
```

Since a magnetorquer is expected to produce a torque, there is a method inside of the object for getting just that. This method takes in the current orientation and magnetic field and returns the induced magnetic moment.

```
%%% Function Format
[Mm] = magneticMoment(obj, B, q)

%%% Function Call
Mm = obj.attitudeSystem.magnetorquer.magneticMoment(...
         1e-9*obj.magnetorquer.magneticField(a,:), q);
```

As mentioned before, the magnetorquer is omitted from use due to its infrequent use. In the code, the magnetorquer object is created, however, the magnetic moment is disregarded. And no, I haven't tested whether it works (that's a joke, I've tested almost everything at least once). But thinking about it more, I'll include a small section about potential errors and bugs along with their solution in the 'Future Work' chapter.

2.3.2 Reaction Wheels

The reaction wheel system consists of a 3-axis reaction wheel system. For the sake of simplicity, the reaction wheels are assumed to be mutually perpendicular, lying parallel to the three cartesian unit vectors of the spacecraft body frame. More complex systems can be modeled at the discretion of the user but there is a higher computational resource use for working with the overactuated system due to the nature of the control law and the extra state variables for the 4th reaction wheel (actual and estimate, each adds one extra state variable). More information about this modification is mentioned in A.2.3 along with the full code of the 'reactionWheel' object.

```
obj = reactionWheel(state0, state0Error, inertiaA, inertiaError, maxMoment)
```

Well this should be more or less self explanatory. Since we are creating two systems, the actual and the estimate, we need to have some error in the parameters and be able to use it when we run the simulation. The 'state0' of the 'reactionWheel' object refers to the initial speed of the reaction wheel and the 'inertiaA' refers to the actual inertia of the ADCS and reaction wheel system. The two associated error variables, 'state0Error' and 'inertiaError', are the deviations that will be used for the estimated system dynamics. Finally, the 'maxMoment' refers to the maximum possible moment that can be generated, inspired by hardware that can have a saturated output.

The most important method in the 'reactionWheel' object is the 'controlTorque'. This is designed to be changed as needed and is based on a simple quaternion-based PD controller. This just happened to be the most simple controller I could find (since I'm not very well versed in the control theory side yet), but I've made it relatively flexible with the number and types of inputs to modify it should one need to, more explanation in A.2.3.

```
%%% Function Format
[Mc] = controlTorque(obj, q, w, K, qd)

%%% Function Call
Mc = obj.attitudeSystem.reactionWheel.controlTorque(...
       obj.stateS(a, 11:14), obj.stateS(a, 15:17),...
       obj.attitudeSystem.K,...
       obj.attitudeSystem.qd(a, :, command));
```

There are many more sub-functions in this object for performing the computations for the control torque, but they aren't as important here and will be mentioned briefly in A.2.3.

2.3.3 Star Tracker

The star tracker is a complex item and can easily be an entire simulation project on its own. Since the main focus of this simulation is overall mission planning and modeling, there is no need for me to create a detailed star tracker model, I can use some of the common modes and make the code simple enough to modify to one's need. In all honesty, this was the trickiest object to model, since there is plenty of area for potential but there is also the pitfall of creating a star tracker simulation that takes too long to compute each step. Unfortunately, I did not create a very advanced star tracker, just a simple one that accounts for two main types of error: a one sigma error and an offset error.

```
obj = starTracker(oneSigma, offset)
```

Each of these error types comes with a method so I will go through those next. The main part of the star tracker is to take in the actual attitude and return the estimated attitude (both are expressed as quaternions in this simulation). The most basic attitude determination is the perfect kind, one with no error. Some might say this can be done directly (using `>> qEstimate = qActual`), but I preferred the consistency here. Anyways, here is the code.

```
%%% Function Format
[qEstimate] = perfectAttitudeAcquisition(obj, qActual)

%%% Function Call
obj.stateS(a,11:14) =
obj.attitudeSystem.starTracker.perfectAttitudeAcquisition(...
        obj.stateS(a,1:4));
```

Probably not the most eye-appealing code nor the most efficient, but definitely the most consistent with the next two code snippets. To add noise with a known standard deviation (one sigma error) is the next kind of attitude determination and here is the method for that.

```
%%% Function Format
[qEstimate] = onesigmaAttitudeAcquisition(obj, qActual)

%%% Function Call
obj.stateS(a,11:14) =
obj.attitudeSystem.starTracker.onesigmaAttitudeAcquisition(...
        obj.stateS(a,1:4));
```

Finally, we can add an offset to our attitude values and without any delay, here is the code.

```
%%% Function Format
[qEstimate] = offsetAttitudeAcquisition(obj, qActual)

%%% Function Call
obj.stateS(a,11:14) =
obj.attitudeSystem.starTracker.offsetAttitudeAcquisition(...
        obj.stateS(a,1:4));
```

Personally, I only use one at a time (usually perfect or one sigma error), but there is nothing wrong with using a combination of them or even defining a new function based on some testing of the star tracker (a potential future update would be to incorporate a simple thermal system and have error associated with it, but that's not happening anytime soon).

2.3.4 Attitude System

And here we have the beast that contains all of the aforementioned objects, functions, and its own system dynamics. We will be using this object to define the systems and make them as versatile as necessary for ease of use and repetitive use. But first we must construct the system using the previous objects and other inputs. The state vector here follows directly from what was mentioned at the start of this chapter, but now I am free to go into detail. There are two state vectors to keep track of, the actual and estimate. The actual is where the system actually is, the estimate is where the system thinks it is. Due to the error terms we have slowly introduced, there will likely be a deviation here. The 'state0' variable here is for the actual initial conditions and consists of the first 10 terms in the state vector (4 spacecraft quaternions, 3 spacecraft angular velocities, and 3 reaction wheel angular velocities). The 'state0error' variable adds in the error associated with it (though it is fine to set it to zero and let the error in parameters add error to the state vector naturally). Something else to note is that the inertia matrix here is that of the whole satellite, not just the reaction wheels and ADCS (as was the case in the 'reactionWheel' object). For the full code for the 'attitudeSystem', see A.3.3.

```
obj = attitudeSystem(state0, state0Error, qd, inertiaA, inertiaError, K,...
                    magnetorquer, reactionWheel, starTracker)
```

The main function for the ‘attitudeSystem’ object is the ‘attitudeSystemDynamics’ method. This function is responsible for simulation attitude dynamics, both actual and estimated. As such, it is able to take in various inputs and simulate the predicted outcome for such a system. Below is the syntax for calling the function as well as how it is called within the main simulation loop.

```
%%% Function Format
[dX] = attitudeSystemDynamics(obj, t, dt, X, a, scI, rwI, M)

%%% Function Call (actual)
obj.stateS(a+1, 1:10) = RK4(@attitudeSystemDynamics, obj.attitudeSystem,...
      obj.time(a), obj.dt, obj.stateS(a, 1:10), a, scIA, rwIA, Mc);

%%% Function Call (estimate)
obj.stateS(a+1,11:20) = RK4(@attitudeSystemDynamics, obj.attitudeSystem,...
      obj.time(a), obj.dt, obj.stateS(a,11:20), a, scIE, rwIE, Mc);
```

Looking at the function format section, we can see there are a few inputs that need to be explained in some detail. The first input is the ‘object’ itself, this would be the ‘attitudeSystem’ that is placed within the ‘satelliteModel’. The next two inputs are ‘t’ and ‘dt’, the current time and the time step. These values are used by the Runge-Kutta Fourth Order (RK4) Solver that solves the dynamics themselves. This will be explained more in section 2.4.2 when discussing the main simulation loop, for now, just know that the format is to please the RK4 solver format[5]. The variable ‘X’ is the ‘state vector’ for the attitude system, consisting of 10 values (4 quaternions, 3 spacecraft angular velocities, 3 reaction wheel angular velocities). The ‘a’ variable is the index variable for the current time (most often used when trying to extract time-dependent data like the magnetic field). The next two variables are ‘scI’ and ‘rwI’, which are spacecraft inertia matrix and reaction wheel inertia matrix. The final variable ‘M’ is the sum of moments (both internal and external).

That took longer to explain than I expected but thank goodness I did. As you can see, this was made purely to facilitate someone conducting spacecraft ADCS research. I’ve made is as easy as possible to add in a custom disturbance torque, account for changing inertia matrices (spacecraft and reaction wheels), and even modify the control torque algorithm (though this one might be a bit tricky to do immediately out of the box depending on how complex the algorithm is).

[5] When an engineer familiar with numerical analysis hears ‘runge-kutta 4th order’, their first thought is usually ‘why not ode45’ and ‘what is your error correction’. To which I say, wait for section 2.4.2 for the full explanation, but for now, the step size to use is 0.01 sec and there are ~360,000 steps in a 2 hour simulation. So the accuracy and accumulated error can be solved from here directly. More details in 2.4.2 QwQ

2.4 Satellite Model

The satellite model is the object that contains all the previously described components and systems. It also contains the main simulation loop. There is nothing special here other than that. This is just where everything sits and simulates.

2.4.1 Creating Model

So the first 'challenge' is putting everything into one object. This is a great challenge that took me hours to figure out. Here is the code for creating the 'satelliteModel' object, see A.3.4 for full code and more information.

```
obj = satelliteModel(time, dt, powerSystem, attitudeSystem, commandSystem)
```

I know. So very beautifully compact. And yet so painful. In the constructor for this object, it loads in all of the 'state0' variables loaded into the other components and systems, which was painful to say the least. This is done by loading the systems themselves and then extracting the 'state0' variables from each one as appropriate. The other variables are 'time' and 'dt', the time span vector and the time step.

2.4.2 Simulation Loop

And here we are. At the main part of this entire documentation: the main simulation loop. The syntax for running the loop is pretty straight forward, see A.3.4 to see the whole 'satelliteModel' object, with the 'simulate' method inside.

```
%%% Function Syntax
[] = simulate(obj)

%%% Function Call
satelliteModel.simulate();
```

Since this is a major component of this project, let's take some time to go through the main components of the loop here. So here is the full method, as called within the 'satelliteModel' object.

```
function [] = simulate(obj)
    %%% simulate
    %       Simulation for satellite model

    %%% PRELIMINARY STUFF
    % WAITING BAR
    f = waitbar(0,'Simulating...', 'Name', 'Simulation Progress');
    % VARIABLES
    ssdCapacity = obj.commandSystem.ssd.capacity;
    scIA = obj.attitudeSystem.inertiaA;
    scIE = obj.attitudeSystem.inertiaE;
    rwIA = obj.attitudeSystem.reactionWheel.inertiaA;
    rwIE = obj.attitudeSystem.reactionWheel.inertiaE;


    %%% SIMULATION LOOP
    % state vector format (update as needed)
      % 1:4         SC Attitude Quaternion (Actual)
      % 5:7         SC Attitude Angular Velocity (Actual)
      % 8:10        Reaction Wheel Angular Velocity (Actual)

      % 11:14       SC Attitude Quaternion (Estimate)
      % 15:17       SC Attitude Angular Velocity (Estimate)
      % 18:20       Reaction Wheel Angular Velocity (Estimate)

      % 21          Battery State of Charge (SOC)
      % 22          Command
      % 23          Data Storage Use

      for a = 1:length(obj.time)-1
            % Command System
            command = obj.commandSystem.command(obj.stateS(a,21),...
                  obj.stateS(a,22)/ssdCapacity, a);
                  obj.stateS(a+1,22) = command;
            obj.stateS(a+1,23) = obj.stateS(a,23) + ...
                  obj.commandSystem.dataGenerated(obj.dt, obj.stateS(a,23),
command);

            % Simulate Power System
            obj.stateS(a+1,21) = obj.powerSystem.step(obj.dt, command);

            % Control Torque
            Mc = obj.attitudeSystem.reactionWheel.controlTorque(...
                  obj.stateS(a, 11:14), obj.stateS(a, 15:17),...
                  obj.attitudeSystem.K,...
                  obj.attitudeSystem.qd(a, :, command));

            % MAGNETIC DISTURBANCE TORQUE
            %{
```

```
            Mm = 
obj.attitudeSystem.magnetorquer.magneticMoment(1e-9*obj.magnetorquer.magnet
icField(a,:), q);
            %}

            % Attitude Determination
            obj.stateS(a,11:14) = 
obj.attitudeSystem.starTracker.onesigmaAttitudeAcquisition(obj.stateS(a,1:4
));
            % [qEstimate] = perfectAttitudeAcquisition(qActual)
            % [qEstimate] = onesigmaAttitudeAcquisition(qActual)
            % [qEstimate] = offsetAttitudeAcquisition(qActual)

            % Actual Attitude Dynamics
            % [dX] = attitudeSystemDynamics(obj, t, dt, X, a, scI, rwI, M)
            obj.stateS(a+1, 1:10) = RK4(@attitudeSystemDynamics,...
                  obj.attitudeSystem, obj.time(a), obj.dt,...
                  obj.stateS(a, 1:10), a, scIA, rwIA, Mc);

            % Estimated Attitude Dynamics
            obj.stateS(a+1,11:20) = RK4(@attitudeSystemDynamics,...
                  obj.attitudeSystem, obj.time(a), obj.dt,...
                  obj.stateS(a,11:20), a, scIE, rwIE, Mc);

            % Update Loading Bar
            if rem(a,1000) == 0
                  percentDone = a/(length(obj.time));
                  waitbar(percentDone, f, sprintf('%.2f', 100*percentDone))
            end

      end
      delete(f)
end
```

So the first part of this loop is just some setup or repeated variables. The loading bar is there to relieve boredom and provide a countdown. The setup variables are the SSD capacity and inertia matrices (actual and estimate). For this simulation, I assume all of these values are constant, however, a system identification oriented simulation would likely update some of these values. After this setup is done, the main simulation loop begins, iterating through for every future step that is possible. In this loop, the series of events is as follows. First, the ‘command’ method in the ‘commandSystem’ is used to generate the current command. This value is used to run the ‘dataGenerated’ method in the ‘commandSystem’ as well as the ‘step’ method in the ‘powerSystem’. This iterates the command and power simulation by one step. Next, the ‘controlTorque’ method in the ‘reactionWheel’ object is run and the output saved. The code for the ‘magneticMoment’ method is also present (albeit commented out). One can simply add this value to the ‘controlTorque’ output and get the total disturbance on the satellite. After this, the ‘attitudeAcquisition’ method from the ‘starTracker’ object is called and a quaternion estimate is given. The syntax for switching acquisition types is left commented out for ease of use. From here, two ‘attitudeSystemDynamics’ are run (one for the actual state and one for the estimated state) using their respective state vector variables. Finally, the loading bar is updated if 1,000 iterations have passed (a 2 hour simulation with a time step of 0.01 sec is 360,000 steps).

2.4.2.1 Attitude Dynamics Error Analysis

A quick error analysis on the attitude dynamics aspect of this simulation goes as follows. The numerical solver used for the attitude dynamics is the Runge-Kutta 4th Order (RK4) put in a function called ‘RK4.m’, the calling of this function is shown below, full code in A.4.1.

```
%%% Function
[X] = RK4(dynamics, obj, t, dt, X, varargin)
```

It can be shown that the RK4 function will produce an error on the order of O(h^5) and an accumulated error of O(h^4). For this simulation, the step size used has been 0.01 sec (1e-2 sec) so the error and accumulated error is 1e-10 and 1e-8, respectively. Recalling that a 2 hour simulation would have 360,000 steps, the total accumulated error has a maximum value of 0.0036. This itself isn’t very useful, but it’s fair to say that the error from the simulation will be greatly outshined by the noise of ‘real components’, making it utterly useless for long-term simulations, but potentially useful for testing ‘short’ maneuvers and the relative behavior of the satellite on the scale of a few hours. Any longer and we run the risk of orbit perturbations and other disturbances (even attitude related) from taking over and causing large deviations. That being said, hopefully, this should be a good approximation for the time scale we need to test it for, a few hours. And that’s it. That’s the justification for using this simulation.

Chapter 3: Visualization and Analysis

Hooray, I'm more than halfway done with my documentation and we're only at page 32. This chapter is dedicated to the code for visualization and analysis. I took the liberty of adding it because I know how annoying it is to have no visualization. Anyways, all of the plotting falls under a singular line, shown below. Full code and more information can be found in A.5.1.

```
%%% Plot Everything
[f] = plotEverything(satelliteModel)
```

In brief, this function simply takes the 'satelliteModel' object (with all computations run), plots the results in a figure, and then returns the figure handle. Obviously there is more going on under the hood, the function has 3 sub-functions for plotting the attitude, power, and command simulations, these can be found in A.5.2, A.5.3, and A.5.4 respectively. There are also some sliders used to make navigation easier, but keep in mind that there is limited accuracy when the simulation is very long (since you cannot move the slider 1 pixel at a time). By default, the 'start' one is set to 0, and the 'length' to the full duration.

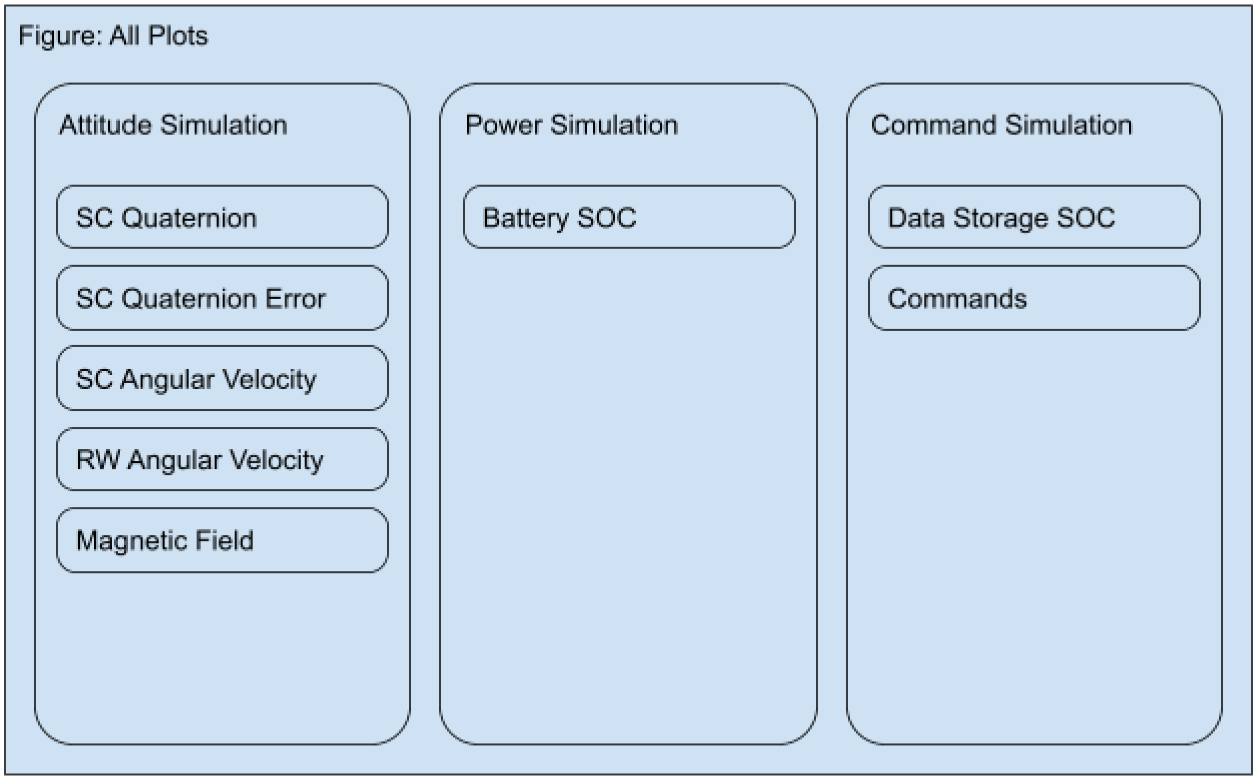

Figure: Organization of the plots (attitude, power, and command simulation)

That's really it. There is nothing special about this. If you want to add more plots or something, you just need to save the data to the 'satelliteModel' object and then use A.5 (yes you need to read it all) to figure out how to add a new tab and plot.

Chapter 4: Running Simulations and Future Work

First is the pressing concern on how to run this entire simulation tool. The two files that are designed to be 'configuration' files are the 'runthisonly.m' in A.0.1 and 'createSatelliteModel.m' in A.0.2. The latter is used to modify the 'satelliteModel' that is made in MATLAB and can be configured accordingly. The former is the script for editing the objects within STK and setting the overall simulation parameters. This does not mean one cannot go in and directly edit the files for their own custom use, they would simply need to make all of the appropriate edits, hopefully this documentation should facilitate that process by providing all of the lower level information. And that is it for how to run the simulation itself. Let it be noted that for most desktop computers, the simulation will take several minutes to run (opening STK, extracting data from it, creating the satellite object in MATLAB, then simulating it, and finally visualizing it) and may require some patience. However, it does run and no major issues have been found (thus far).

There is a lot of room for updates and improvements, however, I do not have the time nor expertise to be doing it. That being said, below are the main areas for improvement that should be made sometime in the future (indefinite deadline for this so don't expect much).

4.1 Variable-Step Size Solver

As mentioned in 2.4.2, the attitude simulation is solved using a Runge-Kutta 4th Order Solver for all steps. However, there are plenty of times in the simulation when the satellite is 'at rest' and can be approximately over a larger time span without needing the explicit computation. I did not bother with this because it is beyond my scope as an engineer, but perhaps a bit of reading into numerical analysis might give one the ability to modify the differential equation solver to speed up the computation of the attitude dynamics.

4.2 Robust Modes/Commands

I chose the past of least resistance when defining the command modes for the satellite to make it easy to perform computation. As mentioned in 1.3.4, the 'desired quaternions' variable is defined explicitly by the command mode structure (and vice versa) so as to make fetching the data easier and faster, however, by no means is this flexible and easy to do. I would highly recommend using actual software engineering practices when implementing this although I'm not too sure what the tradeoff between robust and speed would end up.

4.3 Sun Pointing Charging

Currently, there are variables that can be used to determine charging, however, they are unused. The satellite model will only assume charging when in charging mode. This is relatively unrealistic because the solar panels can charge as long as they are somewhat directed towards the sun. However, doing this computation would require finding the angle between two quaternions (the current orientation and the sun direction) with the added boolean of 'in lighting', which would then be filtered if between -90 to 90 deg, and then used to find the power absorbed by the solar panel. As you can guess, this requires more computational resources and thus has been left out of the current version (version 0.1 if you will). However, it is my hope that the steps mentioned here and some relevant reading makes this a simple implementation should one need it.

4.4 Ground Station Access

As with the sun pointing charging, the current model assumes that communication happens as soon as the satellite enters the ground station field of view. This is unrealistic if one is using a directional antenna (due to the time it takes for the satellite to point itself and then establish a secure connection with the ground station). However, the assumption in the current model is that this effect is negligible. Should one desire to fix it, they would need to create another 'access' object, this time between the satellite sensor and the ground station and then use the 'access time' in there to perform the analysis of the downlink. This would add some complexity to the workflow, since the computation would need to be carried out after the attitude dynamics are computed and uploaded to STK. The access object would then be computed and the results loaded from STK afterwards. This is beyond the scope of my project but this is the general idea of what needs to be done for it.

4.5 Updated Magnetorquer

I mentioned this in A.2.2, but I'll give it a quick section here as well. The magnetorquer object needs to be given a 'cpm' method (exactly like the one in reactionWheel.m) and then the method for obtaining the magnetic moment needs to be modified to include it (rather than doing the explicit cross-product). Also it would be nice to add a condition for when the magnetorquer is active. Currently, it is either stuck on or off, a future update might include using the magnetorquer to desaturate the reaction wheels, however, I currently don't have enough information on reaction wheels, their maximum speeds, normal procedure for desaturation, etc to be able to do this in an effective manner.

4.6 Accurately Plotting Data

Currently, the plots have two sliders at the bottom for interacting with the data and going to certain sections of it. The first slider is for 'start time' and the second is for 'length of data'. Together, this gives a fair bit of control over any major section of the data, but due to the inaccuracy of a human hand controlling a mouse, the sliders cannot be positioned down to the index and so there is some inaccuracy with getting the exact data one might desire. The solution is to add a textbox that only accepts numerical inputs and then plotting the data from there (one may prefer to work in hours or minutes and then convert this value to 'start index' and 'length duration', but this is just my personal preference). I don't know how to do this at the moment so the slider is currently present.

4.7 Adding Horizontal Lines in Plots

There are plenty of helpful lines that would make analyzing the data in the plots a bit easier, however, it depends on the degree of control the user has over the plots. If the user cannot accurately control the start and stop index of the plotted data, then this feature is about as good as estimating from a blank graph (or so I think, I could be wrong). Once the accurate data plotting method is in place, adding the horizontal lines is as simple as plotting the two endpoints with the correct height (i.e. x-data = [0 duration], y-data = [value value], and then plot(x-data, y-data)). Nothing too fancy but I didn't bother with it for the time being.

4.8 Plotting Control Torque

I've considered plotting the control torque value over time as well, but at the moment there is no need to. Since I am using such a simple system, the control torque will likely show up as a series of spikes, making it utterly useless without fine control over the sliders and plots (it would be too difficult to get the sliders on the exact location and so any analysis would be left to whether or not the user can get to that portion of the data within a reasonable amount of time. To add this, one simply needs to store the value in the main simulation loop (preferably to the 'satelliteModel' object) and then plot it with the rest of the attitude simulation data.

Appendix

A.0 Main Scripts

The only script that needs to be run is 'runthisonly.m', mentioned in A.0.1.

A.0.1 runthisonly.m

```
%%% PRELIMINARY STUFF
tic
% ADDING TO PATH (TEMPORARY)
addpath(genpath('lib'));
addpath(genpath('res'));
addpath(genpath('src'));
addpath(genpath('tmp'));


%%% STK LAUNCH
uiApplication = actxserver('STK12.application');
uiApplication.Visible = 1;
root = uiApplication.Personality2;

disp('Started: Systems Tool Kit')


%%% STK SETUP
%{
[scenario, timeVector, dt] = scenarioInfo(root, scenName, scenStartTime,
scenStopTime, dt);
[facility, fSensor] = facilityInfo(root, fName, fLocation, fColor, fsName,
fsCHA, fsRmin, fsRmax, fsElmin, fsElmax)
[satellite, sSensor] = satelliteInfo(root, sName, sSMA, sE, sI, sAP, sAN,
sL, sColor, sModel,...
      ssName, ssCHA, ssRmin, ssRmax)
%}

% SCENARIO
[scenario, timeVector, dt] = scenarioInfo(root, 'solid',...
      '24 Dec 2021 02:00:00.000', '24 Dec 2021 03:00:00.000', 0.01);

% FACILITY AND FACILITY SENSOR
[facility1, fSensor1] = facilityInfo(root, 'rugs',...
      [40.5215 -74.4618 0], [255 0 0],...
      'rugsSensor',  90, 0, 3000, 10, 90);
[facility2, fSensor2] = facilityInfo(root, 'asugs',...
      [33.4242 -111.9280 0], [255 255 0],...
      'asugsSensor', 90, 0, 3000, 10, 90);
[facility3, fSensor3] = facilityInfo(root, 'tamgs',...
      [30.6190 -96.3387 0], [128 0 0],...
      'tamgsSensor', 90, 0, 3000, 10, 90);

facilityArray = [facility1, facility2, facility3];
fSensorArray = [fSensor1, fSensor2, fSensor3];

% SATELLITE AND SATELLITE SENSOR
[satellite, sSensor] = satelliteInfo(root, 'SPICESat',...
    6371+400, 0, 45, 0, 0, 0, [255 0 0],...
    'C:\Program Files\AGI\STK
12\STKData\VO\Models\Space\cubesat_6u.dae',...
    'sSensor', 4, 0, 1500);

disp(['Added all STK Objects: ', num2str(toc), ' seconds'])

% COMPUTE ACCESS
tic
for a = 1:length(facilityArray)
      % NEED TO AVOID MAKING IT LIKE THIS BUT HONESTLY IDK HOW TO
      accessArray(a) = satellite.GetAccessToObject(fSensorArray(a));
      accessArray(a).ComputeAccess();
end
disp(['Computed: Access: ', num2str(toc), ' seconds'])

% FINALIZE AND RESET ANIMATION PERIOD
root.Rewind;


%%% SATELLITE MODEL
% CREATE MODEL (MATLAB)
tic
disp('Creating: Satellite Object in MATLAB')
satelliteModel = createSatelliteModel(root, scenario, satellite,
facilityArray, accessArray, timeVector, dt);
disp(['Created: Satellite Object in MATLAB: ', num2str(toc), ' seconds'])
```

```
% SIMULATE SYSTEM DYNAMICS
disp('Simulating: Satellite System Dynamics')
tic
satelliteModel.simulate();
disp(['Simulated: Satellite System Dynamics: ', num2str(toc), ' seconds'])

% CREATE ATTITUDE FILE
tic
afQ(scenario, timeVector, satelliteModel.stateS(:,1:4),
satelliteModel.stateS(:,5:7));
disp(['Created: Attitude File: ', num2str(toc), ' seconds'])

% LOAD ATTITUDE FILE
toAttitudeFile = [pwd, '\tmp\attitudeQ.a'];
satellite.Attitude.External.Load(toAttitudeFile);



%%% PLOT RESULTS
[f] = plotEverything(satelliteModel);



disp('DONE')
```

The singular script that you must run from the command window in MATLAB. Trying to make the duration longer than 6 hours usually crashes my computer (too little memory) and the time step of 0.01 is able to work with the reaction wheels' jerky accelerations. The afQ function is probably the most intriguing one and there is nowhere to put it in the main text so it is in A.4.2.

A.0.2 createSatelliteModel.m

```
function [SATELLITEMODEL] = createSatelliteModel(root, scenario, satellite,
facilityArray, accessArray, timeVector, dt)
%%% PRELIMINARY
% OBJECT PATHS
addpath(genpath('../../lib'));
addpath(genpath('../../src'));
addpath(genpath('../../tmp'));
addpath(genpath('../../res'));

% TIME
time = 0:dt:dt*(length(timeVector)-1);
t = 1:length(timeVector);
```

```
% BOOLS AND QUATERNIONS
accessQuaternions = zeros(length(t), 4, length(facilityArray));
accessBools = false(length(t), length(facilityArray));
for a = 1:length(facilityArray)
      tic
      [accessBools(:, a), accessQuaternions(:, :, a)] =
getAccessQuaternions(root, scenario, satellite, facilityArray(a),
accessArray(a), timeVector, dt);
      disp(['Computed: Facility ', num2str(a),' Access: ', num2str(toc), '
seconds'])
end

tic
[sunBools, sunQuaternions] = getSunQuaternions(root, scenario, satellite,
timeVector, dt);
sunBools = imresize(sunBools, [length(timeVector) 1], 'nearest');
sunQuaternions = imresize(sunQuaternions, [length(timeVector) 4],
'nearest');
disp(['Computed: Sun Bools and Quaternions: ', num2str(toc), ' seconds'])

% MAGNETIC FIELD
tic
satBField = getMagneticField(scenario, satellite, dt);
satBField = imresize(satBField, [length(timeVector) 3], 'nearest');
disp(['Computed: Magnetic Field: ', num2str(toc), ' seconds'])

%%% COMPONENTS AND SUBSYSTEMS
% Command System
      % 1: Nothing Mode
      % 2: Safety Mode
      % 3: Experiment Mode
      % 4: Charging Mode
      % 5: Access Location 1
      % 6: Access Location 2
      % n+4: Access Location n
qd = zeros(length(timeVector), 4, 4+length(facilityArray));
qd(:,1,1:3) = 1;
qd(:,:,4) = sunQuaternions;   % 4: Charging Mode
for a = 1:length(facilityArray)
      qd(:,:,a+4) = accessQuaternions(:,:,a);
end
```

```
%%% CREATING OBJECTS
% POWER SYSTEM
%{
obj = battery(capacity, soc, R_charge, R_discharge, maxV, maxI, cells,
lineDrop)
obj = solarArray(area, normalVector, efficiency)
obj = electricalSystem(nothingLoadCurrent, safetyLoadCurrent,
experimentLoadCurrent,...
      chargingLoadCurrent, communicationLoadCurrent)
obj = powerSystem(time, battery, batteryData, solarArray, electricalSystem)
%}
batteryFileName = "Moli M.battery";       % Either "Moli M.battery" or "Sony
HC.battery" currently
BATTERY = battery(20, 0.9, 0.01, 0.01, 33.6, 40, 8, 0);
BATTERYDATA = jsondecode(fileread(batteryFileName));
SOLARARRAY = solarArray(0.08, [1,0,0], 1);
ELECTRICALSYSTEM = electricalSystem(0.5, 0.3, 5, 0.3, 5);
POWERSYSTEM = powerSystem(time, BATTERY, BATTERYDATA, SOLARARRAY,
ELECTRICALSYSTEM);
disp('Created: Power System')

% COMMAND SYSTEM
%{
obj = ssd(capacity, nothingDataGen, safetyDataGen, experimentDataGen,
chargingDataGen, communicationDataGen, state0)
obj = commandSystem(socSafe, socUnsafe, ssdSafe, expDuration, dt, sunBools,
accessBools, ssd)
%}
SSD = ssd(100, 0.0002, 0.0002, 0.0002+0.003+0.177, 0.0002, -0.5, 0);
COMMANDSYSTEM = commandSystem(0.75, 0.5, 0.90, 600, dt, sunBools,
accessBools, SSD);
disp('Created: Command System')

% ATTITUDE SYSTEM
%{
obj = magnetorquer(dipoleMagnitude, direction, magneticField)
obj = reactionWheel(state0, state0Error, inertiaA, inertiaError, maxMoment)
obj = attitudeSystem(state0, state0Error, qd, inertiaA, inertiaError, K,...
      magnetorquer, reactionWheel, starTracker)
obj = starTracker(oneSigma, meanError, offset)
%}
MAGNETORQUER = magnetorquer(0.14, [1,0,0], satBField);
REACTIONWHEEL = reactionWheel([1000,1000,1000], [0,0,0],...
      diag(5.6e-6*[1 1 1]), [0,0,0], 7e-3);
STARTRACKER = starTracker(6e-5, [0,0,0,0]);
ATTITUDESYSTEM = attitudeSystem([1,0,0,0,0,0,0], [0,0,0,0,0,0,0], qd,...
      1e-2*diag([5,10,13]), 1e-5*[1,1,1], [1,1], MAGNETORQUER,
REACTIONWHEEL, STARTRACKER);
disp('Created: Attitude System')

% SATELLITE MODEL
%{
obj = satelliteModel(time, dt, powerSystem, attitudeSystem, commandSystem)
%}
SATELLITEMODEL = satelliteModel(time, dt, POWERSYSTEM, ATTITUDESYSTEM,
COMMANDSYSTEM);
disp('Created: Satellite Model')


end
```

This is the function that creates the 'satelliteModel' object. Yes, it is a function. Yes, you are supposed to edit it inside. No, I can't think of a better way to do this. I've left everything labeled as it is used (I'm pretty sure I left everything in standard units [kg, m, s] and the derived units as well).

A.1 STK Extraction Code

A.1.1 scenarioInfo.m

```
function [scenario, timeVector, dt] = scenarioInfo(root, scenName,
scenStartTime, scenStopTime, dt)
%%% SCENARIO INFORMATION
%   Information for Scenario (object) in Systems Tool Kit
%
%   Parameters
%     scenName                Scenario Name (char array: name)
%     scenStartTime           Scenario Start Time (char array: date)
%                                   % Format: 'dd MMM yyyy HH:mm:ss:SSS'
%     scenStopTime            Scenario Stop Time (char array: date)
%                                   % Format: 'dd MMM yyyy HH:mm:ss:SSS'
%     dt                      Time Step (double: seconds)
%
%   Definitions
%     scenStartTime           Analysis Start Time (datetime: date)
%     scenStopTime            Analysis Stop Time (datetime: date)
%     interval                  Time Step (duration: sec)
%     timeVector                Time Vector (column list of datetimes)
%     scenario                  Scenario (object)
%
%   Created by Manav Jadeja on 20220101


%%% DEFINITIONS
% TIME VECTOR
scenStart = datetime(scenStartTime, 'InputFormat', 'dd MMM yyyy
HH:mm:ss.SSS', 'Format', 'dd MMM yyyy HH:mm:ss.SSS');
scenStop = datetime(scenStopTime, 'InputFormat', 'dd MMM yyyy
HH:mm:ss.SSS', 'Format', 'dd MMM yyyy HH:mm:ss.SSS');
interval = seconds(dt);
timeVector = (scenStart:interval:scenStop)';

% SCENARIO (OBJECT)
scenario = root.Children.New('eScenario', scenName);

% SCENARIO PROPERTIES
scenario.SetTimePeriod(scenStartTime, scenStopTime)
scenario.StartTime = scenStartTime;
scenario.StopTime = scenStopTime;

% GRAPHICS STUFF
scenario.VO.MediumFont.Name = 'Comic Sans MS';
scenario.VO.MediumFont.PtSize = 16;
scenario.VO.MediumFont.Bold = false;

disp('Created: Scenario')

end
```

Some quick notes for anyone modifying this function. First of all, yes you can use a different format for the variables `scenStartTime` and `scenStopTime` (as well as `scenStart` and `scenStop`), but these were the times I found most easy to digest. If you do wish to make a change, you may need to go out and find all the datetime variables and change their input format to be the same. Since this isn't the main focus of this program, I've omitted this unnecessary work.

Secondly, yes you can make the default font in STK 'Pt 16, Comic Sans MS' and yes it is absolutely hilarious when you present this and see all your friends lose respect for you. The available fonts can be found in the Scenario Properties menu in STK.

A.1.2 facilityInfo.m

```
function [facility, fSensor] = facilityInfo(root, fName, fLocation, fColor,
fsName, fsCHA, fsRmin, fsRmax, fsElmin, fsElmax)
%%% FACILITY INFORMATION
%   Information for Facility (object) in Systems Tool Kit
%
%   PARAMETERS
%     fName               Facility Name (char array: name)
%     fLocation           Facility Location (3x1 double:
%                               longitude, latitude, altitude)
%     fColor              Facility Color (3x1 double: RGB)
%
%     fsName              Facility Sensor Name (char array: name)
%     fsCHA               Facility Sensor Cone Half Angle (double: deg)
%     fsRmin              Facility Sensor Range Min (double: km)
%     fsRmax              Facility Sensor Range Max (double: km)
%     fsElmin             Sensor Elevation Angle Min (double: deg)
%     fsElmax             Sensor Elevation Angle Max (double: deg)
%
%   DEFINITIONS
%     facility            Facility (object)
%     fSensor             Facility Sensor (object)
%


%%% DEFINITIONS
% FACILITY (OBJECT)
facility = root.CurrentScenario.Children.New('eFacility', fName);

% FACILITY PROPERTIES
facility.Position.AssignGeodetic(fLocation(1), fLocation(2), fLocation(3));
facility.Graphics.Color = rgb2StkColor(fColor);

disp('Created: Facility')

% FACILITY SENSOR (OBJECT)
fSensor = facility.Children.New('eSensor', fsName);

% FACILITY SENSOR PROPERTIES
fSensor.CommonTasks.SetPatternSimpleConic(fsCHA, 1);

fsRange = fSensor.AccessConstraints.AddConstraint('eCstrRange');
```

```
fsRange.EnableMin = true;
fsRange.EnableMax = true;
fsRange.min = fsRmin;
fsRange.max = fsRmax;

angle = fSensor.AccessConstraints.AddConstraint('eCstrElevationAngle');
angle.EnableMin = true;
angle.EnableMax = true;
angle.Min = fsElmin;
angle.Max = fsElmax;

% Graphics Stuff
fSensor.Graphics.Projection.UseDistance = true;
fSensor.Graphics.Projection.UseConstraints = true;
fSensor.Graphics.Projection.EnableConstraint('ElevationAngle');
fSensor.Graphics.Projection.UseConstraints = true;

disp('Created: Facility Sensor')

end
```

Some notes to anyone looking to extend the capabilities of the `facility` and `fSensor`. So first and foremost, this program assumes that the `fSensor` is a simple conic, more specifically defined by a cone half angle (`fsCHA`). Furthermore, the user can put more constraints on its elevation angles (given by `fsElmin` and `fsElmax`). However, STK supports many other formats for sensors (such as rectangular and custom defined). To keep this program simple and relevant to its main purpose (facilitating design and planning for a cubesat mission), there was no need to model anything more complex than a cone sensor and so this is the only option available. Should one want to define a more complex sensor, they would need to add the relevant parameters and set their properties in a similar manner.

Second, the 'Graphics Stuff' section here is actually necessary for seeing the modified sensor shape in the STK window. By default, it will show just the cone half angle constraint, even if the other constraints are placed. The four lines here will ensure that STK uses the updated constraints when creating the sensor graphics.

A.1.3 satelliteInfo.m

```
function [satellite, sSensor] = satelliteInfo(root, sName, sSMA, sE, sI,
sAP, sAN, sL, sColor, sModel, ssName, ssCHA, ssRmin, ssRmax)
%%% SATELLITE INFORMATION
%   Information for Satellite (object) in Systems Tool Kit
%
%   PARAMETERS
%     sName                       Satellite Name (char array: name)
%     sSMA                        Semimajor Axis (double: km)
%     sE                          Eccentricity (double: unitless)
%     sI                          Angle of Inclination (double: deg)
%     sAP                         Argument of Perigee (double: deg)
%     sAN                         Ascending Node (double: deg)
%     sL                          Location in Orbit (double: deg)
%     sModel                      Satellite Model (char array: model path)
%
%     ssName                      Satellite Sensor Name (char array: name)
%     ssCHA                       Satellite Sensor Cone Half Angle
%                                       (double: deg)
%     ssRmin                      Satellite Sensor Range Min (double: km)
%     ssRmax                      Satellite Sensor Range Max (double: km)
%
%   DEFINITIONS
%     satellite                   Satellite (object)
%     sSensor                     Satellite Sensor (object)
%


%%% DEFINITIONS
% SATELLITE (OBJECT)
satellite = root.CurrentScenario.Children.New('eSatellite', sName);

% SATELLITE PROPERTIES
keplerian =
satellite.Propagator.InitialState.Representation.ConvertTo('eOrbitStateClas
sical'); % Classical Elements
keplerian.SizeShapeType = 'eSizeShapeSemimajorAxis';        % Uses
Eccentricity and Inclination
keplerian.LocationType = 'eLocationTrueAnomaly';            % Makes sure
True Anomaly is being used
keplerian.Orientation.AscNodeType = 'eAscNodeRAAN';         % Use RAAN for
data entry
```

```
keplerian.SizeShape.SemimajorAxis = sSMA;
keplerian.SizeShape.Eccentricity = sE;
keplerian.Orientation.Inclination = sI;
keplerian.Orientation.ArgOfPerigee = sAP;
keplerian.Orientation.AscNode.Value = sAN;
keplerian.Location.Value = sL;

satellite.Propagator.InitialState.Representation.Assign(keplerian);
satellite.Propagator.Propagate;

disp('Created: Satellite')

% Satellite Graphics
satellite.VO.Model.ModelData.Filename = sModel;
satellite.Graphics.Resolution.Orbit = 60;

% Satellite Color
graphics = satellite.Graphics;
graphics.SetAttributesType('eAttributesBasic');
attributes = graphics.Attributes;
attributes.Inherit = false;
attributes.Color = rgb2StkColor(sColor);

disp('Updated: Satellite Model')


%%% SATELLITE SENSOR (OBJECT)
sSensor = satellite.Children.New('eSensor', ssName);


%%% SATELLITE SENSOR PROPERTIES
sSensor.CommonTasks.SetPatternSimpleConic(ssCHA, 1);
sSensor.CommonTasks.SetPointingAlongVector(['Satellite/', sName, '
Body.X'], ['Satellite/', sName, ' North'], 0);

ssRange = sSensor.AccessConstraints.AddConstraint('eCstrRange');
ssRange.EnableMin = true;
ssRange.EnableMax = true;
ssRange.min = ssRmin;
ssRange.max = ssRmax;

sSensor.Graphics.Projection.UseConstraints = true;
sSensor.Graphics.Projection.UseDistance = true;
sSensor.Graphics.Projection.DistanceType = 'eRangeConstraint';

disp('Created: Satellite Sensor')

end
```

Some notes for anyone looking to extend the capabilities of `satellite` and `sSensor`. First and foremost is the definition of the orbit. MESS does not simulate an actual orbit, more specifically the error in trajectory during launch and the slow orbital decay. This program is only intended to be run for an analysis period of several days, for which the orbit is relatively unchanged. Should one desire to incorporate this, I would highly recommend using an established orbit simulator, writing the data to a text file with the appropriate format, and then loading that into STK. While I myself did not do this, the procedure should be extremely similar to the afQ.m function, defined later.

Another note about orbit definition is the preferred orbital elements. While it is possible to define an orbit with a cartesian position and velocity vector (as initial conditions), this is so unfathomably inconvenient to use for elliptical orbits that I would strongly urge against anyone doing it. The keplerian orbital elements chosen here are the easiest to understand, especially considering AGI has created an ‘Orbital Parameters’ tutorial on their website (email them if you can’t find it). While these are the most convenient parameters to use for a constant orbit, there are reasons to use other systems. In which case, you would have to redefine the section on ‘Satellite Properties’. It would probably be best to create a mini program that just creates satellite objects in STK and then test it until you are certain it works. An alternative solution would be to use a conversion system to convert from your expected system into the standard keplerian elements.

The final important detail of this function comes from defining the `sSensor`. This sensor is meant to represent the communication system on the satellite (by default it points along the Satellite Body.X direction). You might be wondering what the purpose of the ‘North’ is for. I don’t actually know. The function expects two inputs, an alignment vector, a constraint vector, and an offset. Since I only needed the first, the other two are set to ‘random values’ that have no effect. And as with the `fSensor` from before, it is possible to create a more custom sensor model, it was just avoided here since there is no need to be that specific.

A.1.4 getAccessQuaternions.m

```
function [accessBools, accessQuaternions] = getAccessQuaternions(root,
scenario, satellite, facility, access, timeVector, dt)
%%% getAccessQuaternions
%     Get Quaternions associated with pointing towards an access window
%
%   INPUTS:
%     root                STK (object)
%     scenario            Scenario (object)
%     satellite           Satellite (object)
%     access              Access (object)
%     timeVector          Time Vector
%     dt                  Time Step
%
%   OUTPUTS:
%     accessBool          Boolean for Access Times
%                               0: Access Unavailable
%                               1: Access Available
%     accessQuaternions    Quaternion associated with Access Pointing
%                               Format: <qs, qx, qy, qz>
%

%%% SETUP
% INITIALIZE MATRICES
count = length(timeVector);
accessBools = false(count,1);
accessQuaternions = zeros(count,4);

% DEFINE VECTORS
      % Need to use try because creating it twice throws an error
arAngleName = [facility.InstanceName, 'AccessRotationAngle'];
arAxisName = [facility.InstanceName, 'AccessRotationAxis'];
try
      % Vector and Angle Factory
      vgtSat = satellite.Vgt;
      VectFactory = vgtSat.Vectors.Factory;
      AngFactory = vgtSat.Angles.Factory;

      % Satellite Antenna Vector
      satAntenna = root.CentralBodies.Earth.Vgt.Vectors.Item('Fixed.X');
      % This line assumes the antenna is along the X Body Axes
      % To make this more versatile, need to make it use data from
      % antenna direction instead of this fixed value

      % Displacement Vector From Satellite to Facility
      pointSCenter = satellite.Vgt.Points.Item('Center');
      pointFCenter = facility.Vgt.Points.Item('Center');
      satAccessVector = VectFactory.CreateDisplacementVector(['sat2',
facility.InstanceName], pointSCenter, pointFCenter);

      % Access Rotation Angle
      accessAngle = AngFactory.Create(arAngleName, ['Rotation Angle between
Body X to ', facility.InstanceName], 'eCrdnAngleTypeBetweenVectors');
      accessAngle.FromVector.SetVector(satAntenna);
      accessAngle.ToVector.SetVector(satAccessVector);
      VectFactory.CreateCrossProductVector(arAxisName, satAntenna,
satAccessVector);

      % Add Vector and Angle to Satellite
      vector = satellite.VO.Vector;
      vectorARAxis = vector.RefCrdns.Add('eVectorElem', ['Satellite/',
satellite.InstanceName, ' ', arAxisName]);
      angleARAngle = vector.RefCrdns.Add('eAngleElem', ['Satellite/',
satellite.InstanceName, ' ', arAngleName]);

      % Make Access Rotation Angle Visible
      vector.VectorSizeScale = 0.2;
      vector.AngleSizeScale = 0.5;
      vectorARAxis.Color = facility.Graphics.Color;
      angleARAngle.Color = facility.Graphics.Color;
      vectorARAxis.Visible = true;
      angleARAngle.Visible = true;
      angleARAngle.LabelVisible = true;
      angleARAngle.AngleValueVisible = true;
catch
      disp('Access Rotation Axis and Angle already exist')
end


%%% DATA PROVIDERS
% ACCESS WINDOWS
try
      accessDP = access.DataProviders.Item('Access
Data').Exec(scenario.StartTime, scenario.StopTime);
      accessStart = cell2mat(accessDP.DataSets.GetDataSetByName('Start
Time').GetValues);
      accessStop = cell2mat(accessDP.DataSets.GetDataSetByName('Stop
Time').GetValues);

      % ACCESS ROTATION AXIS AND ANGLE
      accessRotationAxis =
satellite.DataProviders.GetDataPrvTimeVarFromPath(['Vectors(Fixed)/',
arAxisName]).Exec(scenario.StartTime, scenario.StopTime, dt);
      accessRotationAngle =
satellite.DataProviders.GetDataPrvTimeVarFromPath(['Angles/',
arAngleName]).Exec(scenario.StartTime, scenario.StopTime, dt);


      %%% EXTRACT DATA
      % ACCESS ROTATION AXIS
      accessRotationAxis = [

cell2mat(accessRotationAxis.DataSets.GetDataSetByName('x/Magnitude').GetVal
ues),...

cell2mat(accessRotationAxis.DataSets.GetDataSetByName('y/Magnitude').GetVal
ues),...

cell2mat(accessRotationAxis.DataSets.GetDataSetByName('z/Magnitude').GetVal
ues),...
      ];

      % ACCESS ROTATION ANGLE
      accessRotationAngle =
cell2mat(accessRotationAngle.DataSets.GetDataSetByName('Angle').GetValues);


      %%% COMPUTATION
      chunks = 500000;
      % If you start having memory problems, reduce 'size' as needed
      numChunks = floor(count/chunks);

      for a = 1:numChunks
      % ACCESS QUATERNIONS
      accessQuaternions(1+(a-1)*chunks:a*chunks,1) =
cosd(accessRotationAngle(1+(a-1)*chunks:a*chunks)/2);
      accessQuaternions(1+(a-1)*chunks:a*chunks,2:4) =
accessRotationAxis(1+(a-1)*chunks:a*chunks,:).*sind(accessRotationAngle(1+(
a-1)*chunks:a*chunks)/2);
      end

      accessQuaternions(1+numChunks*chunks:end,1) =
cosd(accessRotationAngle(1+numChunks*chunks:end)/2);
      accessQuaternions(1+numChunks*chunks:end,2:4) =
accessRotationAxis(1+numChunks*chunks:end,:).*sind(accessRotationAngle(1+nu
mChunks*chunks:end)/2);


      % ACCESS BOOLS
      for a = 1:size(accessStart, 1)
      accessStarti = datetime(accessStart(a,:), 'InputFormat', 'dd MMM yyyy
HH:mm:ss.SSSSSSSSS', 'Format', 'dd MMM yyy HH:mm:ss.SSS');
      accessStopi = datetime(accessStop(a,:), 'InputFormat', 'dd MMM yyyy
HH:mm:ss.SSSSSSSSS', 'Format', 'dd mmm yyy HH:mm:ss.SSS');

      startIndex = find(dateshift(accessStarti, 'end', 'second') ==
timeVector);
      stopIndex = find(dateshift(accessStopi, 'end', 'second') ==
timeVector);
      accessBools(startIndex:stopIndex) = true;
      end

catch
      disp('NO ACCESS WINDOWS FOUND')
end

end
```

To think I had the sanity to write this code from scratch. Anyways, here are some notes for anyone looking to modify the code. First of all, recall that we defined a satellite antenna vector in the `satellite` and `sSensor` (Appendix A.1.3). Unfortunately, I did not go out to use that definition here directly, opting instead to just put it in by hand (by default, it does the Satellite Body.X direction again). This is nowhere near ideal and some people would have my head for it. But I didn't want to sit here figuring out all the possible cases and so I just left it as is. A simple improvement would be to only make it the main cartesian directions ($\pm$X, $\pm$Y, and $\pm$Z) and then just make it an input to both of these functions. A more thorough improvement would be to define the vector explicitly and then input it to both functions. Good luck to whoever figures it out, I'll copy your code and write your name on it.

At first glance, we can also see the excessive amount of 'try-catch-end' statements made here. There was a lot of suffering that went into this but as a general rule of thumb, don't try to run the same line of code more than once. It may seem obvious to an experienced programmer, but I didn't know this and it took a lot of 'Dispatch Exceptions' for me to understand why. For that reason, if you plan on running the same code more than once, just make it a 'try-catch-end' statement to save yourself the trouble (or just don't run the same line more than one).

Another thing to note here is the unnecessary amount of character array manipulations for defining the required vectors and angles. If someone were trying to understand this, I would recommend searching up the functions (in the STK Help Pages) and making sense of what each one does. The overall process isn't all that complicated, it just looks complicated because there are a lot of lines of code with seemingly random inputs and outputs.

Something that should jump out to anyone reading this raw code is that I am loading in access quaternions for EVERY point in the analysis period. While this is not the most computationally friendly task, I felt it was necessary for the purpose of this program to avoid creating a vector that only has values loaded in for parts of the simulation (with the rest being zero). For someone learning the program and starting out, seeing all of the angles (labeled) even when they are not being used is something that I see as helpful (it helped me a bit with keeping track of multiple objects at once so hopefully it helps someone). Should that learner desire to improve the efficiency of this function, it would be a good exercise to get familiar with the coding required to make life easier (that and I'm lazy).

Another important aspect of this function is the section with 'chunks', where we load the STK vector and angle to compute the access quaternions. As it turns out, using a time step of 10 ms and an analysis period of 6 hours, we have ~2 million data points. This is a lot of data points and trying to load in all of these values will likely cause your computer to crash (it did to mine at least). So to avoid this unfortunate outcome, the code will process the data in 'chunks', reducing the load on your computer on any given cycle. Once again, it is definitely possible to do this computation in parallel (it is just extremely parallel data extraction), but I have yet to go out and figure it out.

In the section where we are extracting the access booleans, one can see why I used the datetime format from the beginning, it just makes life a little easier when trying to go back and modify the function. Something to note here is the deviation from reality. In reality, one could never start talking to a satellite as soon as the access window started. More realistically, they would ping the satellite, telling it which way to orient itself to maximize signal strength, and then send a response. Since there is a speed of light delay between these two events (on the order of 10-30 ms), the simulation is a bit unrealistic in assuming you will get an immediate response. However, the justification for this would be that 10 ms should not make a noticeable difference if you account for the reorientation time the satellite would take to point towards the ground station. Unfortunately, even this is left up to chance, since we don't physically control how the satellite is oriented before a communication window opens up, but assuming the attitude simulator isn't wildly inaccurate, we should be able to determine the shortest and longest possible times with little difficulty.

A.1.5 getSunQuaternions.m

```
function [sunBools, sunQuaternions] = getSunQuaternions(root, scenario,
satellite, timeVector, dt)
%%% getSunQuaternions
%     Get Quaternions associated with pointing towards the sun
%
%   INPUTS:
%     root                STK (object)
%     scenario            Scenario (object)
%     satellite           Satellite (object)
%     timeVector          Time Vector
%     dt                  Time Step
%
%   OUTPUTS:
%     sunBools            Boolean for Lighting Times
%                               0: Lighting Unavailable (sunlight)
%                               1: Lighting Available (sunlight)
%     sunQuaternions      Quaternion associated with Sun Pointing
%                               Format: <qs, qx, qy, qz>
%

%%% SETUP
% INITIALIZE MATRICES
count = 1+(length(timeVector)-1)/10;
sunBools = false(count,1);
sunQuaternions = zeros(count,4);

% DEFINE VECTORS
      % Need to use try because creating it twice throws an error
try
      % Vector and Angle Factory
      vgtSat = satellite.Vgt;
      VectFactory = vgtSat.Vectors.Factory;
      AngFactory = vgtSat.Angles.Factory;

      % Satellite Solar Array Normal Vector
      satSolarArray = root.CentralBodies.Earth.Vgt.Vectors.Item('Fixed.X');
      % This line assumes solar panel normal vector is +X Body Axes
      % To make this more versatile, need to make it use the data from
      % solar array direction instead of this fixed value

      % Satellite Sun Vector
      satSunVector = vgtSat.Vectors.Item('Sun');

      % Create Sun Rotation Angle
      sunAngle = AngFactory.Create('sunRotationAngle', 'Rotation Angle
between Body +X to Sun', 'eCrdnAngleTypeBetweenVectors');
      sunAngle.FromVector.SetVector(satSolarArray);
      sunAngle.ToVector.SetVector(satSunVector);
      VectFactory.CreateCrossProductVector('sunRotationAxis',
satSolarArray, satSunVector);

      % Add Vector and Angle to Satellite
      vector = satellite.VO.Vector;
      vectorSRAxis = vector.RefCrdns.Add('eVectorElem', ['Satellite/',
satellite.InstanceName, ' sunRotationAxis']);
      angleSRAngle = vector.RefCrdns.Add('eAngleElem', ['Satellite/',
satellite.InstanceName, ' sunRotationAngle']);

      % Making Vector and Angle Visible
      vectorSRAxis.Color = 65535;
      angleSRAngle.Color = 65535;
      vectorSRAxis.Visible = true;
      angleSRAngle.Visible = true;
      angleSRAngle.LabelVisible = true;
      angleSRAngle.AngleValueVisible = true;
catch
      disp('Sun Rotation Axis and Angle already exist')
end


%%% DATA PROVIDERS
% LIGHTING
satLTDP = satellite.DataProviders.GetDataPrvIntervalFromPath('Lighting
Times/Sunlight').Exec(scenario.StartTime, scenario.StopTime);
satLTstart = cell2mat(satLTDP.DataSets.GetDataSetByName('Start
Time').GetValues);
satLTstop = cell2mat(satLTDP.DataSets.GetDataSetByName('Stop
Time').GetValues);

% SUN ROTATION AXIS AND ANGLE
sunRotationAxis =
satellite.DataProviders.GetDataPrvTimeVarFromPath('Vectors(Fixed)/sunRotati
onAxis').Exec(scenario.StartTime, scenario.StopTime, 10*dt);
sunRotationAngle =
satellite.DataProviders.GetDataPrvTimeVarFromPath('Angles/sunRotationAngle'
).Exec(scenario.StartTime, scenario.StopTime, 10*dt);


%%% EXTRACT DATA
% SUN ROTATION AXIS
sunRotationAxis = [

cell2mat(sunRotationAxis.DataSets.GetDataSetByName('x/Magnitude').GetValues
),...

cell2mat(sunRotationAxis.DataSets.GetDataSetByName('y/Magnitude').GetValues
),...

cell2mat(sunRotationAxis.DataSets.GetDataSetByName('z/Magnitude').GetValues
),...
];

% SUN ROTATION ANGLE
sunRotationAngle =
cell2mat(sunRotationAngle.DataSets.GetDataSetByName('Angle').GetValues);


%%% COMPUTATION
% SUN BOOLEANS
for a = 1:size(satLTstart, 1)
      sunStarti = datetime(satLTstart(a,:), 'InputFormat', 'dd MMM yyyy
HH:mm:ss.SSSSSSSSS', 'Format', 'dd MMM yyy HH:mm:ss.SSS');
      sunStopi = datetime(satLTstop(a,:), 'InputFormat', 'dd MMM yyyy
HH:mm:ss.SSSSSSSSS', 'Format', 'dd mmm yyy HH:mm:ss.SSS');

      startIndex = find(dateshift(sunStarti, 'end', 'second') ==
timeVector(1:10:end));
      stopIndex = find(dateshift(sunStopi, 'end', 'second') ==
timeVector(1:10:end));
      sunBools(startIndex:stopIndex) = true;
end


% COMPUTATION
chunks = 500000;
      % If you start having memory problems, reduce 'size' as needed
numChunks = floor(count/chunks);
```

```
endPiece = rem(count, chunks);

% SUN QUATERNIONS
for a = 1:numChunks
      sunQuaternions(1+(a-1)*chunks:a*chunks,1) =
cosd(sunRotationAngle(1+(a-1)*chunks:a*chunks)/2);
      sunQuaternions(1+(a-1)*chunks:a*chunks,2:4) =
sunRotationAxis(1+(a-1)*chunks:a*chunks,:).*sind(sunRotationAngle(1+(a-1)*c
hunks:a*chunks)/2);
end

sunQuaternions(1+numChunks*chunks:end,1) =
cosd(sunRotationAngle(1+numChunks*chunks:endPiece+numChunks*chunks)/2);
sunQuaternions(1+numChunks*chunks:end,2:4) =
sunRotationAxis(1+numChunks*chunks:endPiece+numChunks*chunks,:).*sind(sunRo
tationAngle(1+numChunks*chunks:endPiece+numChunks*chunks)/2);

end
```

Some notes for anyone trying to modify this function. So first and foremost, the default setting is to collect 1 in 10 values and then use a stepwise assumption to interpolate between the known values. This was done for the purpose of reducing computation time and while it is not ideal, the difference of 10 ms to 100 ms should not make a noticeable difference. Should one try to revert these changes, they would need to modify the lines that call for the data from STK (i.e. the lines that call the 'Data Providers'). These lines have the following inputs: startTime, stopTime, interval. By changing the interval back to the default, it would also be required to remove the resize function at the end of the section.

Another thing mentioned in the main section is the direction of the solar array normal vector. By default it is set to the satellite body +X direction and it is hard coded into the function itself. Once again, there is likely a way to go about doing this, but I am too lazy to figure out the cases. Something to note however, is that if the whole satellite is surrounded by solar panels, then only the sides facing the sun will be used. For the user of this function, we only need to know the direction with the largest solar panel and the area they cover. As such, I have decided to leave it as is and just modify it whenever needed as opposed to making it an input (once again, I'm lazy).

The rest of the function is very much similar to getAccessQuaternions.m. It uses the 'try-catch-end' methodology to avoid the errors caused by running this code more than once, it also processes the data in 'chunks', and uses the datetime format to find the sun booleans.

A.1.6 getMagneticField.m

```
function [satBField] = getMagneticField(scenario, satellite, dt)
%%% getMagneticField
%     Get Full Magnetic Field
%
%   INPUTS:
%     scenario          Scenario (object)
%     satellite         Satellite (object)
%     dt                Time Step
%
%   OUTPUTS:
%     satBField         B Field at Satellite Location
%                             ECF Frame (double: Tesla)
%

%%% DATA PROVIDER
% SEET MAGNETIC FIELD
satBDP = satellite.DataProviders.GetDataPrvTimeVarFromPath('SEET Magnetic
Field').Exec(scenario.StartTime, scenario.StopTime, 10*dt);


%%% OUTPUT
% SATELLITE MAGNETIC FIELD
satBField = [
      cell2mat(satBDP.DataSets.GetDataSetByName('B Field - ECF
x').GetValues),...
      cell2mat(satBDP.DataSets.GetDataSetByName('B Field - ECF
y').GetValues),...
      cell2mat(satBDP.DataSets.GetDataSetByName('B Field - ECF
z').GetValues),...
];

end
```

As mentioned in the section, there is no real need for this. I collected data for 1 in 10 steps and then interpolated them so yay. Yeah, there is no real need for this other than I wanted to do it. Modify it if you want but there is no real need to do it.

A.2 MATLAB Components

A.2.1 ssd.m

```
classdef ssd < handle
      %%% ssd
      %      Solid State Drive (Object)
      %
      %   Created by Manav Jadeja on 20220515

      properties
      capacity                % Capacity in MB (megabytes)
      state0                  % Initial State Vector (% full)

      dataGenerationRates     % Data Generation Rate (MB/s)
            % BASED ON COMMAND SYSTEM
            % 1: Nothing Mode
            % 2: Safety Mode
            % 3: Experiment Mode
            % 4: Charging Mode
            % 5: Communication Mode

      h
      end

      methods
          function obj = ssd(capacity, nothingDataGen, safetyDataGen,
experimentDataGen, chargingDataGen, communicationDataGen, state0)
            %%% ssd
            %      Create an ssd
            obj.capacity = capacity;
            obj.dataGenerationRates = [
                  nothingDataGen;
                  safetyDataGen;
                  experimentDataGen;
                  chargingDataGen;
                  communicationDataGen
            ];
            obj.state0 = state0;
          end
      end
end
```

There is nothing special about this object, it was only made to store some values and potentially be called for those values. Someone with experience might ask why I used ‘handles’ for something this simple, to which the answer is, I do it for everything because I never learned about when to actually use it. Is this the best way to code? Not at all. So if you are more experienced in programming, change it as needed to make your code faster/shorter. If you are not experienced, then all I can say is “if it isn’t broken, it may not be worth fixing”. How far that logic is taken is upto the user and I bear no responsibility for it.

A.2.2 magnetorquer.m

```
classdef magnetorquer < handle
      %%% magnetorquer
      %     Magnetorquer (Object)
      %
      %   Created by Manav Jadeja on 20220105

      properties
      magneticDipole                  % Magnetic Dipole (3x1 double: A m^2)
      magneticField                   % External Magnetic Field (3x1 double:
kg A^-1 s^-1)

      h                               % Handle
      end

      methods
        function obj = magnetorquer(dipoleMagnitude, direction,
magneticField)
            %%% magnetorquer
            %     Create a magnetorquer

            obj.magneticDipole = dipoleMagnitude*direction/norm(direction);
            obj.magneticField = magneticField;
        end

        function [Mm] = magneticMoment(obj, B, q)
            %%% magneticMoment
            %     Computes Magnetic Moment from current Magnetic Dipole
            %     (rotated with quaternion) and External Magnetic Field
            %   INPUTS:
            %     B           External Magnetic Field
            %     q           Current Quaternion (orientation)
            %   OUTPUTS:
            %     Mm          Moment caused by Magnetic Dipole and Field
            m = obj.magneticDipole;
            mQ = q*[0;m']*[q(1), -q(2), -q(3), -q(4)];
            Mm = [
                  mQ(2)*B(3) - mQ(3)*B(2),...
                  mQ(3)*B(1) - mQ(1)*B(3),...
                  mQ(1)*B(2) - mQ(2)*B(1),...
            ];
        end
      end
end
```

This is the code for the magnetorque object. The breakdown of the code is to create an object with the magnetic field loaded from STK and be able to use that magnetic field with the current orientation to find the magnetic moment caused. One might note that it would be easier to feed just an index for the current time index and load the magnetic field directly within the magneticMoment function. This is true, but my personal preference is to see the data that is being fed into the function. It's not very difficult to change it and I would recommend it if you want to speed up the code a bit. It would also help to shorten the cross product here using the more optimal 'cpm' function (see A.2.3, reactionWheel.m to see this done properly), but I'm a bit rushed to get this finished so I will not do it myself (I'll call it a future update to be made and move on for now).

A.2.3 reactionWheel.m

```
classdef reactionWheel < handle
      %%% reactionWheel
      %     3 Axis Reaction Wheels (Object)
      %
      %   Created by Manav Jadeja on 20220515

      properties
      state0A            % State Vector (Initial Actual)
      state0E            % State Vector (Initial Estimate)

      inertiaA           % Inertia Matrix (Actual)
      inertiaE           % Inertia Matrix (Estimate)
      maxMoment          % Maximum Moment/Torque

      h                  % Handle
      end

      methods
```

```
        function obj = reactionWheel(state0, state0Error, inertiaA,
inertiaError, maxMoment)
            %%% reactionWheel
            %     Create a 3 axis reaction wheel
            obj.state0A = state0;
            obj.state0E = state0 + state0Error;

            obj.inertiaA = inertiaA;
            obj.inertiaE = inertiaA + inertiaError;
            obj.maxMoment = maxMoment;
        end

        function [Mc] = controlTorque(obj, q, w, K, qd)
            %%% controlTorque
            %     Get control torque with a certain orientation
            %   INPUTS:
            %     q           Quaternion (current)
            %     w           Angular Velocity (current)
            %     K           Quaternion Controller Gains
            %     qd          Quaternion (desired)
            %   OUTPUTS:
            %     Mc          Control Moment

            Mc = quatEMc(obj, q, w, K, qd);

            % Saturation
            if norm(Mc) > obj.maxMoment
                  Mc = obj.maxMoment*(Mc/norm(Mc));
            end
        end

        function [Mc] = quatEMc(obj, q, w, K, qd)
            %%% quatEMc
            %     Control Moment from Quaternion Error
            %   INPUTS:
            %     q           Quaternion (current)
            %     w           Angular Velocity (current)
            %     K           Quaternion Controller Gains
            %     qd          Quaternion (desired)
            %   OUTPUTS:
            %     Mc          Control Moment

            %%% COMPUTE CONTROL MOMENT
            % QUATERNION ERROR
            qErr = qp(obj, qd, [q(1), -q(2), -q(3), -q(4)]);

            % CONTROL MOMENT
            Mc = -K(1)*qErr(2:4) -K(2)*w;
        end

        function [pq] = qp(obj, p, q)
            %%% qp
            %     Kronecker (Quaternion) Product
            %   INPUTS:
            %     p           Quaternion 1
            %     q           Quaternion 2
            %   OUTPUTS:
            %     pq          Quaternion Product (p*q)

            pq = [
                  p(1)*q(1) - p(2)*q(2) - p(3)*q(3) - p(4)*q(4),...
                  p(1)*q(2) + p(2)*q(1) + p(3)*q(4) - p(4)*q(3),...
                  p(1)*q(3) - p(2)*q(4) + p(3)*q(1) + p(4)*q(2),...
                  p(1)*q(4) + p(2)*q(3) - p(3)*q(2) + p(4)*q(1),...
            ];
        end

        function mat = cpm(obj, vec)
            %%% cpm
            %     Computes Standard Cross-Product Matrix from Vector
            %   INPUTS:
            %     vec         3x1 Vector
            %   OUTPUTS:
            %     mat         3x3 Standard Cross-Product Matrix

            mat = [
                        0, -vec(3),  vec(2);
                  vec(3),     0, -vec(1);
                  -vec(2),  vec(1),      0;
            ];
        end
      end
end
```

This is the reaction wheel object. As you can see, it is quite long and there was a lot of time spent figuring out what to do. As mentioned, there are two systems at play here, the actual and estimate. The actual will assuming there is no error and the estimate will have error, however, both obey the same equations of motion and so the same functions can be used to avoid creating the same function twice. That being said, a known source of error is the uncertainty in determining the inertia tensor for a given object. Since this is a key area for error, I've built it in that the reaction wheel system will likely have an error in its inertia matrix (assuming a diagonal matrix with only the principal moments of inertia).

Besides that, the function for determining the control torque is based on a quaternion based PD controller. This can be extended pretty easily using some conversions to and from other rotation schemes (but I have avoided this due to the common use of quaternions and the desire to not bother with quaternion to euler angle conversions). Once again, it is possible, but likely not the most computationally friendly task to do that conversion whilst accounting for the singularity points. Quaternions make my life easier and so I use them everywhere and without hesitation (don't be like this without a good reason).

Another thing to mention are the sub-functions. The first one is the 'qp' function. This is a simple function meant to perform a Kronecker Product between two quaternions. This is also how the quaternion error term is generated and there is definitely room to improve upon it. The second subfunction is the 'cpm' function. This is what should be used instead of the standard cross product formula if speed considerations are important (like in the magnetorquer function for generating the magnetic moment).

Finally, this reaction wheel system takes output saturation into account. A realistic reaction wheel system can only output so much torque before it exceeds the capacity of the hardware. This is accounted for in a simple 'if statement' but there are methods for incorporating this into the control algorithm itself and making the most of the resulting torque. Once again, nonlinear control theory is not my area of expertise but it is possible to do this pretty easily. Should one want to extend the control algorithm even further, they would simply need to add more variables to the end and continuously add them to the main simulation loop (explained in 2.4.2) as needed.

A.2.4 starTracker.m

```
classdef starTracker < handle
      %%% starTracker
      %     Star Tracker (Object)
      %
      %   Created by Manav Jadeja on 20220519

      properties
      oneSigma
      offset

      h
      end

      methods
```

```
        function obj = starTracker(oneSigma, offset)
            %%% starTracker
            %     Create a star tracker

            obj.oneSigma = oneSigma;
            obj.offset = offset;
        end

        function [qEstimate] = perfectAttitudeAcquisition(obj, qActual)
            %%% perfectAttitudeAcquisition
            %      Perfect Attitude Acquisition

            qEstimate = qActual;
        end

        function [qEstimate] = onesigmaAttitudeAcquisition(obj, qActual)
            %%% onesigmaAttitudeAcquisitions
            %     One Sigma Error Attitude Acquisition

            qEstimate = qActual + obj.oneSigma*randn(1,4);
        end

        function [qEstimate] = offsetAttitudeAcquisition(obj, qActual)
            %%% offsetAttitudeAcquisition
            %     Offset Error Attitude Acquisition

            qEstimate = qActual + obj.offset;
        end
      end
end
```

I don't think there is anything special here. This object was made as simple as possible because it would be too computationally intensive to simulate a star field and the associated computer vision. Instead, I opted to condense all of the results into the two most common error types (an offset and some standard gaussian noise). However, one can see pretty easily that it would not be very difficult to add their own custom error using this object as a basis. That is not to discourage someone from attempting it, rather just serve as a hint that thorough understanding of this simulation and its code is to be expected before making such a customizable edit.

A.2.5 battery.m

```
classdef battery < handle
      %%% battery
      %     Battery Object
      %
      %   Created by Manav Jadeja on 20220102

      properties
      capacity          % Battery Capacity (Ahr)
      soc               % State of Charge
      R_charge          % Charging Internal Resistance (Ohms)
      R_discharge       % Discharging Internal Resistance (Ohms)
      maxV              % Battery Max Voltage (V)
      maxI              % Battery Max Current (A)
      cells             % Battery Cells (num)
      lineDrop          % Battery Line Drop

      h                 % Handle
      end

      methods
      function obj = battery(capacity, soc, R_charge, R_discharge,...
                  maxV, maxI, cells, lineDrop)
            %%% Battery
            %     Create a battery

            obj.capacity = capacity;
            obj.soc = soc;
            obj.R_charge = R_charge;
            obj.R_discharge = R_discharge;
            obj.maxV = maxV;
            obj.maxI = maxI;
            obj.cells = cells;
            obj.lineDrop = lineDrop;

      end

      end
end
```

See description of the 'power system' object in A.3.2.

A.2.6 solarArray.m

```
classdef solarArray
      %%% solarArray
      %     Solar Array (Object)
      %
      %   Created by Manav Jadeja 20220103

      properties
      area                        % Solar Array Area (Total)
      normalVector                % Normal Vector to Area
      efficiency                  % Efficiency (E Input -> E Output)
      end

      methods
      function obj = solarArray(area, normalVector, efficiency)
            %%% solarArray
            %     Create a satellite solar array
            obj.area = area;
            obj.normalVector = normalVector/norm(normalVector);
            obj.efficiency = efficiency;
      end
      end
end
```

See description of the ‘power system’ object in A.3.2.

A.2.7 electricalSystem.m

```
classdef electricalSystem
      %%% electricalSystem
      %     Electrical System Loads (Object)
      %
      %   Created by Manav Jadeja on 20220110

      properties
      nothingLoadCurrent           % Current Draw for Nothing Mode (Amps)
      safetyLoadCurrent            % Current Draw for Safety Mode (Amps)
      experimentLoadCurrent        % Current Draw for Experiment Mode
(Amps)
      chargingLoadCurrent          % Current Draw for Charging Mode (Amps)
      communicationLoadCurrent     % Current Draw for Communication Mode
(Amps)
```

```
      % BASED ON COMMAND SYSTEM
            % 1: Nothing Mode
            % 2: Safety Mode
            % 3: Experiment Mode
            % 4: Charging Mode
            % 5: Communication Mode

      end

      methods
      function obj = electricalSystem(nothingLoadCurrent,
safetyLoadCurrent, experimentLoadCurrent, chargingLoadCurrent,
communicationLoadCurrent)
            %%% electricalSystem
            %     Create an electrical system
            obj.nothingLoadCurrent = nothingLoadCurrent;
            obj.safetyLoadCurrent = safetyLoadCurrent;
            obj.experimentLoadCurrent = experimentLoadCurrent;
            obj.chargingLoadCurrent = chargingLoadCurrent;
            obj.communicationLoadCurrent = communicationLoadCurrent;
      end
      end
end
```

See description of the ‘power system’ object in A.3.2.

A.3 MATLAB Systems

A.3.1 commandSystem.m

```
classdef commandSystem < handle
      %%% commandSystem
      %     Command System for a Satellite
      %
      %   Created by Manav Jadeja on 20220103

      properties
      time                        % Time

      socSafe                     % State of Charge (safe for access)
      socUnsafe                   % State of Charge (unsafe for access)
      ssdSafe                     % SSD Capacity (safe for experiment)
      expDuration                 % Experiment Duration

      sunBools                    % Sun Bools
      accessBools                 % Access Bools
      expBools                    % Experiment Bools

      ssd                         % SSD (Object)

      state0                      % Initial State
                                        % 1   Commands
                                        % 2   Data storage use
      end

      methods
          function obj = commandSystem(socSafe, socUnsafe, ssdSafe,
expDuration, dt, sunBools, accessBools, ssd)
            %%% commandSystem
            %     Create a Command System

            obj.socSafe = socSafe;
            obj.socUnsafe = socUnsafe;
            obj.ssdSafe = ssdSafe;
            obj.expDuration = expDuration;

            obj.sunBools = sunBools;
            obj.accessBools = accessBools;
            experimentModeBools(obj, dt, expDuration, accessBools,
sunBools);

            obj.ssd = ssd;

            obj.state0 = [1,obj.ssd.state0];
          end

          function [command] = command(obj, batterySOC, ssdSOC, a)
            %%% command
            %     Takes in some data and returns a command
            %   INPUTS:
            %     obj                 % Command System (object)
            %     t                   % Current Time Index
            %     batterySOC          % Battery SOC
            %     ssdSOC              % SSD SOC
            %     a                   % Current Index
            %   OUTPUTS:
            %     command             % Command (integer)
            %                               % 1: Nothing Mode
            %                               % 2: Safety Mode
            %                               % 3: Experiment Mode
            %                               % 4: Charging Mode
            %                               % 5: Access Location 1
            %                               % 6: Access Location 2
            %                               % N+4: Access Location N

            %%% CURRENT BOOLEANS
            socSafeBool = logical(batterySOC >= obj.socSafe);
            socUnsafeBool = logical(batterySOC <= obj.socUnsafe);
            sunBoolT = obj.sunBools(a);
            accessBoolsT = obj.accessBools(a,:);
            expBoolT = obj.expBools(a);

            %%% COMMAND GENERATION
            % Draw Decision Tree to help!
            if socUnsafeBool
                  if sunBoolT
                  command = 4;
                  else
                  command = 2;
                  end
            else
```

```
                  if sum(accessBoolsT)
                  for a = 1:length(accessBoolsT)
                        if accessBoolsT(a) == 1
                              command = 4+a;
                              break
                        end
                  end
                  else
                  if ~socSafeBool
                        if sunBoolT
                              command = 4;
                        else
                              command = 1;
                        end
                  else
                        if sunBoolT
                              command = 4;
                        else
                              if ssdSOC < obj.ssdSafe && expBoolT
                                  command = 3;
                              else
                                  command = 1;
                              end
                        end
                  end
                  end
            end
          end

          function [expBools] = experimentModeBools(obj, dt, expDuration, 
accessBools, sunBools)
            %%% experimentModeBool
            %     Find and make experiment bools

            % Setup
            intervalNeeded = expDuration/dt;

            % Find empty times
            openBools = any([accessBools,sunBools],2);

            % Find All Free Intervals
            expBools = false(1,length(sunBools));
            matchPattern = [1, zeros(1,intervalNeeded)];
            check = true;
            while check
                  pos = strfind(openBools', matchPattern);
                  for a = 1:length(pos)
                  expBools((pos(a)+1):(pos(a)+intervalNeeded)) = true;
                  openBools((pos(a)+1):(pos(a)+intervalNeeded)) = true;
                  end
                  check = ~isempty(pos);
            end
        end

        function [dataGen] = dataGenerated(obj, dt, state, command)
            %%% dataGenerated
            %     Get data generated from a command

            dataGen = dt*obj.ssd.dataGenerationRates(command*(command<=4) +
5*(command>4));
            nextState = state + dataGen;
            if nextState > obj.ssd.capacity || nextState < 0
                  dataGen = 0;
            end
        end
      end
end
```

Wow that’s a lot of code. And thanks to google docs, it is very poorly aligned. Well I would recommend using github if you are trying to get code rather than copy pasting it from here, but this should help get the idea across (or something idk). Anyways, it's pretty easy to see that the command system will accept the access and sun bools generated earlier and use them to create the experiment bools by looking for windows of time when it is possible to perform experiments. It should also be noted here that the experiment mode can only be reached if many variables align for it (i.e. the simulation is longer than the experiment duration, the battery stays charged enough, the ssd is empty, etc). In my experience, you will likely not get experiment mode very often unless you modify your margins of safety (‘socSafe’ and ‘ssdSafe’ are changed) or you oversize your system (modify the power system to keep the battery charged). It is my recommendation to simulate many short bursts of time with expected values of initial conditions rather than a single long simulation with lots of uncertainty.

Moving on, the ‘command’ function is just a really long (and disorganized) decision tree. It goes through the options presented to it and figures out the command that is possible. Should one decide to modify it for their needs, they make sure the decision tree is always closed and it will output a value for command no matter what. Other than that, there is nothing special about this command.

The final method is the ‘dataGenerated’ function. This just connects the current command to the various data generation rates and outputs the amount of data generated for the time step. Unless someone was trying to modify the commands (and indexes), there should be no real reason to modify this. Should someone decide to partake in that adventure, they would have a lot of modification ahead of them, as they would have to find all the times where the command value is used as an index and modify it as they needed to. I do not recommend this to anyone.

A.3.2 powerSystem.m

```
classdef powerSystem < handle
      %%% powerSystem
      %     Power System for a Satellite
      %
      %   Created by Manav Jadeja on 20220102

      properties
      time                % Time Vector
      state0              % Initial State Vector

      battery             % Battery (object)
      batteryVoltage      % Voltage of battery (vector)
      chargingVoltage     % Charging voltage from battery data curve (vector)
      chargingSoc         % Charging SOC from battery data curve (vector)
      dischargingVoltage % Discharging voltage from battery data curve
(vector)
      dischargingSoc      % Discharging SOC from battery data curve (vector)
      solarArray          % Solar Array (object)
      electricalSystem    % Electrical System (object)

      h                   % Handle
      end

      properties (Constant)
      SimpleSolarValue = 1373       % Solar Flux (W/m^2)
      end

      methods
      function obj = powerSystem(time, battery, batteryData, solarArray,
electricalSystem)
            %%% powerSystem
            %     Create a power system

            obj.time = time;
            obj.battery = battery;
            obj.chargingVoltage = transpose([batteryData.battChgRows.cV]);
            obj.chargingSoc = transpose([batteryData.battChgRows.cSOC]);
            obj.dischargingVoltage =
transpose([batteryData.battDchgRows.dV]);
            obj.dischargingSoc =
transpose([batteryData.battDchgRows.dSOC]);
            obj.solarArray = solarArray;
            obj.electricalSystem = electricalSystem;
            obj.state0 = obj.battery.soc;

            obj.batteryVoltage =
obj.chargingVoltage(binarySearch(obj.chargingSoc, obj.state0));
      end

      function [soc] = step(obj, dt, command)
            %%% Simulates a single time step of the power system

            nothingLoadCurrent = obj.electricalSystem.nothingLoadCurrent;
            safetyLoadCurrent = obj.electricalSystem.safetyLoadCurrent;
            experimentLoadCurrent =
obj.electricalSystem.experimentLoadCurrent;
            chargingLoadCurrent = obj.electricalSystem.chargingLoadCurrent;
            communicationLoadCurrent =
obj.electricalSystem.communicationLoadCurrent;

            % Compute current produced by solar array
            solarArrayCurrent = 0;
            action = command;
            if action == 4 % charging mode
                  solarArrayCurrent = obj.SimpleSolarValue
*obj.solarArray.area / obj.batteryVoltage;
            end

            %%% Compute current used by load
            if action >= 5 % Communication Mode
                  loadCurrent = communicationLoadCurrent;
            elseif action == 4 % Charging Mode
                  loadCurrent = chargingLoadCurrent *
obj.solarArray.efficiency;
            elseif action == 3 % Experiment Mode
                  loadCurrent = experimentLoadCurrent;
            elseif action == 2 % Safety Mode
```

```
                    loadCurrent = safetyLoadCurrent;
              else % Nothing Mode
                    loadCurrent = nothingLoadCurrent;
              end

              %%% Net current through the battery
              % Compute the current through the battery
              batteryCurrent = solarArrayCurrent - loadCurrent;

              %%% Voltage data
              if batteryCurrent > 0
                    voltageCharge =
obj.chargingVoltage(binarySearch(obj.chargingSoc, obj.state0));

                    batteryCurrent = min(batteryCurrent, obj.battery.maxI);

                    obj.batteryVoltage = voltageCharge + obj.battery.R_charge
* batteryCurrent;

              else
                    voltageDischarge =
obj.dischargingVoltage(binarySearch(obj.dischargingSoc, obj.state0));

                    obj.batteryVoltage = voltageDischarge +
obj.battery.R_discharge * batteryCurrent;

              end

              % Compute change in the charge of the battery
              ahrChange = batteryCurrent * dt / 60 / 60;
              % Return the new SOC, clamped between 0 and 1
              soc = min(1, max(0, obj.state0 + ahrChange /
obj.battery.capacity));
              obj.state0 = soc;

        end
        end
end
```

As mentioned earlier, the ‘power system’ object was not created directly, rather referenced with permission for the club activity (the original tool was written by Connie Liou, the President of STAR and initiator for the CubeSat problem). The original code can be found on her GitHub and it cannot be stressed enough: https://github.com/connie-liou/powersimstore.

A.3.3 attitudeSystem.m

```
classdef attitudeSystem < handle
      %%% attitudeSystem
      %     Model of a satellite attitude control system
      %
      %   Created by Manav Jadeja on 20220102

      properties
      state0            % State Vector (Initial)
                              % 1:4       SC Quaternions (Actual)
                              % 5:7       SC Angular Velocities (Actual)
                              % 8:10      RW Angular Velocities (Actual)

                              % 11:14     SC Quaternions (Estimate)
                              % 15:17     SC Angular Velocities (Estimate)
                              % 18:20     RW Angular Velocities (Estimate)

      inertiaA          % Inertia Matrix (Actual)
      inertiaE          % Inertia Matrix (Estimate)
      K                 % Quaternion Controller Gains

      magnetorquer      % Magnetorquer (object)
      reactionWheel     % Reaction Wheel (object)
      starTracker       % Star Tracker (object)

      qd                % Desired Attitude Quaternions
                              % 1: Nothing Mode
                              % 2: Safety Mode
                              % 3: Experiment Mode
                              % 4: Charging Mode
                              % 5: Access Location 1
                              % 6: Access Location 2

      h                 % Handle
      end

      methods
        function obj = attitudeSystem(state0, state0Error, qd, inertiaA,
inertiaError, K, magnetorquer, reactionWheel, starTracker)
            %%% attitudeSystem
            %     Create an attitude control system
            obj.state0 = [
                state0,...                 % SC State Vector (Initial 
Actual)
                reactionWheel.state0A,...  % RW State Vector (Initial 
Actual)
                state0+state0Error,...     % SC State Vector (Initial 
Estimate)
                reactionWheel.state0E;     % RW State Vector (Initial 
Estimate)
            ];

            obj.inertiaA = inertiaA;
            obj.inertiaE = inertiaA + inertiaError;
            obj.K = K;

            obj.magnetorquer = magnetorquer;
            obj.reactionWheel = reactionWheel;
            obj.starTracker = starTracker;

            obj.qd = qd;
        end

        function [dX] = attitudeSystemDynamics(obj, t, dt, X, a, scI, rwI, 
M)
            %%% attitudeSystemDynamics
            %     Attitude Control System Dynamics
            %   INPUTS:
            %     obj         attitudeSystem (obj)
            %     t           Current Time
            %     dt          Time Step
            %     X           State Vector
            %     a           Current Index
            %     scI         SC Inertia
            %     rwI         RW Inertia
            %     M           External + Internal Moments
            %   OUTPUTS:
            %     dX          State Vector Derivative

            %%% SETUP
            q = X(1:4);
            w = X(5:7);
            W = X(8:10);
            % KINEMATICS
            % Satellite Motion
            dq = -qp(obj, [0,w], q)/2;
            dw = scI\(-1*cpm(obj, w)*scI*(w') + M');
            % Reaction Wheel Motion
            dW = rwI\(-1*cpm(obj, W)*rwI*(W') - M');

            %%% OUTPUT
            dX = [dq, dw', dW'];
        end

        function [pq] = qp(obj, p, q)
            %%% qp
            %     Kronecker (Quaternion) Product
            %   INPUTS:
            %     p           Quaternion 1
            %     q           Quaternion 2
            %   OUTPUTS:
            %     pq          Quaternion Product (p*q)

            pq = [
                  p(1)*q(1) - p(2)*q(2) - p(3)*q(3) - p(4)*q(4),...
                  p(1)*q(2) + p(2)*q(1) + p(3)*q(4) - p(4)*q(3),...
                  p(1)*q(3) - p(2)*q(4) + p(3)*q(1) + p(4)*q(2),...
                  p(1)*q(4) + p(2)*q(3) - p(3)*q(2) + p(4)*q(1),...
            ];
        end

        function mat = cpm(obj, vec)
            %%% cpm
            %     Computes Standard Cross-Product Matrix from Vector
            %   INPUTS:
            %     vec         3x1 Vector
            %   OUTPUTS:
            %     mat         3x3 Standard Cross-Product Matrix

            mat = [
                        0, -vec(3),  vec(2);
                   vec(3),       0, -vec(1);
                  -vec(2),  vec(1),       0;
            ];
        end
    end
```

```
end
```

Looking at this code again, I am partially ashamed at how long it took me to get the two state system correct (actual and estimate). Anyways, the main room for customization comes from the fact that a lot of the variable inputs are already present and need to be modified in the main simulation loop. One could also extend on the components within the ‘attitudeSystem’ (‘magnetorquer’, ‘reactionWheel’, and ‘starTracker’) and edit them as needed.

As mentioned in A.2.2, the ‘cpm’ method is what one should use to do cross-products efficiently in MATLAB. Multiplication of a matrix with a vector is more efficient than grabbing and using 6 values from memory individually (or so I think of it, I don’t actually know). But this output of ‘cpm’ is multiplied to perform the cross product (if you can’t convince yourself, just do the algebra and work it out).

The ‘attitudeSystemDynamics’ are the equations of motion for a rigid body using quaternions and angular velocities. There is nothing special, I just dug them out of a training session I got from Chris Gnam in Summer of 2021 and verified it with a book my professor recommended I read (Fundamentals of Spacecraft Attitude Determination and Control - F. Landis Markley and John L. Crassidis).

Extending the capabilities of the ‘attitudeSystem’ falls into one of several categories: more accurate disturbance torques, more complex control algorithms, and smarter system identification (I have left out attitude determination because, as I mentioned before, you can make a whole project on attitude determination). For adding custom disturbance torques, one would have to do it in the main simulation loop. It is possible to add in a custom time-varying disturbance there using the simulation index or defining a custom disturbance method. For the more complex control algorithm, it might be a bit tough as one would need to add the necessary variables in both the ‘reactionWheel’ object as well as the main simulation loop and avoid the confusion of ‘misplaced’ variables (i.e. your state vector data is placed under the time index). As for the system identification, there is no ‘nice’ place to put it but my suggestion would be to add an ‘updateSystemParameters’ method in the ‘attitudeSystem’ object for updating the parameters from a given change in state, and then call this variable whenever a noticeable error is formed between the actual and estimated states. These are just suggestions because (as you can probably tell), I’ve never had to do this before.

A.3.4 satelliteModel.m

```
classdef satelliteModel < handle
      %%% satelliteModel
      %     Model of satellite and relevant properties
      %
      %   Created by Manav Jadeja on 20220102

      properties
      time                 % Time Span
      dt                   % Time Step
      state0            % State Vector (Initial)
      stateS            % State Vector (Simulated)

      powerSystem       % Power System (object)
      attitudeSystem    % Attitude Control System (object)
      commandSystem     % Flight Software System (object)

      h                 % Handle
      end

      methods
        function obj = satelliteModel(time, dt, powerSystem,
attitudeSystem, commandSystem)
            %%% satelliteModel
            %     Create a satellite model

            obj.powerSystem = powerSystem;
            obj.attitudeSystem = attitudeSystem;
            obj.commandSystem = commandSystem;

            obj.time = time;
            obj.dt = dt;

            obj.state0 = [
                  attitudeSystem.state0,...
                  powerSystem.state0,...
                  1,...
                  0,...
            ];
            % state vector format (update as needed)
                  % 1:4       SC Attitude Quaternion (Actual)
                  % 5:7       SC Attitude Angular Velocity (Actual)
                  % 8:10      Reaction Wheel Angular Velocity (Actual)
                  % 11:14     SC Attitude Quaternion (Estimate)
                  % 15:17     SC Attitude Angular Velocity (Estimate)
                  % 18:20     Reaction Wheel Angular Velocity (Estimate)

                  % 21        Battery State of Charge (SOC)
                  % 22        Command
                  % 23        Data Storage Use

            obj.stateS = zeros(length(time), length(obj.state0));
            obj.stateS(1,:) = obj.state0;
          end

          function [] = simulate(obj)
            %%% simulate
            %     Simulation for satellite model

            %%% PRELIMINARY STUFF
            % WAITING BAR
            f = waitbar(0,'Simulating...', 'Name', 'Simulation Progress');

            % VARIABLES
            ssdCapacity = obj.commandSystem.ssd.capacity;
            scIA = obj.attitudeSystem.inertiaA;
            scIE = obj.attitudeSystem.inertiaE;
            rwIA = obj.attitudeSystem.reactionWheel.inertiaA;
            rwIE = obj.attitudeSystem.reactionWheel.inertiaE;


            %%% SIMULATION LOOP
                  % state vector format (update as needed)
                  % 1:4       SC Attitude Quaternion (Actual)
                  % 5:7       SC Attitude Angular Velocity (Actual)
                  % 8:10      Reaction Wheel Angular Velocity (Actual)

                  % 11:14     SC Attitude Quaternion (Estimate)
                  % 15:17     SC Attitude Angular Velocity (Estimate)
                  % 18:20     Reaction Wheel Angular Velocity (Estimate)

                  % 21        Battery State of Charge (SOC)
                  % 22        Command
                  % 23        Data Storage Use

            for a = 1:length(obj.time)-1
                  % Command System
                  command = obj.commandSystem.command(obj.stateS(a,21),...
                  obj.stateS(a,22)/ssdCapacity, a);
                  obj.stateS(a+1,22) = command;
                  obj.stateS(a+1,23) = obj.stateS(a,23) + ...
                  obj.commandSystem.dataGenerated(obj.dt, obj.stateS(a,23),
command);

                  % Simulate Power System
                  obj.stateS(a+1,21) = obj.powerSystem.step(obj.dt,
command);

                  % Control Torque
                  Mc = obj.attitudeSystem.reactionWheel.controlTorque(...
                  obj.stateS(a, 11:14), obj.stateS(a, 15:17),...
                  obj.attitudeSystem.K,...
                  obj.attitudeSystem.qd(a, :, command));

                  % MAGNETIC DISTURBANCE TORQUE
                  %{
                  Mm =
obj.attitudeSystem.magnetorquer.magneticMoment(1e-9*obj.magnetorquer.magnet
icField(a,:), q);
                  %}

                  % Attitude Determination
                  obj.stateS(a,11:14) =
obj.attitudeSystem.starTracker.onesigmaAttitudeAcquisition(obj.stateS(a,1:4
));
                  % [qEstimate] = perfectAttitudeAcquisition(qActual)
                  % [qEstimate] = onesigmaAttitudeAcquisition(qActual)
                  % [qEstimate] = offsetAttitudeAcquisition(qActual)

                  % Actual Attitude Dynamics
                  % [dX] = attitudeSystemDynamics(obj, t, dt, X, a, scI,
rwI, M)
                  obj.stateS(a+1, 1:10) = RK4(@attitudeSystemDynamics,
obj.attitudeSystem,...
                  obj.time(a), obj.dt, obj.stateS(a, 1:10), a, scIA, rwIA,
Mc);

                  % Estimated Attitude Dynamics
                  obj.stateS(a+1,11:20) = RK4(@attitudeSystemDynamics,
obj.attitudeSystem,...
                  obj.time(a), obj.dt, obj.stateS(a,11:20), a, scIE, rwIE,
Mc);

                  % Update Loading Bar
                  if rem(a,1000) == 0
                  percentDone = a/(length(obj.time));
                  waitbar(percentDone, f, sprintf('%.2f', 100*percentDone))
                  end
```

```
            end
            delete(f)
        end
      end
end
```

The explanation for the ‘simulate’ method is in the main body of this text as I felt the need to go into detail there. Here is some information about the ‘satelliteModel’ constructor.

A.4 Other Functions

A.4.1 RK4.m

```
function [X] = RK4(dynamics, obj, t, dt, X, varargin)
%%% RK4
%     Generic 4th Order Runge-Kutta Numerical Integrator
%
%   INPUTS:
%     obj                 Handle for Object
%     dynamics            Handle for System Dynamics
%     t                   Time
%     dt                  Time Step
%     X                   State Vector (step n)
%     varargin            Other Input Arguments
%
%   OUTPUTS:
%     X                   State Vector (step n+1)
%
%   Created by Manav Jadeja on 20220102
%   Based heavily on a function made by Chris Gnam


%%% COMPUTATION OF UPDATED STATE VECTOR
% 4TH ORDER RUNGE-KUTTA COEFFICIENTS
k1 = dt*dynamics(obj, t, 0, X, varargin{:});
k2 = dt*dynamics(obj, t, 0.5*dt, X + 0.5*k1, varargin{:});
k3 = dt*dynamics(obj, t, 0.5*dt, X + 0.5*k2, varargin{:});
k4 = dt*dynamics(obj, t, dt, X + k3, varargin{:});

% UPDATED STATE VECTOR
X = X + (k1 + 2*k2 + 2*k3 + k4)/6;

end
```

There isn't much special about this function. It is just a Runge-Kutta 4th Order (RK4) solver based on something Chris Gnam showed me during a training session we had. The 'RK4' function was exclusively designed for the 'attitudeSystemDynamics' method inside the 'attitudeSystem' object (and vice versa). I added the 't' and 'dt' (time and time step) dependence in the event that I needed to add a time dependent force (such as a custom slosh disturbance) and this would make it easier to implement an externally created slosh disturbance profile with a little modification for the '0.5*dt' values.

A.4.2 afQ.m

```
function [] = afQ(scenario, timeVector, quaternions, angularVelocities)
%%% afQ
%     Creates a quaternion attitude file from given information
%
%   INPUTS:
%     scenario                  Scenario (object)
%     timeVector                Time Vector (datetime: n x 1 vector)
%                                     Format: 'dd MMM yyyy HH:mm:ss.S'
%     quaternion                Quaternions (double: n x 4 matrix)
%                                     Format: <qs, qx, qy, qz>
%     angularVelocities         Angular Velocities (double: n x 3 matrix)
%                                     Format: <wx, wy, wz>
%
%   OUTPUTS:
%     attitudeQ.a               Quaternion Attitude File in ECF Frame
%                                     File created in /tmp/attitudeQ.a
%

%%% OPEN FILE
count = length(timeVector);
fid = fopen('tmp\attitudeQ.txt', 'wt');


%%% PRELIMINARY INFORMATION
% See sample .a file on AGI help site for formatting this stuff
fprintf(fid, 'stk.v.12.2\n');
fprintf(fid, 'BEGIN Attitude\n');
fprintf(fid, 'NumberOfAttitudePoints %f\n', count);
fprintf(fid, 'ScenarioEpoch %s\n', scenario.StartTime);
fprintf(fid, 'BlockingFactor 20\n');
fprintf(fid, 'InterpolationOrder 1\n');
fprintf(fid, 'CentralBody Earth\n');
fprintf(fid, 'CoordinateAxes Fixed\n');
fprintf(fid, 'TimeFormat UTCG\n');
fprintf(fid, 'AttitudeTimeQuatAngVels\n');
      % Format: <qx, qy, qz, qs, wx, wy, wz>
fprintf(fid, '\n');
fclose(fid);


%%% ADD QUATERNION AND ANGULAR VELOCITY DATA
```

```
chunks = 500000;
      % If you start having memory problems, reduce 'size' as needed
numChunks = floor(count/chunks);
for a = 1:numChunks
      writematrix([datestr(timeVector(1+(a-1)*chunks:a*chunks), 'dd mmm
yyyy HH:MM:SS.FFF'), repmat(' ',[chunks 1]),...
      num2str(quaternions(1+(a-1)*chunks:a*chunks,2), '%+.6e'), repmat('
',[chunks 1]),...             % qx
      num2str(quaternions(1+(a-1)*chunks:a*chunks,3), '%+.6e'), repmat('
',[chunks 1]),...             % qy
      num2str(quaternions(1+(a-1)*chunks:a*chunks,4), '%+.6e'), repmat('
',[chunks 1]),...             % qz
      num2str(quaternions(1+(a-1)*chunks:a*chunks,1), '%+.6e'), repmat('
',[chunks 1]),...             % qw
      num2str(angularVelocities(1+(a-1)*chunks:a*chunks,1), '%+.6e'),
repmat(' ',[chunks 1]),...    % wx
      num2str(angularVelocities(1+(a-1)*chunks:a*chunks,2), '%+.6e'),
repmat(' ',[chunks 1]),...    % wy
      num2str(angularVelocities(1+(a-1)*chunks:a*chunks,3), '%+.6e'),
repmat(' ',[chunks 1]),...    % wz
      ],...
      'tmp\attitudeQ.txt', 'Delimiter', 'space', 'QuoteStrings', false,
'WriteMode', 'append')

end

endSize = rem(count, chunks);

writematrix([datestr(timeVector(1+numChunks*chunks:end), 'dd mmm yyyy
HH:MM:SS.FFF'), repmat(' ',[endSize 1]),...
      num2str(quaternions(1+numChunks*chunks:end,2), '%+.6e'), repmat('
',[endSize 1]),...            % qx
      num2str(quaternions(1+numChunks*chunks:end,3), '%+.6e'), repmat('
',[endSize 1]),...            % qy
      num2str(quaternions(1+numChunks*chunks:end,4), '%+.6e'), repmat('
',[endSize 1]),...            % qz
      num2str(quaternions(1+numChunks*chunks:end,1), '%+.6e'), repmat('
',[endSize 1]),...            % qw
      num2str(angularVelocities(1+numChunks*chunks:end,1), '%+.6e'),
repmat(' ',[endSize 1]),...   % wx
      num2str(angularVelocities(1+numChunks*chunks:end,2), '%+.6e'),
repmat(' ',[endSize 1]),...   % wy
      num2str(angularVelocities(1+numChunks*chunks:end,3), '%+.6e'),
repmat(' ',[endSize 1]),...   % wz
      ],...
      'tmp\attitudeQ.txt', 'Delimiter', 'space', 'QuoteStrings', false,
'WriteMode', 'append')


%%% FINAL EDITS AND CLOSE FILE
fid = fopen('tmp\attitudeQ.txt', 'a+');
fprintf(fid, 'END Attitude\n');
fclose(fid);


%%% ADD .a EXTENSION
file1 = 'tmp\attitudeQ.txt';
file2 = strrep(file1,'.txt','.a');
copyfile(file1,file2)
delete('tmp\attitudeQ.txt')

disp('Attitude File Created')


end
```

To put it simply, this is just a function that takes in the quaternion data from the simulation and converts it into a format that can be loaded into STK and onto the satellite model in there. It uses the quaternions and angular velocities to ensure a smooth interpolation. The first main chunk just writes the 'file parameters' (i.e. number of points, start time, interpolation order, etc). The second chunk loads in the quaternion and angular velocity data and writes it to a text file in chunks (doing it all at once will often crash your computer). Finally, the file is rewritten to have a '.a' extension because that's how STK wants it. The generated file can be loaded into STK.

A.5 Plotting Functions

A.5.1 plotEverything.m

```
function [f] = plotEverything(satelliteModel)
disp('Plotting...')

%%% PRELIMINARY INFORMATION
% TIME STUFF
duration = length(satelliteModel.time);
dt = satelliteModel.dt;
timeHours = (1:length(satelliteModel.time))*dt/60/60;


%%% MAIN FIGURE
f = figure('Name', 'Plots', 'Position', [100 100 1600 800]);
tabgp = uitabgroup(f, 'Position', [0.01,0.1, 0.99, 0.9]);


%%% PLOTTING FUNCTION CALLBACK
% ATTITUDE TAB
attitudeTab = uitab(tabgp,'Title','Attitude System');
attHandles = plotAttitudeSim(attitudeTab, satelliteModel, timeHours, 1,
duration);

% POWER TAB
powerTab = uitab(tabgp,'Title','Power System');
powHandles = plotPowerSim(powerTab, satelliteModel, timeHours, 1,
duration);

% COMMAND TAB
commandTab = uitab(tabgp,'Title','Command System');
comHandles = plotCommandSim(commandTab, satelliteModel, timeHours, 1,
duration);


%%% UICONTROL
% SLIDERS AND TEXT
startTimeSlider =
uicontrol('Parent',f,'Style','slider','Position',[30,20,700,10],...
      'value',1,'min',1,'max',duration); % NEED TO DEFINE THE CALLBACK
AFTER OTHER SLIDER IS MADE
```

```
startTimeLabel = uicontrol('Parent',f,'Style','text',...
      'Position',[30,40,100,15], 'String','Start Time');

lengthTimeSlider =
uicontrol('Parent',f,'Style','slider','Position',[830,20,700,10],...
      'value',duration,'min',1,'max',duration);
lengthTimeLabel = uicontrol('Parent',f,'Style','text',...
      'Position',[830,40,100,15], 'String','Length Time');

% CALLBACKS
startTimeSlider.Callback = {@startTimeSliderCallback, lengthTimeSlider,...
      dt, attHandles, powHandles, comHandles};
lengthTimeSlider.Callback = {@lengthTimeSliderCallback, startTimeSlider,...
      dt, attHandles, powHandles, comHandles};


%%% CALLBACK FUNCTIONS
function startTimeSliderCallback(slider1, eventData, slider2, dt,...
      attHandles, powHandles, comHandles) %#ok<INUSL>
startIndex = get(slider1, 'Value');
lengthIndex = get(slider2, 'Value');

attFn = fieldnames(attHandles);
for a = 1:numel(attFn)
      set(attHandles.(attFn{a}), 'XLim', [startIndex
startIndex+lengthIndex].*dt/60/60)
end

powFn = fieldnames(powHandles);
for a = 1:numel(powFn)
      set(powHandles.(powFn{a}), 'XLim', [startIndex
startIndex+lengthIndex].*dt/60/60)
end

comFn = fieldnames(comHandles);
for a = 1:numel(comFn)
      set(comHandles.(comFn{a}), 'XLim', [startIndex
startIndex+lengthIndex].*dt/60/60)
end



end
```

```
function lengthTimeSliderCallback(slider1, eventData, slider2, dt,...
      attHandles, powHandles, comHandles) %#ok<INUSL>
lengthIndex = get(slider1, 'Value');
startIndex = get(slider2, 'Value');

attFn = fieldnames(attHandles);
for a = 1:numel(attFn)
      set(attHandles.(attFn{a}), 'XLim', [startIndex
startIndex+lengthIndex].*dt/60/60)
end

powFn = fieldnames(powHandles);
for a = 1:numel(powFn)
      set(powHandles.(powFn{a}), 'XLim', [startIndex
startIndex+lengthIndex].*dt/60/60)
end

comFn = fieldnames(comHandles);
for a = 1:numel(comFn)
      set(comHandles.(comFn{a}), 'XLim', [startIndex
startIndex+lengthIndex].*dt/60/60)
end


end

end
```

This is the main function for the plotting. The first section of the code creates multiple tab groups (one for attitude, power, and command simulation), the second section has the subfunctions for plotting the data and returning the many plot handles, the third section creates the sliders and defines the callback functions for updating the plots using the handles saved earlier.

A.5.2 plotAttitudeSim.m

```
function [attitudeHandles] = plotAttitudeSim(ax, satelliteModel, timeHours,
startTime, lengthTime)

%%% SETUP
% TABS IN ATTITUDE SIM
```

```
attitudeTabgp = uitabgroup(ax, 'Position', [0.01,0.01, 0.99, 0.99]);

% PRELIMINARY INFORMATION
dt = satelliteModel.dt;
duration = length(satelliteModel.time);
qd = zeros(duration, 4);
for i = 1:size(satelliteModel.stateS,1)
      command = satelliteModel.stateS(i,22);
      qd(i,:) = satelliteModel.attitudeSystem.qd(i,:,command);
end
stateS = satelliteModel.stateS;


%%% SPACECRAFT QUATERNIONS
scQuaterTab = uitab(attitudeTabgp,'Title','SC Quaternions');
axes('parent',scQuaterTab);

attitudeHandles.scQuat1Axes = subplot(4,1,1);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,1))
hold(attitudeHandles.scQuat1Axes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,11))
plot(timeHours(startTime:lengthTime), qd(startTime:lengthTime,1))
title('q1')
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([-1.5 1.5])
legend('Actual','Estimate','Desired')
hold(attitudeHandles.scQuat1Axes,'off');

attitudeHandles.scQuat2Axes = subplot(4,1,2);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,2))
hold(attitudeHandles.scQuat2Axes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,12))
plot(timeHours(startTime:lengthTime), qd(startTime:lengthTime,2))
title('q2')
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([-1.5 1.5])
legend('Actual','Estimate','Desired')
hold(attitudeHandles.scQuat2Axes,'off');

attitudeHandles.scQuat3Axes = subplot(4,1,3);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,3))
hold(attitudeHandles.scQuat3Axes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,13))
plot(timeHours(startTime:lengthTime), qd(startTime:lengthTime,3))
title('q3')
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([-1.5 1.5])
legend('Actual','Estimate','Desired')
hold(attitudeHandles.scQuat3Axes,'off');

attitudeHandles.scQuat4Axes = subplot(4,1,4);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,4))
hold(attitudeHandles.scQuat4Axes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,14))
plot(timeHours(startTime:lengthTime), qd(startTime:lengthTime,4))
title('q4')
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([-1.5 1.5])
legend('Actual','Estimate','Desired')
hold(attitudeHandles.scQuat4Axes,'off');


%%% SEMILOG ERROR PLOTS
scQErrorTab = uitab(attitudeTabgp,'Title','SC Quaternion Error');
axes('parent',scQErrorTab);

% ACTUAL - ESTIMATE
attitudeHandles.scQ1AE = subplot(4,2,1);
semilogy(attitudeHandles.scQ1AE, timeHours(startTime:lengthTime),...
      abs(stateS(startTime:lengthTime,1) -
stateS(startTime:lengthTime,11)))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([1e-15 2*1e1])
yticks([1e-15 1e-10 1e-5 1e0])
yticklabels({'1e-15','1e-10','1e-5','1e0'})
title('q1 (Actual - Estimate)')

attitudeHandles.scQ2AE = subplot(4,2,3);
semilogy(attitudeHandles.scQ2AE, timeHours(startTime:lengthTime),...
      abs(stateS(startTime:lengthTime,2) -
stateS(startTime:lengthTime,12)))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([1e-15 2*1e1])
yticks([1e-15 1e-10 1e-5 1e0])
yticklabels({'1e-15','1e-10','1e-5','1e0'})
title('q2 (Actual - Estimate)')
```

```
attitudeHandles.scQ3AE = subplot(4,2,5);
semilogy(attitudeHandles.scQ3AE, timeHours(startTime:lengthTime),...
      abs(stateS(startTime:lengthTime,3) -
stateS(startTime:lengthTime,13)))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([1e-15 2*1e1])
yticks([1e-15 1e-10 1e-5 1e0])
yticklabels({'1e-15','1e-10','1e-5','1e0'})
title('q3 (Actual - Estimate)')

attitudeHandles.scQ4AE = subplot(4,2,7);
semilogy(attitudeHandles.scQ4AE, timeHours(startTime:lengthTime),...
      abs(stateS(startTime:lengthTime,4) -
stateS(startTime:lengthTime,14)))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([1e-15 2*1e1])
yticks([1e-15 1e-10 1e-5 1e0])
yticklabels({'1e-15','1e-10','1e-5','1e0'})
title('q4 (Actual - Estimate)')

% ACTUAL - DESIRED
attitudeHandles.scQ1AD = subplot(4,2,2);
semilogy(attitudeHandles.scQ1AD, timeHours(startTime:lengthTime),...
      abs(stateS(startTime:lengthTime,1) - qd(startTime:lengthTime,1)))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([1e-15 2*1e1])
yticks([1e-15 1e-10 1e-5 1e0])
yticklabels({'1e-15','1e-10','1e-5','1e0'})
title('q1 (Actual - Desired)')

attitudeHandles.scQ2AD = subplot(4,2,4);
semilogy(attitudeHandles.scQ2AD, timeHours(startTime:lengthTime),...
      abs(stateS(startTime:lengthTime,2) - qd(startTime:lengthTime,2)))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([1e-15 2*1e1])
yticks([1e-15 1e-10 1e-5 1e0])
yticklabels({'1e-15','1e-10','1e-5','1e0'})
title('q2 (Actual - Desired)')

attitudeHandles.scQ3AD = subplot(4,2,6);
semilogy(attitudeHandles.scQ3AD, timeHours(startTime:lengthTime),...
      abs(stateS(startTime:lengthTime,3) - qd(startTime:lengthTime,3)))
```

```
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([1e-15 2*1e1])
yticks([1e-15 1e-10 1e-5 1e0])
yticklabels({'1e-15','1e-10','1e-5','1e0'})
title('q3 (Actual - Desired)')

attitudeHandles.scQ4AD = subplot(4,2,8);
semilogy(attitudeHandles.scQ4AD, timeHours(startTime:lengthTime),...
      abs(stateS(startTime:lengthTime,4) - qd(startTime:lengthTime,4)))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([1e-15 2*1e1])
yticks([1e-15 1e-10 1e-5 1e0])
yticklabels({'1e-15','1e-10','1e-5','1e0'})
title('q4 (Actual - Desired)')


%%% SPACECRAFT ANGULAR VELOCITY
scAngVelTab = uitab(attitudeTabgp,'Title','SC Angular Velocity');
axes(scAngVelTab);

attitudeHandles.scAngVelXAxes = subplot(3,1,1);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,5))
hold(attitudeHandles.scAngVelXAxes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,15))
xlim([startTime startTime+lengthTime]*dt/60/60)
xlabel('Time (hours)')
ylabel('Angular Velocity (rad/s)')
title('Satellite Angular Velocity X')
legend('Actual', 'Estimate')
hold(attitudeHandles.scAngVelXAxes,'off');

attitudeHandles.scAngVelYAxes = subplot(3,1,2);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,6))
hold(attitudeHandles.scAngVelYAxes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,16))
xlim([startTime startTime+lengthTime]*dt/60/60)
xlabel('Time (hours)')
ylabel('Angular Velocity (rad/s)')
title('Satellite Angular Velocity Y')
legend('Actual', 'Estimate')
hold(attitudeHandles.scAngVelYAxes,'off');

attitudeHandles.scAngVelZAxes = subplot(3,1,3);
```

```
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,7))
hold(attitudeHandles.scAngVelZAxes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,17))
xlim([startTime startTime+lengthTime]*dt/60/60)
xlabel('Time (hours)')
ylabel('Angular Velocity (rad/s)')
title('Satellite Angular Velocity Z')
legend('Actual', 'Estimate')
hold(attitudeHandles.scAngVelZAxes,'off');


%%% REACTION WHEEL ANGULAR VELOCITY
rwAngVelTab = uitab(attitudeTabgp,'Title','RW Angular Velocity');
axes(rwAngVelTab);

attitudeHandles.rwAngVelXAxes = subplot(3,1,1);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,8))
hold(attitudeHandles.rwAngVelXAxes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,18))
xlim([startTime startTime+lengthTime]*dt/60/60)
title('Reaction Wheel X')
xlabel('Time (hours)')
ylabel('Angular Velocity (rad/s)')
title('Reaction Wheel X Angular Velocity')
legend('Actual','Estimate')
hold(attitudeHandles.rwAngVelXAxes,'off');

attitudeHandles.rwAngVelYAxes = subplot(3,1,2);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,9))
hold(attitudeHandles.rwAngVelYAxes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,19))
xlim([startTime startTime+lengthTime]*dt/60/60)
title('Reaction Wheel Y')
xlabel('Time (hours)')
ylabel('Angular Velocity (rad/s)')
title('Reaction Wheel Y Angular Velocity')
legend('Actual','Estimate')
hold(attitudeHandles.rwAngVelYAxes,'off');

attitudeHandles.rwAngVelZAxes = subplot(3,1,3);
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,10))
hold(attitudeHandles.rwAngVelZAxes,'on');
plot(timeHours(startTime:lengthTime), stateS(startTime:lengthTime,20))
```

```
xlim([startTime startTime+lengthTime]*dt/60/60)
title('Reaction Wheel Z')
xlabel('Time (hours)')
ylabel('Angular Velocity (rad/s)')
title('Reaction Wheel Z Angular Velocity')
legend('Actual','Estimate')
hold(attitudeHandles.rwAngVelZAxes,'off');


%%% MAGNETIC FIELD
magFieldTab = uitab(attitudeTabgp,'Title','Magnetic Field');
magFieldAxes = axes(magFieldTab);

plot(magFieldAxes, timeHours(startTime:lengthTime),...

satelliteModel.attitudeSystem.magnetorquer.magneticField(startTime:lengthTi
me,:))
xlim([startTime startTime+lengthTime]*dt/60/60)
xlabel('Time (hours)')
ylabel('Magnetic Field (nT)')
attitudeHandles.magField = gca;

end
```

Man that is a lot of code. In short, it just creates a bunch of tabs, and within each, the appropriate graph(s). It's really not that special though it may take a second to go through and get what you want. Before you try modifying this, I highly recommend making a figure with several tabs and several graphs within each tab. Use this as a basis, but make sure you can do that before you try adding your own. I'm not going to go into detail here because this 'documentation' is long enough as it is. This just plots the spacecraft quaternions, spacecraft quaternion error, spacecraft angular velocity, reaction wheel speed, and magnetic field.

A.5.3 plotPowerSim.m

```
function [powerHandles] = plotPowerSim(ax, satelliteModel, timeHours,
startTime, lengthTime)

%%% SETUP
% TABS IN ATTITUDE SIM
powerTabgp = uitabgroup(ax, 'Position', [0.01,0.01, 0.99, 0.99]);

% PRELIMINARY INFORMATION
```

```
dt = satelliteModel.dt;


%%% POWER SIMULATION
powerTab = uitab(powerTabgp,'Title','Power Simulation');
powerTabAxes = axes(powerTab);

plot(powerTabAxes, timeHours(startTime:lengthTime),...
      satelliteModel.stateS(startTime:lengthTime,21))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([-0.1 1.1])
title('Battery SOC')
xlabel('Time (hours)')
ylabel('SOC (%)')
powerHandles.power = gca;

end
```

The most simple one, this just has a single plot, the battery state of charge (SOC). I've considered adding a feature that plots the 'socSafe' and 'socUnsafe' values as a solid horizontal line, but there values should be known and the user doesn't have enough resolution on these plots to use that information (it's easier to just eyeball it and see the changed mode). Might be a future update though.

A.5.4 plotCommandSim.m

```
function [commandHandles] = plotCommandSim(ax, satelliteModel, timeHours,
startTime, lengthTime)

%%% SETUP
% TABS IN ATTITUDE SIM
commandTabgp = uitabgroup(ax, 'Position', [0.01,0.01, 0.99, 0.99]);

% PRELIMINARY INFORMATION
dt = satelliteModel.dt;


%%% COMMANDS
commandTab = uitab(commandTabgp,'Title','Commands');
commandAxes = axes(commandTab);

plot(commandAxes, timeHours(startTime:lengthTime),...
      satelliteModel.stateS(startTime:lengthTime,22))
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([0 8])
yticklabels({'', 'Nothing Mode','Safety Mode','Experiment Mode',...
      'Charging Mode','Access 1','Access 2','Access 3'})
xlabel('Time (hours)')
ylabel('Command')
commandHandles.commands = gca;
% Command (integer)
      % 1: Nothing Mode
      % 2: Safety Mode
      % 3: Experiment Mode
      % 4: Charging Mode
      % 5: Access Location 1
      % 6: Access Location 2
      % N+4: Access Location N

%%% DATA STORAGE
dataTab = uitab(commandTabgp,'Title','Data Storage');
dataAxes = axes(dataTab);

plot(dataAxes, timeHours(startTime:lengthTime),...
satelliteModel.stateS(startTime:lengthTime,23)./satelliteModel.commandSyste
m.ssd.capacity)
xlim([startTime startTime+lengthTime]*dt/60/60)
ylim([-0.1 1.1])
title('Data Storage')
xlabel('Time (hours)')
ylabel('Percent Filled')
commandHandles.data = gca;

end
```

The second simplest simulation to plot, the command system plots the SSD storage use and commands over time. The commands are nice to see so you can get a general sense of what the satellite is trying to do and the SSD is for seeing the experiment data slowly fill up and then try to predict how much time it would take to downlink enough data to make space for the next experiment. Once again, I've considered plotting the 'ssdSafe' value as a horizontal line but it's not very useful since once is better off eyeballing this value.