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.

I. Nomenclature

(Ω, i, θ)	=	(3-1-3) Euler angles
$[BN]$	=	direction cosine matrix from $\mathcal{N}$ frame to $\mathcal{B}$ frame
$[C]^T$	=	transpose of the matrix $[C]$
$[I_{3x3}]$	=	a 3x3 identity matrix
$[\tilde{\omega}]$	=	skew-symmetric matrix of the vector $\boldsymbol{\omega}$
${}^B[I]$	=	inertia tensor $[I]$ expressed in $\mathcal{B}$ frame components
${}^B\boldsymbol{u}$	=	control torque vector expressed in $\mathcal{B}$ frame components
${}^B\boldsymbol{\omega}_{R/N}$	=	angular velocity of the $\mathcal{R}$ frame with respect to the $\mathcal{N}$ frame, expressed in $\mathcal{B}$ frame components
${}^N\boldsymbol{r}$	=	vector $\boldsymbol{r}$ expressed in $\mathcal{N}$ frame components
$\lvert\boldsymbol{r}\rvert$	=	magnitude/norm of the vector $\boldsymbol{r}$
β	=	quaternion
${}^BH$	=	angular momentum vector $\boldsymbol{H}$ expressed in $\mathcal{B}$ frame components
$\boldsymbol{r}^T$	=	transpose of the vector $\boldsymbol{r}$
$\boldsymbol{X}$	=	state vector
$\boldsymbol{\sigma}^S$	=	the shadow MRP set of $\boldsymbol{\sigma}$
$\boldsymbol{\sigma}_{B/N}$	=	modified Rodrigues parameters from $\mathcal{N}$ frame to $\mathcal{B}$ frame
σ^2	=	the same as $\boldsymbol{\sigma}^T\boldsymbol{\sigma}$
σ_i	=	the ith component of the MRP set $\boldsymbol{\sigma}$
$\dot{\theta}$	=	derivative of θ
$\hat{\boldsymbol{n}}_1$	=	unit vector of a frame (e.g. $\mathcal{N}$ frame)
$\mathcal{B}$	=	spacecraft body frame - also seen as B
$\mathcal{H}$	=	orbital Hill frame - also seen as H
$\mathcal{N}$	=	mars centered inertial frame - also seen as N
$\mathcal{O} : \{\hat{\boldsymbol{\imath}}_r, \hat{\boldsymbol{\imath}}_\theta, \hat{\boldsymbol{\imath}}_h\}$	=	coordinate frame $\mathcal{O}$ (also seen as O) defined with three unit vector directions $\{\hat{\boldsymbol{\imath}}_r, \hat{\boldsymbol{\imath}}_\theta, \hat{\boldsymbol{\imath}}_h\}$
$\mathcal{R}$	=	generic spacecraft reference frame - also seen as R
$\mathcal{R}_c$	=	GMO-pointing communication reference frame - also seen as R_c
$\mathcal{R}_n$	=	nadir-pointing reference frame - also seen as R_n
$\mathcal{R}_s$	=	sun-pointing reference frame - also seen as R_s
μ	=	gravitational constant
ξ	=	damping ratio
K	=	scalar attitude feedback gain
P	=	scalar angular velocity feedback gain
t	=	time
T	=	rotational kinetic energy
DCM	=	direction cosine matrix

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

GMO	=	geosynchronous Mars orbit satellite
J	=	Joules
kg	=	kilograms
LMO	=	low Mars orbit satellite
m	=	meters
MRPs	=	modified Rodrigues parameters
PD	=	proportional-derivative control
RK4	=	Runge-Kutta 4th order integrator
s	=	seconds

II. 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 $\mathcal{H} = \{\hat{\boldsymbol{\imath}}_r, \hat{\boldsymbol{\imath}}_\theta, \hat{\boldsymbol{\imath}}_h\}$ base vectors are as shown in the following figure.

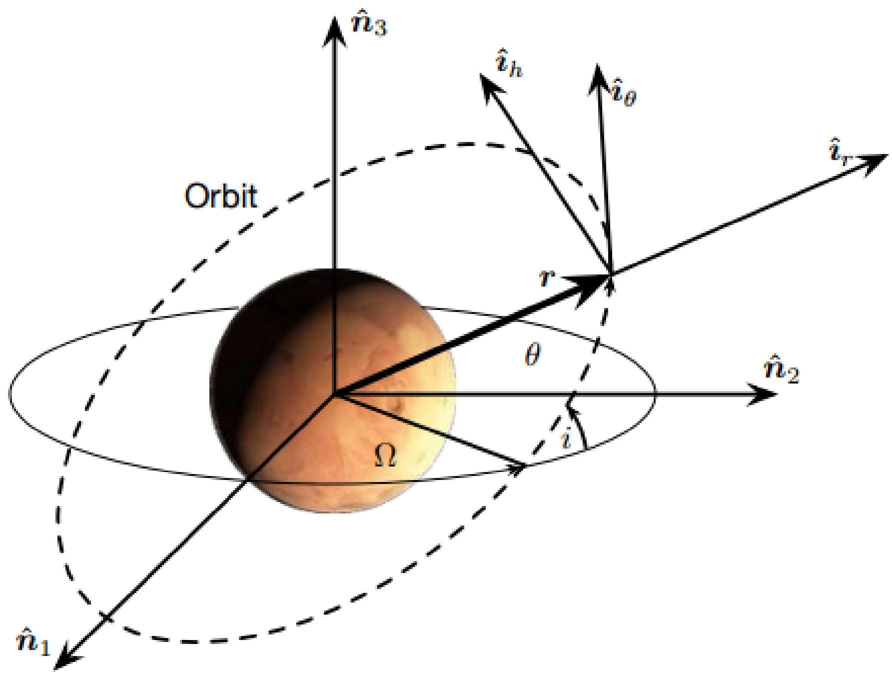

Fig. 1 Illustration of the Inertial frame $\mathcal{N}$ and Hill frame $\mathcal{H}$ base vectors, with the vector $\boldsymbol{r}$ showing the position of the craft in orbit

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 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) $\boldsymbol{\sigma}_{\mathcal{B}/\mathcal{N}}$ and angular velocity $\boldsymbol{\omega}_{\mathcal{B}/\mathcal{N}}$—to track their corresponding reference values $\boldsymbol{\sigma}_{\mathcal{R}/\mathcal{N}}$ and $\boldsymbol{\omega}_{\mathcal{R}/\mathcal{N}}$. This control task includes computing the appropriate torque input $\boldsymbol{u}$ 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:

$$\boldsymbol{\sigma}_{B/N} = \begin{bmatrix} 0.3 \\ -0.4 \\ 0.5 \end{bmatrix}, \quad {}^{B}\boldsymbol{\omega}_{B/N} = \begin{bmatrix} 1.00 \\ 1.75 \\ -2.20 \end{bmatrix} \text{deg/s}$$

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:

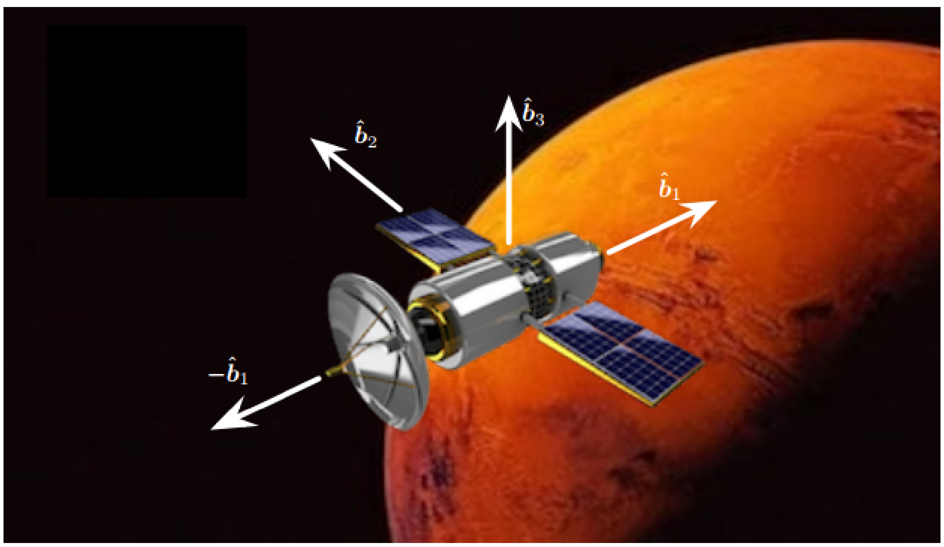

Fig. 2 Illustration of the spacecraft's body frame $\mathcal{B}$ base vectors, showing the alignment of our subsystems of interest with our principal axes.

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 $-\hat{\boldsymbol{b}}_1$) toward the GMO mother satellite when doing comm., its sensor (along $\hat{\boldsymbol{b}}_1$) towards Mars (in the nadir or $-\hat{\boldsymbol{r}}$ direction) when sampling science data, or its solar panels (along $\hat{\boldsymbol{b}}_3$) 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 $\hat{\boldsymbol{n}}_2$ coordinate is positive). Thus, the $\hat{\boldsymbol{b}}_3$ axis will be aligned with $\hat{\boldsymbol{n}}_2$ when the spacecraft is directed at the Sun. To complete the reference frame, the axis $\hat{\boldsymbol{r}}_1$ will point in the $-\hat{\boldsymbol{n}}_1$ direction.

When on the dark side of Mars (i.e., where the $\hat{\boldsymbol{n}}_2$ coordinate is negative), the spacecraft must switch to either communication or science mode. In science mode, the platform's axis $\hat{\boldsymbol{b}}_1$ should point at Mars' center, corresponding to the nadir direction. Additionally, the axis $\hat{\boldsymbol{r}}_2$ will align with the orbital track axis $\hat{\boldsymbol{i}}_\theta$. 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 $-\hat{\boldsymbol{b}}_1$ will be directed toward the GMO mother satellite. These mission pointing scenarios are summarized below in Table 1.

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 1 Spacecraft Pointing Scenario Summary

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

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

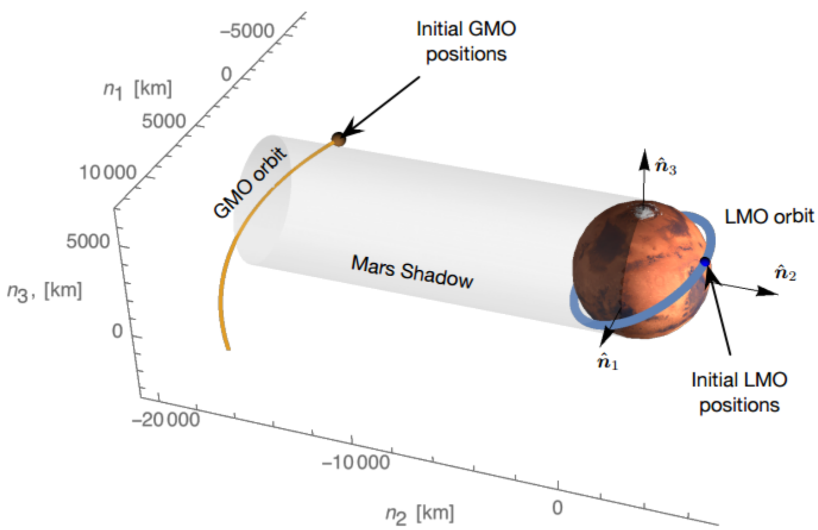

Fig. 3 Illustration of the initial orbital positions of the LMO and GMO crafts, with the Mars centered inertial frame visible and with its $\hat{n}_2$ axis pointing toward the Sun.

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 $\hat{\boldsymbol{n}}_2$ 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 $\mu = 42828.3$ km^3/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:

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

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

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

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.

III. 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 $O : \{\hat{\boldsymbol{i}}_r, \hat{\boldsymbol{i}}_\theta, \hat{\boldsymbol{i}}_h\}$.

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):

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

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 ${}^O\boldsymbol{r} = {}^O\begin{bmatrix} r \\ 0 \\ 0 \end{bmatrix}$ km. We know this because we defined the O frame so that the position vector points in the $\hat{\boldsymbol{i}}_r$ direction. Then we convert it to $\mathcal{N}$ frame by doing ${}^N\boldsymbol{r} = [NO]^O\boldsymbol{r}$. Similarly, we see that the $\hat{\boldsymbol{i}}_\theta$ 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 ${}^N\dot{\boldsymbol{r}} = [NO]\ {}^O\begin{bmatrix} 0 \\ r\dot{\theta} \\ 0 \end{bmatrix}$. 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 ${}^N\boldsymbol{r}$ and ${}^N\dot{\boldsymbol{r}}$.

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**

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

GMO at $t = 1150$ **s**

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

Checking these values through Coursera confirms our calculations are correct.

IV. Task 2: Orbit Frame Orientation

This task calculates the orientation of the orbit frame $\mathcal{H} : \{\hat{\boldsymbol{\imath}}_r, \hat{\boldsymbol{\imath}}_\theta, \hat{\boldsymbol{\imath}}_h\}$ 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.

V. 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 $\hat{\boldsymbol{n}}_2$ coordinate), we want to define the reference frame so that the $+\hat{\boldsymbol{r}}_3$ axis (which will correspond to the satellite's solar panel normal) points to the sun, assumed to be in the $+\hat{\boldsymbol{n}}_2$ direction. To build an orthonormal frame, we use the $-\hat{\boldsymbol{n}}_1$ axis to define $+\hat{\boldsymbol{r}}_1$. Further, we know based on wanting a right handed frame that this means $\hat{\boldsymbol{r}}_2$ will be aligned with the $\hat{\boldsymbol{n}}_3$ 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 $+\hat{\boldsymbol{n}}_2$ direction. This means that the angular velocity is

$$^N\boldsymbol{\omega}_{R_s/N} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{rad/s}$$

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

VI. 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 ($+\hat{\boldsymbol{b}}_1$) directly toward the nadir direction (toward Mars' center). This means our nadir reference frame is constructed such that $+\hat{\boldsymbol{r}}_1$ points nadir to Mars (in the -$\hat{\boldsymbol{\imath}}_r$ direction) and $+\hat{\boldsymbol{r}}_2$ points in the velocity direction $+\hat{\boldsymbol{\imath}}_\theta$. Recall that the $\hat{\boldsymbol{\imath}}$ 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 $\hat{\boldsymbol{r}}_3$ in the -$\hat{\boldsymbol{\imath}}_h$ 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

$$[R_nH] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

From here, we can multiply our two DCMs $[R_nH]$ and $[HN]$ to get our desired DCM of

$$[R_nH][HN] = [R_nN] =$$
$$\begin{bmatrix} \sin(\Omega)\sin(\theta(t))\cos(i) - \cos(\Omega)\cos(\theta(t)) & -\sin(\Omega)\cos(\theta(t)) - \sin(\theta(t))\cos(\Omega)\cos(i) & -\sin(i)\sin(\theta(t)) \\ -\sin(\Omega)\cos(i)\cos(\theta(t)) - \sin(\theta(t))\cos(\Omega) & -\sin(\Omega)\sin(\theta(t)) + \cos(\Omega)\cos(i)\cos(\theta(t)) & \sin(i)\cos(\theta(t)) \\ -\sin(\Omega)\sin(i) & \sin(i)\cos(\Omega) & -\cos(i) \end{bmatrix} \tag{2}$$

Now we need to write a function to do this in our code. Because $[HN]$ 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 $[R_nN]$ DCM at the specified time t.

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

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

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 $[R_nN]$ DCM at that t, computes its transpose $[NR_n]$, and then multiples the $\mathcal{R}$ frame ${}^{R}\boldsymbol{\omega}_{R_n/N}$ by the matrix to get ${}^{N}\boldsymbol{\omega}_{R_n/N}$, which is the vector it returns.

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

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

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

VII. 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 -$-\hat{\boldsymbol{b}}_1$ (antenna direction) with the direction of the GMO mothership. Therefore our reference frame $\mathcal{R}_c$ can be defined as having its $-\hat{\boldsymbol{r}}_1$ 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 $\Delta\boldsymbol{r} = \boldsymbol{r}_{GMO} - \boldsymbol{r}_{LMO}$ means that we can define our first reference direction $\hat{\boldsymbol{r}}_1$ as $-\frac{\Delta \boldsymbol{r}}{|\Delta \boldsymbol{r}|}$ and our second reference direction as $\hat{\boldsymbol{r}}_2 = \frac{\Delta \boldsymbol{r} \times \hat{\boldsymbol{n}}_3}{|\Delta \boldsymbol{r} \times \hat{\boldsymbol{n}}_3|}$. Following the right hand rule, our third reference frame direction will then be $\hat{\boldsymbol{r}}_3 = \hat{\boldsymbol{r}}_1 \times \hat{\boldsymbol{r}}_2$.

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.

Due to how we defined the $\mathcal{H}$ frame, we know our ${}^H\boldsymbol{r}_{LMO}$ and ${}^H\boldsymbol{r}_{GMO}$ vectors are simply $\begin{bmatrix} r_{LMO} \\ 0 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} r_{GMO} \\ 0 \\ 0 \end{bmatrix}$ respectively. Then we can get our ${}^N\boldsymbol{r}_{LMO}$ and ${}^N\boldsymbol{r}_{GMO}$ vectors by doing

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

and

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

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

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

After normalizing, this gives us our first unit vector $\hat{\boldsymbol{r}}_1$ in terms of $\mathcal{N}$ frame components as it is defined above. This also allows us to easily get $\hat{\boldsymbol{r}}_2$, since we have ${}^N\Delta \boldsymbol{r}$ (and $\hat{\boldsymbol{n}}_3$ 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, $\hat{\boldsymbol{r}}_3$ is then just the cross product of the two vectors we just computed, $\hat{\boldsymbol{r}}_1 \times \hat{\boldsymbol{r}}_2$. 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 $t = 330$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 $\boldsymbol{\omega}_{R_c/N}$. To do this, we leverage a rearranged form of the the kinematic differential equation for DCMs:

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

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 $[\dot{R_c N}]$. 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 LaTeXcode. 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 ${}^{\mathcal{R}}\boldsymbol{\omega}_{R_c/N}$ 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:

$$ {}^{\mathcal{N}}\boldsymbol{\omega}_{R_c/N} = [R_cN]^{T\mathcal{R}}\boldsymbol{\omega}_{R_c/N} $$

Finally, we have our ${}^{\mathcal{N}}\boldsymbol{\omega}_{R_c/N}$ 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 $[\dot{R_cN}]$. This function takes a small time step, e.g. $\Delta t = 0.001$s, and evaluates $[R_cN]$ at $[R_cN](t+\Delta t)$, $[R_cN](t-\Delta t)$ and then estimates $[\dot{R_cN}]$ by doing

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

After obtaining $[\dot{R_cN}]$, 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 $t = 330$s 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:

$$ {}^{N}\boldsymbol{\omega}_{R_c/N}(t=330s)_{\text{(Analytical)}} = {}^{N}\boldsymbol{\omega}_{R_c/N}(t=330s)_{\text{(Numerical)}} = \begin{bmatrix} 1.978\times10^{-5} \\ -5.465\times10^{-6} \\ 1.913\times10^{-4} \end{bmatrix} \quad \text{rad/s} $$

VIII. Task 6: Attitude Error Evaluation

Now we have all three reference frames defined and can compute the associated reference DCMs $[R_sN]$, $[R_nN]$, and $[R_cN]$ 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 $\boldsymbol{\sigma}$ (MRPs) to $[C]$ (DCM), we use the vectorial equation

$$ [C] = [I_{3x3}] + \frac{8[\tilde{\boldsymbol{\sigma}}]^2 - 4(1-\sigma^2)[\tilde{\boldsymbol{\sigma}}]}{(1+\sigma^2)^2} \qquad \text{([1], eq. 3.152)} $$

and to go from DCM to MRPs we first compute the quaternions (β_0, β_1, β_2, β_3) using Sheppard's method: This is done by computing the first four β_i^2 terms:

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

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

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

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

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:

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

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

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

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

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

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

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

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

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 $\boldsymbol{\sigma}_{B/N}$ and $\boldsymbol{\omega}_{B/N}$ (converted to radians), given as

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

The `getTrackingErrors` function takes in the current time t, the current $\boldsymbol{\sigma}_{B/N}$ and ${}^{B}\boldsymbol{\omega}_{B/N}$ (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 ${}^{N}\boldsymbol{\omega}_{R/N}$, which will be different for each of the three modes.

The function then converts the provided $\boldsymbol{\sigma}_{B/N}$ 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 $\boldsymbol{\sigma}_{B/R}$ to be returned by the function.

Next the function computes ${}^{B}\boldsymbol{\omega}_{B/R}$ by doing

$${}^{B}\boldsymbol{\omega}_{B/R} = {}^{B}\boldsymbol{\omega}_{B/N} - [BN]\,{}^{N}\boldsymbol{\omega}_{R/N}$$

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

$\texttt{getTrackingErrors}\left(t,\ \boldsymbol{\sigma}_{B/N},\ {}^{B}\boldsymbol{\omega}_{B/N},\ [R_sN],\ {}^{N}\boldsymbol{\omega}_{R_s/N}\right) \to$

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

Nadir-Pointing

$\texttt{getTrackingErrors}\left(t,\ \boldsymbol{\sigma}_{B/N},\ {}^{B}\boldsymbol{\omega}_{B/N},\ [R_nN],\ {}^{N}\boldsymbol{\omega}_{R_n/N}\right) \to$

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

GMO-Pointing

$\texttt{getTrackingErrors}\left(t,\ \boldsymbol{\sigma}_{B/N},\ {}^{B}\boldsymbol{\omega}_{B/N},\ [R_cN],\ {}^{N}\boldsymbol{\omega}_{R_c/N}\right) \to$

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

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

IX. 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 $dt = 1$s and we save our states and rates as the state vector

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{\sigma}_{B/N} \\ {}^{B}\boldsymbol{\omega}_{B/N} \end{bmatrix}$$

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

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

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

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

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

$$\dot{\boldsymbol{\sigma}} = \frac{1}{4}[(1-\sigma^2)[I_{3x3}] + 2[\tilde{\boldsymbol{\sigma}}] + 2\boldsymbol{\sigma}\boldsymbol{\sigma}^T]^{B}\boldsymbol{\omega} \qquad \text{([1], eq. 3.160)}$$

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

```python
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 $\dot{\boldsymbol{\omega}}_{B/N}$ and $\dot{\boldsymbol{\sigma}}$. This function will be passed in as `f` for all calls to our RK4 integrator. The other arguments of `rk4_integrator` are the state $\boldsymbol{X}$, control torque $\boldsymbol{u}$, time-step `dt`, and finally the current simulation time `tn`. It then calls `dynamics` to compute the updated $\dot{\boldsymbol{\omega}}_{B/N}$ and $\dot{\boldsymbol{\sigma}}$ 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 $\boldsymbol{\sigma}_{B/N}$, 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 ${}^{B}\boldsymbol{u} = (0.01, -0.01, 0.02)$ Nm, and simulate again for only 100 seconds this time. For this simulation with control torque, we only need to calculate the attitude $\boldsymbol{\sigma}_{B/N}$ at 100 seconds.

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

$$T = \frac{1}{2}{}^{B}\boldsymbol{\omega}_{B/N}{}^{T}[I]^{B}\boldsymbol{\omega}_{B/N} \qquad \text{([1], eq. 4.55)}$$

and for angular momentum using the equation

$${}^{B}\mathbf{H} = [I]\,{}^{B}\boldsymbol{\omega}_{B/N} \qquad \text{([1], eq. 4.25)}$$

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:

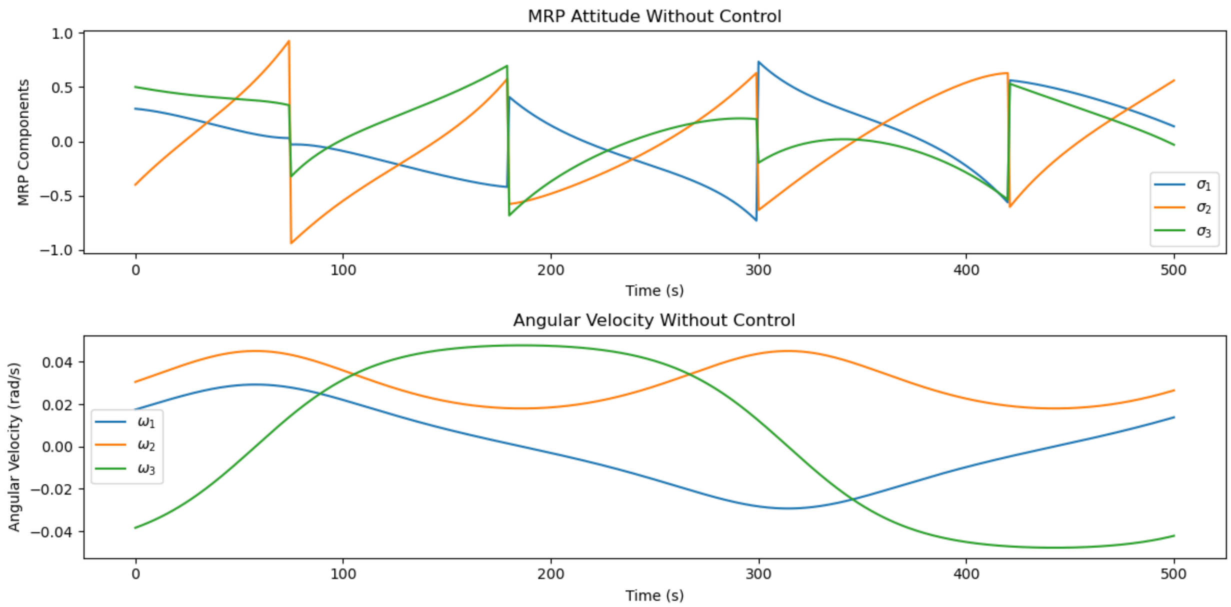

Fig. 4 Attitude and angular velocity of our 500s simulation without control torque

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:

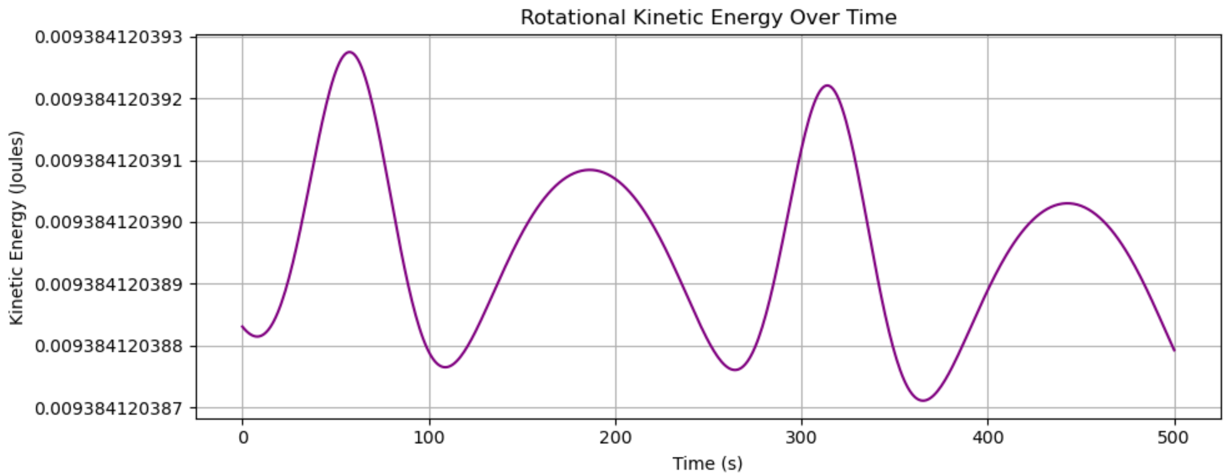

Fig. 5 Kinetic Energy T of our spacecraft over 500s simulation without control torque

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:

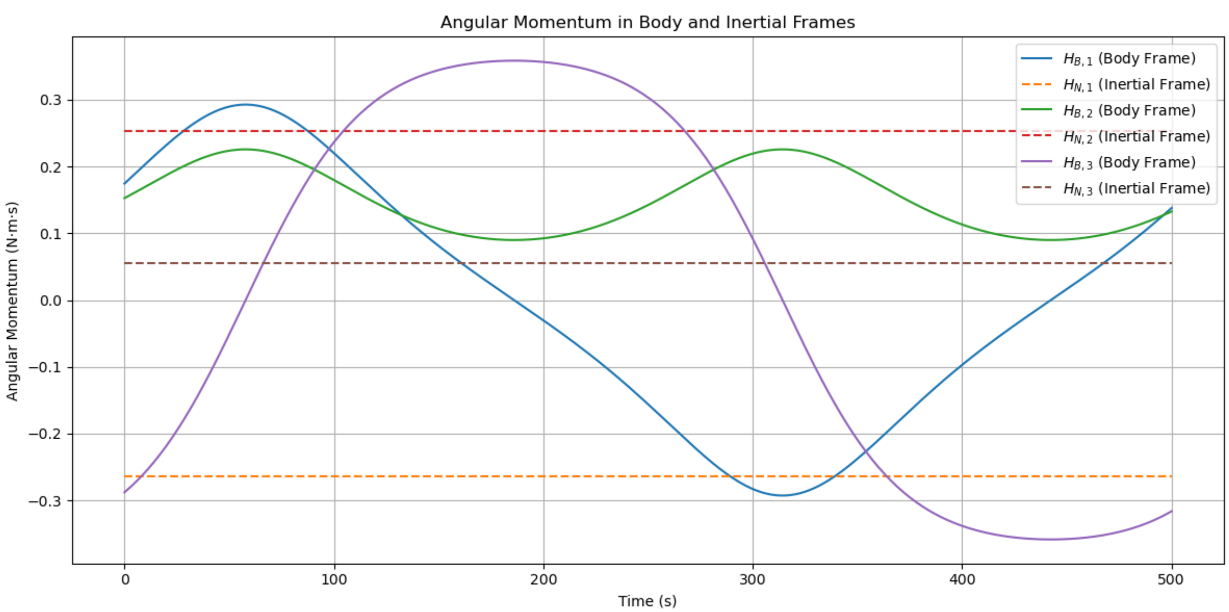

Fig. 6 Angular momentum of our spacecraft in inertial and body fixed frames over 500s simulation without control torque

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 ${}^{B}\boldsymbol{u} = (0.01, -0.01, 0.02)$ Nm. The logic is mostly the same, leveraging our RK4 inegrator 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:

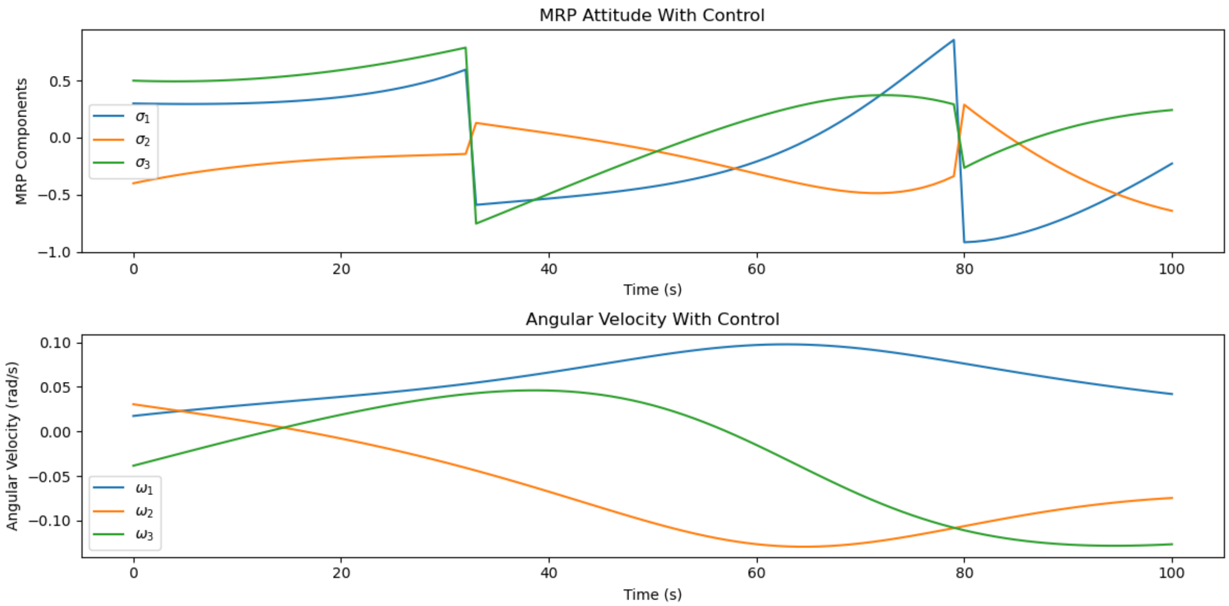

Fig. 7 Attitude and angular velocity of our 100s simulation with control torque

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, $\boldsymbol{\sigma}_{B/N}$ $(500\,\text{s})$:

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

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, $T(500\,\mathrm{s})$:

$$0.0094 \text{ J}$$

Next we do the same for our attitude at $t = 100s$ for our simulation with the control torque added:
MRP Attitude, $\boldsymbol{\sigma}_{B/N}(100\,\mathrm{s})$:

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

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

X. Task 8: Sun Pointing Control

In the next few sections, well 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:

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

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 attitude feedback gain K and scalar angular velocity feedback gain 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

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

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 chose $I_{max} = 10$ kg m^2 to ensure that $T_i \leq 120$s. Plugging this in yields

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

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

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

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 chose $I_{min} = 5$ kg m^2 to ensure that $\xi \leq 1$. Plugging in this along with our calculated value of P yields

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

Now that we have chosen our values for K and P, we can write a function to compute our control torque ${}^{B}\boldsymbol{u}$. This function is called `PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)`. It computes the current control torque vector ${}^{B}\boldsymbol{u}$ 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 ${}^{B}\boldsymbol{u}$, $\boldsymbol{\sigma}_{B/R}$ and ${}^{B}\boldsymbol{\omega}_{B/R}$. 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 ${}^{N}\boldsymbol{\omega}_{R_s/N}$. 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 $\boldsymbol{\sigma}_{B/N}$ and ${}^{B}\boldsymbol{\omega}_{B/N}$ for plotting and calling our RK4 integrator with the `dynamics` function from Task 7. Each iteration, we update the value of the control torque ${}^{B}\boldsymbol{u}$ 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:

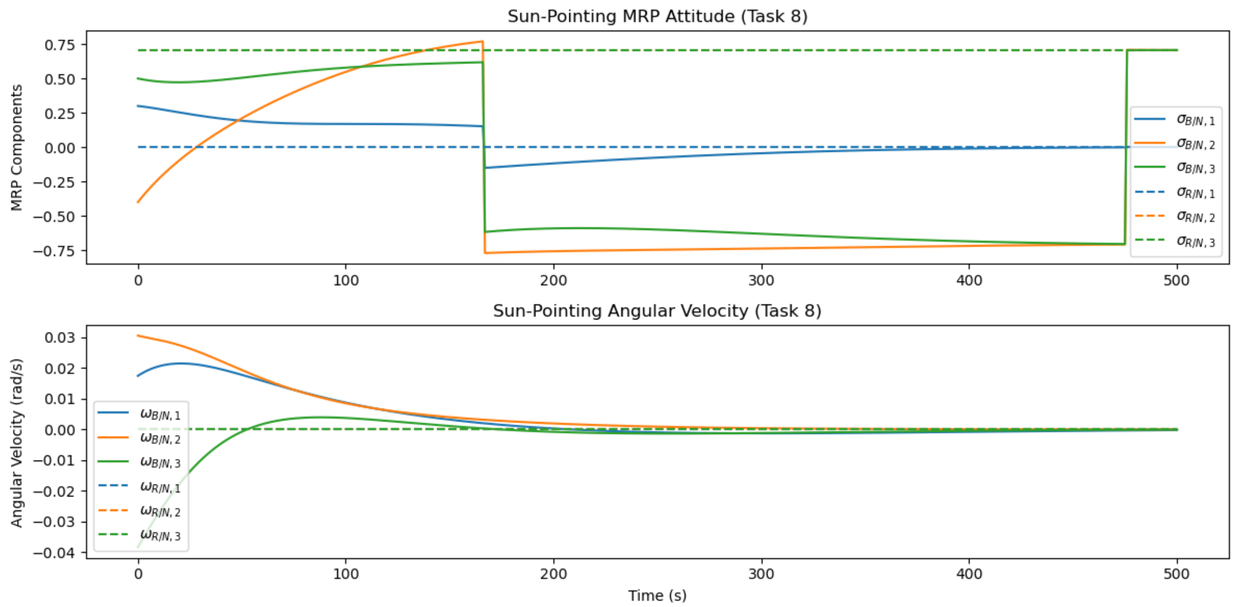

Fig. 8 Attitude and angular velocity plotted against their reference values for a 500s simulation of our control torque driving the craft to the sun-pointing reference frame.

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

$$\boldsymbol{\sigma}_{R_s/N} = \begin{bmatrix} 0 \\ 0.70710678 \\ 0.70710678 \end{bmatrix}$$

Taking the magnitude of this MRP set gives us $|\boldsymbol{\sigma}| = 0.99999999664$. 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:

$\boldsymbol{\sigma}_{B/N}(15\,\text{s})$:

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

$\boldsymbol{\sigma}_{B/N}(100\,\text{s})$:

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

$\boldsymbol{\sigma}_{B/N}(200\,\text{s})$:

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

$\boldsymbol{\sigma}_{B/N}(400\,\text{s})$:

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

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

XI. 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 ${}^N\boldsymbol{\omega}_{R_n/N}$. Afterwards, we can plot our results:

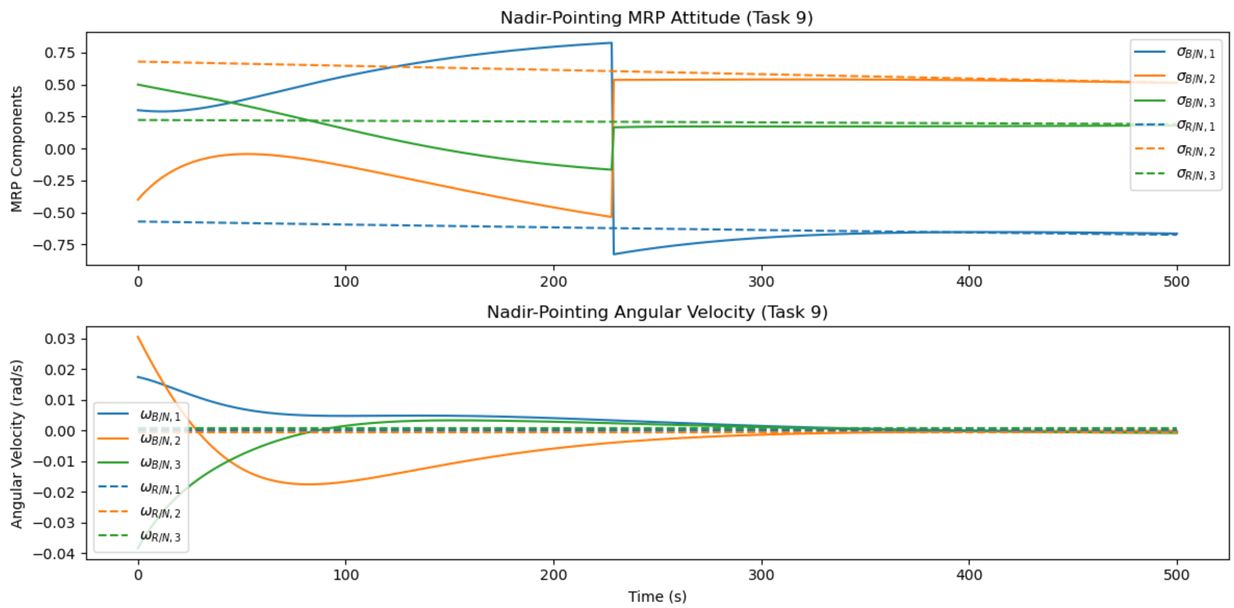

Fig. 9 Attitude and angular velocity plotted against their reference values for a 500s simulation of our control torque driving the craft to the Nadir-pointing reference frame.

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:

$\boldsymbol{\sigma}_{B/N}(15\,\mathrm{s})$:

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

$\boldsymbol{\sigma}_{B/N}(100\,\mathrm{s})$:

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

$\boldsymbol{\sigma}_{B/N}(200\,\mathrm{s})$:

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

$\boldsymbol{\sigma}_{B/N}(400\,\mathrm{s})$:

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

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

XII. 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 $[R_c N]$ and the value passed as `omega_rn` is ${}^N\boldsymbol{\omega}_{R_c/N}$. 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:

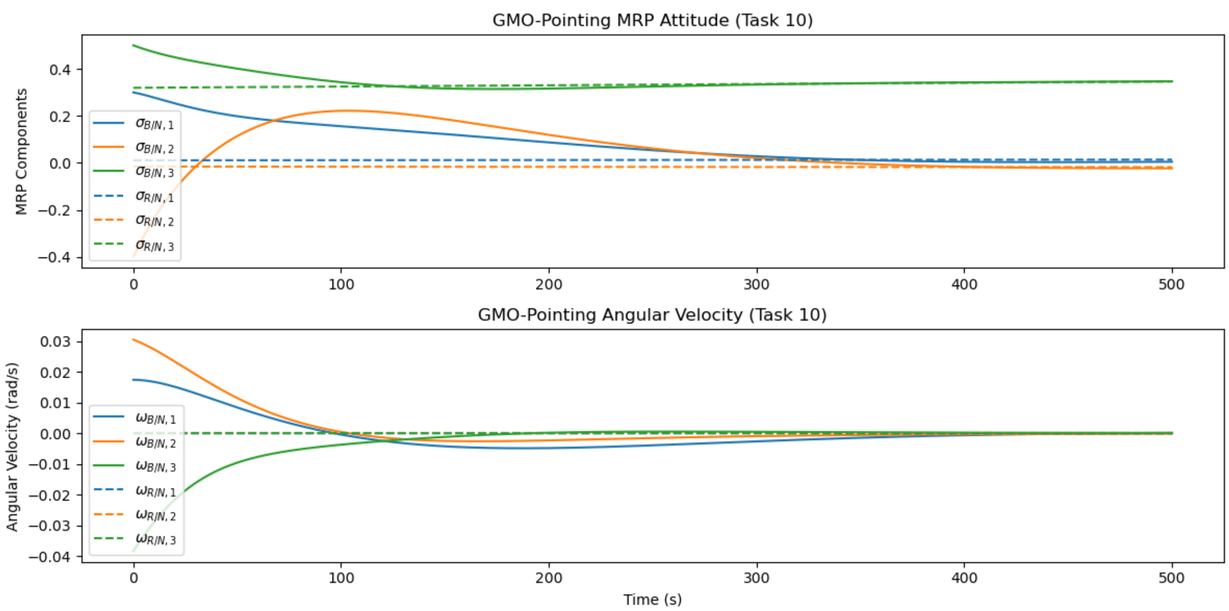

Fig. 10 Attitude and angular velocity plotted against their reference values for a 500s simulation of our control torque driving the craft to the GMO-pointing reference frame.

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:

$\boldsymbol{\sigma}_{B/N}(15\,\mathrm{s})$:

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

$\boldsymbol{\sigma}_{B/N}(100\,\mathrm{s})$:

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

$\boldsymbol{\sigma}_{B/N}(200\,\mathrm{s})$:

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

$\boldsymbol{\sigma}_{B/N}(400\,\mathrm{s})$:

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

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

XIII. 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 `omega_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, ${}^N\boldsymbol{r}_{LMO}$ and ${}^N\boldsymbol{r}_{GMO}$. 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 ${}^N\boldsymbol{r}_{LMO}$ vector. Since this vector is expressed with $\mathcal{N}$ frame components, all we need to do here is check if the second ($\hat{\boldsymbol{n}}_2$) 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 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 ${}^N\boldsymbol{r}_{LMO}$ and ${}^N\boldsymbol{r}_{GMO}$ vectors. We know that

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

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 `omega_rn` values for the `PD_controller`.

Afterwards, we plot our results:

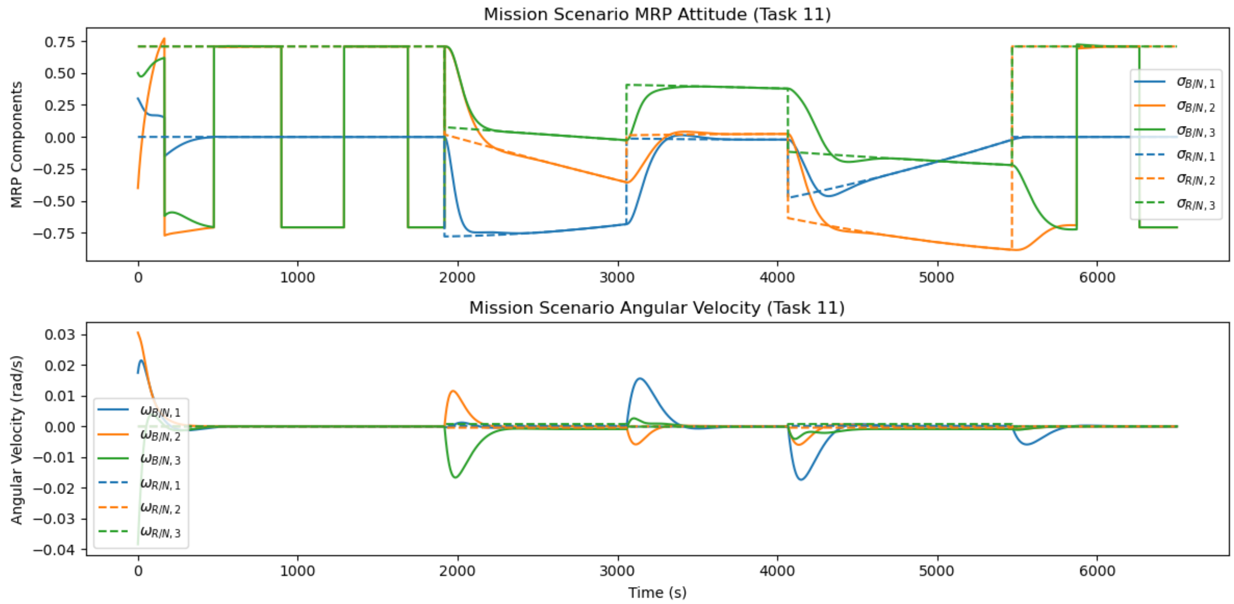

Fig. 11 Attitude and angular velocity plotted against their reference values for a 6500s simulation of our control torque driving the craft to the varying reference frames detailed in our mission.

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 is 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.

XIV. 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

```python
import numpy as np
from sympy import symbols, pprint, cos, sin, Matrix, pi
import sympy as sp
import matplotlib.pyplot as plt

############################## Welcome ##############################
print("Welcome to the ASEN 5010 Capstone Project")
print("Attitude Dynamics and Control of a Nano-Satellite Orbiting Mars")

############################## Initial state ##############################
h = 400  # km
R_mars = 3396.19  # km
r_lmo = R_mars + h
mu = 42828.3  # km^3/s^2
theta_lmo_rate = 0.000884797  # rad/s
mars_period_min = (24 * 60) + 37
mars_period_sec = mars_period_min * 60
mars_period_hr = mars_period_min / 60
gmo_period_sec = mars_period_sec
r_gmo = 20424.2  # km
theta_gmo_rate = 0.0000709003  # rad/s
tmax = 6500  # Set the value of tmax
dt = 0.1  # Set the value of t
t_0 = 0.0  # Initial time
sigma_bn_0 = np.array([0.3, -0.4, 0.5])  # MRPs
omega_bn_0 = np.array([1.00, 1.75, -2.20])  # deg/s
omega_bn_0_rad = np.deg2rad(omega_bn_0)
I_b = np.array([[10, 0, 0], [0, 5, 0], [0, 0, 7.5]])  # kg*m^2
I_b_inv = np.linalg.inv(I_b)
X_0 = np.concatenate((sigma_bn_0, omega_bn_0_rad))  # Initial conditions
omega_lmo_0 = np.deg2rad(20)
i_lmo_0 = np.deg2rad(30)
theta_lmo_0 = np.deg2rad(60)  # function of time
omega_gmo_0 = 0
i_gmo_0 = 0
theta_gmo_0 = np.deg2rad(250)  # function of time
comm_angle_threshold = 35  # deg


############################## Helper Functions ##############################
def writeToFile(path, data):
    str_to_write = ""

    # Convert scalar to list
    if np.isscalar(data):
        str_to_write = str(data)
    else:
        if data.ndim == 2:
            for row in data:
                for element in row:
                    str_to_write += str(element) + " "
        else:
            for element in data:
                str_to_write += str(element) + " "
    with open(path, "w+") as file:
        file.write(str_to_write.rstrip())


def theta_lmo(t):
    return theta_lmo_0 + t * theta_lmo_rate


def theta_gmo(t):
```

```python
    return theta_gmo_0 + t * theta_gmo_rate


def s(theta):
    return np.sin(theta)


def c(theta):
    return np.cos(theta)


# Returns a skew-symmetric matrix from a vector
def tilde(v):
    return np.array([[0, -v[2], v[1]], [v[2], 0, -v[0]], [-v[1], v[0], 0]])


def Euler313toDCM(t1, t2, t3):
    # Convert 313 angles to DCM - From appendix B.1
    return np.array(
        [
            [
                c(t3) * c(t1) - s(t3) * c(t2) * s(t1),
                c(t3) * s(t1) + s(t3) * c(t2) * c(t1),
                s(t3) * s(t2),
            ],
            [
                -s(t3) * c(t1) - c(t3) * c(t2) * s(t1),
                -s(t3) * s(t1) + c(t3) * c(t2) * c(t1),
                c(t3) * s(t2),
            ],
            [s(t2) * s(t1), -s(t2) * c(t1), c(t2)],
        ]
    )


def DCM2Quaternion(C):
    # Initialize stuff
    B = np.zeros(4)  # B^2 array
    b = np.zeros(4)  # resulting set of quaternions
    trc = np.trace(C)

    B[0] = (1 + trc) / 4
    B[1] = (1 + 2 * C[0, 0] - trc) / 4
    B[2] = (1 + 2 * C[1, 1] - trc) / 4
    B[3] = (1 + 2 * C[2, 2] - trc) / 4

    # Find the index of the maximum value in B2
    i = np.argmax(B)

    # Calculate quaternion based on sheppard's method
    if i == 0:
        b[0] = np.sqrt(B[0])
        b[1] = (C[1, 2] - C[2, 1]) / (4 * b[0])
        b[2] = (C[2, 0] - C[0, 2]) / (4 * b[0])
        b[3] = (C[0, 1] - C[1, 0]) / (4 * b[0])
    elif i == 1:
        b[1] = np.sqrt(B[1])
        b[0] = (C[1, 2] - C[2, 1]) / (4 * b[1])
        if b[0] < 0:
            b[1] = -b[1]
            b[0] = -b[0]
        b[2] = (C[0, 1] + C[1, 0]) / (4 * b[1])
        b[3] = (C[2, 0] + C[0, 2]) / (4 * b[1])
    elif i == 2:
        b[2] = np.sqrt(B[2])
        b[0] = (C[2, 0] - C[0, 2]) / (4 * b[2])
        if b[0] < 0:
            b[2] = -b[2]
```

```python
            b[0] = -b[0]
        b[1] = (C[0, 1] + C[1, 0]) / (4 * b[2])
        b[3] = (C[1, 2] + C[2, 1]) / (4 * b[2])
    elif i == 3:
        b[3] = np.sqrt(B[3])
        b[0] = (C[0, 1] - C[1, 0]) / (4 * b[3])
        if b[0] < 0:
            b[3] = -b[3]
            b[0] = -b[0]
        b[1] = (C[2, 0] + C[0, 2]) / (4 * b[3])
        b[2] = (C[1, 2] + C[2, 1]) / (4 * b[3])

    return b


def MRP2DCM(sigma):
    tilde_sigma = tilde(sigma)
    return np.eye(3) + (
        8 * tilde_sigma @ tilde_sigma - 4 * (1 - sigma @ sigma) * tilde_sigma
    ) / ((1 + sigma @ sigma) ** 2)


def checkShadowSet(sigma):
    if np.linalg.norm(sigma) > 1:
        return -sigma / (sigma @ sigma)
    else:
        return sigma


def DCM2MRP(C):
    b = DCM2Quaternion(C)
    divisor = 1 + b[0]
    sigma = np.array([b[1], b[2], b[3]]) / divisor
    return checkShadowSet(sigma)


############################## Task 1: Orbit Simulation (5 points) ##############################
# pos = r*i_r
# Derive inertial s/c velocity r_dot.
# Note that for circular orbits, theta_dot is constant
print("\n\nBEGIN TASK 1")


# Write a function whose inputs are radius r and 313 angles omega, i, theta,
# and outputs are the inertial pos vector N_r and vel N_r_dot
# Calculate the inertial position vector N_r and velocity N_r_dot
def getInertialPositionVectors(r, omega, i, theta):
    # O : {i_r, i_theta, i_h} aka H frame
    # N : {n_1, n_2, n_3}
    ON = Euler313toDCM(omega, i, theta)
    NO = ON.T
    # Convert direction of i_r to N
    N_r = NO @ np.array([r, 0, 0])
    # Convert direction of i_theta to N
    if r == r_lmo:
        N_r_dot = NO @ np.array([0, r * theta_lmo_rate, 0])
    if r == r_gmo:
        N_r_dot = NO @ np.array([0, r * theta_gmo_rate, 0])
    return N_r, N_r_dot


# confirm the operation by checking getInertialVectors(r_lmo, omega_lmo, i_lmo, theta_lmo(450))
# and getInertialVectors(r_gmo, omega_gmo, i_gmo, theta_gmo(1150))
N_r_lmo, N_r_lmo_dot = getInertialPositionVectors(
    r_lmo, omega_lmo_0, i_lmo_0, theta_lmo(450)
)
N_r_gmo, N_r_gmo_dot = getInertialPositionVectors(
    r_gmo, omega_gmo_0, i_gmo_0, theta_gmo(1150)
```

```python
)
print("rLMO = ", N_r_lmo)
print("vLMO = ", N_r_lmo_dot)
print("rGMO = ", N_r_gmo)
print("vGMO = ", N_r_gmo_dot)
writeToFile("./tasks/task 1/rLMO.txt", N_r_lmo)
writeToFile("./tasks/task 1/vLMO.txt", N_r_lmo_dot)
writeToFile("./tasks/task 1/rGMO.txt", N_r_gmo)
writeToFile("./tasks/task 1/vGMO.txt", N_r_gmo_dot)


############################## Task 2: Orbit Frame Orientation (5 points) ##############################
print("\n\nBEGIN TASK 2")
Omega, i = symbols("Omega i")
t = symbols("t", positive=True)
theta = sp.Function(symbols("theta"))
HN = Matrix(
    [
        [
            cos(theta(t)) * cos(Omega) - sin(theta(t)) * cos(i) * sin(Omega),
            cos(theta(t)) * sin(Omega) + sin(theta(t)) * cos(i) * cos(Omega),
            sin(theta(t)) * sin(i),
        ],
        [
            -sin(theta(t)) * cos(Omega) - cos(theta(t)) * cos(i) * sin(Omega),
            -sin(theta(t)) * sin(Omega) + cos(theta(t)) * cos(i) * cos(Omega),
            cos(theta(t)) * sin(i),
        ],
        [sin(i) * sin(Omega), -sin(i) * cos(Omega), cos(i)],
    ]
)
with open("./latex/task_2_HN.tex", "w+") as file:
    file.write((sp.latex(HN)))


# Write a function whose input is time t and output is DCM HN(t) for the LMO
def getHNforLMO(t):
    return Euler313toDCM(omega_lmo_0, i_lmo_0, theta_lmo(t))


# Validate the operation by computing HN(300)
t = 300  # sec
HN_at_t = getHNforLMO(t)
print("HN(t = " + str(t) + "s) = ", HN_at_t)
writeToFile("./tasks/task 2/HN.txt", HN_at_t)


############################## Task 3: Sun-Pointing Reference Frame Orientation (10 points)
↪ ##############################
print("\n\nBEGIN TASK 3")
# First determine and analytic expression for Rs by defining DCM [RsN]
RsN = np.array([[-1, 0, 0], [0, 0, 1], [0, 1, 0]])
with open("./latex/RsN.tex", "w+") as file:
    file.write((sp.latex(RsN)))


# Write a function that returns RsN
def getRsN():
    return RsN


def getOmegaRsN():
    return np.array([0, 0, 0])


# Validate the evalutation of RsN by providing numerical values for t=0s
print("RsN(t = 0s) = ", getRsN())
writeToFile("./tasks/task 3/RsN.txt", getRsN())
```

```python

# Angular velocity is [0, 0, 0] since the DCM is not a function of time
print("N__Rn/N = ", getOmegaRsN())
writeToFile("./tasks/task 3/omega_rs_n.txt", getOmegaRsN())


############################## Task 4: Nadir-Pointing Reference Frame Orientation (10 points) ##############################
print("\n\nBEGIN TASK 4")
# First determine and analytic expression for Rn by defining DCM [RnN]
RnH = np.array([[-1, 0, 0], [0, 1, 0], [0, 0, -1]])
with open("./latex/RnH.tex", "w+") as file:
    file.write((sp.latex(RnH)))
with open("./latex/RnN.tex", "w+") as file:
    file.write((sp.latex(RnH @ HN)))


# Write a function that returns RnN
def getRnN(t):
    return RnH @ getHNforLMO(t)


# Write a function that determines angular velocity vector omega_rn_n
def getOmegaRnN(t):
    NRn = getRnN(t).T
    return NRn @ [0, 0, -theta_lmo_rate]  #  = _dot*i_h = -_dot*r_3


t = 330  # sec
# Validate the evalutation of RnN by providing numerical values for t = 330s
RnN_at_t = getRnN(t)
print("RnN(t = " + str(t) + "s) = ", RnN_at_t)
writeToFile("./tasks/task 4/RnN.txt", RnN_at_t)

# What is the angular velocity @ t = 330s
omega_rnn_at_t = getOmegaRnN(t)
print("N__Rn/N(t = " + str(t) + "s) = ", omega_rnn_at_t)
writeToFile("./tasks/task 4/omega_rn_n.txt", omega_rnn_at_t)


############################## Task 5: GMO-Pointing Reference Frame Orientation (10 points) ##############################
print("\n\nBEGIN TASK 5")


# dr = r_gmo - r_lmo  => lets get these in N frame to make cross product easy
# H : {i_r, i_theta, i_h} - Note theres one for GMO, one for LMO, depending on theta
# N : {n_1, n_2, n_3}
# Write a function that returns RcN
def getRcNExpr():
    NH = HN.T
    theta_gmo_expr = sp.Function(symbols("theta_GMO"))
    theta_lmo_expr = sp.Function(symbols("theta_LMO"))
    NH_gmo = NH.subs([(Omega, omega_gmo_0), (i, i_gmo_0), (theta, theta_gmo_expr)])
    NH_lmo = NH.subs([(Omega, omega_lmo_0), (i, i_lmo_0), (theta, theta_lmo_expr)])
    N_r_gmo_col = NH_gmo @ sp.Matrix([r_gmo, 0, 0])
    N_r_lmo_col = NH_lmo @ sp.Matrix([r_lmo, 0, 0])

    N_dr_col = N_r_gmo_col - N_r_lmo_col
    N_r1_col = -N_dr_col.normalized()
    N_r2_col = (N_dr_col.cross(sp.Matrix([0, 0, 1]))).normalized()
    N_r3_col = N_r1_col.cross(N_r2_col).normalized()

    return (sp.Matrix.hstack(N_r1_col, N_r2_col, N_r3_col)).T


def getRcN(t):
    RcN = getRcNExpr()
```

```python
    RcN_f = sp.lambdify(
        ["t"],
        RcN,
        modules=["numpy", {"theta_GMO": theta_gmo}, {"theta_LMO": theta_lmo}],
    )
    return RcN_f(t)


# Write a function that determines angular velocity vector omega_rc_n
def getOmegaRcNAnalytically(time):
    t = symbols("t", positive=True)
    RcN = getRcNExpr()
    # RcNT = RcN.T

    # Replace with base functions so sympy knows how to derive wrt t
    theta_gmo_expr = sp.Function(symbols("theta_GMO"))
    theta_lmo_expr = sp.Function(symbols("theta_LMO"))
    replace_gmo = theta_gmo_0 + t * theta_gmo_rate
    replace_lmo = theta_lmo_0 + t * theta_lmo_rate
    replace_dict = {theta_gmo_expr(t): replace_gmo, theta_lmo_expr(t): replace_lmo}
    RcN_rep = RcN.subs(replace_dict)

    # Now sympy can take derivative wrt t
    RcN_dot = sp.diff(RcN_rep, t)
    omega_tilde = -RcN_dot @ RcN.T
    # with open('./latex/insane_matrix.tex', "w+") as file:
    #     file.write((sp.latex(omega_tilde)))
    omega_tilde_f = sp.lambdify(
        ["t"],
        omega_tilde,
        modules=["numpy", {"theta_GMO": theta_gmo}, {"theta_LMO": theta_lmo}],
    )

    ssm = omega_tilde_f(time)
    ssm = (ssm - ssm.T) / 2  # Force diagonals to 0
    R_omega_rcn = sp.Matrix([-ssm[1, 2], ssm[0, 2], -ssm[0, 1]])
    N_omega_rcn = RcN.T @ R_omega_rcn
    N_omega_rcn_f = sp.lambdify(
        ["t"],
        N_omega_rcn,
        modules=["numpy", {"theta_GMO": theta_gmo}, {"theta_LMO": theta_lmo}],
    )
    N_omega_rcn_real = N_omega_rcn_f(time)
    return N_omega_rcn_real.flatten()


def getOmegaRcN(time):
    dt = 1e-6

    RcN_t = getRcN(time)
    RcN_plus = getRcN(time + dt)
    RcN_minus = getRcN(time - dt)
    RcN_dot = (RcN_plus - RcN_minus) / (2 * dt)

    ssm = -RcN_dot @ RcN_t.T
    ssm = (ssm + ssm) / 2  # Force diagonals to 0
    R_omega_rcn = np.array([-ssm[1, 2], ssm[0, 2], -ssm[0, 1]])
    N_omega_rcn = RcN_t.T @ R_omega_rcn
    return N_omega_rcn


t = 330
RcN_at_t = getRcN(t)
omega_RcN_num_at_t = getOmegaRcN(t)
omega_RcN_anal_at_t = getOmegaRcNAnalytically(t)
print("RcN = ", RcN_at_t)
print("Numerical  = ", omega_RcN_num_at_t)
print("Analytical  = ", omega_RcN_anal_at_t)
```

```
writeToFile("./tasks/task 5/RcN.txt", RcN_at_t)
writeToFile("./tasks/task 5/omega_rc_n_num.txt", omega_RcN_num_at_t)
writeToFile("./tasks/task 5/omega_rc_n_anal.txt", omega_RcN_anal_at_t)


############################## Task 6: Attitude Error Evaluation (10 points) ##############################
print("\n\nBEGIN TASK 6")


# Write function that returns tracking errors sigma_br and omega_br
def getTrackingErrors(t, sigma_bn, B_omega_bn, RN, N_omega_rn):
    # Get _BR from _BN and RN DCM
    BN = MRP2DCM(sigma_bn)
    BR = BN @ (RN.T)
    sigma_br = DCM2MRP(BR)

    # Get _br from _bn and _rn
    B_omega_br = B_omega_bn - (BN @ N_omega_rn)

    return sigma_br, B_omega_br


# Sun-pointing
t = 0
sigma, omega = getTrackingErrors(t, sigma_bn_0, omega_bn_0_rad, getRsN(), getOmegaRsN())
print("Sun-Pointing Orientation")
print("_B/R = ", sigma)
print("_B/R = ", omega)
writeToFile("./tasks/task 6/sun-sigma.txt", sigma)
writeToFile("./tasks/task 6/sun-omega.txt", omega)

# Nadir-pointing
sigma, omega = getTrackingErrors(
    t, sigma_bn_0, omega_bn_0_rad, getRnN(t), getOmegaRnN(t)
)
print("Nadir-Pointing Orientation")
print("_B/R = ", sigma)
print("_B/R = ", omega)
writeToFile("./tasks/task 6/nad-sigma.txt", sigma)
writeToFile("./tasks/task 6/nad-omega.txt", omega)

# GMO-pointing
sigma, omega = getTrackingErrors(
    t, sigma_bn_0, omega_bn_0_rad, getRcN(t), getOmegaRcN(t)
)
print("GMO-Pointing Orientation")
print("_B/R = ", sigma)
print("_B/R = ", omega)
writeToFile("./tasks/task 6/gmo-sigma.txt", sigma)
writeToFile("./tasks/task 6/gmo-omega.txt", omega)


############################## Task 7: Numerical Attitude Simulator (10 points) ##############################
print("\n\nBEGIN TASK 7")

# Write your own numerical integrator using RK45


# Define the spacecraft dynamics (equation of motion)
def dynamics(X, dt, u):
    sigma_BN = X[:3]  # MRP attitude
    sigma_BN_skew = tilde(sigma_BN)
    sigma_BN = sigma_BN.reshape((3, 1))  # make col
    omega_BN = X[3:]  # Angular velocity
    omega_BN_skew = tilde(omega_BN)
    omega_BN = omega_BN.reshape((3, 1))  # make col
    if isinstance(u, np.ndarray):
        u = u.reshape((3, 1))
```

```python
    d_omega_BN = I_b_inv @ (-omega_BN_skew @ (I_b @ omega_BN) + u)
    d_sigma_BN = (
        0.25
        * (
            (1 - (sigma_BN.T @ sigma_BN)) * np.eye(3)
            + 2 * sigma_BN_skew
            + 2 * sigma_BN @ sigma_BN.T
        )
        @ omega_BN
    )
    return np.concatenate((d_sigma_BN.flatten(), d_omega_BN.flatten()))


# Runge-Kutta 4th order integrator
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


# Time settings
dt = 1  # 1 second time step
t_final = 500.0  # Total time for the integration
time_steps = int(t_final / dt)

# Control torque (zero initially)
u = np.zeros(3)  # Control torque vector

# Arrays to store results for plotting
sigma_BN_history = []
omega_BN_history = []
T_history = []
B_H_history = []
N_H_history = []

# Integration loop (u = 0 for this part)
X = X_0
for t in np.arange(0, t_final + dt, dt):
    B_sigma_BN = X[:3]  # MRP attitude
    # B_sigma_BN = checkShadowSet(B_sigma_BN)
    B_omega_BN = X[3:]  # Angular velocity
    sigma_BN_history.append(B_sigma_BN)
    omega_BN_history.append(B_omega_BN)

    # Kinetic Energy
    T = 0.5 * (B_omega_BN.T @ (I_b @ B_omega_BN))
    T_history.append(T)

    # Compute angular momentum H
    H = I_b @ B_omega_BN
    B_H_history.append(H)
    N_H_history.append(MRP2DCM(B_sigma_BN).T @ H)

    # Update attitude using RK4
    X = rk4_integrator(dynamics, X, u, dt, t)

    X[:3] = checkShadowSet(X[:3])


# Xn, t_500, sigmas, omegas, B_H, N_H, T = RK4(dynamics, X_0, 1, 500, 0)
# print(sigmas[0:3, 500])
# Results at 500 seconds
sigma_BN_500 = sigma_BN_history[-1]
omega_BN_500 = omega_BN_history[-1]
T_500 = T_history[-1]
```

```python
B_H_500 = B_H_history[-1]
BN_500 = MRP2DCM(sigma_BN_500)
N_H_500 = BN_500.T @ B_H_500
N_H_500s = N_H_history[-1]

print(f"MRP attitude at 500s: {sigma_BN_500}")
print(f"Kinetic energy at 500s: {T_500}")
print(f"B-Frame Angular momentum at 500s: {B_H_500}")
print(f"N-Frame Angular momentum at 500s: {N_H_500}")
print(f"N-Frame Angular momentum at 500s: {N_H_500s}")
writeToFile("./tasks/task 7/sigma_500s.txt", sigma_BN_500)
writeToFile("./tasks/task 7/T_500s.txt", T_500)
writeToFile("./tasks/task 7/H_500s_B_frame.txt", B_H_500)
writeToFile("./tasks/task 7/H_500s_N_frame.txt", N_H_500)


# Now apply control torque u = (0.01, -0.01, 0.02) Nm and integrate again for 100s
u_fixed = np.array([0.01, -0.01, 0.02])  # Fixed control torque
X = X_0  # Reset initial conditions
sigma_BN_100_history = []
omega_BN_100_history = []
t_final_control = 100  # update to only 100s this time
# Run integration with control torque for 100 seconds
for t in np.arange(0, t_final_control + dt, dt):
    B_sigma_BN = X[:3]  # MRP attitude
    B_omega_BN = X[3:]
    # B_sigma_BN = checkShadowSet(B_sigma_BN)

    sigma_BN_100_history.append(B_sigma_BN)
    omega_BN_100_history.append(B_omega_BN)

    # Update attitude using RK4
    X = rk4_integrator(dynamics, X, u_fixed, dt, t)

    X[:3] = checkShadowSet(X[:3])


sigma_BN_100 = sigma_BN_100_history[-1]
print(f"MRP attitude at 100s with control torque: {sigma_BN_100}")
writeToFile("./tasks/task 7/sigma_100s_with_control.txt", sigma_BN_100)

# Plot the results for visualization
sigma_BN_history = np.array(sigma_BN_history)
omega_BN_history = np.array(omega_BN_history)

fig, axs = plt.subplots(2, 1, figsize=(12, 6))  # Create 2 vertically stacked subplots

# Time vectors for plotting
t_sigma = np.linspace(0, t_final, len(sigma_BN_history))
t_omega = np.linspace(0, t_final, len(omega_BN_history))

# Plot MRP attitude history
axs[0].plot(t_sigma, sigma_BN_history[:, 0], label=r"$\sigma_1$")
axs[0].plot(t_sigma, sigma_BN_history[:, 1], label=r"$\sigma_2$")
axs[0].plot(t_sigma, sigma_BN_history[:, 2], label=r"$\sigma_3$")
axs[0].set_title("MRP Attitude Without Control")
axs[0].set_xlabel("Time (s)")
axs[0].set_ylabel("MRP Components")
axs[0].legend()

# Plot Angular velocity history
axs[1].plot(t_omega, omega_BN_history[:, 0], label=r"$\omega_1$")
axs[1].plot(t_omega, omega_BN_history[:, 1], label=r"$\omega_2$")
axs[1].plot(t_omega, omega_BN_history[:, 2], label=r"$\omega_3$")
axs[1].set_title("Angular Velocity Without Control")
axs[1].set_xlabel("Time (s)")
axs[1].set_ylabel("Angular Velocity (rad/s)")
axs[1].legend()
```

```python

plt.tight_layout()
plt.savefig("task7_without_control.png")

# Plot the results for visualization
sigma_BN_100_history = np.array(sigma_BN_100_history)
omega_BN_100_history = np.array(omega_BN_100_history)

# Time vectors
t_sigma = np.linspace(0, t_final_control, len(sigma_BN_100_history))
t_omega = np.linspace(0, t_final_control, len(omega_BN_100_history))

# Set up subplots
fig, axs = plt.subplots(2, 1, figsize=(12, 6))

# Plot MRP attitude history
axs[0].plot(t_sigma, sigma_BN_100_history[:, 0], label=r"$\sigma_1$")
axs[0].plot(t_sigma, sigma_BN_100_history[:, 1], label=r"$\sigma_2$")
axs[0].plot(t_sigma, sigma_BN_100_history[:, 2], label=r"$\sigma_3$")
axs[0].set_title("MRP Attitude With Control")
axs[0].set_xlabel("Time (s)")
axs[0].set_ylabel("MRP Components")
axs[0].legend()

# Plot Angular velocity history
axs[1].plot(t_omega, omega_BN_100_history[:, 0], label=r"$\omega_1$")
axs[1].plot(t_omega, omega_BN_100_history[:, 1], label=r"$\omega_2$")
axs[1].plot(t_omega, omega_BN_100_history[:, 2], label=r"$\omega_3$")
axs[1].set_title("Angular Velocity With Control")
axs[1].set_xlabel("Time (s)")
axs[1].set_ylabel("Angular Velocity (rad/s)")
axs[1].legend()

# Layout and save
plt.tight_layout()
plt.savefig("task7_with_control.png")


# Convert history arrays for plotting
B_H_history = np.array(B_H_history)
N_H_history = np.array(N_H_history)
T_history = np.array(T_history)
t_H = np.linspace(0, t_final, len(B_H_history))  # Same time vector for all

# Plot angular momentum in B and N frames
plt.figure(figsize=(12, 6))
for j in range(3):
    plt.plot(t_H, B_H_history[:, j], label=rf"$H_{{B,{j+1}}}$ (Body Frame)")
    plt.plot(
        t_H,
        N_H_history[:, j],
        label=rf"$H_{{N,{j+1}}}$ (Inertial Frame)",
        linestyle="--",
    )
plt.title("Angular Momentum in Body and Inertial Frames")
plt.xlabel("Time (s)")
plt.ylabel("Angular Momentum (N·m·s)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("task7_angular_momentum.png")

# Plot kinetic energy
plt.figure(figsize=(10, 4))
plt.plot(t_H, T_history, label="Kinetic Energy $T$", color="purple")
plt.title("Rotational Kinetic Energy Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Kinetic Energy (Joules)")
```

```python
yax = plt.gca()
yax.get_yaxis().get_major_formatter().set_useOffset(False)
yax.ticklabel_format(style="plain", axis="y")
plt.grid(True)
plt.tight_layout()
plt.savefig("task7_kinetic_energy.png")


############################## Task 8: Sun Pointing Control (10 points) ##############################
print("\n\nBEGIN TASK 8")
# # B_u = K_B/R  P * B__B/R

# Compute PD gains based on critical damping and 120s time constant
I_max = np.max(np.diag(I_b))
I_min = np.min(np.diag(I_b))
tau = 120  # seconds
P = (2 * I_max) / tau
K = P**2 / I_min

print("Chosen P = " + str(P) + " and K = " + str(K))
writeToFile(f"./tasks/task 8/gains.txt", np.array([P, K]))

# Define time settings
dt = 1.0  # 1-second step
t_final = 500  # seconds
time_points = np.arange(0, t_final + dt, dt)

# Storage arrays
sigma_BN_task8_history = []
omega_BN_task8_history = []
sigma_RN_task8_history = []
omega_RN_task8_history = []

log_times = [15, 100, 200, 400]


# PD control function
def PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P):
    sigma_br, omega_br = getTrackingErrors(t, sigma_bn, omega_bn, RN, omega_rn)
    u = -K * sigma_br - P * omega_br
    return u


# Initialize state
X = X_0

# Run control simulation
for t in time_points:
    sigma_bn = X[:3]
    omega_bn = X[3:]
    RN = getRsN()
    omega_rn = getOmegaRsN()
    u = PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)

    sigma_BN_task8_history.append(sigma_bn)
    omega_BN_task8_history.append(omega_bn)
    sigma_RN_task8_history.append(DCM2MRP(RN))
    omega_RN_task8_history.append(omega_rn)

    # Log MRPs at requested times (short rotation set)
    if int(t) in log_times:
        print(f"B/N at t = {int(t)}s {sigma_bn}")
        writeToFile(f"./tasks/task 8/sigma_{int(t)}s.txt", sigma_bn)

    # Integrate using RK4
    X = rk4_integrator(dynamics, X, u, dt, t)

    # Enforce MRP shadow set condition after update
```

```python
        X[:3] = checkShadowSet(X[:3])

    # Optional: convert to np.array if plotting
    sigma_BN_task8_history = np.array(sigma_BN_task8_history)
    omega_BN_task8_history = np.array(omega_BN_task8_history)
    sigma_RN_task8_history = np.array(sigma_RN_task8_history)
    omega_RN_task8_history = np.array(omega_RN_task8_history)

    # Save final plots
    fig, axs = plt.subplots(2, 1, figsize=(12, 6))

    # Plot MRP history (B/N)
    (line1,) = axs[0].plot(
        time_points, sigma_BN_task8_history[:, 0], label=r"$\sigma_{B/N,1}$"
    )
    (line2,) = axs[0].plot(
        time_points, sigma_BN_task8_history[:, 1], label=r"$\sigma_{B/N,2}$"
    )
    (line3,) = axs[0].plot(
        time_points, sigma_BN_task8_history[:, 2], label=r"$\sigma_{B/N,3}$"
    )

    # Plot MRP history (R/N) with matching colors and dashed lines
    axs[0].plot(
        time_points,
        sigma_RN_task8_history[:, 0],
        "--",
        color=line1.get_color(),
        label=r"$\sigma_{R/N,1}$",
    )
    axs[0].plot(
        time_points,
        sigma_RN_task8_history[:, 1],
        "--",
        color=line2.get_color(),
        label=r"$\sigma_{R/N,2}$",
    )
    axs[0].plot(
        time_points,
        sigma_RN_task8_history[:, 2],
        "--",
        color=line3.get_color(),
        label=r"$\sigma_{R/N,3}$",
    )

    axs[0].set_title("Sun-Pointing MRP Attitude (Task 8)")
    axs[0].set_ylabel("MRP Components")
    axs[0].legend()

    # Plot Angular Velocity history (B/N)
    (line1w,) = axs[1].plot(
        time_points, omega_BN_task8_history[:, 0], label=r"$\omega_{B/N,1}$"
    )
    (line2w,) = axs[1].plot(
        time_points, omega_BN_task8_history[:, 1], label=r"$\omega_{B/N,2}$"
    )
    (line3w,) = axs[1].plot(
        time_points, omega_BN_task8_history[:, 2], label=r"$\omega_{B/N,3}$"
    )

    # Plot Angular Velocity history (R/N) with matching colors and dashed lines
    axs[1].plot(
        time_points,
        omega_RN_task8_history[:, 0],
        "--",
        color=line1w.get_color(),
        label=r"$\omega_{R/N,1}$",
    )
```

```python
axs[1].plot(
    time_points,
    omega_RN_task8_history[:, 1],
    "--",
    color=line2w.get_color(),
    label=r"$\omega_{R/N,2}$",
)
axs[1].plot(
    time_points,
    omega_RN_task8_history[:, 2],
    "--",
    color=line3w.get_color(),
    label=r"$\omega_{R/N,3}$",
)

axs[1].set_title("Sun-Pointing Angular Velocity (Task 8)")
axs[1].set_xlabel("Time (s)")
axs[1].set_ylabel("Angular Velocity (rad/s)")
axs[1].legend()

plt.tight_layout()
plt.savefig("task8_sun_pointing_control.png")


############################## Task 9: Nadir Pointing Control (10 points) ##############################
print("\n\nBEGIN TASK 9")

# Initialize state
X = X_0
sigma_BN_task9_history = []
omega_BN_task9_history = []
sigma_RN_task9_history = []
omega_RN_task9_history = []

# Run control simulation
for t in time_points:
    sigma_bn = X[:3]
    omega_bn = X[3:]
    RN = getRnN(t)
    omega_rn = getOmegaRnN(t)
    u = PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)

    sigma_BN_task9_history.append(sigma_bn)
    omega_BN_task9_history.append(omega_bn)
    sigma_RN_task9_history.append(DCM2MRP(RN))
    omega_RN_task9_history.append(omega_rn)

    # Log MRPs at requested times (short rotation set)
    if int(t) in log_times:
        print(f"B/N at t = {int(t)}s {sigma_bn}")
        writeToFile(f"./tasks/task 9/sigma_{int(t)}s.txt", sigma_bn)

    # Integrate using RK4
    X = rk4_integrator(dynamics, X, u, dt, t)

    # Enforce MRP shadow set condition after update
    X[:3] = checkShadowSet(X[:3])

# Optional: convert to np.array if plotting
sigma_BN_task9_history = np.array(sigma_BN_task9_history)
omega_BN_task9_history = np.array(omega_BN_task9_history)
sigma_RN_task9_history = np.array(sigma_RN_task9_history)
omega_RN_task9_history = np.array(omega_RN_task9_history)

# Save final plots
fig, axs = plt.subplots(2, 1, figsize=(12, 6))

# Plot MRP history (B/N)
```

```
(line1,) = axs[0].plot(
    time_points, sigma_BN_task9_history[:, 0], label=r"$\sigma_{B/N,1}$"
)
(line2,) = axs[0].plot(
    time_points, sigma_BN_task9_history[:, 1], label=r"$\sigma_{B/N,2}$"
)
(line3,) = axs[0].plot(
    time_points, sigma_BN_task9_history[:, 2], label=r"$\sigma_{B/N,3}$"
)

# Plot MRP history (R/N) with matching colors and dashed lines
axs[0].plot(
    time_points,
    sigma_RN_task9_history[:, 0],
    "--",
    color=line1.get_color(),
    label=r"$\sigma_{R/N,1}$",
)
axs[0].plot(
    time_points,
    sigma_RN_task9_history[:, 1],
    "--",
    color=line2.get_color(),
    label=r"$\sigma_{R/N,2}$",
)
axs[0].plot(
    time_points,
    sigma_RN_task9_history[:, 2],
    "--",
    color=line3.get_color(),
    label=r"$\sigma_{R/N,3}$",
)

axs[0].set_title("Nadir-Pointing MRP Attitude (Task 9)")
axs[0].set_ylabel("MRP Components")
axs[0].legend()

# Plot Angular Velocity history (B/N)
(line1w,) = axs[1].plot(
    time_points, omega_BN_task9_history[:, 0], label=r"$\omega_{B/N,1}$"
)
(line2w,) = axs[1].plot(
    time_points, omega_BN_task9_history[:, 1], label=r"$\omega_{B/N,2}$"
)
(line3w,) = axs[1].plot(
    time_points, omega_BN_task9_history[:, 2], label=r"$\omega_{B/N,3}$"
)

# Plot Angular Velocity history (R/N) with matching colors and dashed lines
axs[1].plot(
    time_points,
    omega_RN_task9_history[:, 0],
    "--",
    color=line1w.get_color(),
    label=r"$\omega_{R/N,1}$",
)
axs[1].plot(
    time_points,
    omega_RN_task9_history[:, 1],
    "--",
    color=line2w.get_color(),
    label=r"$\omega_{R/N,2}$",
)
axs[1].plot(
    time_points,
    omega_RN_task9_history[:, 2],
    "--",
    color=line3w.get_color(),
```

```
    label=r"$\omega_{R/N,3}$",
)

axs[1].set_title("Nadir-Pointing Angular Velocity (Task 9)")
axs[1].set_xlabel("Time (s)")
axs[1].set_ylabel("Angular Velocity (rad/s)")
axs[1].legend()

plt.tight_layout()
plt.savefig("task9_nadir_pointing_control.png")


############################## Task 10: GMO Pointing Control (10 points) ##############################
print("\n\nBEGIN TASK 10")

# Initialize state
X = X_0
sigma_BN_task10_history = []
omega_BN_task10_history = []
sigma_RN_task10_history = []
omega_RN_task10_history = []

# Run control simulation
for t in time_points:
    sigma_bn = X[:3]
    omega_bn = X[3:]
    RN = getRcN(t)
    omega_rn = getOmegaRcN(t)
    u = PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)

    sigma_BN_task10_history.append(sigma_bn)
    omega_BN_task10_history.append(omega_bn)
    sigma_RN_task10_history.append(DCM2MRP(RN))
    omega_RN_task10_history.append(omega_rn)

    # Log MRPs at requested times (short rotation set)
    if int(t) in log_times:
        print(f"B/N at t = {int(t)}s {sigma_bn}")
        writeToFile(f"./tasks/task 10/sigma_{int(t)}s.txt", sigma_bn)

    # Integrate using RK4
    X = rk4_integrator(dynamics, X, u, dt, t)

    # Enforce MRP shadow set condition after update
    X[:3] = checkShadowSet(X[:3])

# Optional: convert to np.array if plotting
sigma_BN_task10_history = np.array(sigma_BN_task10_history)
omega_BN_task10_history = np.array(omega_BN_task10_history)
sigma_RN_task10_history = np.array(sigma_RN_task10_history)
omega_RN_task10_history = np.array(omega_RN_task10_history)

# Save final plots
fig, axs = plt.subplots(2, 1, figsize=(12, 6))

# Plot MRP history (B/N)
(line1,) = axs[0].plot(
    time_points, sigma_BN_task10_history[:, 0], label=r"$\sigma_{B/N,1}$"
)
(line2,) = axs[0].plot(
    time_points, sigma_BN_task10_history[:, 1], label=r"$\sigma_{B/N,2}$"
)
(line3,) = axs[0].plot(
    time_points, sigma_BN_task10_history[:, 2], label=r"$\sigma_{B/N,3}$"
)

# Plot MRP history (R/N) with matching colors and dashed lines
axs[0].plot(
```

```
    time_points,
    sigma_RN_task10_history[:, 0],
    "--",
    color=line1.get_color(),
    label=r"$\sigma_{R/N,1}$",
)
axs[0].plot(
    time_points,
    sigma_RN_task10_history[:, 1],
    "--",
    color=line2.get_color(),
    label=r"$\sigma_{R/N,2}$",
)
axs[0].plot(
    time_points,
    sigma_RN_task10_history[:, 2],
    "--",
    color=line3.get_color(),
    label=r"$\sigma_{R/N,3}$",
)

axs[0].set_title("GMO-Pointing MRP Attitude (Task 10)")
axs[0].set_ylabel("MRP Components")
axs[0].legend()

# Plot Angular Velocity history (B/N)
(line1w,) = axs[1].plot(
    time_points, omega_BN_task10_history[:, 0], label=r"$\omega_{B/N,1}$"
)
(line2w,) = axs[1].plot(
    time_points, omega_BN_task10_history[:, 1], label=r"$\omega_{B/N,2}$"
)
(line3w,) = axs[1].plot(
    time_points, omega_BN_task10_history[:, 2], label=r"$\omega_{B/N,3}$"
)

# Plot Angular Velocity history (R/N) with matching colors and dashed lines
axs[1].plot(
    time_points,
    omega_RN_task10_history[:, 0],
    "--",
    color=line1w.get_color(),
    label=r"$\omega_{R/N,1}$",
)
axs[1].plot(
    time_points,
    omega_RN_task10_history[:, 1],
    "--",
    color=line2w.get_color(),
    label=r"$\omega_{R/N,2}$",
)
axs[1].plot(
    time_points,
    omega_RN_task10_history[:, 2],
    "--",
    color=line3w.get_color(),
    label=r"$\omega_{R/N,3}$",
)

axs[1].set_title("GMO-Pointing Angular Velocity (Task 10)")
axs[1].set_xlabel("Time (s)")
axs[1].set_ylabel("Angular Velocity (rad/s)")
axs[1].legend()

plt.tight_layout()
plt.savefig("task10_gmo_pointing_control.png")
```

```python
############################## Task 11: Mission Scenario Simulation (10 points) ##############################
print("\n\nBEGIN TASK 11")

# Define mission duration
t_final = 6500  # seconds
time_points_11 = np.arange(0, t_final + dt, dt)

# Define time points to log MRPs
log_times_11 = [300, 2100, 3400, 4400, 5600]
sigma_BN_task11_history = []
omega_BN_task11_history = []
sigma_RN_task11_history = []
omega_RN_task11_history = []

# Reset initial state
X = np.copy(X_0)


# Mission logic: determine mode based on positions
def determine_control_mode(r_LMO_inertial, r_GMO_inertial):
    in_sunlight = r_LMO_inertial[1] > 0
    r_LMO_unit = r_LMO_inertial / np.linalg.norm(r_LMO_inertial)
    r_GMO_unit = r_GMO_inertial / np.linalg.norm(r_GMO_inertial)
    angle = np.arccos(np.dot(r_LMO_unit, r_GMO_unit))
    in_comm_window = angle < np.deg2rad(comm_angle_threshold)

    if in_sunlight:
        return "sun"
    elif in_comm_window:
        return "gmo"
    else:
        return "nadir"


# Run mission simulation
for t in time_points_11:
    sigma_bn = X[:3]
    omega_bn = X[3:]

    # Compute inertial positions of LMO and GMO
    r_LMO_inertial, _ = getInertialPositionVectors(
        r_lmo, omega_lmo_0, i_lmo_0, theta_lmo(t)
    )
    r_GMO_inertial, _ = getInertialPositionVectors(
        r_gmo, omega_gmo_0, i_gmo_0, theta_gmo(t)
    )

    # Determine control mode
    mode = determine_control_mode(r_LMO_inertial, r_GMO_inertial)

    # Get reference attitude and rate based on mode
    if mode == "sun":
        RN = getRsN()
        omega_rn = getOmegaRsN()
    elif mode == "nadir":
        RN = getRnN(t)
        omega_rn = getOmegaRnN(t)
    elif mode == "gmo":
        RN = getRcN(t)
        omega_rn = getOmegaRcN(t)

    # Control torque
    u = PD_controller(t, sigma_bn, omega_bn, RN, omega_rn, K, P)

    # Log state
    sigma_BN_task11_history.append(sigma_bn)
    omega_BN_task11_history.append(omega_bn)
    sigma_RN_task11_history.append(DCM2MRP(RN))
```

```python
    omega_RN_task11_history.append(omega_rn)

    if int(t) in log_times_11:
        print(f"B/N at t = {int(t)}s {sigma_bn}")
        writeToFile(f"./tasks/task 11/sigma_{int(t)}s.txt", sigma_bn)

    # Integrate dynamics and apply MRP shadow set
    X = rk4_integrator(dynamics, X, u, dt, t)
    X[:3] = checkShadowSet(X[:3])

# Optional: convert for plotting
sigma_BN_task11_history = np.array(sigma_BN_task11_history)
omega_BN_task11_history = np.array(omega_BN_task11_history)
sigma_RN_task11_history = np.array(sigma_RN_task11_history)
omega_RN_task11_history = np.array(omega_RN_task11_history)

# Save final plots
fig, axs = plt.subplots(2, 1, figsize=(12, 6))

# Plot MRP history (B/N)
(line1,) = axs[0].plot(
    time_points_11, sigma_BN_task11_history[:, 0], label=r"$\sigma_{B/N,1}$"
)
(line2,) = axs[0].plot(
    time_points_11, sigma_BN_task11_history[:, 1], label=r"$\sigma_{B/N,2}$"
)
(line3,) = axs[0].plot(
    time_points_11, sigma_BN_task11_history[:, 2], label=r"$\sigma_{B/N,3}$"
)

# Plot MRP history (R/N) with matching colors and dashed lines
axs[0].plot(
    time_points_11,
    sigma_RN_task11_history[:, 0],
    "--",
    color=line1.get_color(),
    label=r"$\sigma_{R/N,1}$",
)
axs[0].plot(
    time_points_11,
    sigma_RN_task11_history[:, 1],
    "--",
    color=line2.get_color(),
    label=r"$\sigma_{R/N,2}$",
)
axs[0].plot(
    time_points_11,
    sigma_RN_task11_history[:, 2],
    "--",
    color=line3.get_color(),
    label=r"$\sigma_{R/N,3}$",
)

axs[0].set_title("Mission Scenario MRP Attitude (Task 11)")
axs[0].set_ylabel("MRP Components")
axs[0].legend()

# Plot Angular Velocity history (B/N)
(line1w,) = axs[1].plot(
    time_points_11, omega_BN_task11_history[:, 0], label=r"$\omega_{B/N,1}$"
)
(line2w,) = axs[1].plot(
    time_points_11, omega_BN_task11_history[:, 1], label=r"$\omega_{B/N,2}$"
)
(line3w,) = axs[1].plot(
    time_points_11, omega_BN_task11_history[:, 2], label=r"$\omega_{B/N,3}$"
)
```

```python
# Plot Angular Velocity history (R/N) with matching colors and dashed lines
axs[1].plot(
    time_points_11,
    omega_RN_task11_history[:, 0],
    "--",
    color=line1w.get_color(),
    label=r"$\omega_{R/N,1}$",
)
axs[1].plot(
    time_points_11,
    omega_RN_task11_history[:, 1],
    "--",
    color=line2w.get_color(),
    label=r"$\omega_{R/N,2}$",
)
axs[1].plot(
    time_points_11,
    omega_RN_task11_history[:, 2],
    "--",
    color=line3w.get_color(),
    label=r"$\omega_{R/N,3}$",
)

axs[1].set_title("Mission Scenario Angular Velocity (Task 11)")
axs[1].set_xlabel("Time (s)")
axs[1].set_ylabel("Angular Velocity (rad/s)")
axs[1].legend()

plt.tight_layout()
plt.savefig("task11_mission_scenario.png")
```

References

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