VISORS Mission Orbit & Dynamics Simulation Using a Realtime Dynamics Processor

Elizabeth Kimmel[1] and E. Glenn Lightsey[2]
Georgia Institute of Technology, Atlanta, GA, 30318

VIrtual Super-resolution Optics using Reconfigurable Swarms (VISORS) is a precision formation-flying mission which uses two 6U CubeSats with a Science Mode separation distance of 40 meters to emulate a 40-meter focal length diffractive optic telescope. Due to the novelty of the technology used to achieve the stringent relative positioning requirements, the dynamics of these orbits must be simulated to verify the concept of operations (ConOps), the commercial spacecraft bus flight software (FSW), the guidance, navigation, and control (GNC) formation-keeping algorithm, and the attitude determination and control system (ADCS) performance, among others. Verifying these aspects helps ensure that issues such as reaction wheel saturation, pointing errors, or collision risks, among others, do not arise during the mission. This paper describes the work done in simulating the spacecraft dynamics during the mission's Science Operations using COSMOS to interface with the Realtime Dynamics Processor (RDP) and spacecraft bus Engineering Design Unit (EDU) provided by Blue Canyon Technologies (BCT).

I. Acronyms

ADCS = Attitude Dynamics and Control System
BCT = Blue Canyon Technologies
CDH = Command and Data Handling
ConOps = Concept of Operations
DCM = Direction cosine matrix
DSC = Detector Spacecraft
ECEF = Earth-centered Earth-fixed
ECI = Earth-centered inertial
EDU = Engineering Development Unit
EPS = Electrical Power System
FKA = Formation-keeping algorithm
FSW = Flight Software
GNC = Guidance, Navigation, and Control
ISL = Intersatellite link
NavSen = Navigation sensors
OSC = Optics Spacecraft
RAAN = Right ascension of the ascending node
RF = Radio frequency
RDP = Realtime Dynamics Processor
RTN = Radial-tangential-normal
SSO = Sun Synchronous Orbit
TAI = International Atomic Time

[1] Graduate Research Assistant, Daniel Guggenheim School of Aerospace Engineering, ekimmel6@gatech.edu
[2] Professor, Daniel Guggenheim School of Aerospace Engineering, glenn.lightsey@gatech.edu

UUT = Unit Under Test
VISORS = Virtual Super-Resolution Optics using Reconfigurable Swarms

II. Variables

a = semi-major axis
i = inclination
J_2 = J_2 coefficient representing oblateness of the Earth
R_E = radius of the Earth
T = orbital period
β = orbit beta angle
δ_s = declination of the Sun
μ = Earth standard gravitational parameter
Ω = RAAN
Ω_s = right ascension of the Sun

III. VISORS Mission Overview

The Virtual Super-Resolution Optics using Reconfigurable Swarms (VISORS) is a precision formation-flying mission which uses two 6U CubeSats in low Earth orbit with a 40-meter separation distance during Science Mode to emulate a monolithic, 40-meter focal length diffractive optic telescope for studying high energy release regions of the solar corona. The Optics spacecraft (OSC) houses the diffractive optics instrument, a photon sieve, which creates a specific pattern of light that lands on the detector, housed in the detector spacecraft (DSC) 40 meters away. The photons are measured and post-processed to form a high-resolution image of the solar corona. This mission requires high precision formation-flying capabilities for both the OSC and the DSC, and as a result, a high-fidelity simulation of attitude and orbital dynamics is required during testing to verify the performance of the payload GNC algorithms, bus attitude determination and control (ADCS) algorithms, and mission concept of operations (CONOPS) before flight.

A. ConOps

Before describing the test setup and simulations, a fundamental understanding of the mission ConOps is required. The mission begins with preliminary operations, during which both spacecraft will power on and check out their subsystems, then establish a safe initial relative orbit configuration after sharing sufficient relative navigation information between the spacecraft over the radio frequency (RF) intersatellite link (ISL) system, at which point they will transition to Standby Mode. In this configuration, the spacecraft maintain a minimum relative cross-track separation of 200 meters. This configuration is safe for the spacecraft because there is a large margin of passive safety; more specifically, if the spacecraft lost the ability to maneuver within this configuration, it would be a matter of days before there would be any collision risk. After the ground identifies a potentially interesting science opportunity, the ground will command a transition into Transfer Mode, which reconfigures the relative orbit geometry over the course of 10-20 orbits such that the spacecraft maintain a minimum cross-track separation of 20 meters and can move into their Science configuration with a 40-meter separation for high resolution imaging of the Sun. For ten orbits, the spacecraft maintain their Science Mode configuration where once during each orbit they achieve the proper alignment to record images. After these ten orbits are completed, the spacecraft transition into Transfer Mode where once again over the course of 10-20 orbits they are reconfigured through a series of propulsive maneuvers back into the Standby Mode onfiguration. The cyclic flow between Standby, Transfer, and Science defines the mission's nominal science operations. In the case of off-nominal scenarios, the spacecraft will enter a Safe mode in which actions are taken to protect the spacecraft and determine whether or not there is a collision risk requiring the execution of an escape maneuver. These off-nominal modes are part of the complex contingency operations in development for this mission [1,2]. This concept of operations flow is shown in Figure 1, and the Science Mode relative geometry is visualized in Figure 2.

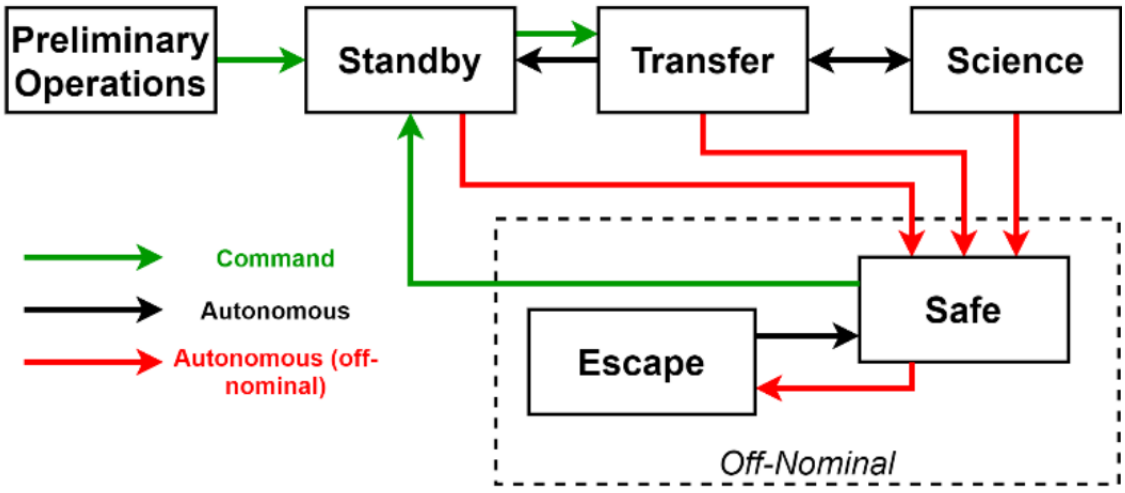

Figure 1. Concept of Operations state flow [3].

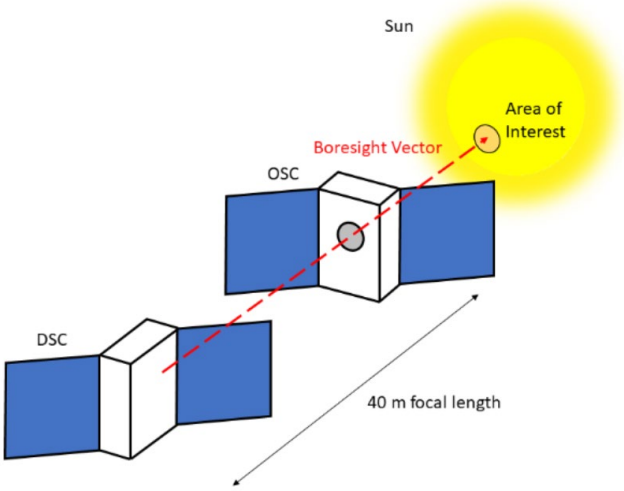

Figure 2. Science operations distributed spacecraft configuration [3].

B. Project Organization

This mission was selected as part of a National Science Foundation (NSF) Ideas workshop and each subsystem is developed by a different organization, whether commercial, government, or university. The chassis and primary avionics, ADCS, structure, communications, and Electrical Power System (EPS) system is provided through the commercial vendor Blue Canyon Technologies (BCT). Georgia Tech is the systems engineering team responsible for project management, overseeing subsystem development, interfacing with BCT and the other subsystem groups, and system integration and testing. As a few examples of some of the subsystems and their respective contributors, the science instrumentation (the detector and the photon sieve) is developed by NASA Goddard Spaceflight Center, the precision formation-keeping GNC algorithms are developed by Stanford, and the ISL system is developed by Washington State University and Ohio State University.

IV. Testbed Setup Overview

In order to verify mission aspects such as the ConOps, the commercial spacecraft bus FSW, the GNC formation-keeping algorithm, and the ADCS performance, among others, a testbed setup is required which can emulate the mission components as well as their interactions, mode transitions, orbital dynamics, and attitude dynamics of the spacecraft. This particular test configuration consists of an XB1 Engineering Development Unit (EDU) provided by

BCT to simulate the functionality of the Generation-3 XB1 flight bus which is used for the VISORS mission. The XB1 EDU and the XB1 flight bus (DSC) are pictured in Figure 3.

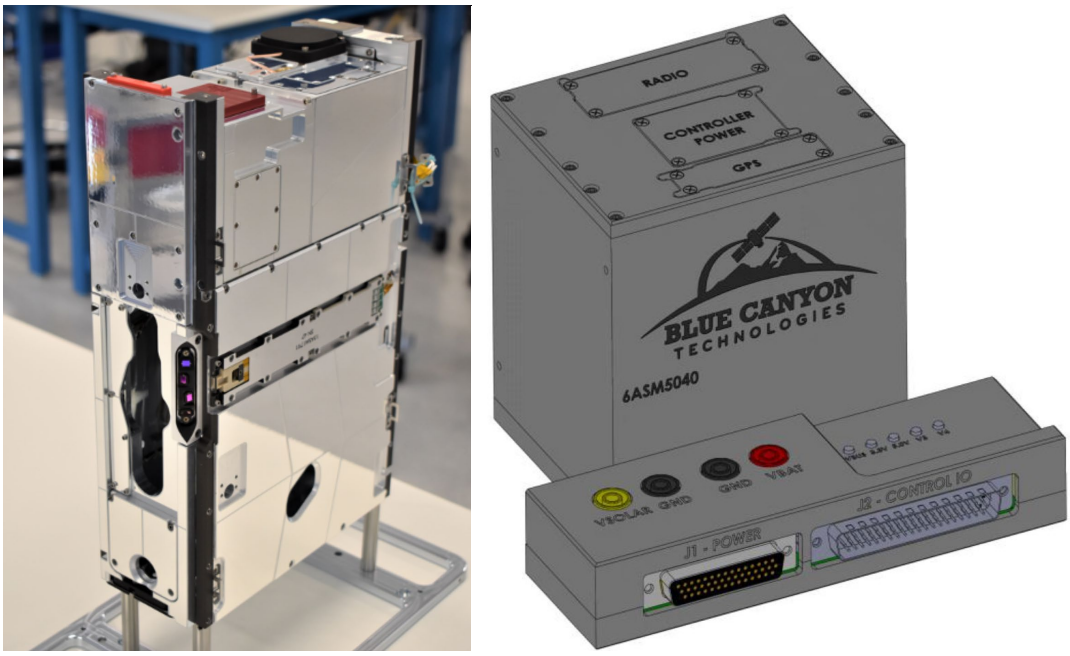

Figure 3. Left: XB1 spacecraft bus (DSC). Right: XB1 EDU. Photos courtesy of Blue Canyon Technologies (BCT).

In order for the XB1 EDU to simulate the spacecraft dynamics, there needs to be a simulated orbital environment to which the spacecraft can respond. A Realtime Dynamics Processor (RDP) is used for this purpose. The RDP simulates the attitude dynamics of the spacecraft, including the reaction wheel speeds and spacecraft pointing profiles, and it interfaces with the unit under test (UUT). In this case, the UUT is the XB1 EDU. The RDP has an ethernet port to communicate its commands and telemetry with a test computer, as well as a USB port through which command and telemetry data is passed between the UUT and the test computer. It also has a +12 V power input port which is solely used to power the RDP in the setup for VISORS. The XB1 EDU is powered using a separate power supply [4]. This setup is shown in Figure 4.

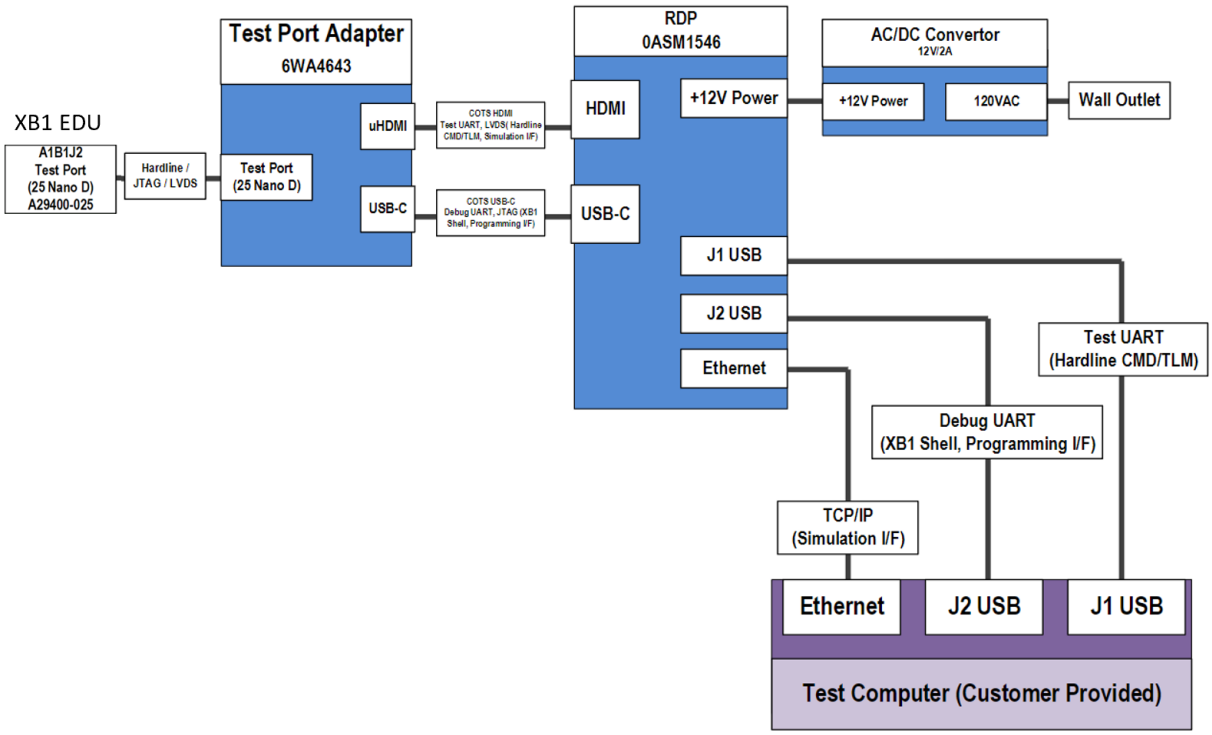

Figure 4. Testbed setup [5].

Through the test computer, the user can interface with the RDP and the EDU simultaneously using the open-source COSMOS software [6]. This software is a command and telemetry system that has a graphical user interface through which the user is able to power on and off the RDP and the UUT, send individual commands and modify parameters, run scripts, and plot and log live telemetry during simulations. At the point at which this work was completed (December 2022), no other subsystem EDUs had been delivered. For this reason, simulations in this work focus on GNC, ADCS, FSW, and ConOps verification.

A. Basic RDP Script Writing and Execution

The RDP and the UUT both use COSMOS as their command and telemetry interface with the test computer. The Ruby programming language is used to write COSMOS executable scripts as well as COSMOS applications and libraries [6]. Upon initialization of the COSMOS application, the user sees the window shown in Figure 5.

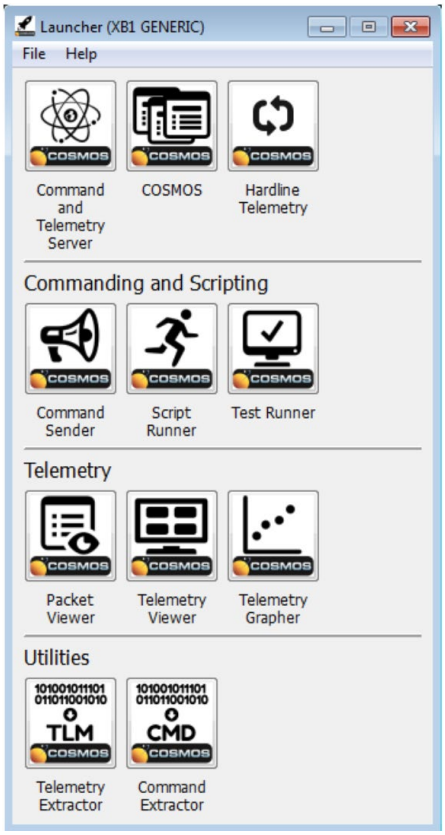

Figure 5. COSMOS launcher window.

For completeness, a brief overview of the various COSMOS functionalities is provided here, however, the Ball Aerospace documentation provides more detailed information regarding COSMOS functionality[3] [6,7]. The command sender icon opens a window through which the user can send individual commands to the UUT and to the RDP. The packet viewer allows the user to view telemetry values during a simulation which runs 1-to-1 with real time and allows telemetry to be viewed by packet. The telemetry grapher allows for plotting of real time telemetry. The telemetry extractor allows the user to extract and save simulation telemetry log files for data post-processing following a simulation.

The script runner allows the user to run test scripts and has a feature which highlights the current line being executed in green so the user can follow the execution of the script step-by-step. The user may also choose to only run certain lines of the script or start from any point in the script. The user may use this interface to develop the test script, however, it is recommended to use VS Code for the development of Ruby scripts for COSMOS execution; although COSMOS commands cannot be executed in the VS Code terminal and can only be run through COSMOS. The script runner and VS Code were both relied on heavily for this work in order to develop and execute test scripts which simulated GNC and ADCS performance during nominal operations.

As part of this work, two setup scripts were written to allow for ease of testbed initialization. The first script, called calibrate_LVDS_delay.rb, determines the low voltage differential signaling (LVDS) signal delay through the micro-HDMI cable following the procedure outlined in the BCT-provided RDP User's Guide. This script can be executed through COSMOS on the test computer using the script runner once after powering on the hardware. Once this script has been run, successful calibration can be confirmed through viewing the PRBS_SUCCESS telemetry point in the RDP (target) GENERAL packet.

The second setup script follows the steps outlined in the BCT-provided RDP User's Guide to initialize an RDP simulation. The first step in this script initiates the RDP simulation by sending the RUN_SIM command to the RDP. Next, the spacecraft attitude and simulation universal time is initiated to the same value as the user defined pointing and time. The orbit is then initialized either with position and velocity vectors or orbital elements. The simulated time

[3] For GT SSDL students working on VISORS, the BUS-EC-010-A_COSMOS_User_Guide.pdf document provides information related to BCTs implementation of COSMOS.

as well as the spacecraft position and velocity are copied to the UUT, and the simulated UUT spacecraft components are configured into test mode. Finally, simulated sensor biases are calibrated through a simulation consisting of 100 samples of the unit's physical readings. These steps are all executed sequentially within the RDP setup script (with no user input required) to initialize an orbit simulation for a single spacecraft. To confirm that critical steps are completed successfully, multiple wait_check methods are used to validate orbit initialization parameters and confirm that the script is running. These validations can also be conducted manually by viewing the telemetry points using the packet viewer.

B. Viewing Telemetry

The RDP and UUT telemetry are separate. The RDP and the UUT are defined as "targets," and in order to send a command or view telemetry, the user must select the "target." Once the target has been selected, only telemetry from that target can be viewed unless the target is changed. The telemetry for each target is divided into packets. When extracting data and exporting it into an Excel file for analysis, it is recommended to extract telemetry from only one target and only one packet to each Excel file. If telemetry from multiple packets is extracted and exported into the same Excel file, it will create a difficult-to-navigate file due to the way in which it separates the telemetry by packet and target. Another useful way of viewing telemetry is to view it live using the telemetry grapher application. This allows the user to view the telemetry in real time and is more efficient than having to run a full simulation, go through the telemetry extraction process afterwards, and then create plots manually to verify a test script.

V. Science Orbit Attitude Simulation

The purpose of the Science orbit attitude simulation is for Georgia Tech to verify the BCT-provided software, verify the pointing profiles are as expected using the BCT COSMOS commands, and monitor the reaction wheel speeds for any potential reaction wheel saturation concerns. The BCT-provided software has been previously tested by BCT before delivery, but as the mission systems engineers and future VISORS mission operators, it is prudent that the Georgia Tech team verify the functionality of the software and command and telemetry interface before and after integration. Beyond this verification, becoming well-versed in the command and telemetry interface and usage is also important.

Starting from a high level, this simulation is intended to simulate the pointing profile of the spacecraft over an entire Science Campaign. A Science Campaign is defined as the 10 orbits that occur in Science Mode during which, once per orbit, the spacecraft aligns properly to record images.

A. Script Architecture & Computations

For the DSC, the primary observation pointing constraint vector is that the detector must point to the Sun. The secondary observation constraint vector is that the GPS antenna must point to Zenith. This requirement enables the detector to stay aligned with the Sun during a Science observation while also maintaining the proper relative positioning and alignment with the OSC. When the DSC is in Science Mode but not actively taking an observation, the primary pointing constraint vector is that the GPS antenna must point to zenith, and the secondary pointing constraint vector is that the solar panel normal vector must point to the Sun. During the time in between observations, the relative positioning knowledge is most important, and the second priority is charging the batteries. The spacecraft pointing profiles over the course of a Science Campaign are shown in Figure 6.

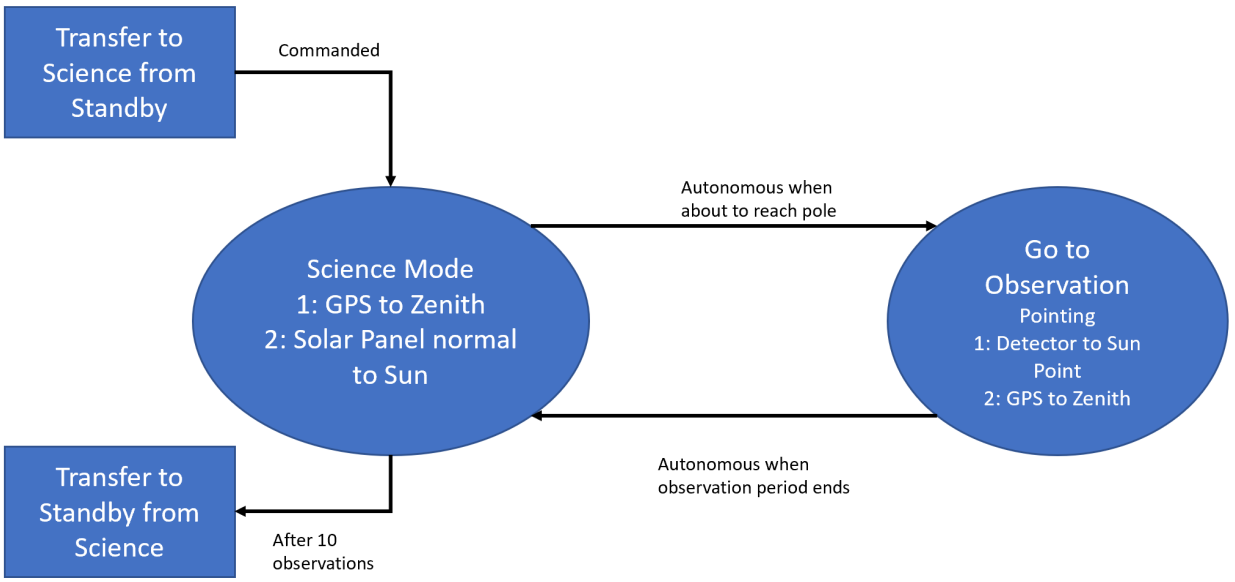

Figure 6. State flow of a baseline Science Campaign.

The orbits chosen for the OSC and DSC are described based on their beta angle, orbit altitude, true anomaly, eccentricity, argument of periapsis, and the time since 01/01/2000 (J2000). The most straightforward BCT defined COSMOS command to initialize the simulated orbit accepts the classical orbital elements as the inputs. This convention requires the initial portion of the script to take the user inputs and calculate the missing classical orbital elements: the RAAN and orbital plane inclination angle.

Calculating the inclination angle from the user input parameters utilizes an important characteristic of the orbits defined for the VISORS mission, namely, that they are Sun-synchronous orbits (SSOs). This property means that the orbital plane remains fixed relative to the Sun regardless of the position of the Earth in its orbit around the Sun. As a result, it can be deduced that the RAAN drifts exactly 360 degrees over the course of one Earth orbital period, or one year. This relationship is shown in equation (1):

$$\frac{\Delta\Omega}{T} = \frac{360°}{365.25\ days} \tag{1}$$

Using this rate as defined for an SSO and comparing it to the derived J2 RAAN precession rate [8], which is a function of the Earth's radius, the user defined orbit altitude, and the inclination of the orbital plane, an equation can be derived to express the inclination as a function of the altitude and the SSO RAAN precession rate as shown in equation (2):

$$i = cos^{-1}\left(\frac{-2a^{\frac{7}{2}}}{3J_2R_E^2\sqrt{\mu}}\frac{\Delta\Omega}{T}\right) \tag{2}$$

The second missing orbital parameter is the RAAN, which is a function of the orbit inclination, beta angle, declination of the Sun, and right ascension of the Sun [9] as shown in equation (3):

$$\Omega = sin^{-1}\left(\frac{sin\,\beta\ - sin\,\delta_s\,cos\,i}{cos\,\delta_s\,sin\,i}\right) + \Omega_s$$
$$\beta = beta\ angle, i = inclination, \tag{3}$$
$$\delta_s = declination\ of\ the\ sun,\ \Omega_s = right\ ascension\ of\ the\ sun$$

The declination of the Sun and the right ascension of the Sun are found as functions of the days past vernal equinox (taken here as March 20th regardless of the year for simplicity and assuming Earth's orbit as perfectly circular) as shown in equation (4):

$$\Omega_s = \frac{d}{365.25} 2\pi$$

$$\delta_s = -23.45\, sin(\frac{d}{365.25}) \frac{\pi}{180} \tag{4}$$

$$d = days\ past\ vernal\ equinox$$

This process of going from the user defined inputs to the accepted command inputs is visualized in Figure 7, starting from the user inputs on the left and ending with the desired Keplerian elements on the right.

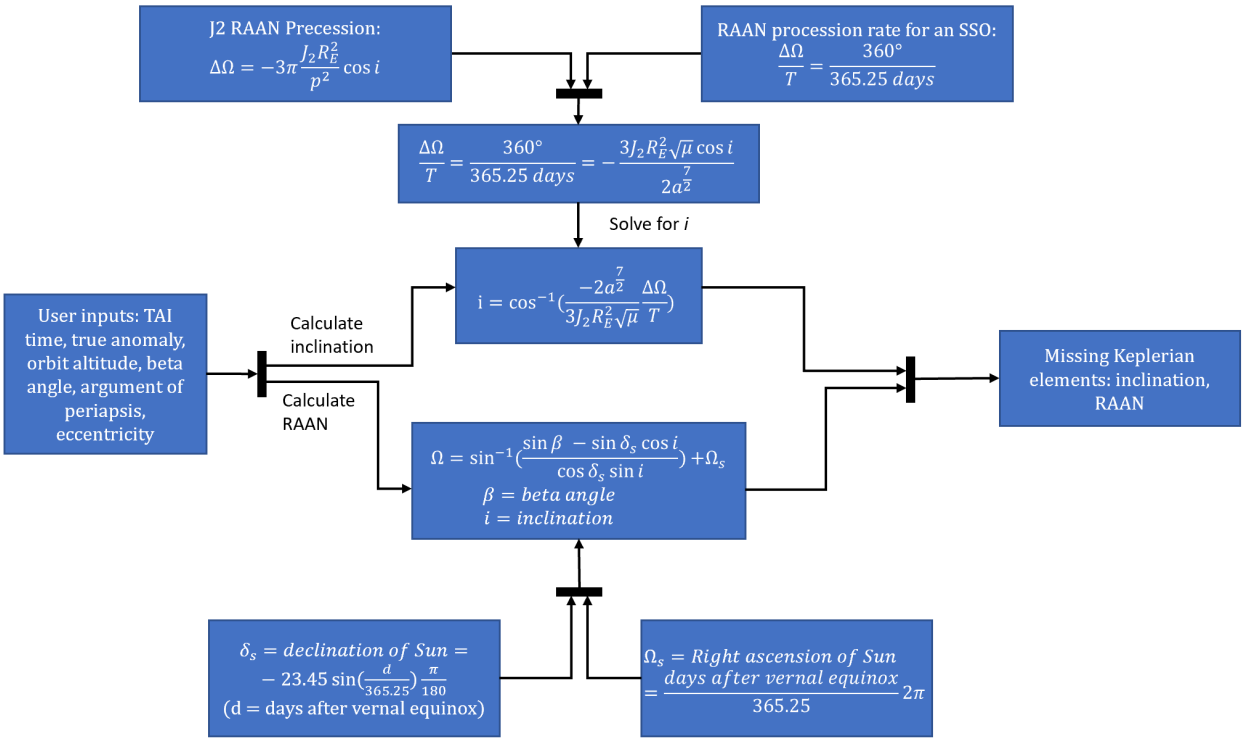

Figure 7. Flow diagram showing how user input parameters are converted to orbit initialization command accepted parameters.

Once the missing elements have been calculated, the aforementioned RDP simulation startup process is executed to initialize the orbit and the spacecraft pointing profile, as well as initialize or calibrate the simulated sensors and modeled mechanical components. At this point, the script enters the primary logic for loop, which will simulate the pointing profiles of the spacecraft over the course of 10 orbits in Science Mode, which is equivalent to a full Science Campaign. The primary command of importance in this simulation is the GOTO_TARGET command. This command is structured such that the user can input a primary and secondary reference direction and a primary and secondary command direction. The command works such that the primary command direction in the body frame is aligned with the primary reference direction as the primary pointing constraint, and the secondary command direction in the body frame is aligned with the secondary reference direction as the secondary pointing constraint. This approach means that the primary pointing constraint should always be met given that it is physically possible, and the angle between the secondary command direction in the body frame and the secondary reference direction will always be minimized. The relevant body frame direction vectors are defined in Table 1.

Table 1. Relevant spacecraft body frame directions.

Body Vector	Corresponding Spacecraft Direction
$[0, \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}]$	Solar panel normal
$[1,0,0]$	GPS Antenna Boresight
$[0,0,-1]$.	Detector Boresight

The first pointing commands that are given are the pointing vectors corresponding to the defined Science Mode (as defined previously and shown in Figure 6). After a couple of minutes, the spacecraft settles into the expected Science pointing profile. Logically, the next series of steps that needs to be executed is for the script to wait until the correct time to slew to observation pointing, and then once that time is reached, the script must command the spacecraft to reorient into observation pointing. The way this sequence is implemented is by using the COSMOS wait_check_expression command which works by continually checking the expression contained in the method argument until it is true, and only when it returns true does the script move to the next line. The logical expression that was implemented for this wait_check_expression was chosen such that the expression would only return true between 120 and 119 seconds before the observation window begins. The beginning of the observation window is approximately at a true anomaly of 90 degrees, or when the spacecraft is over the pole (either the North or South pole). Once this returns true, the next command is a GOTO_TARGET command which slews to the observation pointing direction (inertially pointing during the 10 second observation window). After this command, the next expression is another wait_check_expression which waits for 130 seconds (the 120 seconds before the observation from the previous wait time plus 10 seconds for the observation to be complete) before it returns a true value. After this condition is true, the spacecraft returns to its Science pointing and this cycle repeats ten times. The overall script logic and architecture is visualized in Figure 8.

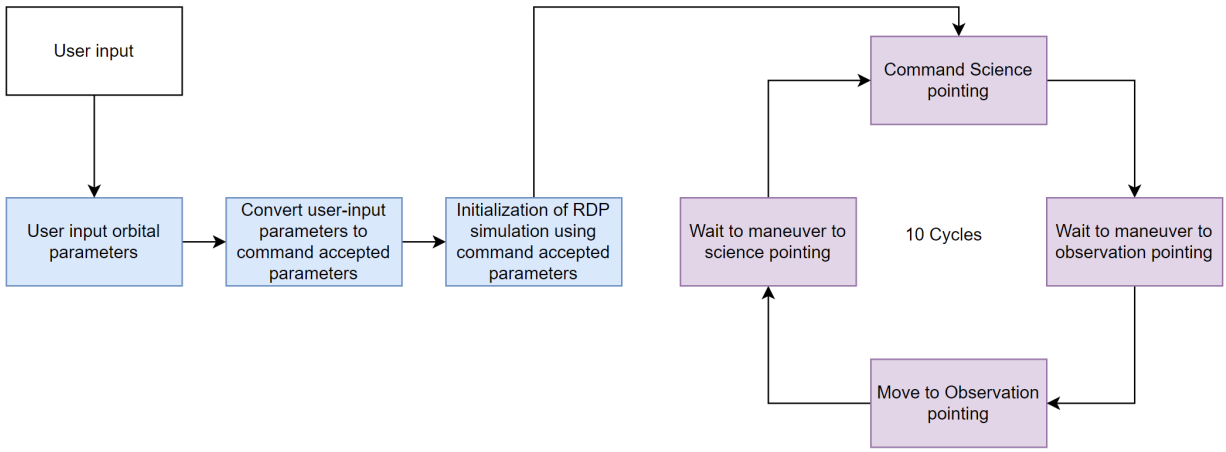

Figure 8. Logic and flow of Science Campaign simulation

B. Script Verification

To verify that the script is producing results that are consistent with the ConOps, specific telemetry points were chosen to plot and analyze over some period of time. Figure 9 shows the pointing profile over the course of approximately six minutes. This time interval includes approximately two minutes prior to the command to slew to observation pointing, two minutes and ten seconds of observation pointing, and two minutes after the command to slew back to Science pointing.

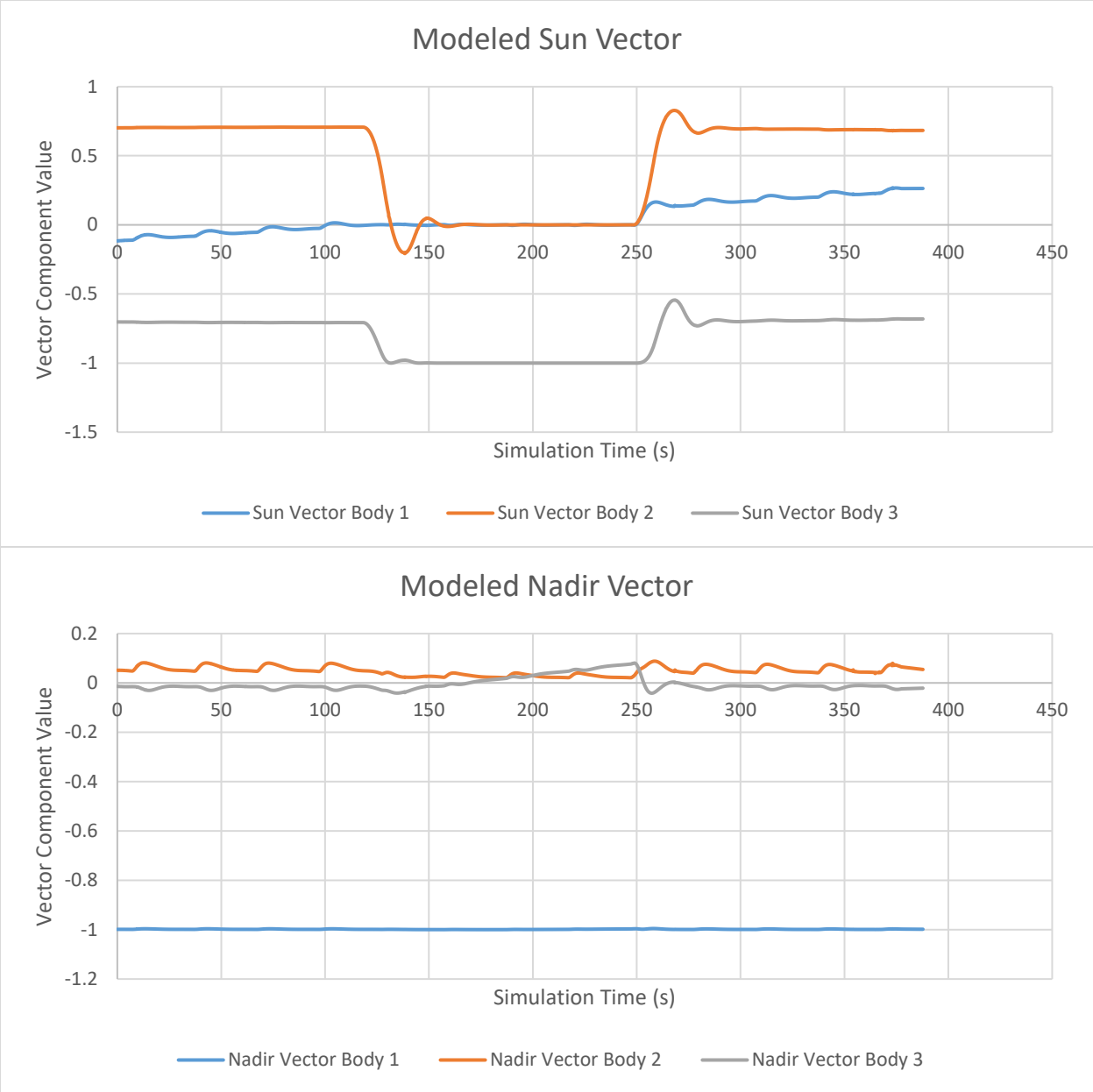

Figure 9. Pointing profile of DSC Science Orbit simulation.

As seen in the upper plot in Figure 9, there are two clear and abrupt changes in the spacecraft pointing, the first of which corresponds to the slew to observation pointing, and the second of which corresponds to the slew to Science pointing. The two minutes and ten seconds of pointing data in the middle of each plot corresponds to the time that the spacecraft should be in its observation pointing, and the two minutes outside of that on either side corresponds to when the spacecraft is in its Science pointing configuration. Examining the Science pointing portion of the top plot, the spacecraft body vector of approximately [0, 0.7, -0.7] is aligned with the Sun, with some clear oscillation particularly of the body 1 component of the vector. This result is exactly as expected, since the secondary pointing vector constraint for Science is the solar panel normal vector to the Sun, and the solar panel normal vector in the body frame is defined as $[0, \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}]$. Within the observation pointing timeframe, the body vector [0, 0, -1] is aligned with the Sun. This vector as defined in the spacecraft body frame corresponds to the detector boresight direction, which is consistent with

the primary pointing constraint during an observation. In this range, any fluctuations are very small compared to the fluctuations that are seen in the secondary Science pointing vector during the Science timeframe.

The second (bottom) plot in Figure 9 shows the body vector that is aligned with the nadir direction at any point in the simulation. This plot captures the same time window and shows that the body vector [-1, 0, 0] is approximately aligned with the nadir direction. The body vector [-1, 0, 0] is the direction opposite the defined GPS antenna direction. This result means that the GPS antenna is aligned with zenith, as required by the pointing constraints. Beyond general verification of the pointing profiles and FSW pointing commands, this script also shows preliminary verification of certain aspects of the ADCS system performance. Figure 10 shows the simulated reaction wheel rates and spacecraft body rates over the same time period as captured in Figure 9.

Figure 10. DSC Science Campaign slewing (simulated body rates and reaction wheel rates).

The top plot in Figure 10 shows the simulated reaction wheel speeds in units of RPM with enough time on the front and back end of the window to allow for settling. Some maneuvering happens after approximately two minutes,

and another maneuver occurs two minutes and ten seconds later. This plot also confirms the proper timing of the Science and observation pointing slews. The exact timing was confirmed through comparison of the raw data with the command log. The reaction wheel speed about the Z-axis in the body frame has the greatest excursion as shown in the plot, reaching slightly over 4000 RPM in opposite directions. This speed is significantly underneath the wheel saturation limit, indicating that the maneuvers between Science Mode pointing profiles during a Science Campaign should not cause reaction wheel saturation. Additional simulation and post processing work done adjacent to the work detailed here has also shown that the reaction wheels rates decrease over time even with a significantly non-zero (2000-3000 RPM) initial rate and commanded slews, and that they do not pass their saturation limit. The second (bottom) plot in Figure 10 shows the spacecraft body rates in units of degrees per second (DPS). This plot also has spikes that line up with the excursions seen in the reaction wheel speeds and the changing pointing profiles, as expected. Furthermore, the largest spike in body rate is seen in the body defined X direction. This result is also logical since the GPS antenna direction (the X direction) is always aligned with zenith, meaning that the only way the spacecraft is reorienting in any significant way is by rotating about that body vector. The results from these plots confirm that the Science Mode slewing does not present a concern for reaction wheel saturation, that the commanded slews produce the expected responses from the ADCS system, and that the performance of the system is acceptable since the slews settle relatively quickly compared to the available time windows.

VI. Science Mode Full Control Simulation

While simulating only the ADCS during a Science Campaign does provide useful information and allows for some high-level verification of the ConOps and ADCS performance, there is significantly more that remains to be verified which requires a longer, more complex, and higher fidelity simulation. This section describes work performed to incorporate propulsive maneuvers that will be executed by the VISORS cold gas propulsion systems into the simulation of Science operations and dynamics to investigate momentum accumulation.

A. Delta V to External Torque

Similar to the Science Campaign simulation, the parameters that are accepted by the BCT-provided maneuvering command do not align with the parameters that are input to the simulation. The maneuvers are provided as a comma separated file (CSV) list of time-stamped delta-v maneuvers in the radial-tangential-normal (RTN) frame to be executed during a full Transfer-Science-Transfer sequence, but the maneuvering command requires external torques in the body frame. As a result, each delta-v maneuver must be converted to external torques. In this case, the problem with the orbital parameters is still present, but the work described previously to address this matter can be reused. The process for converting each delta-v maneuver in the RTN frame to an external torque in the body frame can be visualized in Figure 11. The flow starts from the user input in red and terminates with the desired command input parameter in green. The flow diagram also shows what on-board telemetry is utilized in each step.

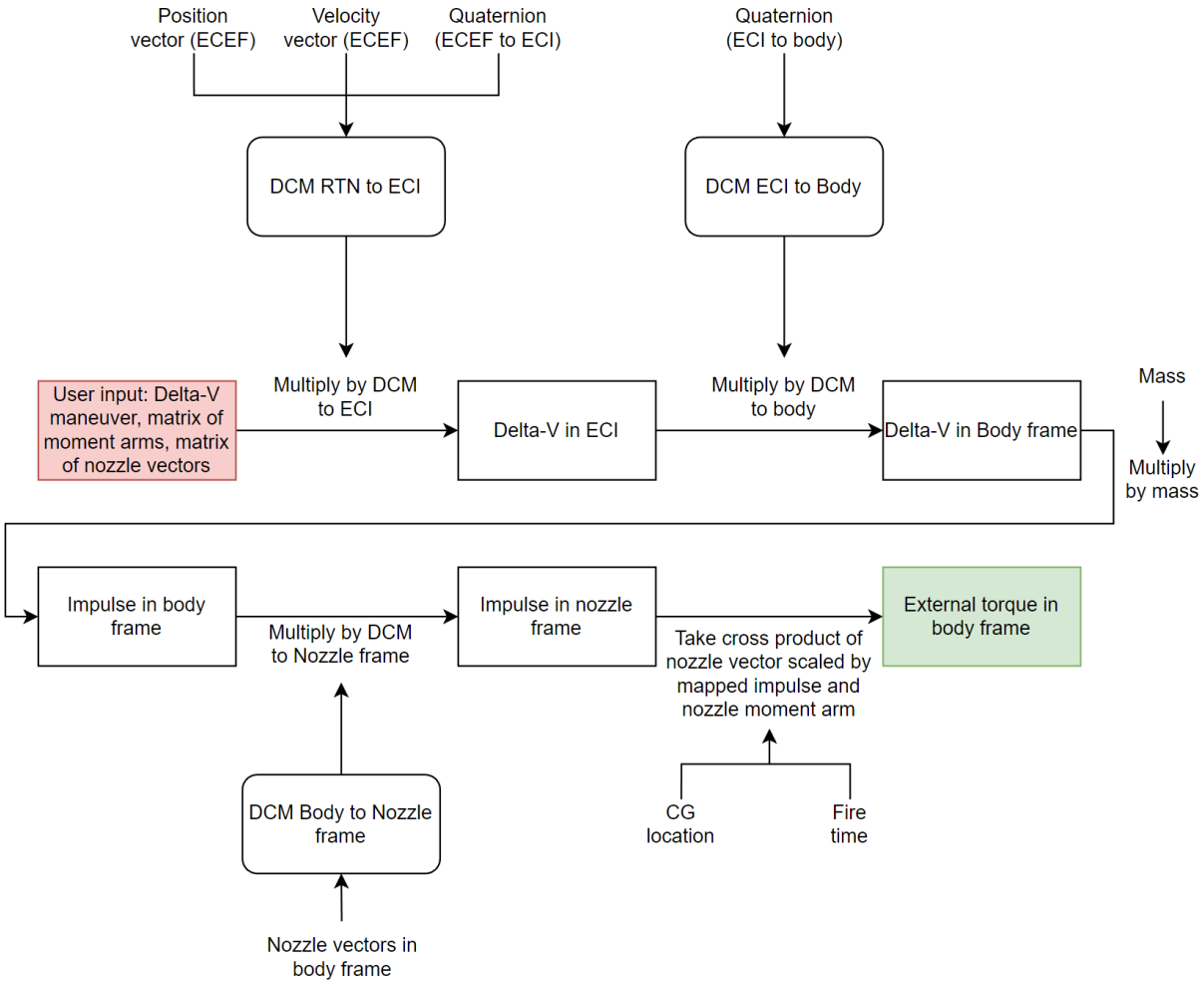

Figure 11. Algorithm for converting Delta-v inputs to external torques in the spacecraft body frame.

The first step in converting the delta-v to an external torque is to transform the maneuver from the RTN frame to the earth-centered inertial (ECI) frame using a direction cosine matrix (DCM). There is no DCM telemetry point produced by the system, so this matrix must be constructed using the measured position and velocity vectors in the Earth-centered-Earth-fixed (ECEF) non inertial reference frame as well as the quaternion that rotates from the ECEF frame to the ECI frame. To determine these values, the position and velocity vector must first be transformed from ECEF to ECI using the quaternion ECEF to ECI telemetry point. Next, the columns of the RTN to ECI DCM must be constructed. The position vector (now in the ECI frame) is normalized and used as the first column of the DCM. The normalized cross product of the ECI position and velocity vectors (now in the ECI frame) produces the third column of the DCM, and the cross product of the third and first columns of the DCM, respectively, form the second column of the DCM. This process for construction of the DCM is shown in equations 5-8:

$$\hat{\boldsymbol{x}} = \frac{\boldsymbol{r}}{||\boldsymbol{r}||} \tag{5}$$

$$\hat{\boldsymbol{z}} = \frac{\boldsymbol{r} \times \boldsymbol{v}}{||\boldsymbol{r} \times \boldsymbol{v}||} \tag{6}$$

$$\hat{\boldsymbol{y}} = \hat{\boldsymbol{z}} \times \hat{\boldsymbol{x}} \tag{7}$$

$$DCM = [\hat{x};\ \hat{y};\ \hat{z}] \tag{8}$$

The delta-v vector can now be transformed to the ECI frame by multiplying by this DCM. Once the delta-v vector is in the ECI frame, it must be transformed to the body frame to properly map impulses to the individual nozzles as they are defined in the body frame. This method requires multiplying by a DCM or quaternion that transforms from the ECI frame to the body frame. A quaternion which maps from ECI to the body frame is produced as live telemetry on-board, so this quaternion is used to make this frame transformation.

The next step in this process is to multiply the delta-v vector in the body frame by the spacecraft mass to obtain the impulse in the body frame and then transform the impulse in the body frame to the impulse in the nozzle frame. The purpose of the latter is to map the impulse to each particular nozzle. In this case, the nozzles form a proper orthogonal set of axes to which the impulse can easily be transformed by creating a DCM using the nozzle vectors as defined in the body frame. The final step of this process is going from the impulse in the nozzle frame to the external torque in the body frame. This can be done by taking the cross product of the nozzle vector scaled by its mapped impulses with the corresponding nozzle moment arm and summing them to produce the net external torque on the system. This process is visualized in Figure 11.

B. Script Architecture

The first version of the script involving propulsive maneuvers was used as a diagnostic for debugging the delta-v to external torque conversation script. This version simply replaced the primary logical loop responsible for changing between Science and observation pointing with a loop that simulated the propulsive maneuvers as the simulation reached their timestamps. In other words, this script only simulated propulsive maneuvers during Science operations instead of simulating pointing slews. The architecture is shown in Figure 12.

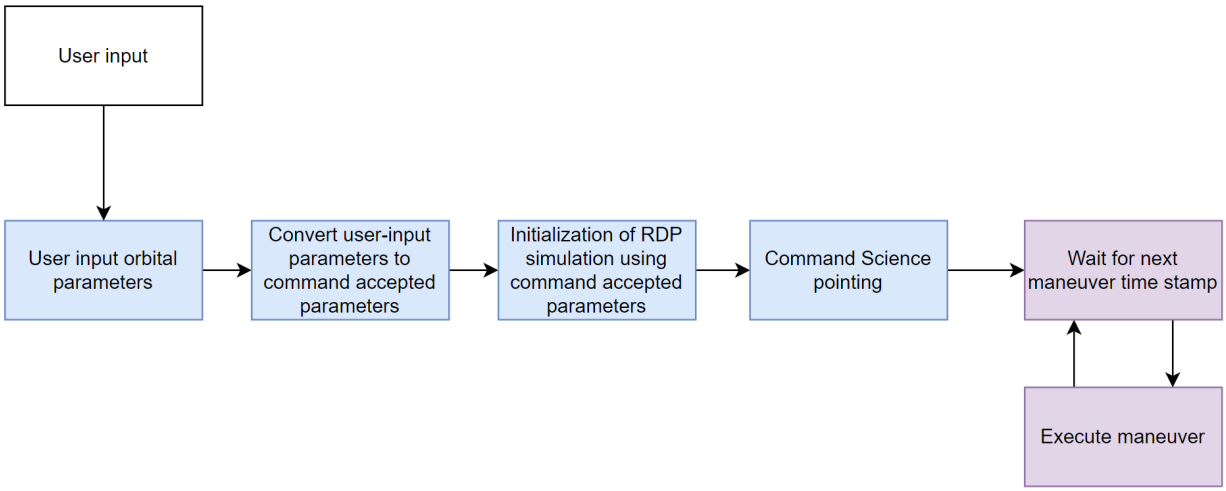

Figure 12. Architecture of Science Campaign simulation script with only propulsive maneuvers.

This intermediary script successfully facilitated debugging the implementation of the propulsive maneuvers. It allowed for real time telemetry monitoring of maneuver timing, magnitude, and system response. Once the implementation of the propulsive maneuvers was verified, the architecture of the script to simulate a single Science Campaign with propulsive maneuvers and pointing profiles was designed. The architecture of this script had the added challenge of having to constantly check for both reaching maneuver timestamps *and* for the indicated times to switch pointing profiles. This was accomplished using two while loops (see Figure 13 for logic flow diagram). One of the while loops monitored for propulsive maneuvers or a slew during the Science pointing timeframe using a single wait_check_expression with an OR logic that would return true when either a maneuver timestamp *or* the end of the Science pointing timeframe was reached. The second while loop monitored for propulsive maneuvers or a slew during the observation pointing timeframe using a single wait_check_expression with an OR logic that would return true when either a maneuver timestamp *or* the end of the observation pointing timeframe was reached. This cycle should

be repeated ten times to simulate a full Science Campaign. Figure 13 visualizes the recommended architecture for this simulation script. The beginning point is the blue square where the RDP simulation is initialized as described previously.

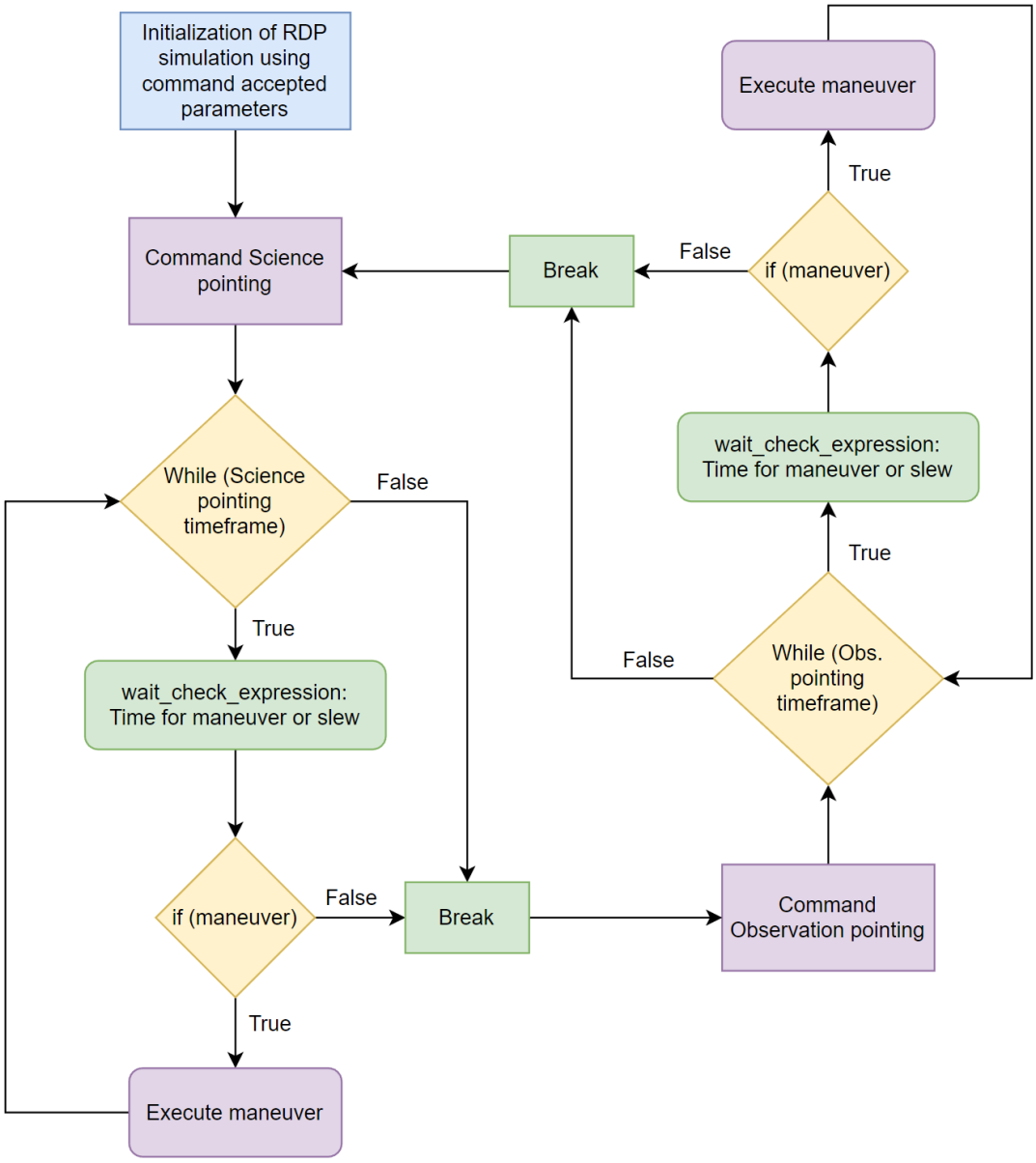

Figure 13. Full control Science Campaign script architecture.

Using this figure as a reference, this simulation architecture can be described in slightly more detail. The while loops will use the simulation time as the primary telemetry point to indicate when the current pointing timeframe is no longer valid, at which point the loop will break and transition to the other while loop. Within the while loops, the wait_check_expression methods will also use the simulation time telemetry point to return true only when either a slew or a maneuver is required. When the wait_check_expression returns true, it is known that it is either time to execute a maneuver or a slew, but it is not known which. In order to determine the next action, an if statement can be used to check if it is time for a maneuver or a slew. If the logic determines that a maneuver triggered the wait_check_expression to return true, then the corresponding maneuver is executed, and the while loop continues. If

a slew triggered the return true, then the script should break from the while loop, command the slew, and switch to the other while loop.

In this script architecture it is also necessary to keep track of the next maneuver timestamp. This value is compared to the simulation time to determine when a maneuver should be commanded. At the time that this work was completed (December 2022), this script had not yet been written, but the architecture presented here is one potential way of simulating the full control and dynamics of a single spacecraft during a Science Campaign. This script will allow for a higher fidelity model of the ADCS performance and will help verify the GNC algorithm.

C. Script Verification

At the time of this paper, only a portion of the required verification of these simulations has been completed. The full script of propulsive maneuvers (whose suggested architecture is detailed in Figure 13) has not been verified in any way since it has not been written, but many of the primary functions used in this script have been written and verified. The most important of these functions converts the delta-v maneuvers in the RTN frame to external torques in the body frame. This function, in order to facilitate the verification process, was broken down such that most of the steps shown in Figure 11 call another function to execute that step. By designing the script this way, this allowed for script verification through the use of unit tests designed to thoroughly test each of the interior functions. The Ruby programming language has a useful unit testing framework built in which was utilized to test the functions. To utilize this framework for Ruby function unit testing, the "test/unit" framework must be included in the script, along with any functions that are being tested.

A simple example to demonstrate this process is the set of unit tests for the function that takes the date provided (year, month, day) and returns the number of days that have passed since vernal equinox, which is defined here as March 20th at midnight. For simplification, March 20th is used regardless of the year, although it can also occur on March 21st some years depending on the exact timing of when the plane of Earth's equator passes through the center of the Sun. The unit tests for this function are relatively simple since there is only one output value which needs to be verified. Figure 14 shows three of the unit tests which were written for this function. There are a total of four unit tests for this function, each written to test a slightly different case based on how the function was written. One important aspect of the function that is verified by the unit tests shown is the functionality that implements leap years. In the case of the integer output, the method which is used to verify the output of the function using the built in Ruby unit testing framework is the assert_equal(value_1, value_2) method. This method will compare the exact values and will pass if the values are equal. The values against which the function output is being compared are hand calculated values. In this case, assert_equal works well since the numbers are integers, but for other cases, a different unit-test method must be used.

```
134     def test_days_past_vernal_equinox
135         #first test case
136         #user input date since 1/1/2000
137         year = 24
138         mon = 3
139         day = 25
140         compare = 5
141         assert_equal(compare,days_past_vernal_equinox(year,mon,day))
142 
143         #second test case
144         year = 24
145         mon = 5
146         day = 25
147         compare = 66
148         assert_equal(compare,days_past_vernal_equinox(year,mon,day))
149 
150         #third test case (can change this between leap year to see the extra
151         #day is in fact counted, to do this change year to 24 and compare value
152         # to 365 and test will come back successful)
153         year = 25
154         mon = 3
155         day = 19
156         compare = 364
157         assert_equal(compare,days_past_vernal_equinox(year,mon,day))
```

Figure 14. Unit tests for days_past_vernal_equinox function.

Another example to illustrate the unit testing verification process is the unit test for the function which converts a quaternion to a DCM. In this case, the function takes a quaternion input (a 4 element Ruby Vector object) and returns a DCM (a 3x3 Ruby Matrix object). For this unit test, each output was tested as compared to the expected value. In other words, this method required nine individual comparisons instead of just one. Additionally, a different method had to be used in place of the assert_equal method to account for floating point errors which would force the assert_equal to fail since the values in this case are not integers. The assert_in_delta(value_1, value_2, delta) method was used to account for this. The first two arguments of this method, value_1 and value_2, are the two values being compared. The third argument, delta, is the maximum allowable difference between the two values for the test to still pass (usually set as 0.000001).

In total, there were six functions which were verified through unit tests. These functions, their purpose, and the test cases are shown in Table 2 (this is *not* a comprehensive list of every test case run). The implementations of these unit tests can be reviewed in detail with the code provided in Appendix A.

Table 2. Function unit test cases.

Function	input	output	Test input	Expected out	Pass?
quat2dcm	quaternion	DCM	[.707, 0, .707, 0]	[[0, 0, 1] [0, 1, 0] [-1, 0, 0]]	Y
			[.707, .707, 0, 0]	[[1, 0, 0] [0, 0, -1] [0, 1, 0]]	Y
conj	quaternion	Conjugate rotation quaternion	[.707, .707, 0, 0]	[.707, -.707, 0, 0]	Y
			[-.577, .577, -.577, 0]	[-.577, -.577, .577, 0]	Y
dcmrtn2eci	position vector (ECEF), velocity vector (ECEF), quaternion (ECI to ECEF)	DCM RTN to ECI	R = [0, 0, 1] V = [1, 0, 0] q = [1, 0, 0, 0]	[[0, 1, 0] [0, 0, 1] [1, 0, 0]]	Y
			R = [1, 0, 0] V = [0, -1, 0] q = [1, 0 , 0, 0]	[[1, 0, 0] [0, -1, 0] [0, 0, -1]]	Y
quatrot	direction vector, quaterion	direction vector rotated by quaternion	V = [1, 1, 1] q = [-.707, 0, .707, 0]	[1, 1, -1]	Y
			V = [1, 1, 1] q = [.707, 0, .707, 0]	[-1, 1, 1]	Y
full_impulse_vector	1x3 impulse vector in nozzle frame	1x6 impulse mapped to nozzle	[-4, 2, -6]	[0, 2, 0, 4, 0, 6]	Y
			[0, -2, 6]	[0, 0, 6, 0, 2, 0]	Y
days_past_vernal_equinox	year, month, day (since 1/1/2000)	number of days past VE	yr = 24, mon = 3, day = 25	5	Y
			yr = 24, mon = 5, day = 25	66	Y
			yr = 25, mon = 3, day = 19	364	Y
			yr = 25, mon = 1, day = 19	305	Y

During implementation of the test cases shown in Table 2, bugs were discovered and corrected within the functions. This verification process has provided a baseline level of confidence that each piece of the delta-v to external torque algorithm produces the expected results and can handle potential edge cases. In addition, the mathematics and reasoning behind these functions were peer-reviewed in detail among three different systems engineering team members and compared with portions of the ADCS team's process. During these reviews, the high-level process used to make this conversion was verified. Finally, the delta-v limiting cases (reaction wheel saturation limit) were tested in the script to ensure that inputting the limiting delta-vs returned a torque value that was less than or equal to the saturation limit. The results of this test were also as expected.

While care was taken to choose unit tests which represented a range of cases, there are plenty of limitations with the verification work done thus far. First, not all edge cases have been tested, meaning that it is possible that singularities exist which could produce undefined values and lead to off-nominal performance or failure of the ADCS. Furthermore, a few test cases are not enough to verify any of these functions in 100% of cases. In addition, hand calculations which step through the entirety of the algorithm should be done and compared to the results produced by the algorithm (including comparing each result of the interior functions throughout the execution of the algorithm as a whole). While the verification work done thus far does provide a baseline of confidence in the algorithm, the level of confidence is limited by the lack of complete sets of test cases, full hand calculation comparisons, and thorough, well-documented peer reviews.

D. Future Work

A significant amount of work remains to be completed for creating a high-fidelity simulation of the full mission Science operations. The next step in this process is to conduct a higher fidelity verification of the delta-v conversion script. This approach would consist of converting a number of delta-v maneuvers in the RTN frame to external torques in the body frame step-by-step by hand and then comparing the final external torques to what the script outputs. This

test should be done for a number of cases including any potential edge cases in order to verify that there are no weak points in the script that could lead to unexpected results or singularities.

The next step is to implement the architecture described in Figure 13 in a script. Then, this script should be run for a maneuver plan CSV file that runs from Transfer Mode to Science Mode and back to Transfer Mode, simulating an entire Science Campaign. It is critical that this simulation is run and the results are analyzed in order to provide confidence in the ADCS system performance throughout this entire timeframe. This is particularly important in Transfer Mode where there are larger magnitude maneuvers whose effects on the reaction wheels have not yet been examined.

Since navigation data from one spacecraft is required for successful performance of the formation-keeping GNC algorithms of the other spacecraft, the data from this simulation can be used to simulate and verify formation level GNC algorithm performance. The data can be used to verify the performance of the formation-keeping GNC algorithms by allowing for simulated orbit data from one spacecraft to be transmitted over a simulated ISL to another spacecraft in-the-loop. This test setup design is captured in Figure 15 where FKA is the formation-keeping algorithm and NavSen is the navigation sensor. The yellow box labelled "playback from rest of formation" will play back the pre-recorded data from one spacecraft's Transfer-Science-Transfer simulation over a simulated ISL to the other spacecraft for input to its GNC FKA in order to simulate the dynamics of the formation. There are shortcomings in this design, most obviously being that the pre-recorded data is not real-time, meaning that this simulation cannot simulate the response of the spacecraft providing the pre-recorded data due to the open loop nature of the design. While this is a pitfall of the design, the "real-time" responses of the spacecraft under test should generally provide confidence in the performance of the GNC FKAs.

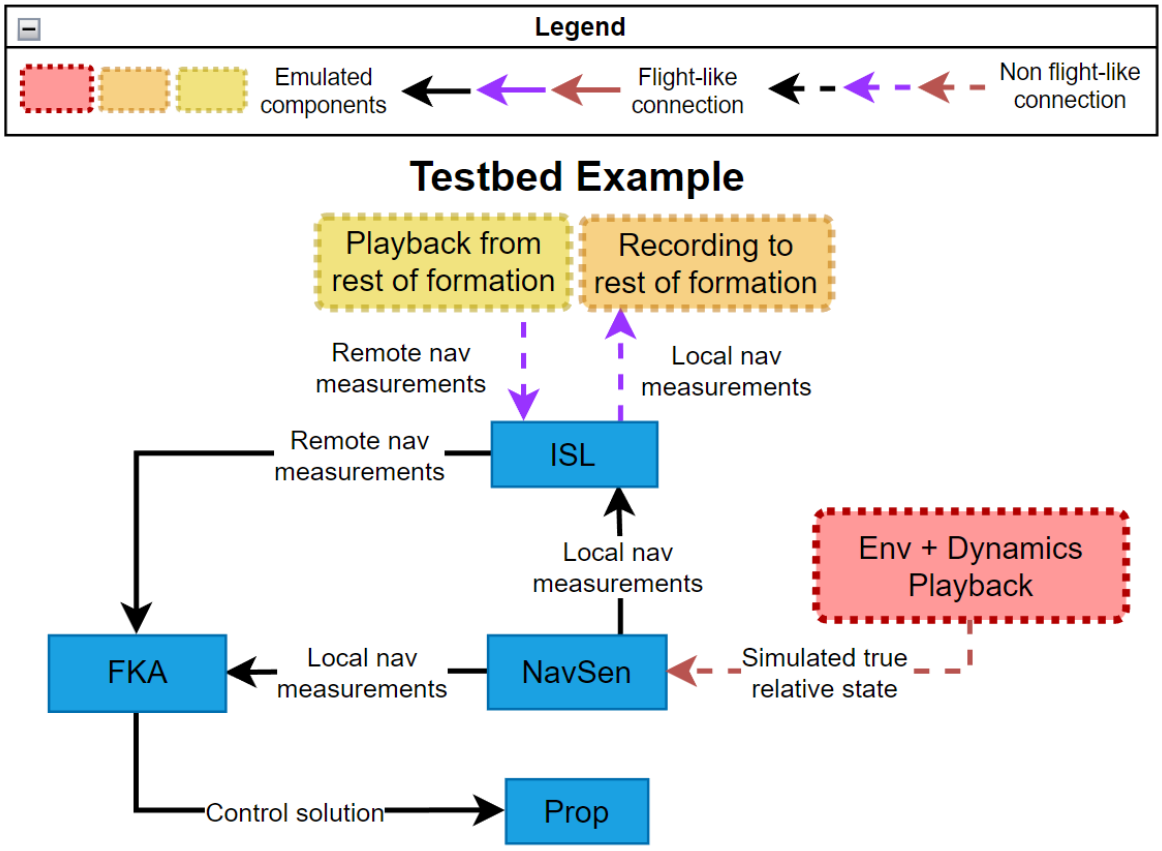

Figure 15. Example of a formation-flying testbed that uses prerecorded orbit simulation data to reconstruct the formation level interaction loop and dynamics [10].

These next steps in this work will be important in verifying subsystem performance, verifying on-board software and algorithm execution, evaluating any potentially risky situations that the spacecraft could encounter on orbit, and developing strategies to mitigate any such situations.

VII. Conclusion

This paper summarizes the current state of the verification of the VISORS ConOps, spacecraft ADCS performance, GNC, and FSW. A script currently exists which fully simulates the pointing profiles of the spacecraft during a full Science Campaign. This script has been partially verified and the results are as expected, with reaction wheel momentum staying well below the saturation limit and settling times within acceptable limits. Another script was created that takes the delta-v maneuver inputs that are produced by the GNC algorithm and converts them to external torques. This script was broken into a handful of smaller functions to facilitate unit testing of each sub function. The algorithm behind this script has also undergone preliminary peer review. A high-level architecture suggestion for a script that simulates the Science Campaign and includes the pointing profiles as well as the maneuvers has also been developed for future implementation.

Work remains to be done in verifying the performance of these scripts, as well as further development of a high-fidelity Transfer-Science-Transfer script as described in the Future Work section previously. The work that has been done so far exists on the VISORS project GitHub (Georgia Tech research GitHub) in the ConOps repository. The repository also has text file documentation describing the orbit simulation scripts, and each script is also well commented.

Appendix A

Delta-v to external torque script:

```
def ThrustSolver(maneuver,q_bi,q_fi,recef,vecef,m,cg)
# Solves for Thrust based on attitude, position, and maneuver plan.

# Inputs:
# maneuver: dV command in mm/s in RTN frame, 1D vector object
# quaternions (scalar, [v1, v2, v3])
# q_bi: Attitude Quaternion, spacecraft body frame with respect to ECI
# (J2000) frame, Quaternion object, 1D vector object
# q_fi: Quaternion of ECEF with respect to ECI, 1D Vector object
# R_ECEF: Orbit Position, reported in the ECEF frame, 1D Vector object
# V_ECEF: Orbit Velocity, reported in the ECEF frame, 1D Vector object
# m: mass in kg [Can be replaced with constant]
# CG: Spacecraft center of gravity in mm. [Can be replaced with constant], Vector
object
# [CHANGE CG FRAME BEFORE SENDING]

# Output:
# Torque as 1D 3-element vector in Nm

# Constants
# m = 10.1620; % Vehicle mass (kg)
# CG = [0 0 0]; % Vehicle center of gravity (mm) [PLACEHOLDER VALUE]

#import necessary classes and methods
#requrie matrix class
require "matrix"
```

```ruby
#require all files in Functions folder (current folder)
#Dir[File.join(__dir__, '', '*.rb')].each { |file| require file }
require_relative "quat2dcm"
require_relative "conj"
require_relative "quatrot"
require_relative "dcmrtn2eci"
require_relative "full_impulse_vector"
require_relative "days_past_vernal_equinox"
require_relative "initThrustSolver"

#set constants
fire_time = 1 #time that the nozzle fires for, set as 2 ms here

#initialize moment arm and nozzle vectors in the body frame
initialization = initThrustSolver(cg)
moment_arms = initialization[0]
mf = initialization[1]

#reference frame conversions, get maneuver delta-v from RTN to body frame
dcm_il = dcmrtn2eci(recef,vecef,q_fi)# dcm lvlh into inertial
dcm_bi = quat2dcm(q_bi) # dcm inertial into body. % quaternion into dcm Function
comes from the aerospace toolbox

dcm_bl = dcm_bi*dcm_il # dcm lvlh into body.
maneuver = Matrix.column_vector([maneuver[0],maneuver[1],maneuver[2]])
maneuver_b = dcm_bl*maneuver # take maneuver from lvlh to body frame

# Solve for Tdt
impulse_b = m*maneuver_b; # Impulse in body frame

#tdt is a 2D column vector (matrix object)
#access elements by tdt[row,column]
tdt = mf.inverse*impulse_b

tdtfull = full_impulse_vector(tdt) #tdt is impulse
#positions 1, 2, and 3 correspond to the prop vectors defined in t1v, t2v, and
t3v
#positions 4, 5, and 6 correspond to prop vectors opposite 1, 2, and 3,
respectively

#.column Matrix object function returns a vector
#THIS FUNCTION DOES NOT EXIST IN THIS COSMOS RUBY VERSION

#converting from impulse to external torque by dividing by fire time then taking
cross product
```

```
#with the moment arm
ext_torque = Vector.[](0,0,0)
for a in 0..5 do
    if tdtfull[a] != 0 && a < 3
        ext_torque +=
tdtfull[a]/fire_time*mf.column(a).cross_product(moment_arms.column(a))
    elsif tdtfull[a] != 0 && a >= 3
        ext_torque += tdtfull[a]/fire_time*-1*mf.column(a-
3).cross_product(moment_arms.column(a))
    end
end

#return the external torque as required
return ext_torque

end
```

Unit test script:

```
#performs unit tests on each of the functions

require "test/unit"
require "matrix"
require_relative "../quat2dcm"
require_relative "../conj"
require_relative "../quatrot"
require_relative "../dcmrtn2eci"
require_relative "../full_impulse_vector"
require_relative "../days_past_vernal_equinox"

class TestFunctions < Test::Unit::TestCase
    def test_quat2dcm
        #first test case
        mat_compare = Matrix[[0.0, 0.0, 1.0],
            [0.0, 1.0, 0.0],
            [-1.0, 0.0, 0.0]]
        q1 = Vector.[](Math.sqrt(2.0)/2.0, 0.0, Math.sqrt(2.0)/2.0, 0.0)
        assert_in_delta(mat_compare[0,0], quat2dcm(q1).element(0,0), 0.000001)
        assert_in_delta(mat_compare[0,1], quat2dcm(q1).element(0,1), 0.000001)
        assert_in_delta(mat_compare[0,2], quat2dcm(q1).element(0,2), 0.000001)
        assert_in_delta(mat_compare[1,0], quat2dcm(q1).element(1,0), 0.000001)
        assert_in_delta(mat_compare[1,1], quat2dcm(q1).element(1,1), 0.000001)
        assert_in_delta(mat_compare[1,2], quat2dcm(q1).element(1,2), 0.000001)
        assert_in_delta(mat_compare[2,0], quat2dcm(q1).element(2,0), 0.000001)
        assert_in_delta(mat_compare[2,1], quat2dcm(q1).element(2,1), 0.000001)
        assert_in_delta(mat_compare[2,2], quat2dcm(q1).element(2,2), 0.000001)
```

```
        #second test case
        q1 = Vector.[](Math.sqrt(2)/2, Math.sqrt(2)/2, 0, 0)
        mat_compare = Matrix[[1.0, 0.0, 0.0],
            [0.0, 0.0, -1.0],
            [0.0, 1.0, 0.0]]
        assert_in_delta(mat_compare[0,0], quat2dcm(q1).element(0,0), 0.000001)
        assert_in_delta(mat_compare[0,1], quat2dcm(q1).element(0,1), 0.000001)
        assert_in_delta(mat_compare[0,2], quat2dcm(q1).element(0,2), 0.000001)
        assert_in_delta(mat_compare[1,0], quat2dcm(q1).element(1,0), 0.000001)
        assert_in_delta(mat_compare[1,1], quat2dcm(q1).element(1,1), 0.000001)
        assert_in_delta(mat_compare[1,2], quat2dcm(q1).element(1,2), 0.000001)
        assert_in_delta(mat_compare[2,0], quat2dcm(q1).element(2,0), 0.000001)
        assert_in_delta(mat_compare[2,1], quat2dcm(q1).element(2,1), 0.000001)
        assert_in_delta(mat_compare[2,2], quat2dcm(q1).element(2,2), 0.000001)
    end

    def test_conj
        #first test case
        compare = Vector.[](Math.sqrt(2)/2,-Math.sqrt(2)/2,0,0)
        q = Vector.[](Math.sqrt(2)/2,Math.sqrt(2)/2,0,0)
        assert_equal(compare,(conj(q)))

        #second test case
        compare = Vector.[](-Math.sqrt(3)/3,Math.sqrt(3)/3,-Math.sqrt(3)/3,0)
        q = Vector.[](-Math.sqrt(3)/3,-Math.sqrt(3)/3,Math.sqrt(3)/3,0)
        assert_equal(compare,(conj(q)))
    end

    def test_dcmrtn2eci
        #first test case
        compare = Matrix[[0.0, 1.0, 0.0],
            [0.0, 0.0, 1.0],
            [1.0, 0.0, 0.0]]
        recef = Vector.[](0,0,1)
        vecef = Vector.[](1,0,0)
        q = Vector.[](1,0,0,0)
        assert_in_delta(compare[0,0], dcmrtn2eci(recef,vecef,q).element(0,0),
0.000001)
        assert_in_delta(compare[0,1], dcmrtn2eci(recef,vecef,q).element(0,1),
0.000001)
        assert_in_delta(compare[0,2], dcmrtn2eci(recef,vecef,q).element(0,2),
0.000001)
        assert_in_delta(compare[1,0], dcmrtn2eci(recef,vecef,q).element(1,0),
0.000001)
```

```
        assert_in_delta(compare[1,1], dcmrtn2eci(recef,vecef,q).element(1,1),
0.000001)
        assert_in_delta(compare[1,2], dcmrtn2eci(recef,vecef,q).element(1,2),
0.000001)
        assert_in_delta(compare[2,0], dcmrtn2eci(recef,vecef,q).element(2,0),
0.000001)
        assert_in_delta(compare[2,1], dcmrtn2eci(recef,vecef,q).element(2,1),
0.000001)
        assert_in_delta(compare[2,2], dcmrtn2eci(recef,vecef,q).element(2,2),
0.000001)

        #second test case
        compare = Matrix[[1.0, 0.0, 0.0],
        [0.0, -1.0, 0.0],
        [0.0, 0.0, -1.0]]
        recef = Vector.[](1,0,0)
        vecef = Vector.[](0,-1,0)
        q = Vector.[](1,0,0,0)
        assert_in_delta(compare[0,0], dcmrtn2eci(recef,vecef,q).element(0,0),
0.000001)
        assert_in_delta(compare[0,1], dcmrtn2eci(recef,vecef,q).element(0,1),
0.000001)
        assert_in_delta(compare[0,2], dcmrtn2eci(recef,vecef,q).element(0,2),
0.000001)
        assert_in_delta(compare[1,0], dcmrtn2eci(recef,vecef,q).element(1,0),
0.000001)
        assert_in_delta(compare[1,1], dcmrtn2eci(recef,vecef,q).element(1,1),
0.000001)
        assert_in_delta(compare[1,2], dcmrtn2eci(recef,vecef,q).element(1,2),
0.000001)
        assert_in_delta(compare[2,0], dcmrtn2eci(recef,vecef,q).element(2,0),
0.000001)
        assert_in_delta(compare[2,1], dcmrtn2eci(recef,vecef,q).element(2,1),
0.000001)
        assert_in_delta(compare[2,2], dcmrtn2eci(recef,vecef,q).element(2,2),
0.000001)
    end

    def test_quatrot
        #first test case
        compare = Vector.[](1.0000, 1.0000, -1.0000)
        v = Vector.[](1,1,1)
        q = Vector.[](-Math.sqrt(2)/2, 0, Math.sqrt(2)/2, 0)
        assert_in_delta(compare[0],quatrot(q,v).element(0),0.000001)
        assert_in_delta(compare[1],quatrot(q,v).element(1),0.000001)
```

```
        assert_in_delta(compare[2],quatrot(q,v).element(2),0.000001)

        #second test case
        compare = Vector.[](-1.0000, 1.0000, 1.0000)
        q = Vector.[](Math.sqrt(2)/2,0,Math.sqrt(2)/2,0)
        assert_in_delta(compare[0],quatrot(q,v).element(0),0.000001)
        assert_in_delta(compare[1],quatrot(q,v).element(1),0.000001)
        assert_in_delta(compare[2],quatrot(q,v).element(2),0.000001)
    end

    def test_full_impulse_vector
        #first test case
        compare = Vector.[](0,2,0,4,0,6)
        tdt = Vector.[](-4,2,-6).to_matrix()
        output = full_impulse_vector(tdt)
        assert_in_delta(compare[0],output[0],0.000001)
        assert_in_delta(compare[1],output[1],0.000001)
        assert_in_delta(compare[2],output[2],0.000001)
        assert_in_delta(compare[3],output[3],0.000001)
        assert_in_delta(compare[4],output[4],0.000001)
        assert_in_delta(compare[5],output[5],0.000001)

        #second test case
        compare = Vector.[](0,0,6,0,2,0)
        tdt = Vector.[](0,-2,6).to_matrix()
        output = full_impulse_vector(tdt)
        assert_in_delta(compare[0],output[0],0.000001)
        assert_in_delta(compare[1],output[1],0.000001)
        assert_in_delta(compare[2],output[2],0.000001)
        assert_in_delta(compare[3],output[3],0.000001)
        assert_in_delta(compare[4],output[4],0.000001)
        assert_in_delta(compare[5],output[5],0.000001)
    end

    def test_days_past_vernal_equinox
        #first test case
        #user input date since 1/1/2000
        year = 24
        mon = 3
        day = 25
        compare = 5
        assert_equal(compare,days_past_vernal_equinox(year,mon,day))

        #second test case
        year = 24
```

```
        mon = 5
        day = 25
        compare = 66
        assert_equal(compare,days_past_vernal_equinox(year,mon,day))

        #third test case (can change this between leap year to see the extra
        #day is in fact counted, to do this change year to 24 and compare value
        # to 365 and test will come back successful)
        year = 25
        mon = 3
        day = 19
        compare = 364
        assert_equal(compare,days_past_vernal_equinox(year,mon,day))

        #fourth test case
        year = 25
        mon = 1
        day = 19
        compare = 305
        assert_equal(compare,days_past_vernal_equinox(year,mon,day))
    end
end
```


Acknowledgments

I would like to first especially thank some of the people who went above and beyond in helping me to transition to working on a space flight project coming from a highly theoretical mechanical engineering undergraduate program; thank you to Michael, Shan, Mark, Sam, Will, Antoine, and Max, among others, for their guidance and insight into spacecraft engineering and for their patience with me in this transition.

I would like to also thank some VISORS team members at Georgia Tech – Will, Antoine, Ebenezer, and Grace – who were so supportive in the development of the work detailed in this paper. Finally, thank you Dr. Lightsey for bringing me into this program and for the opportunity to work on projects that I am passionate about. Thank you for your guidance and mentorship during my time so far as a graduate student at Georgia Tech.

References

[1] Paletta, A., et al. (2022). "A Contingency Operations Architecture for the VISORS Formation Flying Space Telescope." 36th Annual Conference on Small Satellites. Logan, UT: American Institute of Aeronautics and Astronautics.

[2] Paletta, A., and Lightsey, E. G., "Development of an Autonomous Distributed Fault Management Architecture for the VISORS Mission," Masters Report, Space Systems Design Laboratory, Georgia Institute of Technology, Atlanta, GA, May, 2023.

[3] Lightsey, E. G., et al. (2022). "Concept of Operations for the VISORS Mission: A Two Satellite CubeSat Formation Flying Telescope." AAS Guidance, Navigation, and Control Conference. Breckenridge, CO: American Astronautical Society.

[4] Fill, D., and Lightsey, E. G., "Design of Test Bed for Subsystem Integration for the VISORS CubeSat Mission," Masters Report, Space Systems Design Laboratory, Georgia Institute of Technology, Atlanta, GA, May, 2022.

[5] Payne, Jacob, "XB1 Gen 3 EDU Interface Control Document, Virtual Super-resolution Optics with Reconfigurable Swarms (VISORS)," Rev A, Blue Canyon Technologies, Boulder, CO, 12 May 2021

[6] Melton, R. (2016). "Ball Aerospace COSMOS Open Source Command and Control System." 30th Annual AIAA/USU Conference on Small Satellites. Logan, UT: American Institute of Aeronautics and Astronautics.

[7] Ball Aerospace, "Cosmos," *Ball Aerospace COSMOS* Available: https://ballaerospace.github.io/cosmos-website/.

[8] Conway, B. A., Prussing, J. E., "Perturbation," *Orbital Mechanics*, New York, New York, 1993, pp. 164-168.

[9] Sumanth, R. M. (2019), "Computation of Eclipse Time for Low-Earth Orbiting Small Satellites," *International Journal of Avioation, Aeronautics, and Aerospace*, Vol. 6, Iss. 5, Art. 15, 2019. https://doi.org/10.15394/ijaaa.2019.1412

[10] Kimmel, E., et al. (2023). "Testing Methodology for Spacecraft Precision Formation Flying Missions." AAS Guidance, Navigation, and Control Conference. Breckenridge, CO: American Astronautical Society.