

Attitude Dynamics and Control of a Nano-Satellite Orbiting Mars

Landry Matthews*

University of Colorado Boulder, Boulder, CO, 80309

This document outlines the work completed for Tasks 1 through 11 of the ASEN 5010 Capstone Project - Attitude Dynamics and Control of a nano-Satellite Orbiting Mars. These tasks focus on simulating and analyzing the orbit and attitude dynamics of a nano-satellite in low Mars orbit. Results are derived using Python-based simulation frameworks and validated through analytical and numerical approaches, as well as through Coursera.

Introduction

This project involves the design and implementation of an attitude control system for a small nano-satellite in a circular low Mars orbit (LMO). The satellite performs three key mission functions: gathering science data by pointing a sensor nadir towards Mars, recharging via solar panels by pointing its solar array towards the Sun, and transmitting data to a geosynchronous Mars orbit (GMO) mother satellite by aligning its communication platform.

We assume the Mars-inertial frame \mathcal{N} and Hill frame $\mathcal{H} = \{\hat{i}_r, \hat{i}_\theta, \hat{i}_h\}$ base vectors are as shown in the following figure.

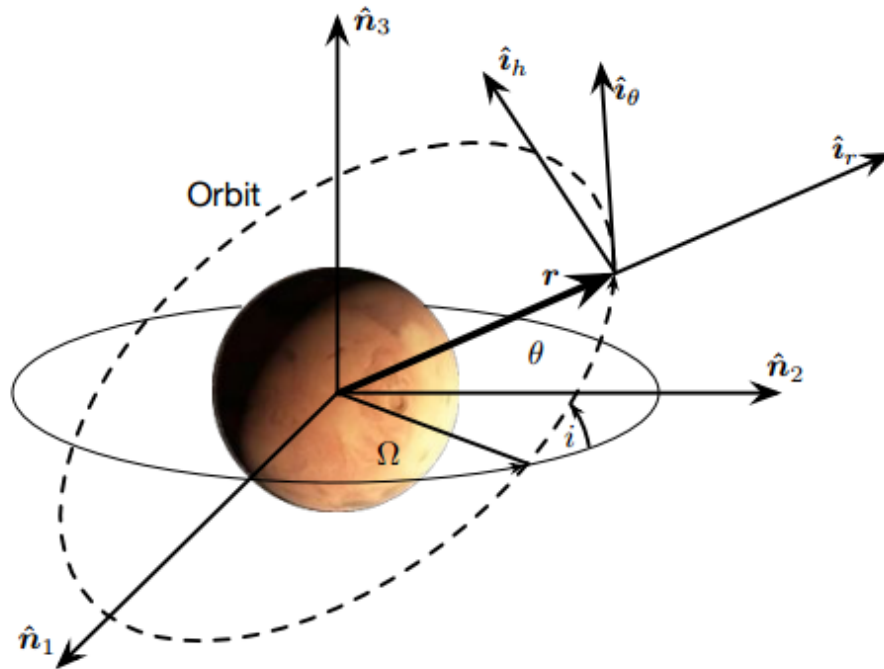


Fig. 1 Illustration of the Inertial frame \mathcal{N} and Hill frame \mathcal{H} base vectors, with the vector r showing the position of the craft in orbit

The attitude of the satellite is controlled using a thruster-based system that orients the spacecraft body frame \mathcal{B} to align with a reference frame \mathcal{R} , which changes depending on the current operational mode (science, charging, or

*Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder.

communication). These reference frames are computed on the basis of the known orbital parameters of the spacecraft and mother satellite, as well as the Sun's position relative to Mars.

The primary objective is to implement a feedback control law that drives the spacecraft's attitude and angular velocity—described by the modified Rodrigues parameters (MRPs) $\sigma_{\mathcal{B}/N}$ and angular velocity $\omega_{\mathcal{B}/N}$ —to track their corresponding reference values $\sigma_{\mathcal{R}/N}$ and $\omega_{\mathcal{R}/N}$. This control task includes computing the appropriate torque input \mathbf{u} and switching between control modes based on the orbital position of the spacecraft.

We are given the initial attitude MRP set and angular velocity conditions:

$$\sigma_{\mathcal{B}/N} = \begin{bmatrix} 0.3 \\ -0.4 \\ 0.5 \end{bmatrix}, \quad {}^B\omega_{\mathcal{B}/N} = \begin{bmatrix} 1.00 \\ 1.75 \\ -2.20 \end{bmatrix} \text{ deg/s}$$

Additionally, our spacecraft's inertia tensor is also provided:

$${}^B[I] = {}^B \begin{bmatrix} 10 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 7.5 \end{bmatrix} \text{ kg m}^2$$

We are required to follow certain protocol for when to align our spacecraft \mathcal{B} frame with our 3 reference \mathcal{R} frames. Depicted below is a figure illustrating our spacecraft body frame, showing its 3 principal axes aligning with the instrument sensors, solar panel normal, and antenna:

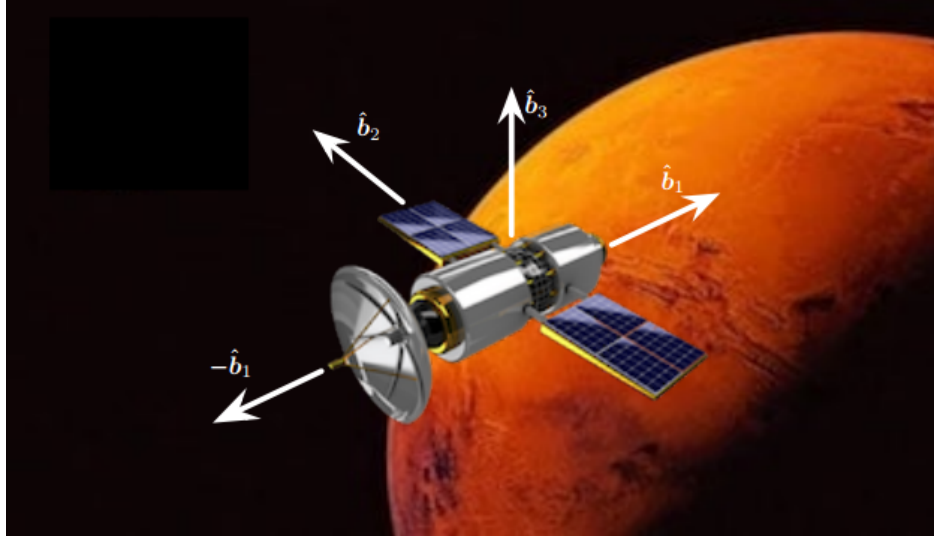


Fig. 2 Illustration of the spacecraft's body frame \mathcal{B} base vectors, showing the alignment of our subsystems of interest with our principal axes.

During the mission, the spacecraft must follow a control protocol to stabilize its rotation and align its components along the desired reference axes. Specifically, the spacecraft will aim its antenna (along $-\hat{b}_1$) toward the GMO mother satellite when doing comm., its sensor (along \hat{b}_1) towards Mars (in the nadir or $-\hat{r}$ direction) when sampling science data, or its solar panels (along \hat{b}_3) toward the sun when charging. Given the telescope's high power consumption, the solar panels should always be pointed toward the sun when the spacecraft is on the sunlit hemisphere of Mars (i.e., when the \hat{n}_2 coordinate is positive). Thus, the \hat{b}_3 axis will be aligned with \hat{n}_2 when the spacecraft is directed at the Sun. To complete the reference frame, the axis \hat{r}_1 will point in the $-\hat{n}_1$ direction.

When on the dark side of Mars (i.e., where the \hat{n}_2 coordinate is negative), the spacecraft must switch to either communication or science mode. In science mode, the platform's axis \hat{b}_1 should point at Mars' center, corresponding to the nadir direction. Additionally, the axis \hat{r}_2 will align with the orbital track axis \hat{i}_θ . In communication mode, the satellite needs to maintain a position where the angular separation between the LMO and GMO satellites is less than 35

degrees. If this angle is exceeded, we switch back to science mode. In this GMO facing mode, the communication axis $-\hat{b}_1$ will be directed toward the GMO mother satellite. These mission pointing scenarios are summarized below in Table 1.

Table 1 Spacecraft Pointing Scenario Summary

Orbital Situation	Pointing Goals
SC on sunlit side	Point \hat{b}_3 at the Sun for Solar
SC on dark side & GMO in sight	Point $-\hat{b}_1$ at the GMO for comm.
SC on dark side & GMO not in sight	Point \hat{b}_1 nadir for sensors

The initial orbit frame angles of both the LMO and GMO satellites are listed in Table 2. The corresponding initial orbital positions are illustrated in Figure 3.

Table 2 Initial Orbit Frame Orientation Angles

Spacecraft	Ω	i	$\theta(t_0)$
LMO	0.349066 rad	0.523599 rad	1.0472 rad
GMO	0 rad	0 rad	4.36332 rad

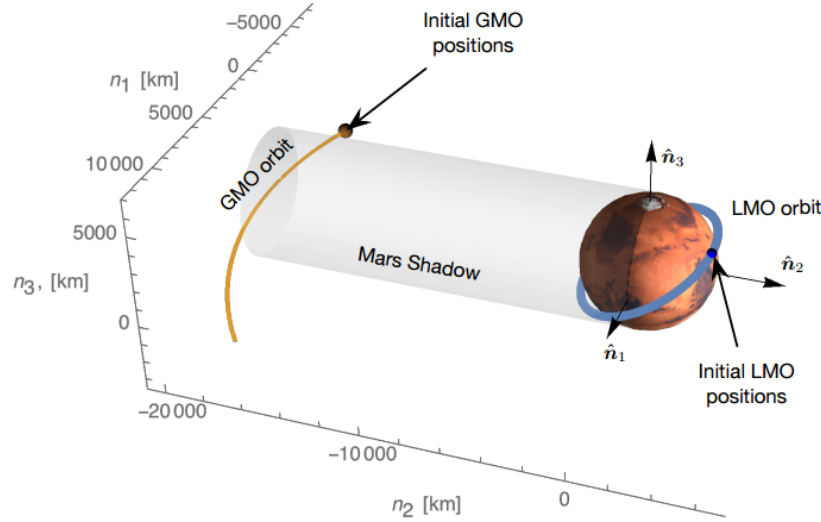


Fig. 3 Illustration of the initial orbital positions of the LMO and GMO crafts, with the Mars centered inertial frame visible and with its \hat{n}_2 axis pointing toward the Sun.

Here we can see a visualization of the entire mission overview, showing that our LMO craft will start on the sunlit side of Mars before orbiting into the shadow. For simplicity, it is assumed that the sun is always positioned in the \hat{n}_2 direction, as depicted in Figure 3.

We are also given a few more orbit characteristics: The nano-satellite orbit has an altitude $h = 400$ km, which when added to Mars' radius of 3396.19 gives the LMO orbital radius of $r_{LMO} = 3796.19$ km. We are always given the GMO orbital radius of $r_{GMO} = 20424.2$ km, which is calculated via the fact that it is geosynchronous and thus must have the same 1 day and 37 minute orbital period as Mars' rotational period. To calculate the orbital rates, we are given Mars' gravity constant as $\mu = 42828.3 \text{ km}^3/\text{s}^2$. The orbital rate of the LMO craft is then

calculated through the equation $\dot{\theta}_{LMO} = \sqrt{u/r_{LMO}^3} = 0.000884797$ rad/s and the GMO craft via the same math giving $\dot{\theta}_{GMO} = \sqrt{u/r_{GMO}^3} = 0.0000709003$ rad/s. Since we are told they are in circular orbit, we know that $\dot{\theta}$ is constant for both crafts.

In this project, we employ a simple proportional-derivative (PD) attitude control law:

$${}^B\mathbf{u} = -K\boldsymbol{\sigma}_{B/R} - P^B\boldsymbol{\omega}_{B/R}$$

This control law is used to drive the body frame \mathcal{B} towards its reference frame \mathcal{R} . Note that both feedback gains P and K are scalars.

Throughout this project, practical experience is gained in reference frame generation, spacecraft attitude representation using MRPs, and the development of feedback controllers. The work involves both analytical derivations and implementation in simulation software, culminating in a full mission scenario demonstrating autonomous control switching.

It should be noted here that for all tasks, MRP calculations and results will always switch to the shadow set to avoid singularities when the shadow set condition $|\sigma| > 1$ is met. To switch we employ the shadow set equation

$$\sigma_i^S = \frac{-\beta_i}{1 - \beta_0} = \frac{-\sigma_i}{\sigma^2}, \quad i = 1, 2, 3 \quad ([1], \text{eq. 3.147})$$

I have written a helper function in Python `checkShadowSet(sigma)` which takes in the current MRP attitude set and returns the shadow set, if the condition is met. This function is called throughout all tasks where switching to shadow set is applicable.

Each of the following tasks contributes to a component of this overall attitude control simulation.

I. Task 1: Orbit Simulation

For this task, we assume the general orbit frame \mathcal{N} is Mars-centered inertial. We also have the general orbit frame $\mathcal{O} : \{\hat{\mathbf{i}}_r, \hat{\mathbf{i}}_\theta, \hat{\mathbf{i}}_h\}$.

Next we had to write a function whose inputs are the radius r and the (3-1-3) Euler angles (Ω, i, θ) , and whose outputs are the inertial position vector ${}^N\mathbf{r}$ and velocity ${}^N\dot{\mathbf{r}}$ of the associated circular orbit.

To do this, we start by writing two functions, `theta_lmo(t)` and `theta_gmo(t)`. They both work by simply computing and returning $\theta(t_0) + t\dot{\theta}$, which gives the updated value of θ for time t . Note that the θ values here are respective to LMO or GMO (depending on which function is called).

From here, we can now obtain all 3 Euler angles for both our orbits at any time t . We can then write a helper function `Euler313toDCM(t1, t2, t3)`. Note for this function that $t1 = \Omega$, $t2 = i$, and $t3 = \theta(t)$. This function converts 313 Euler angles to their corresponding directional cosine matrix (DCM). It uses the following DCM matrix written in terms of 313 Euler angles to do so (from Schaub and Junkins [1] Appendix B):

$$\begin{bmatrix} -\sin(\Omega)\sin(\theta(t))\cos(i) + \cos(\Omega)\cos(\theta(t)) & \sin(\Omega)\cos(\theta(t)) + \sin(\theta(t))\cos(\Omega)\cos(i) & \sin(i)\sin(\theta(t)) \\ -\sin(\Omega)\cos(i)\cos(\theta(t)) - \sin(\theta(t))\cos(\Omega) & -\sin(\Omega)\sin(\theta(t)) + \cos(\Omega)\cos(i)\cos(\theta(t)) & \sin(i)\cos(\theta(t)) \\ \sin(\Omega)\sin(i) & -\sin(i)\cos(\Omega) & \cos(i) \end{bmatrix} \quad (1)$$

Now that we have a function to convert from Euler angles to DCMs, and a function to get our θ values at any time t , we can now write our main function for this task. This function is called `getInertialPositionVectors(r, omega, i, theta)`. It takes the orbital radius (which will either be r_{LMO} or r_{GMO}), and the current 313 Euler angles. Note that the argument `theta` must be pre-calculated before being passed into `getInertialPositionVectors`. Within the function, the DCM is calculated by plugging in the provided Euler angles into `Euler313toDCM()`. This gives the DCM $[ON]$ based on the frames defined above. We then take the transpose to get $[NO]$. We can then write our position

vector in the form ${}^O\mathbf{r} = {}^O \begin{bmatrix} r \\ 0 \\ 0 \end{bmatrix}$ km. We know this because we defined the \mathcal{O} frame so that the position vector points

in the $\hat{\mathbf{i}}_r$ direction. Then we convert it to \mathcal{N} frame by doing ${}^N\mathbf{r} = [NO]{}^O\mathbf{r}$. Similarly, we see that the $\hat{\mathbf{i}}_\theta$ direction is

defined as the direction tangential to the orbit. This means our velocity will be in this direction which we know for our circular orbit is just the radius multiplied by the rate. Thus we can calculate velocity ${}^N\dot{\mathbf{r}} = [NO] \circ \begin{bmatrix} 0 \\ r\dot{\theta} \\ 0 \end{bmatrix}$. Note that in

these equations, the value of r and $\dot{\theta}$ change depending on if we are doing LMO or GMO calculations. The function then returns the two computed values of ${}^N\mathbf{r}$ and ${}^N\dot{\mathbf{r}}$.

Next we validate our results. By plugging in the prescribed values of $t = 450$ for LMO and $t = 1150$ for GMO, we see the following results:

LMO at $t = 450$ s

$${}^N\mathbf{r} = \begin{bmatrix} -669.29 \\ 3227.50 \\ 1883.18 \end{bmatrix} \text{ (km)}, \quad {}^N\dot{\mathbf{r}} = \begin{bmatrix} -3.256 \\ -0.798 \\ 0.210 \end{bmatrix} \text{ (km/s)}$$

GMO at $t = 1150$ s

$${}^N\mathbf{r} = \begin{bmatrix} -5399.15 \\ -19697.64 \\ 0 \end{bmatrix} \text{ (km)}, \quad {}^N\dot{\mathbf{r}} = \begin{bmatrix} 1.397 \\ -0.383 \\ 0 \end{bmatrix} \text{ (km/s)}$$

Checking these values through Coursera confirms our calculations are correct.

II. Task 2: Orbit Frame Orientation

This task calculates the orientation of the orbit frame $\mathcal{H} : \{\hat{\mathbf{i}}_r, \hat{\mathbf{i}}_\theta, \hat{\mathbf{i}}_h\}$ with respect to the inertial frame \mathcal{N} . Our main function for this task, `getHNforLMO(t)`, takes a time t and returns the DCM for the LMO at that time. As discussed above in Task 1, the `Euler313toDCM()` function, and its associated matrix, is used here. Since only θ changes with time, `getHNforLMO(t)` can take in the current time and calculate the 3 Euler angles, then feed those into the `Euler313toDCM()` function. The function then returns the resulting DCM matrix. Note that the analytical expression for $[HN]$ is shown above in Eq. (1). In Task 2, we save a symbolic version of this matrix in our code for use in future tasks.

Next we validate our results. The DCM $[HN]$ is computed and evaluated at $t = 300$ s.

$$HN(t = 300s) = \begin{bmatrix} -0.0465 & 0.8741 & 0.4834 \\ -0.9842 & -0.1229 & 0.1277 \\ 0.1710 & -0.4698 & 0.8660 \end{bmatrix}$$

Checking these values through Coursera confirms our calculations are correct.

III. Task 3: Sun-Pointing Reference Frame Orientation

Now we begin with the first of our three reference frame implementations. When the spacecraft is on the sunlit side (positive $\hat{\mathbf{n}}_2$ coordinate), we want to define the reference frame so that the $+\hat{\mathbf{r}}_3$ axis (which will correspond to the satellite's solar panel normal) points to the sun, assumed to be in the $+\hat{\mathbf{n}}_2$ direction. To build an orthonormal frame, we use the $-\hat{\mathbf{n}}_1$ axis to define $+\hat{\mathbf{r}}_1$. Further, we know based on wanting a right handed frame that this means $\hat{\mathbf{r}}_2$ will be aligned with the $\hat{\mathbf{n}}_3$ direction. Based on these frame and unit vector definitions/relationships, we can derive the trivial DCM for the sun reference frame without needing any calculation:

$$R_s N(t = 0s) = R_s N = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

We see here that the DCM for this reference frame is not dependent on time, as the sun is assumed to always be in the same inertial direction.

Next, we wish to derive the angular velocity with respect to the inertial frame. Given that we just figured out that the DCM was not time dependent for this reference frame, that means it is not rotationally moving with respect to the inertial frame. It will always be the same fixed rotation. In other words, no matter how far into the mission we are, the sun will always be in the $+\hat{n}_2$ direction. This means that the angular velocity is

$${}^N\omega_{R_s/N} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \text{ rad/s}$$

Finally, we check these values through Coursera and see that our logic and derivations are correct.

IV. Task 4: Nadir-Pointing Reference Frame Orientation

The next reference frame we must assemble is the nadir one. On the shadowed side of Mars, and with the GMO mothership out of the 35° threshold to perform communication, the satellite will instead collect science data using its sensors. To do this, it must point its sensor ($+\hat{b}_1$) directly toward the nadir direction (toward Mars' center). This means our nadir reference frame is constructed such that $+\hat{r}_1$ points nadir to Mars (in the $-\hat{i}_r$ direction) and $+\hat{r}_2$ points in the velocity direction $+\hat{i}_\theta$. Recall that the \hat{i} unit vectors are part of the \mathcal{H} frame. We can then complete the reference frame by using a right handed system, which gives us \hat{r}_3 in the $-\hat{i}_h$ direction. From here, we can again easily define the DCM from our reference frame \mathcal{H} to the \mathcal{R}_n frame defined above as

$$R_n H = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

From here, we can multiply our two DCMs $[R_n H]$ and $[H N]$ to get our desired DCM of

$$\begin{aligned} [R_n H][H N] &= [R_n N] = \\ \begin{bmatrix} \sin(\Omega) \sin(\theta(t)) \cos(i) - \cos(\Omega) \cos(\theta(t)) & -\sin(\Omega) \cos(\theta(t)) - \sin(\theta(t)) \cos(\Omega) \cos(i) & -\sin(i) \sin(\theta(t)) \\ -\sin(\Omega) \cos(i) \cos(\theta(t)) - \sin(\theta(t)) \cos(\Omega) & -\sin(\Omega) \sin(\theta(t)) + \cos(\Omega) \cos(i) \cos(\theta(t)) & \sin(i) \cos(\theta(t)) \\ -\sin(\Omega) \sin(i) & \sin(i) \cos(\Omega) & -\cos(i) \end{bmatrix} \end{aligned} \quad (2)$$

Now we need to write a function to do this in our code. Because $[H N]$ is a function of time, we will need to pass in time t as a parameter. We define `getRnN(t)` which does the exact math outlined above, and returns a numeric value of the $[R_n N]$ DCM at the specified time t .

Next we need to write a function to get our inertial angular velocity. First, we recall that our angular velocity in the \mathcal{H} frame is just $\dot{\theta}\hat{i}_h$. Since we know that $\hat{r}_3 = -\hat{i}_h$ we can easily rewrite our angular velocity in the \mathcal{R}_n frame as $-\dot{\theta}\hat{r}_3$. From here, we can use the transpose of our $[R_n N]$ DCM to express our angular velocity in the N frame by doing

$${}^N\omega_{R_n/N} = [R_n N]^T (-\dot{\theta}\hat{r}_3) = [N R_n] \begin{bmatrix} 0 \\ 0 \\ -\dot{\theta}_{LMO} \end{bmatrix}$$

Now all we need to do is implement this in the code, which is done in the `getOmegaRnN(t)` function. This function does exactly as outlined above — takes in a time t , gets the $[R_n N]$ DCM at that t , computes its transpose $[N R_n]$, and then multiplies the \mathcal{R} frame ${}^R\omega_{R_n/N}$ by the matrix to get ${}^N\omega_{R_n/N}$, which is the vector it returns.

We test this at 330 seconds and are given the following results from our two functions:

$$R_n N(t = 330s) = \begin{bmatrix} 0.0726 & -0.8706 & -0.4866 \\ -0.9826 & -0.1461 & 0.1148 \\ -0.1710 & 0.4698 & -0.8660 \end{bmatrix}, \quad {}^N\omega_{R_n/N} = \begin{bmatrix} 0.000151 \\ -0.000416 \\ 0.000766 \end{bmatrix} \text{ rad/s}$$

Checking these results through Coursera confirms our math and code are correct.

V. Task 5: GMO-Pointing Reference Frame Orientation

It's now time to develop our final reference frame. When the LMO craft is on the dark side of Mars and the GMO satellite is less than 35° away, the satellite enters communication mode. In this mode, the satellite aligns its $-\hat{\mathbf{b}}_1$ (antenna direction) with the direction of the GMO mothership. Therefore our reference frame \mathcal{R}_c can be defined as having its $-\hat{\mathbf{r}}_1$ as pointing to the GMO satellite. The relative vector between the spacecraft and GMO is used to compute this direction and the remaining reference directions. Referring to this relative vector as $\Delta\mathbf{r} = \mathbf{r}_{GMO} - \mathbf{r}_{LMO}$ means that we can define our first reference direction $\hat{\mathbf{r}}_1$ as $-\frac{\Delta\mathbf{r}}{|\Delta\mathbf{r}|}$ and our second reference direction as $\hat{\mathbf{r}}_2 = \frac{\Delta\mathbf{r} \times \hat{\mathbf{n}}_3}{|\Delta\mathbf{r} \times \hat{\mathbf{n}}_3|}$. Following the right hand rule, our third reference frame direction will then be $\hat{\mathbf{r}}_3 = \hat{\mathbf{r}}_1 \times \hat{\mathbf{r}}_2$.

Our first goal here is to derive an expression for the GMO-pointing reference frame DCM $[R_c N]$. This means we essentially need to write all the \mathcal{R}_c unit vectors in terms of the \mathcal{N} frame unit vectors. We can then stack them to form the DCM.

For simplicity, we will first start by getting all of our desired values and vectors in the \mathcal{N} frame. To do this we will need to get two $[NH]$ DCMs, $[NH_{LMO}]$ and $[NH_{GMO}]$. Recall that we have the $[HN]$ DCM saved symbolically in our code. Since this matrix preserves the 313 Euler angles symbolically, we can use it for both the LMO and GMO \mathcal{H} frames. Thus we can obtain our two desired DCMs by doing $[H_{LMO} N]^T$ and $[H_{GMO} N]^T$. To clarify, the difference in these matrices is the input angles, $(\Omega, i, \theta(t))$, where Ω and i will just be their respective LMO or GMO angles at time $t = 0$, and $\theta(t)$ will be the value from `theta_lmo(t)` or `theta_gmo(t)` which were described and implemented in Task 1.

$$\text{Due to how we defined the } \mathcal{H} \text{ frame, we know our } {}^H\mathbf{r}_{LMO} \text{ and } {}^H\mathbf{r}_{GMO} \text{ vectors are simply } \begin{bmatrix} \mathbf{r}_{LMO} \\ 0 \\ 0 \end{bmatrix} \text{ and } \begin{bmatrix} \mathbf{r}_{GMO} \\ 0 \\ 0 \end{bmatrix}$$

respectively. Then we can get our ${}^N\mathbf{r}_{LMO}$ and ${}^N\mathbf{r}_{GMO}$ vectors by doing

$${}^N\mathbf{r}_{LMO} = [NH_{LMO}]^H \mathbf{r}_{LMO}$$

and

$${}^N\mathbf{r}_{GMO} = [NH_{GMO}]^H \mathbf{r}_{GMO}$$

We then follow our definition of $\Delta\mathbf{r}$ to get

$${}^N\Delta\mathbf{r} = {}^N\mathbf{r}_{GMO} - {}^N\mathbf{r}_{LMO}$$

After normalizing, this gives us our first unit vector $\hat{\mathbf{r}}_1$ in terms of \mathcal{N} frame components as it is defined above. This also allows us to easily get $\hat{\mathbf{r}}_2$, since we have ${}^N\Delta\mathbf{r}$ (and $\hat{\mathbf{n}}_3$ is of course also defined in \mathcal{N} frame), we can easily follow the definition above and take the cross product and divide by magnitude. And finally, $\hat{\mathbf{r}}_3$ is then just the cross product of the two vectors we just computed, $\hat{\mathbf{r}}_1 \times \hat{\mathbf{r}}_2$. We can now stack these three \mathcal{R} frame vectors expressed in terms of \mathcal{N} to obtain the expression for our DCM $[R_c N]$. This is implemented exactly as described in the code, which gives us a function `getRcNExpr()` that returns $[R_c N]$ as a expression in terms of time t . Note that while I did obtain a LaTeX expression for this matrix as a function of time, it is quite large at over 25,000 characters of text, so it will not be displayed here.

Further, I created a wrapper function `getRcN(t)` which takes in time and evaluates and returns a numeric version of the $[R_c N]$ matrix from `getRcNExpr()`. Evaluating this function at $t = 330\text{s}$ returns the following matrix:

$$R_c N = \begin{bmatrix} 0.2655 & 0.9609 & 0.0784 \\ -0.9639 & 0.2663 & 0 \\ -0.0209 & -0.0755 & 0.9969 \end{bmatrix}$$

Next, we need to find the angular velocity $\omega_{R_c/N}$. To do this, we leverage a rearranged form of the the kinematic differential equation for DCMs:

$$[\dot{C}] = -[\tilde{\omega}][C] \quad ([1], \text{eq. 3.27})$$

Since we know DCMs are orthogonal matrices, we can rearrange this equation by multiplying by $[C]^T$ and moving the negative sign over. Plugging in our matrix $[R_c N]$ for $[C]$ gives us the equation with the angular velocity matrix isolated:

$$[\tilde{\omega}] = -[\dot{R}_c N][R_c N]^T$$

Here, $[\tilde{\omega}]$ is the skew-symmetric matrix of the angular velocity vector. While the $[R_c N]$ matrix as a function of time is not feasibly possible to derive by hand, we can leverage Python's symbolic manipulation library `sympy` to take

the derivative with respect to time for us, giving us $[R_c \dot{N}]$. We then multiply this by the transpose of the original $[R_c N]$ matrix and multiply by -1 , resulting in a final expression of $[\tilde{\omega}]$ in terms of time t . It is fun to note here that printing this matrix as an expression of time resulted in over 800,000 characters of L^AT_EXcode. We will not be displaying it in this report. Finally, after evaluating at our time t , the code gives us back the skew symmetric matrix $[\tilde{\omega}]$. we can extract the off diagonal terms to re-build the original ${}^{\mathcal{R}}\omega_{R_c/N}$ vector. Note the \mathcal{R} frame in this vector, which is a result of the form of the kinematic differential equation we used. To convert, we do the following math:

$${}^N\omega_{R_c/N} = [R_c N]^T {}^{\mathcal{R}}\omega_{R_c/N}$$

Finally, we have our ${}^N\omega_{R_c/N}$ vector and can return its value from the function. The function that implements this is called `getOmegaRcNAnalytically`.

Additionally, we evaluate using numerical differentiation in `getOmegaRcN` to estimate the value of $[R_c \dot{N}]$. This function takes a small time step, e.g. $\Delta t = 0.001s$, and evaluates $[R_c N]$ at $[R_c N](t + \Delta t)$, $[R_c N](t - \Delta t)$ and then estimates $[R_c \dot{N}]$ by doing

$$[R_c \dot{N}] = \frac{[R_c N](t + \Delta t) - [R_c N](t - \Delta t)}{2\Delta t}$$

After obtaining $[R_c \dot{N}]$, the rest of the function's code follows the same logic as described for `getOmegaRcNAnalytically`.

With both of these functions ready to test, we can plug in $t = 330s$ and compare the answers returned. The angular velocity is evaluated both analytically and numerically for verification of each method. Running the code gives the following results:

$${}^N\omega_{R_c/N}(t = 330s)_{(\text{Analytical})} = {}^N\omega_{R_c/N}(t = 330s)_{(\text{Numerical})} = \begin{bmatrix} 1.978 \times 10^{-5} \\ -5.465 \times 10^{-6} \\ 1.913 \times 10^{-4} \end{bmatrix} \text{ rad/s}$$

VI. Task 6: Attitude Error Evaluation

Now we have all three reference frames defined and can compute the associated reference DCMs $[R_s N]$, $[R_n N]$, and $[R_c N]$ for any time t . For this task we will be computing the attitude and angular velocity tracking errors of the spacecraft body frame \mathcal{B} with respect to the idealized reference frames \mathcal{R} . We will write a function called `getTrackingErrors(t, sigma_bn, B_omega_bn, RN, N_omega_rn)`. This will use two helper functions to convert to and from MRPs and DCM. These functions will be called `MRP2DCM(sigma)` and `DCM2MRP(C)`. To get from σ (MRPs) to $[C]$ (DCM), we use the vectorial equation

$$[C] = [I_{3 \times 3}] + \frac{8[\tilde{\sigma}]^2 - 4(1 - \sigma^2)[\tilde{\sigma}]}{(1 + \sigma^2)^2} \quad ([1], \text{eq. 3.152})$$

and to go from DCM to MRPs we first compute the quaternions $(\beta_0, \beta_1, \beta_2, \beta_3)$ using Sheppard's method: This is done by computing the first four β_i^2 terms:

$$\beta_0^2 = \frac{1}{4}(1 + \text{trace}[C]) \quad ([1], \text{eq. 3.100a})$$

$$\beta_1^2 = \frac{1}{4}(1 + 2C_{11} - \text{trace}[C]) \quad ([1], \text{eq. 3.100b})$$

$$\beta_2^2 = \frac{1}{4}(1 + 2C_{22} - \text{trace}[C]) \quad ([1], \text{eq. 3.100c})$$

$$\beta_3^2 = \frac{1}{4}(1 + 2C_{33} - \text{trace}[C]) \quad ([1], \text{eq. 3.100d})$$

We then take the square root of the largest β_i^2 term and arbitrarily choose β_i to be positive. The other β_j terms are found by dividing the appropriate three of the following six equations by the chosen largest β_i coordinate:

$$\beta_0\beta_1 = (C_{23} - C_{32})/4 \quad ([1], \text{eq. 3.101a})$$

$$\beta_0\beta_2 = (C_{31} - C_{13})/4 \quad ([1], \text{eq. 3.101b})$$

$$\beta_0\beta_3 = (C_{12} - C_{21})/4 \quad ([1], \text{eq. 3.101c})$$

$$\beta_2\beta_3 = (C_{23} + C_{32})/4 \quad ([1], \text{eq. 3.101d})$$

$$\beta_3\beta_1 = (C_{31} + C_{13})/4 \quad ([1], \text{eq. 3.101e})$$

$$\beta_1\beta_2 = (C_{12} + C_{21})/4 \quad ([1], \text{eq. 3.101f})$$

This is implemented in the `DCM2Quaternion(C)` function. Note that this function is only called as a helper function from our `DCM2MRP(C)` method, and is never called directly.

We then use the formula

$$\sigma_i = \frac{\beta_i}{1 + \beta_0}, \quad i = 1, 2, 3 \quad ([1], \text{eq. 3.142})$$

to compute the final MRP set, and check for the shadow set before returning the values.

For our simulation in this task, we are instructed to use the initial values of $\sigma_{B/N}$ and ${}^B\omega_{B/N}$ (converted to radians), given as

$$\sigma_{B/N} = \begin{bmatrix} 0.3 \\ -0.4 \\ 0.5 \end{bmatrix}, \quad {}^B\omega_{B/N} = \begin{bmatrix} 0.01745330 \\ 0.03054326 \\ -0.03839720 \end{bmatrix} \text{ rad/s}$$

The `getTrackingErrors` function takes in the current time t , the current $\sigma_{B/N}$ and ${}^B\omega_{B/N}$ (which we are instructed to assume are just the values given to us for time $t = 0$ for this task), and finally the reference frame values constructed and calculated in Tasks 3-5, $[RN]$ and ${}^N\omega_{R/N}$, which will be different for each of the three modes.

The function then converts the provided $\sigma_{B/N}$ to a $[BN]$ DCM matrix, and uses the provided $[RN]$ matrix to compute the $[BR]$ matrix as

$$[BR] = [BN][RN]^T$$

We then convert this $[BR]$ DCM back to MRPs, checking for shadow set where applicable, which gives us our $\sigma_{B/R}$ to be returned by the function.

Next the function computes ${}^B\omega_{B/R}$ by doing

$${}^B\omega_{B/R} = {}^B\omega_{B/N} - [BN]{}^N\omega_{R/N}$$

We want the R/N values since the body frame with respect to our derived reference frames gives our tracking error. Our function `getTrackingErrors` is now ready to test. We provide it values for each of the three modes and check the results:

Sun-Pointing

$$\text{getTrackingErrors}(t, \sigma_{B/N}, {}^B\omega_{B/N}, [R_sN], {}^N\omega_{R_s/N}) \rightarrow$$

$$\sigma_{B/R} = \begin{bmatrix} -0.7754 \\ -0.4739 \\ 0.0431 \end{bmatrix}, \quad {}^B\omega_{B/R} = \begin{bmatrix} 0.01745 \\ 0.03054 \\ -0.03840 \end{bmatrix} \text{ rad/s}$$

Nadir-Pointing

$$\text{getTrackingErrors}(t, \sigma_{B/N}, {}^B\omega_{B/N}, [R_nN], {}^N\omega_{R_n/N}) \rightarrow$$

$$\sigma_{B/R} = \begin{bmatrix} 0.2623 \\ 0.5547 \\ 0.0394 \end{bmatrix}, \quad {}^B\omega_{B/R} = \begin{bmatrix} 0.01685 \\ 0.03093 \\ -0.03892 \end{bmatrix} \text{ rad/s}$$

GMO-Pointing

`getTrackingErrors(t, $\sigma_{B/N}$, ${}^B\omega_{B/N}$, $[R_c N]$, ${}^N\omega_{R_c/N}$) \rightarrow`

$$\sigma_{B/R} = \begin{bmatrix} 0.0170 \\ -0.3828 \\ 0.2076 \end{bmatrix}, \quad {}^B\omega_{B/R} = \begin{bmatrix} 0.01730 \\ 0.03066 \\ -0.03844 \end{bmatrix} \text{ rad/s}$$

Checking each all six of these results in Coursera confirms our math and derivations are correct.

VII. Task 7: Numerical Attitude Simulator

Next, we finally come to our simulation of the dynamics over a specified period of time. For this task, we set up a lot of functions and equations that will be re-used throughout the remainder of the tasks. We are asked to write our own Runge-Kutta 4th order (RK4) integrator to integrate our attitudes and rates. For this task (and all remaining tasks), we use a integration time step of $dt = 1\text{s}$ and we save our states and rates as the state vector

$$X = \begin{bmatrix} \sigma_{B/N} \\ {}^B\omega_{B/N} \end{bmatrix}$$

We are told to assume the satellite is rigid and obeys the following dynamical system

$$[I]\dot{\omega}_{B/N} = -[\tilde{\omega}_{B/N}][I]\omega_{B/N} + u$$

Here $[I]$ is our inertia tensor matrix given by

$${}^B[I] = {}^B \begin{bmatrix} 10 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 7.5 \end{bmatrix} \text{ kg m}^2$$

By multiplying each side by the inverse of this matrix, we can isolate the angular velocity rates as

$$\dot{\omega}_{B/N} = -[I]^{-1}([\tilde{\omega}_{B/N}][I]\omega_{B/N} + u)$$

For the attitude rates, we use the following equation from our textbook

$$\dot{\sigma} = \frac{1}{4}[(1 - \sigma^2)[I_{3 \times 3}] + 2[\tilde{\sigma}] + 2\sigma\sigma^T]{}^B\omega \quad ([1], \text{eq. 3.160})$$

To accomplish this task, we first implement our RK4 integrator as given to us in the handout:

```
def rk4_integrator(f, X, u, dt, tn):
    k1 = dt*f(X, tn, u)
    k2 = dt*f(X+(k1/2), tn+(dt/2), u)
    k3 = dt*f(X+(k2/2), tn+(dt/2), u)
    k4 = dt*f(X+k3, tn+dt, u)
    X = X + (1/6)*(k1+2*k2+2*k3+k4)
    return X
```

Then we write another function `dynamics(X, dt, u)` which solves the two equations of motions for $\dot{\omega}_{B/N}$ and $\dot{\sigma}$. This function will be passed in as `f` for all calls to our RK4 integrator. The other arguments of `rk4_integrator` are the state X , control torque u , time-step `dt`, and finally the current simulation time `tn`. It then calls `dynamics` to compute the updated $\dot{\omega}_{B/N}$ and $\dot{\sigma}$ values based on the current state and control torque. Finally it returns the updated state based on the results from the `dynamics` function.

We now have most of what we need for the remaining tasks. For Task 7, we are asked to simulate these equations of motion with our initial conditions for 500 seconds and with no control torque. We then want to find the MRPs $\sigma_{B/N}$, kinetic energy T , \mathcal{B} frame angular momentum ${}^B H$, and \mathcal{N} frame angular momentum ${}^N H$. Additionally, we are tasked

with applying a fixed control torque ${}^B\mathbf{u} = (0.01, -0.01, 0.02)$ Nm, and simulate again for only 100 seconds this time. For this simulation with control torque, we only need to calculate the attitude $\sigma_{B/N}$ at 100 seconds.

We start by iterating on our simulation time from 0 to 500 seconds. In each iteration, we extract $\sigma_{B/N}$ and ${}^B\omega_{B/N}$ from the state vector \mathbf{X} and save their history for plotting. We do the same for kinetic energy using the equation

$$T = \frac{1}{2} {}^B\omega_{B/N}^T [I] {}^B\omega_{B/N} \quad ([1], \text{eq. 4.55})$$

and for angular momentum using the equation

$${}^B\mathbf{H} = [I] {}^B\omega_{B/N} \quad ([1], \text{eq. 4.25})$$

For ${}^N\mathbf{H}$ we can simply convert this to \mathcal{N} frame by applying the DCM:

$${}^N\mathbf{H} = [BN][I] {}^B\omega_{B/N}$$

Here, the \mathcal{N} frame angular momentum is constant (as expected), however, we chart its history anyway to confirm our model and calculations are correct.

We then plot all these values for analysis:

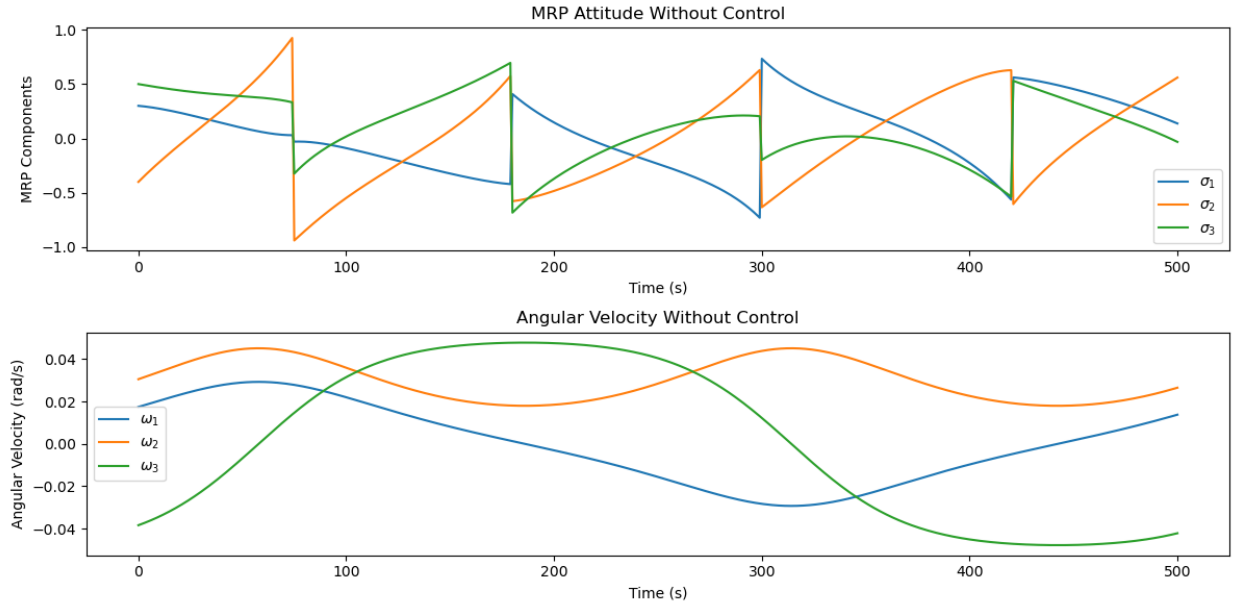


Fig. 4 Attitude and angular velocity of our 500s simulation without control torque

In Fig.4 we can see the MRPs evolve smoothly with periodic sharp transitions – this is the shadow set switching of MRPs to avoid singularities. This behavior is expected and shows the system rotating naturally without a control torque. Meanwhile, the angular velocity values exhibit an expected smooth oscillatory behavior. No component stays constant, which is typical for a free rigid body rotation with asymmetric inertia and no control torque.

Next we look at kinetic energy:

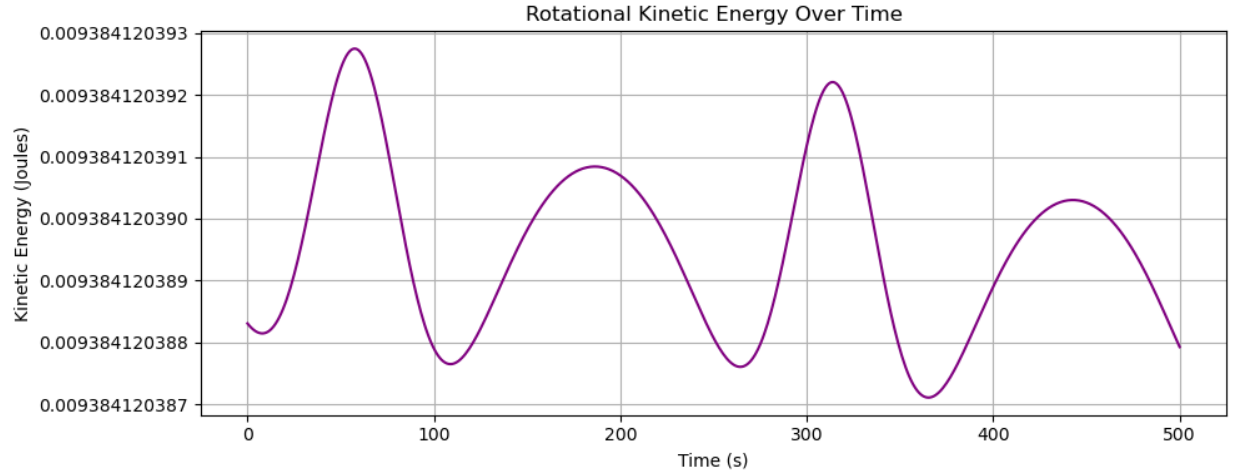


Fig. 5 Kinetic Energy T of our spacecraft over 500s simulation without control torque

In Fig.5, we see the rotational kinetic energy is almost perfectly constant, with only minor numerical fluctuations (on the order of 1×10^{-13}). This further confirms that the system is torque-free, since no energy is being added or removed. Next we plot both of the angular momentum vectors together to compare:

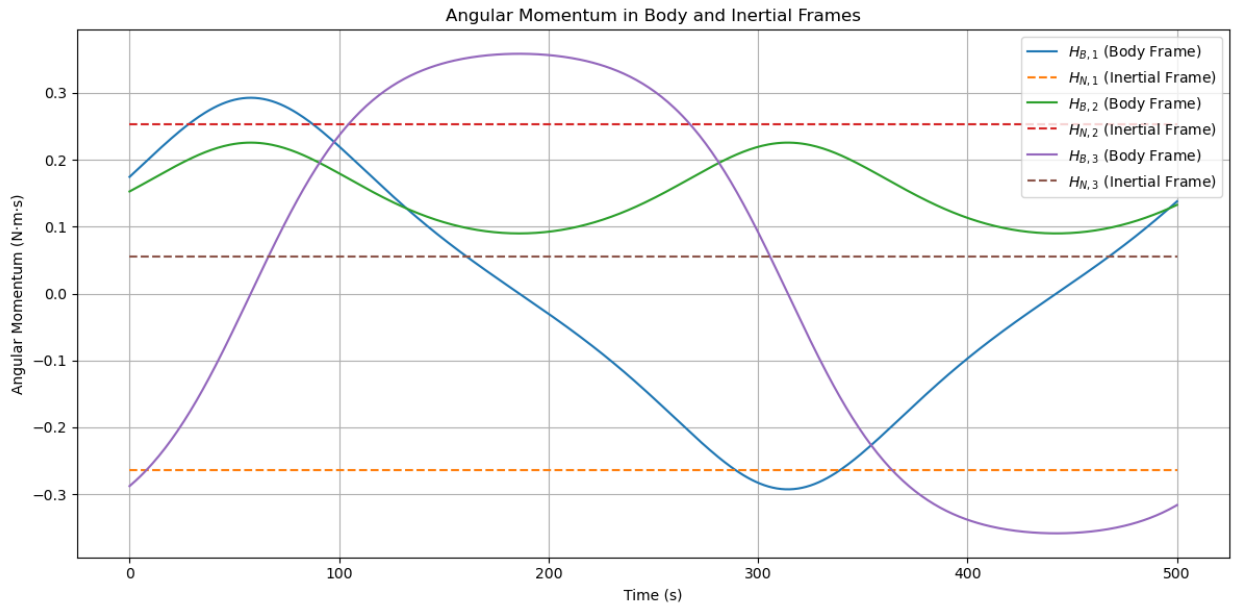


Fig. 6 Angular momentum of our spacecraft in inertial and body fixed frames over 500s simulation without control torque

The angular momentum vector in the body frame ${}^B\mathbf{H}$ clearly changes over time, which is expected since the body frame is rotating. The angular momentum in the inertial frame ${}^N\mathbf{H}$ remains constant (flat dotted lines) throughout the entire simulation. This confirms conservation of angular momentum in an inertial frame when no external torques act on the system.

In summary, we can see that in the absence of external torques, the spacecraft demonstrates classic free rigid body dynamics. Angular momentum is conserved in the inertial frame, while its components in the rotating body frame vary over time. The rotational kinetic energy remains essentially constant, reinforcing the system's energy conservation.

Looking at the attitude, the MRP evolution shows smooth lines except for the expected shadow set behavior, and angular velocities oscillate, all consistent with torque-free motion of an asymmetric rigid body.

Next we perform our analysis with a control torque ${}^B\mathbf{u} = (0.01, -0.01, 0.02)$ Nm. The logic is mostly the same, leveraging our RK4 integrator while iterating over the simulation time from 0-100 seconds. During this simulation, we only save the history of the attitude and angular velocity. We can see the results below:

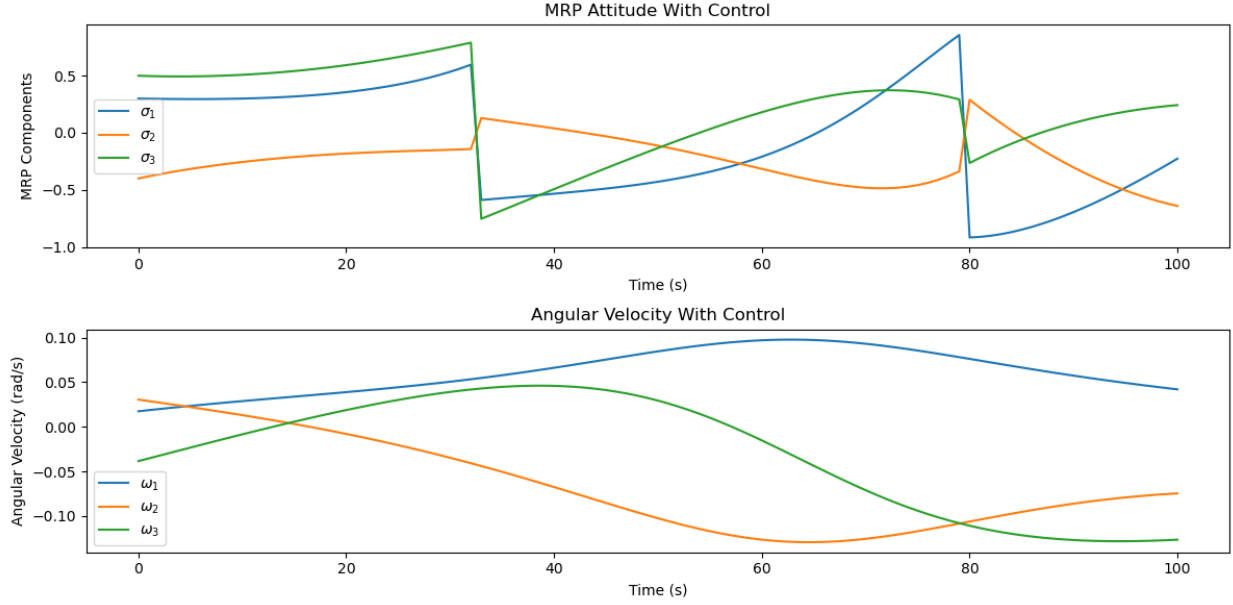


Fig. 7 Attitude and angular velocity of our 100s simulation with control torque

In this simulation we expect that the control torque is driving attitude changes more rapidly, which is evident in Fig.7 by the shadow set switching almost twice as frequently than in the un-controlled simulation. This more rapid orientation is a direct effect of the control torque actively rotating the spacecraft toward a desired state, as opposed to the more passive, natural motion observed in the torque-free case. Additionally, The angular velocity exhibits slightly more damped behavior, in contrast to the sustained oscillations seen without control. Over the 100-second period, the angular rates trend toward more steady values, confirming that the control input is successfully regulating rotational motion.

While these plots allow us to analyze the simulation, we are also tasked with providing their specific values at $t = 500$. Doing so gives the following results:

MRP Attitude, $\sigma_{B/N}(500\text{ s})$:

$$\begin{bmatrix} 0.1377 \\ 0.5603 \\ -0.0322 \end{bmatrix}$$

Angular Momentum in Body Frame, ${}^B\mathbf{H}(500\text{ s})$ ($\text{kg} \cdot \text{m}^2/\text{s}$):

$$\begin{bmatrix} 0.1379 \\ 0.1327 \\ -0.3164 \end{bmatrix}$$

Angular Momentum in Inertial Frame, ${}^N\mathbf{H}(500\text{ s})$ ($\text{kg} \cdot \text{m}^2/\text{s}$):

$$\begin{bmatrix} -0.2641 \\ 0.2528 \\ 0.0553 \end{bmatrix}$$

Rotational Kinetic Energy, $T(500\text{ s})$:

$$0.0094\text{ J}$$

Next we do the same for our attitude at $t = 100\text{ s}$ for our simulation with the control torque added:

MRP Attitude, $\sigma_{B/N}(100\text{ s})$:

$$\begin{bmatrix} -0.2269 \\ -0.6414 \\ 0.2425 \end{bmatrix}$$

Finally, we plug in these results to Coursera and see that our simulations and derivations are all working correctly.

VIII. Task 8: Sun Pointing Control

In the next few sections, we'll be testing each control pointing mode individually, with our final control torque incorporated, and then finally combine them all in the final task. For Task 8, we develop a simple PD control as shown in the following equation which was provided to us:

$${}^B\mathbf{u} = -K\sigma_{B/R} - P^B\omega_{B/R}$$

For the next three tasks, we assume the spacecraft engages with the corresponding pointing control mode at time $t = 0$. For all remaining tasks, we must integrate the control torque equation into our simulation, ensuring that our desired pointing is achieved with closed-loop performance. In this task, that desired pointing is the sun-pointing mode. We want to use linearized closed loop dynamics to determine the scalar feedback gains K and P such that the slowest decay response time is 120 seconds. Additionally, the closed loop response for all attitude components should be either critically damped or underdamped. This means our damping ratio must be $\xi \leq 1$. From that information, we can start by deriving our P value using the time decay equation

$$T_i = \frac{2I_i}{P_i} \implies P_i = \frac{2I_i}{T_i} \quad ([1], \text{eq. 8.119})$$

In this equation, we want I_i to be the maximum inertia in our inertia tensor $[I]$ since the value of T_i is maximized when I is maximized. Since 120 seconds is our maximum decay time, we want to choose $I_{max} = 10\text{ kg m}^2$ to ensure that $T_i \leq 120\text{ s}$. Plugging this in yields

$$P_i = \frac{2(10)}{120} = \frac{20}{120} = \frac{1}{6} = 0.1\bar{6} \text{ kg m}^2/\text{s}$$

In a similar fashion we use this result to derive our value for K

$$\xi_i = \frac{P_i}{\sqrt{KI_i}} \implies \sqrt{KI_i} = \frac{P_i}{\xi_i} \implies KI_i = \frac{P_i^2}{\xi_i^2} \implies K = \frac{P_i^2}{\xi_i^2 I_i} \quad ([1], \text{eq. 8.118})$$

Here, we want I_i to be the minimum inertia in our inertia tensor $[I]$ since the value of ξ_i is maximized when I is minimized. Since $\xi \leq 1$ is our maximum damping ratio, we want to choose $I_{min} = 5\text{ kg m}^2$ to ensure that $\xi \leq 1$. Plugging in this along with our calculated value of P yields

$$K = \frac{(\frac{1}{6})^2}{1^2(5)} = \frac{\frac{1}{36}}{5} = \frac{1}{180} = 0.00\bar{5} \text{ kg m}^2/\text{s}^2$$

Now that we have chosen our values for K and P , we can write a function to compute our control torque ${}^B\mathbf{u}$. This function is called `PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)`. It computes the current control torque vector ${}^B\mathbf{u}$ as defined in the equation above. To do this, it first has to call our function from Task 6, `getTrackingErrors`. Recall that this function returns the two values needed to compute ${}^B\mathbf{u}$, $\sigma_{B/R}$ and ${}^B\omega_{B/R}$. A full description is given in Task 6. The main thing to note here is that for Task 8, the `PD_controller` DCM $[RN]$ is the sun-pointing DCM $[R_sN]$ and the value passed as `omega_rn` is ${}^N\omega_{R_s/N}$. After obtaining these values, the control torque is computed and returned by simply plugging in all of our values.

We can now start our simulation loop, similarly to how we completed Task 7. This time, we run the simulation for $t = 500$ s. We iterate through each timestep, saving both $\sigma_{B/N}$ and ${}^B\omega_{B/N}$ for plotting and calling our RK4 integrator with the `dynamics` function from Task 7. Each iteration, we update the value of the control torque ${}^B\mathbf{u}$ before passing it to the integrator. Note that as said earlier in the report, we are always checking for the shadow set condition in all of these simulations.

Finally, we can view our results:

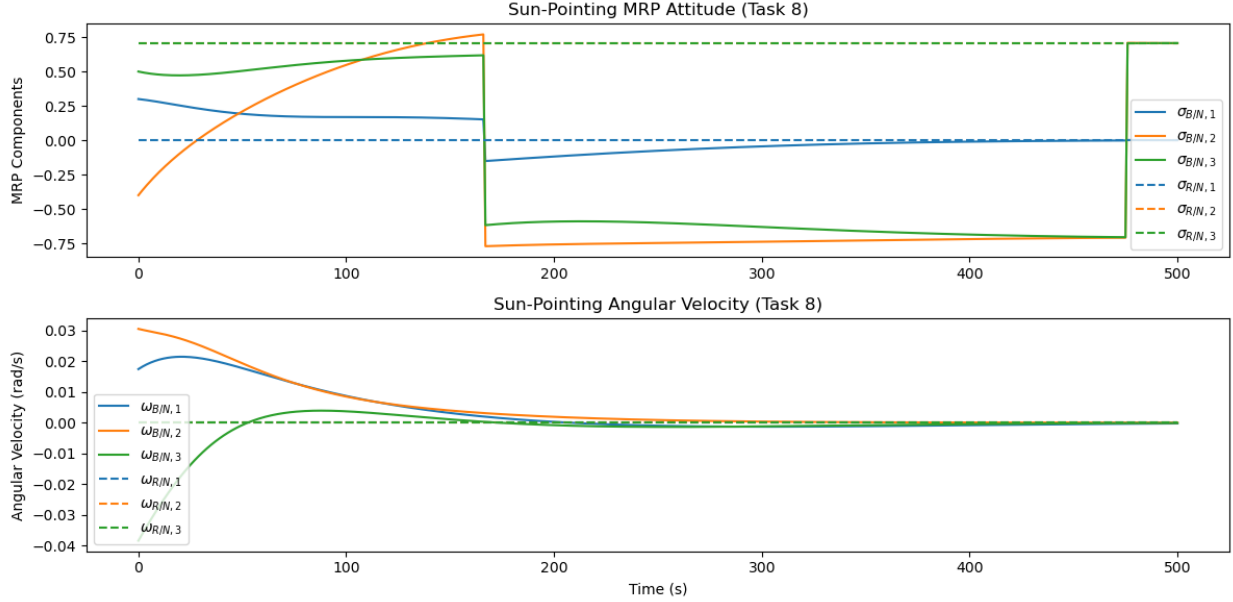


Fig. 8 Attitude and angular velocity plotted against their reference values for a 500s simulation of our control torque driving the craft to the sun-pointing reference frame.

In this simulation we see the attitudes and velocities being driven to their reference values (dotted lines) as expected. Here, the attitudes are constant straight lines since, as we found Task 3, the sun pointing reference from is not a function of time. However, something interesting is also happening here. If a closer look is taken at the MRPs, it can be observed that they actually flip to being driven to their shadow set version of the reference frame at ~180 seconds. Looking into this, we investigate the sun-pointing reference frame defined in Task 3 to find that its MRP equivalent is

$$\sigma_{R_s/N} = \begin{bmatrix} 0 \\ 0.70710678 \\ 0.70710678 \end{bmatrix}$$

Taking the magnitude of this MRP set gives us $|\sigma| = 0.99999999664$. Considering that the shadow set threshold for the norm is exactly 1, we see that we are only a few billionths of a decimal off from switching to the shadow set for this reference frame attitude set. This explains the odd behavior in the plot. Basically, as the spacecraft is being driven to the shadow set threshold, tiny numerical errors on the order of 1×10^{-9} cause the attitude representation to flip sets. We can see further down the simulation it flips back to the reference value. This will likely continuously occur if we extend our simulation. Remember that this is just the representation of the attitude state. Since these flips do not signify the spacecraft undergoing any significant rotation, these are a perfectly fine anomaly to see in our plots. Furthermore, if it was desired, we could even mitigate this effect by adjusting our shadow set threshold from its current value of 1, to a number slightly larger than 1 e.g. $1 + 1 \times 10^{-7}$. This should keep other results mostly the same, while not allowing this particular case to flip back and forth since the numerical precision error that would cause shadow set flipping on this attitude set is on an order that is smaller than 1×10^{-7} .

Looking at the angular velocity values, we can see they are all driven to 0, which is what we expect for the the sun-pointing frame. Here, the green dotted line is the only reference line shown, since all the reference values for

angular velocity are exactly 0 in this case.

Next, we are tasked with providing specific MRP values at various time steps. Doing so gives the following results:

$\sigma_{B/N}(15 \text{ s})$:

$$\begin{bmatrix} 0.2656 \\ -0.1598 \\ 0.4733 \end{bmatrix}$$

$\sigma_{B/N}(100 \text{ s})$:

$$\begin{bmatrix} 0.1688 \\ 0.5482 \\ 0.5789 \end{bmatrix}$$

$\sigma_{B/N}(200 \text{ s})$:

$$\begin{bmatrix} -0.1181 \\ -0.7579 \\ -0.5915 \end{bmatrix}$$

$\sigma_{B/N}(400 \text{ s})$:

$$\begin{bmatrix} -0.0101 \\ -0.7188 \\ -0.6861 \end{bmatrix}$$

Finally, we plug in these results to Coursera and see that our simulations and derivations are all working correctly.

IX. Task 9: Nadir Pointing Control

The next two tasks, including this one, are very similar to Task 8. We do the exact same process we did there, but are now running the simulation for the remaining two reference frames. Here, we will simulate the nadir-pointing frame \mathcal{R}_n . Our K and P values have already been derived, and we use them again here, and for the remainder of the assignment. We also keep our PD control the same as well.

We start our simulation loop again, saving states and rates and calling the RK4 integrator. The main difference for this task is that the `PD_controller` DCM $[RN]$ is the nadir-pointing DCM $[R_nN]$ and the value passed as `omega_rn` is ${}^N\omega_{R_n/N}$. Afterwards, we can plot our results:

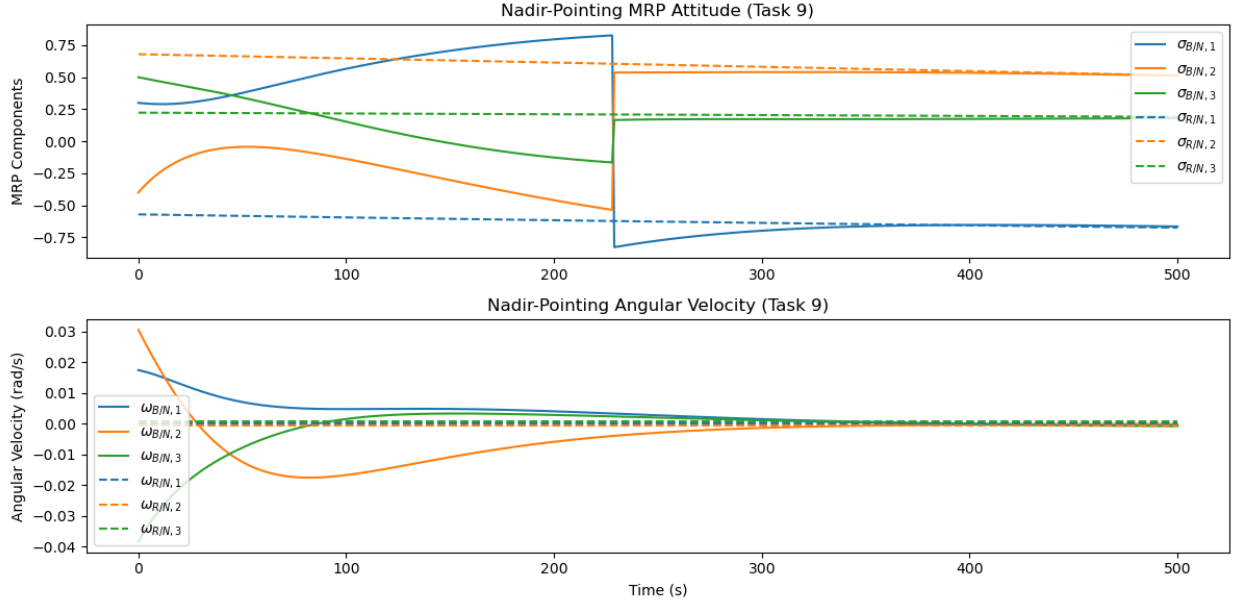


Fig. 9 Attitude and angular velocity plotted against their reference values for a 500s simulation of our control torque driving the craft to the Nadir-pointing reference frame.

These plots are in line with all of our expectations. We can see the attitude being driven to its reference value, and the angular velocities being driven to their near 0 reference values as well. Clearly our control torque is working correctly. Note that while the attitudes look flat, they have a very subtle slope/curvature to them. This contrasts with the actual flat lines seen in the sun-pointing analysis. This is expected as the attitude should be changing very slowly over time, indirectly proportional to the LMO's orbital period for nadir-pointing. This also explains why the angular velocities are near 0, since we are rotating at a relatively slow rate. If you look closely, you can see the reference velocities have small fluctuations around 0 rad/s, as expected. We can also observe some noticeable under-damping in the ${}^B\omega_2$ value. This is expected since this is about the axis of least inertia for our spacecraft. This is fine since we are still meeting our control law requirement of being critically damped or under-damped.

We are also tasked with providing specific MRP values for this task. We need the values at the same time-steps as Task 8. The results are as follows:

$\sigma_{B/N}(15\text{ s})$:

$$\begin{bmatrix} 0.2911 \\ -0.1912 \\ 0.4535 \end{bmatrix}$$

$\sigma_{B/N}(100\text{ s})$:

$$\begin{bmatrix} 0.5661 \\ -0.1374 \\ 0.1522 \end{bmatrix}$$

$\sigma_{B/N}(200\text{ s})$:

$$\begin{bmatrix} 0.7958 \\ -0.4598 \\ -0.1265 \end{bmatrix}$$

$\sigma_{B/N}(400 \text{ s})$:

$$\begin{bmatrix} -0.6528 \\ 0.5349 \\ 0.1746 \end{bmatrix}$$

Again, we plug these into Coursera and confirm everything is working as expected.

X. Task 10: GMO Pointing Control

For this task, we complete the reference frame simulations with the GMO/Communication reference frame \mathcal{R}_c . The same details regarding control torque and simulation parameters that were discussed in Tasks 8 and 9 also apply here. Again, the only difference for this task is that for the `PD_controller` function, the DCM $[RN]$ is the GMO-pointing DCM $[R_cN]$ and the value passed as `omega_rn` is ${}^N\omega_{R_c/N}$. Using this, we start our simulation and save our state and rates for each time step, the same as before.

Afterwards, we plot our results:

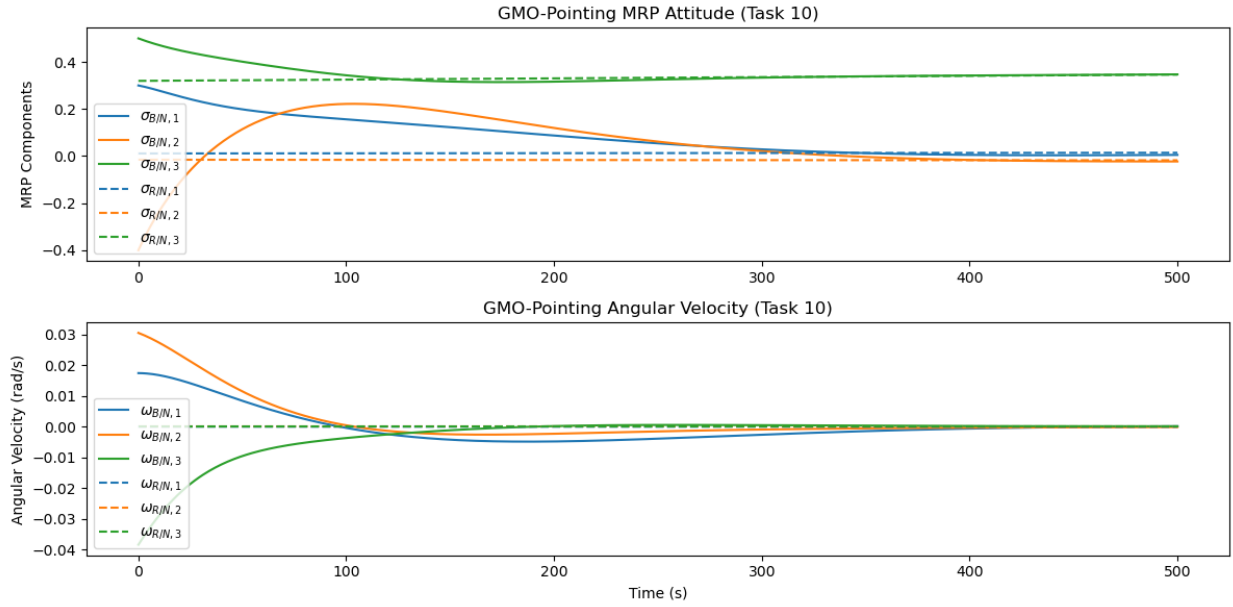


Fig. 10 Attitude and angular velocity plotted against their reference values for a 500s simulation of our control torque driving the craft to the GMO-pointing reference frame.

Here, we see similar results to Tasks 8 and 9. The states and rates are driven to their reference values. We can observe some noticeable under-damping in the σ_2 component which is acceptable. The angular velocity values are all minimally under-damped and very close to critical damping. Similarly to Task 9, the reference attitudes appear flat, but do have a measurable slope/curvature. Since we are GMO pointing, the reference value rates are indirectly proportional to both the LMO and GMO orbital periods. The same is true for the angular velocities. They are near 0 since we are rotating slowly, but do have real values that are changing over time.

Last, we are tasked with providing specific MRP values at various time steps. Doing so gives the following results:

$\sigma_{B/N}(15 \text{ s})$:

$$\begin{bmatrix} 0.2654 \\ -0.1688 \\ 0.4595 \end{bmatrix}$$

$\sigma_{B/N}(100 \text{ s})$:

$$\begin{bmatrix} 0.1561 \\ 0.2216 \\ 0.3432 \end{bmatrix}$$

$\sigma_{B/N}(200 \text{ s})$:

$$\begin{bmatrix} 0.0873 \\ 0.1193 \\ 0.3162 \end{bmatrix}$$

$\sigma_{B/N}(400 \text{ s})$:

$$\begin{bmatrix} 0.0050 \\ -0.0165 \\ 0.3424 \end{bmatrix}$$

Finally, we plug in these results to Coursera and see that our simulations and derivations are all working correctly.

XI. Task 11: Mission Scenario Simulation

Finally we arrive at our final task: full mission scenario simulation.

This task differs from the previous 3 in that we are finally going to control the mode switching of the spacecraft during our simulation loop. In previous tasks, we assumed a single reference frame (or no reference in Task 7) and stayed with it for the entire duration of the simulation. Here, we need to implement logic to switch reference frames when certain criteria are met. When that happens, the the DCM $[RN]$ and ω_{rn} values for the `PD_controller` function are updated to the new reference frame. This allows our control law to always drive us to our currently desired reference frame.

To implement these mode-switching criteria, we can look back to Table 1 from the Introduction. This gives an overview of the control logic. We will write a function called `determine_control_mode(r_LMO_inertial, r_GMO_inertial)` to implement this logic. The two arguments represent the positional vectors of each spacecraft, ${}^N\mathbf{r}_{LMO}$ and ${}^N\mathbf{r}_{GMO}$. To get these two vectors, we of course use our function from Task 1, `getInertialPositionVectors`, which was written to do this exact task. For a full description, refer to Task 1. Once we get both position vectors returned, we are ready to call our `determine_control_mode` function. Now we just need to figure out how it should work.

Looking at the table, we see that if the satellite is on the sunlit side of Mars, it will always be charging. To check this, we need the ${}^N\mathbf{r}_{LMO}$ vector. Since this vector is expressed with N frame components, all we need to do here is check if the second (\hat{n}_2) component of the vector is > 0 . If it is, we are guaranteed to be in charging mode, and our reference frame \mathcal{R} will be the sun-pointing frame \mathcal{R}_s .

If this check fails, we know we are on the dark side of Mars. Here, we always want to first attempt communication with the GMO satellite, but only if we are within our 35° angular difference threshold for communication with the GMO satellite. To check this, we need both the ${}^N\mathbf{r}_{LMO}$ and ${}^N\mathbf{r}_{GMO}$ vectors. We know that

$$\cos(\theta) = \frac{{}^N\mathbf{r}_{LMO} \cdot {}^N\mathbf{r}_{GMO}}{|{}^N\mathbf{r}_{LMO}| |{}^N\mathbf{r}_{GMO}|}$$

Here, θ is the angle between the two satellite position vectors. Since we have all the information for the right hand side of the equation, we simply take the arccos to solve for the angle θ . We then check if $\theta < 35$. If it is, we will be in communication mode and our reference frame \mathcal{R} will be the GMO-pointing frame \mathcal{R}_c .

If neither of these criteria are met, then we know we are on the dark side of Mars and out of range for communication with GMO. If this is the case, we default to pointing our sensors in the nadir direction to get science data from Mars' surface. This means our reference frame \mathcal{R} will be the nadir-pointing frame \mathcal{R}_n .

We implement these criteria with an `if-else` block in our function, giving us the logic needed for switching control modes of the LMO satellite. Finally, our `determine_control_mode` function is ready.

We start our simulation and save our state and rates for each time step, the same as before. This time, we are simulating for 6500 seconds. Remember, we do everything the same as before, the only difference for this task is we are switching the reference frame DCM $[RN]$ and ω_{rn} values for the `PD_controller`.

Afterwards, we plot our results:

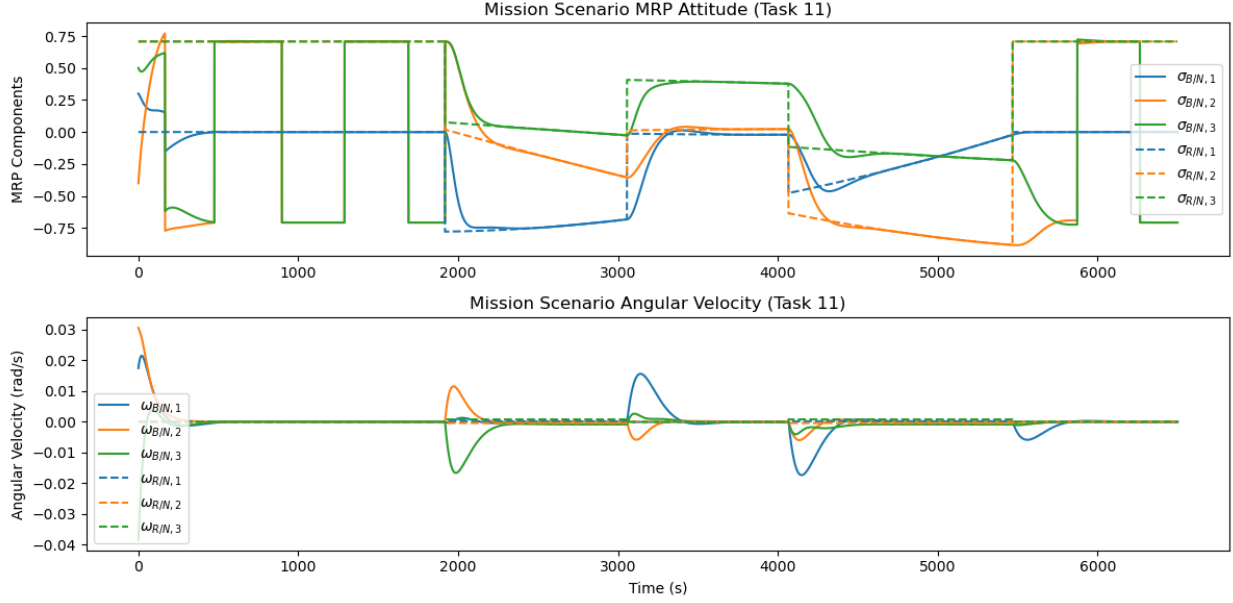


Fig. 11 Attitude and angular velocity plotted against their reference values for a 6500s simulation of our control torque driving the craft to the varying reference frames detailed in our mission.

Looking at this figure gives interesting, but expected, results. Comparing to the figures presented in Tasks 8-10, we can identify each of the modes. We start in the sun-pointing mode, and undergo a total of 4 control mode changes during our 6500s mission simulation time. The control mode changes are evident in both plots, but each mode change is most obviously distinguishable in the angular velocity chart. Each blip in the plot represents our control torque changing, and shows how our satellite is re-orienting into the next reference frame.

The first mode change can be seen at ~2000 seconds. Looking back at the plots in the previous tasks, we see this is to the nadir reference frame. Here, we can finally see the slopes in the attitudes that have been discussed previously. This is because we are running a longer simulation and thus have a zoomed out view.

At just over ~3000 seconds, we see our second mode change. Again, this can be confirmed as the GMO-pointing mode. We can see the reference MRP values match those of our Task 10 plot perfectly. The MRP slope/rate of change is very subtle here, which aligns with our results in Task 5, which determined that the angular velocity values of the GMO-pointing frame were very small, on the order of 1×10^{-5} rad/s.

At roughly ~4000 seconds, we see our third switch. This time we can tell it is going back to the nadir-pointing reference. We can observe how in both of the nadir modes in the plot, the attitudes have a sharper slopes compared to GMO mode, which aligns with our results for the nadir mode angular velocity values from Task 4 as well.

Finally, we have our final mode switch, back to sun-pointing mode, at ~5500 seconds. In both this instance and in the sun-pointing instance at the beginning of the simulation, we can see the MRP shadow set switching discussed extensively in Task 9. This helps further confirm what mode we are looking for these two segments of the simulation. The spacecraft appears to stay in this mode until 6500 seconds, when our simulation ends.

All of these findings and discussions also help explain the angular velocity plot seen on the bottom. For this plot, we can see the perfectly 0 (green) reference values for both of the sun-pointing modes. When we switch to the nadir modes, we see disturbances and fluctuations in the reference velocity values, confirming our small but non 0 values calculated during previous simulations. Finally, for the GMO-pointing mode, we see nearly invisible fluctuations, but are just able to make out the switching of the green and orange color on the plot.

Our last observation concerns the damping. Looking at the plot, we see successful under- or critical damping for all mode switching across every axis of the spacecraft. The values are driven to their reference values, but don't exhibit over-damped behavior or any highly visible under-damping. This is exactly what we designed our control law to do, confirming that our simulation implementation is working correctly.

For our final requirement, we are tasked with providing specific MRP values at various time steps. Doing so gives

the following results:

$\sigma_{B/N}(300 \text{ s})$:

$$\begin{bmatrix} -0.0442 \\ -0.7386 \\ -0.6307 \end{bmatrix}$$

$\sigma_{B/N}(2100 \text{ s})$:

$$\begin{bmatrix} -0.7458 \\ 0.1139 \\ 0.1581 \end{bmatrix}$$

$\sigma_{B/N}(3400 \text{ s})$:

$$\begin{bmatrix} 0.0132 \\ 0.0398 \\ 0.3907 \end{bmatrix}$$

$\sigma_{B/N}(4400 \text{ s})$:

$$\begin{bmatrix} -0.4331 \\ -0.7323 \\ -0.1877 \end{bmatrix}$$

$\sigma_{B/N}(5600 \text{ s})$:

$$\begin{bmatrix} -0.0012 \\ -0.8260 \\ -0.5044 \end{bmatrix}$$

Finally, we plug in these results to Coursera and see that our simulations and derivations are all working correctly.

XII. Conclusion

This report presents the complete development and analysis of a nano-satellite's attitude dynamics and control system in a Mars-centered orbit. Through Tasks 1–11, we designed, implemented, and verified a comprehensive simulation framework capable of modeling the satellite's behavior across multiple mission modes—science, charging, and communication.

Starting with orbital mechanics, we derived inertial positions and velocities from orbital elements and constructed key reference frames including sun-pointing, nadir-pointing, and GMO-pointing attitudes. By leveraging modified Rodrigues parameters (MRPs) and associated angular velocity vectors, we were able to represent spacecraft attitude robustly while accounting for shadow set switching to avoid singularities.

A custom Runge-Kutta 4 integrator was implemented to propagate both attitude and angular velocity over time, and we developed a PD control law that stabilized the spacecraft orientation with respect to changing reference frames. Simulation results from Tasks 7 through 11 demonstrated the dynamic response of the spacecraft under both uncontrolled and controlled conditions. The controller was successfully tuned to achieve critical or under-damped performance while satisfying specified decay times and mode-switching criteria.

In the final mission scenario, the spacecraft responded appropriately to environmental and positional cues, autonomously switching between operational modes. Our results showed effective convergence to reference frames, minimal overshoot, and accurate long-duration tracking—all validated both visually through plots and quantitatively by extracting and verifying state values at specific time steps.

Overall, this project provided valuable hands-on experience in space vehicle dynamics, attitude control, and reference frame logic. The combination of analytical derivations, numerical methods, and simulation-based validation offers a rigorous and realistic demonstration of how modern spacecraft achieve precise orientation and reconfiguration in planetary orbits.

Appendix

A. Code Listing

```
1 import numpy as np
2 from sympy import symbols, pprint, cos, sin, Matrix, pi
3 import sympy as sp
4 import matplotlib.pyplot as plt
5
6 ##### Welcome #####
7 print("Welcome to the ASEN 5010 Capstone Project")
8 print("Attitude Dynamics and Control of a Nano-Satellite Orbiting Mars")
9
10 ##### Initial state #####
11 h = 400 # km
12 R_mars = 3396.19 # km
13 r_lmo = R_mars + h
14 mu = 42828.3 # km^3/s^2
15 theta_lmo_rate = 0.000884797 # rad/s
16 mars_period_min = (24 * 60) + 37
17 mars_period_sec = mars_period_min * 60
18 mars_period_hr = mars_period_min / 60
19 gmo_period_sec = mars_period_sec
20 r_gmo = 20424.2 # km
21 theta_gmo_rate = 0.0000709003 # rad/s
22 tmax = 6500 # Set the value of tmax
23 dt = 0.1 # Set the value of t
24 t_0 = 0.0 # Initial time
25 sigma_bn_0 = np.array([0.3, -0.4, 0.5]) # MRPs
26 omega_bn_0 = np.array([1.00, 1.75, -2.20]) # deg/s
27 omega_bn_0_rad = np.deg2rad(omega_bn_0)
28 I_b = np.array([[10, 0, 0], [0, 5, 0], [0, 0, 7.5]]) # kg*m^2
29 I_b_inv = np.linalg.inv(I_b)
30 X_0 = np.concatenate((sigma_bn_0, omega_bn_0_rad)) # Initial conditions
31 omega_lmo_0 = np.deg2rad(20)
32 i_lmo_0 = np.deg2rad(30)
33 theta_lmo_0 = np.deg2rad(60) # function of time
34 omega_gmo_0 = 0
35 i_gmo_0 = 0
36 theta_gmo_0 = np.deg2rad(250) # function of time
37 comm_angle_threshold = 35 # deg
38
39
40 ##### Helper Functions #####
41 def writeToFile(path, data):
42     str_to_write = ""
43
44     # Convert scalar to list
45     if np.isscalar(data):
46         str_to_write = str(data)
47     else:
48         if data.ndim == 2:
49             for row in data:
50                 for element in row:
51                     str_to_write += str(element) + " "
52         else:
53             for element in data:
54                 str_to_write += str(element) + " "
55     with open(path, "w+") as file:
56         file.write(str_to_write.rstrip())
57
58
59 def theta_lmo(t):
60     return theta_lmo_0 + t * theta_lmo_rate
61
62
63 def theta_gmo(t):
```

```

64     return theta_gmo_0 + t * theta_gmo_rate
65
66
67 def s(theta):
68     return np.sin(theta)
69
70
71 def c(theta):
72     return np.cos(theta)
73
74
75 # Returns a skew-symmetric matrix from a vector
76 def tilde(v):
77     return np.array([[0, -v[2], v[1]], [v[2], 0, -v[0]], [-v[1], v[0], 0]])
78
79
80 def Euler313toDCM(t1, t2, t3):
81     # Convert 313 angles to DCM - From appendix B.1
82     return np.array(
83         [
84             [
85                 c(t3) * c(t1) - s(t3) * c(t2) * s(t1),
86                 c(t3) * s(t1) + s(t3) * c(t2) * c(t1),
87                 s(t3) * s(t2),
88             ],
89             [
90                 -s(t3) * c(t1) - c(t3) * c(t2) * s(t1),
91                 -s(t3) * s(t1) + c(t3) * c(t2) * c(t1),
92                 c(t3) * s(t2),
93             ],
94             [s(t2) * s(t1), -s(t2) * c(t1), c(t2)],
95         ]
96     )
97
98
99 def DCM2Quaternion(C):
100     # Initialize stuff
101     B = np.zeros(4) # B^2 array
102     b = np.zeros(4) # resulting set of quaternions
103     trc = np.trace(C)
104
105     B[0] = (1 + trc) / 4
106     B[1] = (1 + 2 * C[0, 0] - trc) / 4
107     B[2] = (1 + 2 * C[1, 1] - trc) / 4
108     B[3] = (1 + 2 * C[2, 2] - trc) / 4
109
110     # Find the index of the maximum value in B2
111     i = np.argmax(B)
112
113     # Calculate quaternion based on sheppard's method
114     if i == 0:
115         b[0] = np.sqrt(B[0])
116         b[1] = (C[1, 2] - C[2, 1]) / (4 * b[0])
117         b[2] = (C[2, 0] - C[0, 2]) / (4 * b[0])
118         b[3] = (C[0, 1] - C[1, 0]) / (4 * b[0])
119     elif i == 1:
120         b[1] = np.sqrt(B[1])
121         b[0] = (C[1, 2] - C[2, 1]) / (4 * b[1])
122         if b[0] < 0:
123             b[1] = -b[1]
124             b[0] = -b[0]
125         b[2] = (C[0, 1] + C[1, 0]) / (4 * b[1])
126         b[3] = (C[2, 0] + C[0, 2]) / (4 * b[1])
127     elif i == 2:
128         b[2] = np.sqrt(B[2])
129         b[0] = (C[2, 0] - C[0, 2]) / (4 * b[2])
130         if b[0] < 0:
131             b[2] = -b[2]

```

```

132         b[0] = -b[0]
133         b[1] = (C[0, 1] + C[1, 0]) / (4 * b[2])
134         b[3] = (C[1, 2] + C[2, 1]) / (4 * b[2])
135     elif i == 3:
136         b[3] = np.sqrt(B[3])
137         b[0] = (C[0, 1] - C[1, 0]) / (4 * b[3])
138         if b[0] < 0:
139             b[3] = -b[3]
140             b[0] = -b[0]
141         b[1] = (C[2, 0] + C[0, 2]) / (4 * b[3])
142         b[2] = (C[1, 2] + C[2, 1]) / (4 * b[3])
143
144     return b
145
146
147 def MRP2DCM(sigma):
148     tilde_sigma = tilde(sigma)
149     return np.eye(3) + (
150         8 * tilde_sigma @ tilde_sigma - 4 * (1 - sigma @ sigma) * tilde_sigma
151     ) / ((1 + sigma @ sigma) ** 2)
152
153
154 def checkShadowSet(sigma):
155     if np.linalg.norm(sigma) > 1:
156         return -sigma / (sigma @ sigma)
157     else:
158         return sigma
159
160
161 def DCM2MRP(C):
162     b = DCM2Quaternion(C)
163     divisor = 1 + b[0]
164     sigma = np.array([b[1], b[2], b[3]]) / divisor
165     return checkShadowSet(sigma)
166
167
168 ##### Task 1: Orbit Simulation (5 points) #####
169 # pos = r*i_r
170 # Derive inertial s/c velocity r_dot.
171 # Note that for circular orbits, theta_dot is constant
172 print("\n\nBEGIN TASK 1")
173
174
175 # Write a function whose inputs are radius r and 313 angles omega, i, theta,
176 # and outputs are the inertial pos vector N_r and vel N_r_dot
177 # Calculate the inertial position vector N_r and velocity N_r_dot
178 def getInertialPositionVectors(r, omega, i, theta):
179     # O : {i_r, i_theta, i_h} aka H frame
180     # N : {n_1, n_2, n_3}
181     ON = Euler313toDCM(omega, i, theta)
182     NO = ON.T
183     # Convert direction of i_r to N
184     N_r = NO @ np.array([r, 0, 0])
185     # Convert direction of i_theta to N
186     if r == r_lmo:
187         N_r_dot = NO @ np.array([0, r * theta_lmo_rate, 0])
188     if r == r_gmo:
189         N_r_dot = NO @ np.array([0, r * theta_gmo_rate, 0])
190     return N_r, N_r_dot
191
192
193 # confirm the operation by checking getInertialVectors(r_lmo, omega_lmo, i_lmo, theta_lmo(450))
194 # and getInertialVectors(r_gmo, omega_gmo, i_gmo, theta_gmo(1150))
195 N_r_lmo, N_r_lmo_dot = getInertialPositionVectors(
196     r_lmo, omega_lmo_0, i_lmo_0, theta_lmo(450)
197 )
198 N_r_gmo, N_r_gmo_dot = getInertialPositionVectors(
199     r_gmo, omega_gmo_0, i_gmo_0, theta_gmo(1150)

```



```

200 )
201 print("rLMO = ", N_r_lmo)
202 print("vLMO = ", N_r_lmo_dot)
203 print("rGMO = ", N_r_gmo)
204 print("vGMO = ", N_r_gmo_dot)
205 writeToFile("./tasks/task 1/rLMO.txt", N_r_lmo)
206 writeToFile("./tasks/task 1/vLMO.txt", N_r_lmo_dot)
207 writeToFile("./tasks/task 1/rGMO.txt", N_r_gmo)
208 writeToFile("./tasks/task 1/vGMO.txt", N_r_gmo_dot)
209
210
211 ##### Task 2: Orbit Frame Orientation (5 points) #####
212 print("\n\nBEGIN TASK 2")
213 Omega, i = symbols("Omega i")
214 t = symbols("t", positive=True)
215 theta = sp.Function(symbols("theta"))
216 HN = Matrix(
217     [
218         [
219             cos(theta(t)) * cos(Omega) - sin(theta(t)) * cos(i) * sin(Omega),
220             cos(theta(t)) * sin(Omega) + sin(theta(t)) * cos(i) * cos(Omega),
221             sin(theta(t)) * sin(i),
222         ],
223         [
224             -sin(theta(t)) * cos(Omega) - cos(theta(t)) * cos(i) * sin(Omega),
225             -sin(theta(t)) * sin(Omega) + cos(theta(t)) * cos(i) * cos(Omega),
226             cos(theta(t)) * sin(i),
227         ],
228         [sin(i) * sin(Omega), -sin(i) * cos(Omega), cos(i)],
229     ]
230 )
231 with open("./latex/task_2_HN.tex", "w+") as file:
232     file.write((sp.latex(HN)))
233
234
235 # Write a function whose input is time t and output is DCM HN(t) for the LMO
236 def getHNforLMO(t):
237     return Euler313toDCM(omega_lmo_0, i_lmo_0, theta_lmo(t))
238
239
240 # Validate the operation by computing HN(300)
241 t = 300 # sec
242 HN_at_t = getHNforLMO(t)
243 print("HN(t = " + str(t) + "s) = ", HN_at_t)
244 writeToFile("./tasks/task 2/HN.txt", HN_at_t)
245
246
247 ##### Task 3: Sun-Pointing Reference Frame Orientation (10 points)
248 ↪ #####
249 print("\n\nBEGIN TASK 3")
250 # First determine and analytic expression for Rs by defining DCM [RsN]
251 RsN = np.array([-1, 0, 0], [0, 0, 1], [0, 1, 0])
252 with open("./latex/RsN.tex", "w+") as file:
253     file.write((sp.latex(RsN)))
254
255
256 # Write a function that returns RsN
257 def getRsN():
258     return RsN
259
260 def getOmegaRsN():
261     return np.array([0, 0, 0])
262
263
264 # Validate the evaluation of RsN by providing numerical values for t=0s
265 print("RsN(t = 0s) = ", getRsN())
266 writeToFile("./tasks/task 3/RsN.txt", getRsN())

```

```

267
268 # Angular velocity is [0, 0, 0] since the DCM is not a function of time
269 print("N_Rn/N = ", getOmegaRsN())
270 writeToFile("./tasks/task 3/omega_rs_n.txt", getOmegaRsN())
271
272
273 ##### Task 4: Nadir-Pointing Reference Frame Orientation (10 points)
274 ↪ #####
275 print("\n\nBEGIN TASK 4")
276 # First determine and analytic expression for Rn by defining DCM [RnN]
277 RnH = np.array([[ -1, 0, 0], [0, 1, 0], [0, 0, -1]])
278 with open("./latex/RnH.tex", "w+") as file:
279     file.write((sp.latex(RnH)))
280 with open("./latex/RnN.tex", "w+") as file:
281     file.write((sp.latex(RnH @ HN)))
282
283 # Write a function that returns RnN
284 def getRnN(t):
285     return RnH @ getHNforLMO(t)
286
287
288 # Write a function that determines angular velocity vector omega_rn_n
289 def getOmegaRnN(t):
290     NRn = getRnN(t).T
291     return NRn @ [0, 0, -theta_lmo_rate] # = _dot*i_h = -_dot*r_3
292
293
294 t = 330 # sec
295 # Validate the evaluation of RnN by providing numerical values for t = 330s
296 RnN_at_t = getRnN(t)
297 print("RnN(t = " + str(t) + "s) = ", RnN_at_t)
298 writeToFile("./tasks/task 4/RnN.txt", RnN_at_t)
299
300 # What is the angular velocity @ t = 330s
301 omega_rnn_at_t = getOmegaRnN(t)
302 print("N_Rn/N(t = " + str(t) + "s) = ", omega_rnn_at_t)
303 writeToFile("./tasks/task 4/omega_rn_n.txt", omega_rnn_at_t)
304
305
306 ##### Task 5: GMO-Pointing Reference Frame Orientation (10 points)
307 ↪ #####
308 print("\n\nBEGIN TASK 5")
309
310 # dr = r_gmo - r_lmo => lets get these in N frame to make cross product easy
311 # H : {i_r, i_theta, i_h} - Note theres one for GMO, one for LMO, depending on theta
312 # N : {n_1, n_2, n_3}
313 # Write a function that returns RcN
314 def getRcNExpr():
315     NH = HN.T
316     theta_gmo_expr = sp.Function(symbols("theta_GMO"))
317     theta_lmo_expr = sp.Function(symbols("theta_LMO"))
318     NH_gmo = NH.subs([(Omega, omega_gmo_0), (i, i_gmo_0), (theta, theta_gmo_expr)])
319     NH_lmo = NH.subs([(Omega, omega_lmo_0), (i, i_lmo_0), (theta, theta_lmo_expr)])
320     N_r_gmo_col = NH_gmo @ sp.Matrix([r_gmo, 0, 0])
321     N_r_lmo_col = NH_lmo @ sp.Matrix([r_lmo, 0, 0])
322
323     N_dr_col = N_r_gmo_col - N_r_lmo_col
324     N_r1_col = -N_dr_col.normalized()
325     N_r2_col = (N_dr_col.cross(sp.Matrix([0, 0, 1]))).normalized()
326     N_r3_col = N_r1_col.cross(N_r2_col).normalized()
327
328     return (sp.Matrix.hstack(N_r1_col, N_r2_col, N_r3_col)).T
329
330
331 def getRcN(t):
332     RcN = getRcNExpr()

```

```

333     RcN_f = sp.lambdify(
334         ["t"],
335         RcN,
336         modules=["numpy", {"theta_GMO": theta_gmo}, {"theta_LMO": theta_lmo}],
337     )
338     return RcN_f(t)
339
340
341 # Write a function that determines angular velocity vector omega_rc_n
342 def getOmegaRcNAnalytically(time):
343     t = symbols("t", positive=True)
344     RcN = getRcNExpr()
345     # RcNT = RcN.T
346
347     # Replace with base functions so sympy knows how to derive wrt t
348     theta_gmo_expr = sp.Function(symbols("theta_GMO"))
349     theta_lmo_expr = sp.Function(symbols("theta_LMO"))
350     replace_gmo = theta_gmo_0 + t * theta_gmo_rate
351     replace_lmo = theta_lmo_0 + t * theta_lmo_rate
352     replace_dict = {theta_gmo_expr(t): replace_gmo, theta_lmo_expr(t): replace_lmo}
353     RcN_rep = RcN.subs(replace_dict)
354
355     # Now sympy can take derivative wrt t
356     RcN_dot = sp.diff(RcN_rep, t)
357     omega_tilde = -RcN_dot @ RcN.T
358     # with open('./latex/insane_matrix.tex', "w+") as file:
359     #     file.write((sp.latex(omega_tilde)))
360     omega_tilde_f = sp.lambdify(
361         ["t"],
362         omega_tilde,
363         modules=["numpy", {"theta_GMO": theta_gmo}, {"theta_LMO": theta_lmo}],
364     )
365
366     ssm = omega_tilde_f(time)
367     ssm = (ssm - ssm.T) / 2 # Force diagonals to 0
368     R_omega_rcn = sp.Matrix([-ssm[1, 2], ssm[0, 2], -ssm[0, 1]])
369     N_omega_rcn = RcN.T @ R_omega_rcn
370     N_omega_rcn_f = sp.lambdify(
371         ["t"],
372         N_omega_rcn,
373         modules=["numpy", {"theta_GMO": theta_gmo}, {"theta_LMO": theta_lmo}],
374     )
375     N_omega_rcn_real = N_omega_rcn_f(time)
376     return N_omega_rcn_real.flatten()
377
378
379 def getOmegaRcN(time):
380     dt = 1e-6
381
382     RcN_t = getRcN(time)
383     RcN_plus = getRcN(time + dt)
384     RcN_minus = getRcN(time - dt)
385     RcN_dot = (RcN_plus - RcN_minus) / (2 * dt)
386
387     ssm = -RcN_dot @ RcN_t.T
388     ssm = (ssm + ssm) / 2 # Force diagonals to 0
389     R_omega_rcn = np.array([-ssm[1, 2], ssm[0, 2], -ssm[0, 1]])
390     N_omega_rcn = RcN_t.T @ R_omega_rcn
391     return N_omega_rcn
392
393
394 t = 330
395 RcN_at_t = getRcN(t)
396 omega_RcN_num_at_t = getOmegaRcN(t)
397 omega_RcN_anal_at_t = getOmegaRcNAnalytically(t)
398 print("RcN = ", RcN_at_t)
399 print("Numerical = ", omega_RcN_num_at_t)
400 print("Analytical = ", omega_RcN_anal_at_t)

```

```

401 writeToFile("./tasks/task 5/RcN.txt", RcN_at_t)
402 writeToFile("./tasks/task 5/omega_rc_n_num.txt", omega_RcN_num_at_t)
403 writeToFile("./tasks/task 5/omega_rc_n_anal.txt", omega_RcN_anal_at_t)
404
405
406 ##### Task 6: Attitude Error Evaluation (10 points) #####
407 print("\n\nBEGIN TASK 6")
408
409
410 # Write function that returns tracking errors sigma_br and omega_br
411 def getTrackingErrors(t, sigma_bn, B_omega_bn, RN, N_omega_rn):
412     # Get _BR from _BN and RN DCM
413     BN = MRP2DCM(sigma_bn)
414     BR = BN @ (RN.T)
415     sigma_br = DCM2MRP(BR)
416
417     # Get _br from _bn and _rn
418     B_omega_br = B_omega_bn - (BN @ N_omega_rn)
419
420     return sigma_br, B_omega_br
421
422
423 # Sun-pointing
424 t = 0
425 sigma, omega = getTrackingErrors(t, sigma_bn_0, omega_bn_0_rad, getRsN(), getOmegaRsN())
426 print("Sun-Pointing Orientation")
427 print("_B/R = ", sigma)
428 print("_B/R = ", omega)
429 writeToFile("./tasks/task 6/sun-sigma.txt", sigma)
430 writeToFile("./tasks/task 6/sun-omega.txt", omega)
431
432 # Nadir-pointing
433 sigma, omega = getTrackingErrors(
434     t, sigma_bn_0, omega_bn_0_rad, getRnN(t), getOmegaRnN(t)
435 )
436 print("Nadir-Pointing Orientation")
437 print("_B/R = ", sigma)
438 print("_B/R = ", omega)
439 writeToFile("./tasks/task 6/nad-sigma.txt", sigma)
440 writeToFile("./tasks/task 6/nad-omega.txt", omega)
441
442 # GMO-pointing
443 sigma, omega = getTrackingErrors(
444     t, sigma_bn_0, omega_bn_0_rad, getRcN(t), getOmegaRcN(t)
445 )
446 print("GMO-Pointing Orientation")
447 print("_B/R = ", sigma)
448 print("_B/R = ", omega)
449 writeToFile("./tasks/task 6/gmo-sigma.txt", sigma)
450 writeToFile("./tasks/task 6/gmo-omega.txt", omega)
451
452
453 ##### Task 7: Numerical Attitude Simulator (10 points) #####
454 print("\n\nBEGIN TASK 7")
455
456 # Write your own numerical integrator using RK45
457
458
459 # Define the spacecraft dynamics (equation of motion)
460 def dynamics(X, dt, u):
461     sigma_BN = X[:3] # MRP attitude
462     sigma_BN_skew = tilde(sigma_BN)
463     sigma_BN = sigma_BN.reshape((3, 1)) # make col
464     omega_BN = X[3:] # Angular velocity
465     omega_BN_skew = tilde(omega_BN)
466     omega_BN = omega_BN.reshape((3, 1)) # make col
467     if isinstance(u, np.ndarray):
468         u = u.reshape((3, 1))

```

```

469     d_omega_BN = I_b_inv @ (-omega_BN_skew @ (I_b @ omega_BN) + u)
470     d_sigma_BN = (
471         0.25
472         * (
473             (1 - (sigma_BN.T @ sigma_BN)) * np.eye(3)
474             + 2 * sigma_BN_skew
475             + 2 * sigma_BN @ sigma_BN.T
476         )
477         @ omega_BN
478     )
479     return np.concatenate((d_sigma_BN.flatten(), d_omega_BN.flatten()))
480
481
482     # Runge-Kutta 4th order integrator
483     def rk4_integrator(f, X, u, dt, tn):
484         k1 = dt * f(X, tn, u)
485         k2 = dt * f(X + (k1 / 2), tn + (dt / 2), u)
486         k3 = dt * f(X + (k2 / 2), tn + (dt / 2), u)
487         k4 = dt * f(X + k3, tn + dt, u)
488         X = X + (1 / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
489         return X
490
491
492     # Time settings
493     dt = 1 # 1 second time step
494     t_final = 500.0 # Total time for the integration
495     time_steps = int(t_final / dt)
496
497     # Control torque (zero initially)
498     u = np.zeros(3) # Control torque vector
499
500     # Arrays to store results for plotting
501     sigma_BN_history = []
502     omega_BN_history = []
503     T_history = []
504     B_H_history = []
505     N_H_history = []
506
507     # Integration loop (u = 0 for this part)
508     X = X_0
509     for t in np.arange(0, t_final + dt, dt):
510         B_sigma_BN = X[:3] # MRP attitude
511         # B_sigma_BN = checkShadowSet(B_sigma_BN)
512         B_omega_BN = X[3:] # Angular velocity
513         sigma_BN_history.append(B_sigma_BN)
514         omega_BN_history.append(B_omega_BN)
515
516         # Kinetic Energy
517         T = 0.5 * (B_omega_BN.T @ (I_b @ B_omega_BN))
518         T_history.append(T)
519
520         # Compute angular momentum H
521         H = I_b @ B_omega_BN
522         B_H_history.append(H)
523         N_H_history.append(MRP2DCM(B_sigma_BN).T @ H)
524
525         # Update attitude using RK4
526         X = rk4_integrator(dynamics, X, u, dt, t)
527
528         X[:3] = checkShadowSet(X[:3])
529
530
531     # Xn, t_500, sigmas, omegas, B_H, N_H, T = RK4(dynamics, X_0, 1, 500, 0)
532     # print(sigmas[0:3, 500])
533     # Results at 500 seconds
534     sigma_BN_500 = sigma_BN_history[-1]
535     omega_BN_500 = omega_BN_history[-1]
536     T_500 = T_history[-1]

```

```

537 B_H_500 = B_H_history[-1]
538 BN_500 = MRP2DCM(sigma_BN_500)
539 N_H_500 = BN_500.T @ B_H_500
540 N_H_500s = N_H_history[-1]
541
542 print(f"MRP attitude at 500s: {sigma_BN_500}")
543 print(f"Kinetic energy at 500s: {T_500}")
544 print(f"B-Frame Angular momentum at 500s: {B_H_500}")
545 print(f"N-Frame Angular momentum at 500s: {N_H_500}")
546 print(f"N-Frame Angular momentum at 500s: {N_H_500s}")
547 writeToFile("./tasks/task 7/sigma_500s.txt", sigma_BN_500)
548 writeToFile("./tasks/task 7/T_500s.txt", T_500)
549 writeToFile("./tasks/task 7/H_500s_B_frame.txt", B_H_500)
550 writeToFile("./tasks/task 7/H_500s_N_frame.txt", N_H_500)
551
552
553 # Now apply control torque u = (0.01, -0.01, 0.02) Nm and integrate again for 100s
554 u_fixed = np.array([0.01, -0.01, 0.02]) # Fixed control torque
555 X = X_0 # Reset initial conditions
556 sigma_BN_100_history = []
557 omega_BN_100_history = []
558 t_final_control = 100 # update to only 100s this time
559 # Run integration with control torque for 100 seconds
560 for t in np.arange(0, t_final_control + dt, dt):
561     B_sigma_BN = X[:3] # MRP attitude
562     B_omega_BN = X[3:]
563     # B_sigma_BN = checkShadowSet(B_sigma_BN)
564
565     sigma_BN_100_history.append(B_sigma_BN)
566     omega_BN_100_history.append(B_omega_BN)
567
568     # Update attitude using RK4
569     X = rk4_integrator(dynamics, X, u_fixed, dt, t)
570
571     X[:3] = checkShadowSet(X[:3])
572
573
574 sigma_BN_100 = sigma_BN_100_history[-1]
575 print(f"MRP attitude at 100s with control torque: {sigma_BN_100}")
576 writeToFile("./tasks/task 7/sigma_100s_with_control.txt", sigma_BN_100)
577
578 # Plot the results for visualization
579 sigma_BN_history = np.array(sigma_BN_history)
580 omega_BN_history = np.array(omega_BN_history)
581
582 fig, axs = plt.subplots(2, 1, figsize=(12, 6)) # Create 2 vertically stacked subplots
583
584 # Time vectors for plotting
585 t_sigma = np.linspace(0, t_final, len(sigma_BN_history))
586 t_omega = np.linspace(0, t_final, len(omega_BN_history))
587
588 # Plot MRP attitude history
589 axs[0].plot(t_sigma, sigma_BN_history[:, 0], label=r"$\sigma_1$")
590 axs[0].plot(t_sigma, sigma_BN_history[:, 1], label=r"$\sigma_2$")
591 axs[0].plot(t_sigma, sigma_BN_history[:, 2], label=r"$\sigma_3$")
592 axs[0].set_title("MRP Attitude Without Control")
593 axs[0].set_xlabel("Time (s)")
594 axs[0].set_ylabel("MRP Components")
595 axs[0].legend()
596
597 # Plot Angular velocity history
598 axs[1].plot(t_omega, omega_BN_history[:, 0], label=r"$\omega_1$")
599 axs[1].plot(t_omega, omega_BN_history[:, 1], label=r"$\omega_2$")
600 axs[1].plot(t_omega, omega_BN_history[:, 2], label=r"$\omega_3$")
601 axs[1].set_title("Angular Velocity Without Control")
602 axs[1].set_xlabel("Time (s)")
603 axs[1].set_ylabel("Angular Velocity (rad/s)")
604 axs[1].legend()

```

```

605
606 plt.tight_layout()
607 plt.savefig("task7_without_control.png")
608
609 # Plot the results for visualization
610 sigma_BN_100_history = np.array(sigma_BN_100_history)
611 omega_BN_100_history = np.array(omega_BN_100_history)
612
613 # Time vectors
614 t_sigma = np.linspace(0, t_final_control, len(sigma_BN_100_history))
615 t_omega = np.linspace(0, t_final_control, len(omega_BN_100_history))
616
617 # Set up subplots
618 fig, axs = plt.subplots(2, 1, figsize=(12, 6))
619
620 # Plot MRP attitude history
621 axs[0].plot(t_sigma, sigma_BN_100_history[:, 0], label=r"$\sigma_1$")
622 axs[0].plot(t_sigma, sigma_BN_100_history[:, 1], label=r"$\sigma_2$")
623 axs[0].plot(t_sigma, sigma_BN_100_history[:, 2], label=r"$\sigma_3$")
624 axs[0].set_title("MRP Attitude With Control")
625 axs[0].set_xlabel("Time (s)")
626 axs[0].set_ylabel("MRP Components")
627 axs[0].legend()
628
629 # Plot Angular velocity history
630 axs[1].plot(t_omega, omega_BN_100_history[:, 0], label=r"$\omega_1$")
631 axs[1].plot(t_omega, omega_BN_100_history[:, 1], label=r"$\omega_2$")
632 axs[1].plot(t_omega, omega_BN_100_history[:, 2], label=r"$\omega_3$")
633 axs[1].set_title("Angular Velocity With Control")
634 axs[1].set_xlabel("Time (s)")
635 axs[1].set_ylabel("Angular Velocity (rad/s)")
636 axs[1].legend()
637
638 # Layout and save
639 plt.tight_layout()
640 plt.savefig("task7_with_control.png")
641
642
643 # Convert history arrays for plotting
644 B_H_history = np.array(B_H_history)
645 N_H_history = np.array(N_H_history)
646 T_history = np.array(T_history)
647 t_H = np.linspace(0, t_final, len(B_H_history)) # Same time vector for all
648
649 # Plot angular momentum in B and N frames
650 plt.figure(figsize=(12, 6))
651 for j in range(3):
652     plt.plot(t_H, B_H_history[:, j], label=rf"$H_{{B, {j+1}}}$ (Body Frame)")
653     plt.plot(
654         t_H,
655         N_H_history[:, j],
656         label=rf"$H_{{N, {j+1}}}$ (Inertial Frame)",
657         linestyle="--",
658     )
659 plt.title("Angular Momentum in Body and Inertial Frames")
660 plt.xlabel("Time (s)")
661 plt.ylabel("Angular Momentum (N·m·s)")
662 plt.legend()
663 plt.grid(True)
664 plt.tight_layout()
665 plt.savefig("task7_angular_momentum.png")
666
667 # Plot kinetic energy
668 plt.figure(figsize=(10, 4))
669 plt.plot(t_H, T_history, label="Kinetic Energy $T$", color="purple")
670 plt.title("Rotational Kinetic Energy Over Time")
671 plt.xlabel("Time (s)")
672 plt.ylabel("Kinetic Energy (Joules)")

```

```

673 yax = plt.gca()
674 yax.get_yaxis().get_major_formatter().set_useOffset(False)
675 yax.ticklabel_format(style="plain", axis="y")
676 plt.grid(True)
677 plt.tight_layout()
678 plt.savefig("task7_kinetic_energy.png")
679
680
681 ##### Task 8: Sun Pointing Control (10 points) #####
682 print("\n\nBEGIN TASK 8")
683 # #  $B_u = K_B/R$   $P * B_{-}B/R$ 
684
685 # Compute PD gains based on critical damping and 120s time constant
686 I_max = np.max(np.diag(I_b))
687 I_min = np.min(np.diag(I_b))
688 tau = 120 # seconds
689 P = (2 * I_max) / tau
690 K = P**2 / I_min
691
692 print("Chosen P = " + str(P) + " and K = " + str(K))
693 writeToFile(f"./tasks/task 8/gains.txt", np.array([P, K]))
694
695 # Define time settings
696 dt = 1.0 # 1-second step
697 t_final = 500 # seconds
698 time_points = np.arange(0, t_final + dt, dt)
699
700 # Storage arrays
701 sigma_BN_task8_history = []
702 omega_BN_task8_history = []
703 sigma_RN_task8_history = []
704 omega_RN_task8_history = []
705
706 log_times = [15, 100, 200, 400]
707
708
709 # PD control function
710 def PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P):
711     sigma_br, omega_br = getTrackingErrors(t, sigma_bn, omega_bn, RN, omega_rn)
712     u = -K * sigma_br - P * omega_br
713     return u
714
715
716 # Initialize state
717 X = X_0
718
719 # Run control simulation
720 for t in time_points:
721     sigma_bn = X[:3]
722     omega_bn = X[3:]
723     RN = getRsN()
724     omega_rn = getOmegaRsN()
725     u = PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)
726
727     sigma_BN_task8_history.append(sigma_bn)
728     omega_BN_task8_history.append(omega_bn)
729     sigma_RN_task8_history.append(DCM2MRP(RN))
730     omega_RN_task8_history.append(omega_rn)
731
732 # Log MRPs at requested times (short rotation set)
733 if int(t) in log_times:
734     print(f"B/N at t = {int(t)}s {sigma_bn}")
735     writeToFile(f"./tasks/task 8/sigma_{int(t)}s.txt", sigma_bn)
736
737 # Integrate using RK4
738 X = rk4_integrator(dynamics, X, u, dt, t)
739
740 # Enforce MRP shadow set condition after update

```



```

741     X[:3] = checkShadowSet(X[:3])
742
743     # Optional: convert to np.array if plotting
744     sigma_BN_task8_history = np.array(sigma_BN_task8_history)
745     omega_BN_task8_history = np.array(omega_BN_task8_history)
746     sigma_RN_task8_history = np.array(sigma_RN_task8_history)
747     omega_RN_task8_history = np.array(omega_RN_task8_history)
748
749     # Save final plots
750     fig, axs = plt.subplots(2, 1, figsize=(12, 6))
751
752     # Plot MRP history (B/N)
753     (line1,) = axs[0].plot(
754         time_points, sigma_BN_task8_history[:, 0], label=r"$\sigma_{B/N,1}$"
755     )
756     (line2,) = axs[0].plot(
757         time_points, sigma_BN_task8_history[:, 1], label=r"$\sigma_{B/N,2}$"
758     )
759     (line3,) = axs[0].plot(
760         time_points, sigma_BN_task8_history[:, 2], label=r"$\sigma_{B/N,3}$"
761     )
762
763     # Plot MRP history (R/N) with matching colors and dashed lines
764     axs[0].plot(
765         time_points,
766         sigma_RN_task8_history[:, 0],
767         "--",
768         color=line1.get_color(),
769         label=r"$\sigma_{R/N,1}$",
770     )
771     axs[0].plot(
772         time_points,
773         sigma_RN_task8_history[:, 1],
774         "--",
775         color=line2.get_color(),
776         label=r"$\sigma_{R/N,2}$",
777     )
778     axs[0].plot(
779         time_points,
780         sigma_RN_task8_history[:, 2],
781         "--",
782         color=line3.get_color(),
783         label=r"$\sigma_{R/N,3}$",
784     )
785
786     axs[0].set_title("Sun-Pointing MRP Attitude (Task 8)")
787     axs[0].set_ylabel("MRP Components")
788     axs[0].legend()
789
790     # Plot Angular Velocity history (B/N)
791     (line1w,) = axs[1].plot(
792         time_points, omega_BN_task8_history[:, 0], label=r"$\omega_{B/N,1}$"
793     )
794     (line2w,) = axs[1].plot(
795         time_points, omega_BN_task8_history[:, 1], label=r"$\omega_{B/N,2}$"
796     )
797     (line3w,) = axs[1].plot(
798         time_points, omega_BN_task8_history[:, 2], label=r"$\omega_{B/N,3}$"
799     )
800
801     # Plot Angular Velocity history (R/N) with matching colors and dashed lines
802     axs[1].plot(
803         time_points,
804         omega_RN_task8_history[:, 0],
805         "--",
806         color=line1w.get_color(),
807         label=r"$\omega_{R/N,1}$",
808     )

```

```

809     axes[1].plot(
810         time_points,
811         omega_RN_task8_history[:, 1],
812         "--",
813         color=line2w.get_color(),
814         label=r"$\omega_{R/N,2}$",
815     )
816     axes[1].plot(
817         time_points,
818         omega_RN_task8_history[:, 2],
819         "--",
820         color=line3w.get_color(),
821         label=r"$\omega_{R/N,3}$",
822     )
823
824     axes[1].set_title("Sun-Pointing Angular Velocity (Task 8)")
825     axes[1].set_xlabel("Time (s)")
826     axes[1].set_ylabel("Angular Velocity (rad/s)")
827     axes[1].legend()
828
829     plt.tight_layout()
830     plt.savefig("task8_sun_pointing_control.png")
831
832
833     ##### Task 9: Nadir Pointing Control (10 points) #####
834     print("\n\nBEGIN TASK 9")
835
836     # Initialize state
837     X = X_0
838     sigma_BN_task9_history = []
839     omega_BN_task9_history = []
840     sigma_RN_task9_history = []
841     omega_RN_task9_history = []
842
843     # Run control simulation
844     for t in time_points:
845         sigma_bn = X[:3]
846         omega_bn = X[3:]
847         RN = getRnN(t)
848         omega_rn = getOmegaRnN(t)
849         u = PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)
850
851         sigma_BN_task9_history.append(sigma_bn)
852         omega_BN_task9_history.append(omega_bn)
853         sigma_RN_task9_history.append(DCM2MRP(RN))
854         omega_RN_task9_history.append(omega_rn)
855
856     # Log MRPs at requested times (short rotation set)
857     if int(t) in log_times:
858         print(f"B/N at t = {int(t)}s {sigma_bn}")
859         writeToFile(f"./tasks/task 9/sigma_{int(t)}s.txt", sigma_bn)
860
861     # Integrate using RK4
862     X = rk4_integrator(dynamics, X, u, dt, t)
863
864     # Enforce MRP shadow set condition after update
865     X[:3] = checkShadowSet(X[:3])
866
867     # Optional: convert to np.array if plotting
868     sigma_BN_task9_history = np.array(sigma_BN_task9_history)
869     omega_BN_task9_history = np.array(omega_BN_task9_history)
870     sigma_RN_task9_history = np.array(sigma_RN_task9_history)
871     omega_RN_task9_history = np.array(omega_RN_task9_history)
872
873     # Save final plots
874     fig, axes = plt.subplots(2, 1, figsize=(12, 6))
875
876     # Plot MRP history (B/N)

```

```

877 (line1,) = axs[0].plot(
878     time_points, sigma_BN_task9_history[:, 0], label=r"$\sigma_{B/N,1}$"
879 )
880 (line2,) = axs[0].plot(
881     time_points, sigma_BN_task9_history[:, 1], label=r"$\sigma_{B/N,2}$"
882 )
883 (line3,) = axs[0].plot(
884     time_points, sigma_BN_task9_history[:, 2], label=r"$\sigma_{B/N,3}$"
885 )
886
887 # Plot MRP history (R/N) with matching colors and dashed lines
888 axs[0].plot(
889     time_points,
890     sigma_RN_task9_history[:, 0],
891     "--",
892     color=line1.get_color(),
893     label=r"$\sigma_{R/N,1}$",
894 )
895 axs[0].plot(
896     time_points,
897     sigma_RN_task9_history[:, 1],
898     "--",
899     color=line2.get_color(),
900     label=r"$\sigma_{R/N,2}$",
901 )
902 axs[0].plot(
903     time_points,
904     sigma_RN_task9_history[:, 2],
905     "--",
906     color=line3.get_color(),
907     label=r"$\sigma_{R/N,3}$",
908 )
909
910 axs[0].set_title("Nadir-Pointing MRP Attitude (Task 9)")
911 axs[0].set_ylabel("MRP Components")
912 axs[0].legend()
913
914 # Plot Angular Velocity history (B/N)
915 (line1w,) = axs[1].plot(
916     time_points, omega_BN_task9_history[:, 0], label=r"$\omega_{B/N,1}$"
917 )
918 (line2w,) = axs[1].plot(
919     time_points, omega_BN_task9_history[:, 1], label=r"$\omega_{B/N,2}$"
920 )
921 (line3w,) = axs[1].plot(
922     time_points, omega_BN_task9_history[:, 2], label=r"$\omega_{B/N,3}$"
923 )
924
925 # Plot Angular Velocity history (R/N) with matching colors and dashed lines
926 axs[1].plot(
927     time_points,
928     omega_RN_task9_history[:, 0],
929     "--",
930     color=line1w.get_color(),
931     label=r"$\omega_{R/N,1}$",
932 )
933 axs[1].plot(
934     time_points,
935     omega_RN_task9_history[:, 1],
936     "--",
937     color=line2w.get_color(),
938     label=r"$\omega_{R/N,2}$",
939 )
940 axs[1].plot(
941     time_points,
942     omega_RN_task9_history[:, 2],
943     "--",
944     color=line3w.get_color(),

```

```

945     label=r"$\omega_{R/N,3}$",
946 )
947
948 axs[1].set_title("Nadir-Pointing Angular Velocity (Task 9)")
949 axs[1].set_xlabel("Time (s)")
950 axs[1].set_ylabel("Angular Velocity (rad/s)")
951 axs[1].legend()
952
953 plt.tight_layout()
954 plt.savefig("task9_nadir_pointing_control.png")
955
956
957 ##### Task 10: GMP Pointing Control (10 points) #####
958 print("\n\nBEGIN TASK 10")
959
960 # Initialize state
961 X = X_0
962 sigma_BN_task10_history = []
963 omega_BN_task10_history = []
964 sigma_RN_task10_history = []
965 omega_RN_task10_history = []
966
967 # Run control simulation
968 for t in time_points:
969     sigma_bn = X[:3]
970     omega_bn = X[3:]
971     RN = getRcN(t)
972     omega_rn = getOmegaRcN(t)
973     u = PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)
974
975     sigma_BN_task10_history.append(sigma_bn)
976     omega_BN_task10_history.append(omega_bn)
977     sigma_RN_task10_history.append(DCM2MRP(RN))
978     omega_RN_task10_history.append(omega_rn)
979
980 # Log MRPs at requested times (short rotation set)
981 if int(t) in log_times:
982     print(f"B/N at t = {int(t)}s {sigma_bn}")
983     writeToFIle(f"./tasks/task 10/sigma_{int(t)}.txt", sigma_bn)
984
985 # Integrate using RK4
986 X = rk4_integrator(dynamics, X, u, dt, t)
987
988 # Enforce MRP shadow set condition after update
989 X[:3] = checkShadowSet(X[:3])
990
991 # Optional: convert to np.array if plotting
992 sigma_BN_task10_history = np.array(sigma_BN_task10_history)
993 omega_BN_task10_history = np.array(omega_BN_task10_history)
994 sigma_RN_task10_history = np.array(sigma_RN_task10_history)
995 omega_RN_task10_history = np.array(omega_RN_task10_history)
996
997 # Save final plots
998 fig, axs = plt.subplots(2, 1, figsize=(12, 6))
999
1000 # Plot MRP history (B/N)
1001 (line1,) = axs[0].plot(
1002     time_points, sigma_BN_task10_history[:, 0], label=r"$\sigma_{B/N,1}$"
1003 )
1004 (line2,) = axs[0].plot(
1005     time_points, sigma_BN_task10_history[:, 1], label=r"$\sigma_{B/N,2}$"
1006 )
1007 (line3,) = axs[0].plot(
1008     time_points, sigma_BN_task10_history[:, 2], label=r"$\sigma_{B/N,3}$"
1009 )
1010
1011 # Plot MRP history (R/N) with matching colors and dashed lines
1012 axs[0].plot(

```

```

1013     time_points,
1014     sigma_RN_task10_history[:, 0],
1015     "--",
1016     color=line1.get_color(),
1017     label=r"$\sigma_{R/N,1}$",
1018 )
1019 axs[0].plot(
1020     time_points,
1021     sigma_RN_task10_history[:, 1],
1022     "--",
1023     color=line2.get_color(),
1024     label=r"$\sigma_{R/N,2}$",
1025 )
1026 axs[0].plot(
1027     time_points,
1028     sigma_RN_task10_history[:, 2],
1029     "--",
1030     color=line3.get_color(),
1031     label=r"$\sigma_{R/N,3}$",
1032 )
1033
1034 axs[0].set_title("GMO-Pointing MRP Attitude (Task 10)")
1035 axs[0].set_ylabel("MRP Components")
1036 axs[0].legend()
1037
1038 # Plot Angular Velocity history (B/N)
1039 (line1w,) = axs[1].plot(
1040     time_points, omega_BN_task10_history[:, 0], label=r"$\omega_{B/N,1}$"
1041 )
1042 (line2w,) = axs[1].plot(
1043     time_points, omega_BN_task10_history[:, 1], label=r"$\omega_{B/N,2}$"
1044 )
1045 (line3w,) = axs[1].plot(
1046     time_points, omega_BN_task10_history[:, 2], label=r"$\omega_{B/N,3}$"
1047 )
1048
1049 # Plot Angular Velocity history (R/N) with matching colors and dashed lines
1050 axs[1].plot(
1051     time_points,
1052     omega_RN_task10_history[:, 0],
1053     "--",
1054     color=line1w.get_color(),
1055     label=r"$\omega_{R/N,1}$",
1056 )
1057 axs[1].plot(
1058     time_points,
1059     omega_RN_task10_history[:, 1],
1060     "--",
1061     color=line2w.get_color(),
1062     label=r"$\omega_{R/N,2}$",
1063 )
1064 axs[1].plot(
1065     time_points,
1066     omega_RN_task10_history[:, 2],
1067     "--",
1068     color=line3w.get_color(),
1069     label=r"$\omega_{R/N,3}$",
1070 )
1071
1072 axs[1].set_title("GMO-Pointing Angular Velocity (Task 10)")
1073 axs[1].set_xlabel("Time (s)")
1074 axs[1].set_ylabel("Angular Velocity (rad/s)")
1075 axs[1].legend()
1076
1077 plt.tight_layout()
1078 plt.savefig("task10_gmo_pointing_control.png")
1079
1080

```

```

1081 ##### Task 11: Mission Scenario Simulation (10 points) #####
1082 print("\n\nBEGIN TASK 11")
1083
1084 # Define mission duration
1085 t_final = 6500 # seconds
1086 time_points_11 = np.arange(0, t_final + dt, dt)
1087
1088 # Define time points to log MRPs
1089 log_times_11 = [300, 2100, 3400, 4400, 5600]
1090 sigma_BN_task11_history = []
1091 omega_BN_task11_history = []
1092 sigma_RN_task11_history = []
1093 omega_RN_task11_history = []
1094
1095 # Reset initial state
1096 X = np.copy(X_0)
1097
1098
1099 # Mission logic: determine mode based on positions
1100 def determine_control_mode(r_LMO_inertial, r_GMO_inertial):
1101     in_sunlight = r_LMO_inertial[1] > 0
1102     r_LMO_unit = r_LMO_inertial / np.linalg.norm(r_LMO_inertial)
1103     r_GMO_unit = r_GMO_inertial / np.linalg.norm(r_GMO_inertial)
1104     angle = np.arccos(np.dot(r_LMO_unit, r_GMO_unit))
1105     in_comm_window = angle < np.deg2rad(comm_angle_threshold)
1106
1107     if in_sunlight:
1108         return "sun"
1109     elif in_comm_window:
1110         return "gmo"
1111     else:
1112         return "nadir"
1113
1114
1115 # Run mission simulation
1116 for t in time_points_11:
1117     sigma_bn = X[:3]
1118     omega_bn = X[3:]
1119
1120     # Compute inertial positions of LMO and GMO
1121     r_LMO_inertial, _ = getInertialPositionVectors(
1122         r_lmo, omega_lmo_0, i_lmo_0, theta_lmo(t)
1123     )
1124     r_GMO_inertial, _ = getInertialPositionVectors(
1125         r_gmo, omega_gmo_0, i_gmo_0, theta_gmo(t)
1126     )
1127
1128     # Determine control mode
1129     mode = determine_control_mode(r_LMO_inertial, r_GMO_inertial)
1130
1131     # Get reference attitude and rate based on mode
1132     if mode == "sun":
1133         RN = getRsN()
1134         omega_rn = getOmegaRsN()
1135     elif mode == "nadir":
1136         RN = getRnN(t)
1137         omega_rn = getOmegaRnN(t)
1138     elif mode == "gmo":
1139         RN = getRcN(t)
1140         omega_rn = getOmegaRcN(t)
1141
1142     # Control torque
1143     u = PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)
1144
1145     # Log state
1146     sigma_BN_task11_history.append(sigma_bn)
1147     omega_BN_task11_history.append(omega_bn)
1148     sigma_RN_task11_history.append(DCM2MRP(RN))

```

```

1149     omega_RN_task11_history.append(omega_rn)
1150
1151     if int(t) in log_times_11:
1152         print(f"B/N at t = {int(t)}s {sigma_bn}")
1153         writeToFile(f"./tasks/task 11/sigma_{int(t)}s.txt", sigma_bn)
1154
1155     # Integrate dynamics and apply MRP shadow set
1156     X = rk4_integrator(dynamics, X, u, dt, t)
1157     X[:3] = checkShadowSet(X[:3])
1158
1159     # Optional: convert for plotting
1160     sigma_BN_task11_history = np.array(sigma_BN_task11_history)
1161     omega_BN_task11_history = np.array(omega_BN_task11_history)
1162     sigma_RN_task11_history = np.array(sigma_RN_task11_history)
1163     omega_RN_task11_history = np.array(omega_RN_task11_history)
1164
1165     # Save final plots
1166     fig, axs = plt.subplots(2, 1, figsize=(12, 6))
1167
1168     # Plot MRP history (B/N)
1169     (line1,) = axs[0].plot(
1170         time_points_11, sigma_BN_task11_history[:, 0], label=r"$\sigma_{B/N,1}$"
1171     )
1172     (line2,) = axs[0].plot(
1173         time_points_11, sigma_BN_task11_history[:, 1], label=r"$\sigma_{B/N,2}$"
1174     )
1175     (line3,) = axs[0].plot(
1176         time_points_11, sigma_BN_task11_history[:, 2], label=r"$\sigma_{B/N,3}$"
1177     )
1178
1179     # Plot MRP history (R/N) with matching colors and dashed lines
1180     axs[0].plot(
1181         time_points_11,
1182         sigma_RN_task11_history[:, 0],
1183         "--",
1184         color=line1.get_color(),
1185         label=r"$\sigma_{R/N,1}$",
1186     )
1187     axs[0].plot(
1188         time_points_11,
1189         sigma_RN_task11_history[:, 1],
1190         "--",
1191         color=line2.get_color(),
1192         label=r"$\sigma_{R/N,2}$",
1193     )
1194     axs[0].plot(
1195         time_points_11,
1196         sigma_RN_task11_history[:, 2],
1197         "--",
1198         color=line3.get_color(),
1199         label=r"$\sigma_{R/N,3}$",
1200     )
1201
1202     axs[0].set_title("Mission Scenario MRP Attitude (Task 11)")
1203     axs[0].set_ylabel("MRP Components")
1204     axs[0].legend()
1205
1206     # Plot Angular Velocity history (B/N)
1207     (line1w,) = axs[1].plot(
1208         time_points_11, omega_BN_task11_history[:, 0], label=r"$\omega_{B/N,1}$"
1209     )
1210     (line2w,) = axs[1].plot(
1211         time_points_11, omega_BN_task11_history[:, 1], label=r"$\omega_{B/N,2}$"
1212     )
1213     (line3w,) = axs[1].plot(
1214         time_points_11, omega_BN_task11_history[:, 2], label=r"$\omega_{B/N,3}$"
1215     )
1216

```

```

1217 # Plot Angular Velocity history (R/N) with matching colors and dashed lines
1218 axs[1].plot(
1219     time_points_11,
1220     omega_RN_task11_history[:, 0],
1221     "--",
1222     color=line1w.get_color(),
1223     label=r"$\omega_{R/N,1}$",
1224 )
1225 axs[1].plot(
1226     time_points_11,
1227     omega_RN_task11_history[:, 1],
1228     "--",
1229     color=line2w.get_color(),
1230     label=r"$\omega_{R/N,2}$",
1231 )
1232 axs[1].plot(
1233     time_points_11,
1234     omega_RN_task11_history[:, 2],
1235     "--",
1236     color=line3w.get_color(),
1237     label=r"$\omega_{R/N,3}$",
1238 )
1239
1240 axs[1].set_title("Mission Scenario Angular Velocity (Task 11)")
1241 axs[1].set_xlabel("Time (s)")
1242 axs[1].set_ylabel("Angular Velocity (rad/s)")
1243 axs[1].legend()
1244
1245 plt.tight_layout()
1246 plt.savefig("task11_mission_scenario.png")
1247

```

References

- [1] Schaub, H., and Junkins, J. L., *Analytical Mechanics of Space Systems*, 4th ed., AIAA Education Series, 2018.