# Attitude Dynamics and Control of a Nano-Satellite Orbiting Mars

Timothy J. Russell*
*University of Colorado, Boulder, CO, 80303*

## I. Introduction

The mission studied in this report is that of a nano-satellite in a circular, inclined orbit about Mars. The Keplerian elements of this low-Mars orbit (LMO) spacecraft are

| Element | Symbol | Value | Units |
|---|---|---|---|
| RAAN | $\Omega$ | 20° | - |
| Inclination | $i$ | 30° | - |
| True Anomaly (t=0s) | $\theta_0$ | 60° | - |
| Mean Motion | $n$ | 0.000884797 | $rad/s$ |
| Altitude | $z$ | 400 | $km$ |

**Table 1 LMO Spacecraft Orbit Description**

Due to strict power requirements, whenever the spacecraft is in sunlight it must point its solar array assembly at the sun, assumed to be in a constant inertial direction. This is referred to as "sun-pointing mode".

In order to relay science data and spacecraft bus telemetry back to Earth, a second spacecraft in a circular geosynchronous Mars orbit (GMO) is used as a communications relay. The elements of the GMO spacecraft orbit are

| Element | Symbol | Value | Units |
|---|---|---|---|
| RAAN | $\Omega$ | 0° | - |
| Inclination | $i$ | 0° | - |
| True Anomaly (t=0s) | $\theta_0$ | 250° | - |
| Mean Motion | $n$ | 0.0000709003 | $rad/s$ |
| Altitude | $z$ | 17028.01 | $km$ |

**Table 2 GMO Spacecraft Orbit Description**

If the LMO spacecraft is in shadow and within 35° of the GMO spacecraft as measured from the center of Mars, the LMO spacecraft must point its communication payload boresight at the GMO spacecraft, which is called "GMO-point mode". If neither of the above two conditions apply, the LMO spacecraft must point its science payload boresight at the point on the surface of Mars immediately below the spacecraft, which is referred to as "nadir-point mode."

Mars is assumed to be perfectly spherical with a radius of $3396.19km$ and also uniformly dense. Both spacecraft orbits are assumed to be unperturbed, and infinite, consistent control is assumed.

The LMO spacecraft body axes are aligned with the principal inertia axes, with inertia tensor

$$^{\mathcal{B}}I = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 7.5 \end{bmatrix}^{\mathcal{B}} kgm^2$$

At time $t = 0$, the spacecraft has the following initial attitude and angular rates representing a tumble:

*ProMS Student, Aerospace Engineering Sciences

$$\boldsymbol{\sigma}_{B/N}(t=0) = \begin{bmatrix} 0.3 \\ -0.4 \\ 0.5 \end{bmatrix}$$

$${}^{\mathcal{B}}\boldsymbol{\omega}_{B/N}(t=0) = {}^{\mathcal{B}}\begin{bmatrix} 1.00 \\ 1.75 \\ -2.20 \end{bmatrix} deg/s$$

Attitude is represented using Modified Rodrigues Parameters (MRPs).

## II. Reference Frame Description and Translation

At any time during the mission there are six reference frames of interest:

- Body frame $\mathcal{B}$ - A reference frame aligned with the LMO spacecraft primary axes of inertia, and fortunately also the critical components of the LMO spacecraft. Vector $\hat{\boldsymbol{b}}_1$ is the axis of maximum inertia and is aligned with the science payload boresight. It is also anti-aligned with the communication payload boresight. The solar arrays point in the positive $\hat{\boldsymbol{b}}_3$ direction, which is also the axis of intermediate inertia. Vector $\hat{\boldsymbol{b}}_2 = \hat{\boldsymbol{b}}_3 \times \hat{\boldsymbol{b}}_1$ and is the axis of minimum inertia.
- Inertial frame $\mathcal{N}$ - A reference frame in which $\hat{\boldsymbol{n}}_2$ points from the Mars center of mass towards the Sun (assumed infinitely far away), $\hat{\boldsymbol{n}}_3$ points towards the north pole, and $\hat{\boldsymbol{n}}_1$ satisfied a right-handed system by $\hat{\boldsymbol{n}}_1 = \hat{\boldsymbol{n}}_2 \times \hat{\boldsymbol{n}}_3$.
- Hill frame $\mathcal{O}$ - A reference frame designed to easily calculate derivatives of orbit position and velocity. Vector $\hat{\boldsymbol{i}}_r$ points from the Mars center of mass towards the spacecraft center of mass, $\hat{\boldsymbol{i}}_h$ points in the direction of the orbit angular momentum, and $\hat{\boldsymbol{i}}_\theta = \hat{\boldsymbol{i}}_h \times \hat{\boldsymbol{i}}_r$. For circular orbits such as the ones in this mission, $\hat{\boldsymbol{i}}_\theta$ and the velocity of the spacecraft are co-linear.
- Sun-Pointing frame $\mathcal{S}$ - Used to control the spacecraft such that $\hat{\boldsymbol{b}}_3$ is pointed at the Sun. Vector $\hat{\boldsymbol{s}}_3$ is co-aligned with $\hat{\boldsymbol{n}}_2$, and $\hat{\boldsymbol{s}}_1$ is anti-aligned with $\hat{\boldsymbol{n}}_1$, with $\hat{\boldsymbol{s}}_2 = \hat{\boldsymbol{s}}_3 \times \hat{\boldsymbol{s}}_1$.
- Nadir-Pointing frame $\mathcal{G}$ - Used to control the spacecraft such that $\hat{\boldsymbol{b}}_1$ is pointed at the Mars center of mass. Vector $\hat{\boldsymbol{g}}_1$ is anti-aligned with $\hat{\boldsymbol{i}}_r$, and $\hat{\boldsymbol{g}}_2$ is co-aligned with $\hat{\boldsymbol{i}}_\theta$, with $\hat{\boldsymbol{g}}_3 = \hat{\boldsymbol{g}}_1 \times \hat{\boldsymbol{g}}_2$.
- GMO-Pointing frame $\mathcal{M}$ - Used to control the spacecraft such that -$\hat{\boldsymbol{b}}_1$ is pointed at the GMO spacecraft. Vector $\hat{\boldsymbol{m}}_1$ is aligned with this line-of-sight. Vector $\hat{\boldsymbol{m}}_2$ is selected such that it is orthogonal to both $\hat{\boldsymbol{m}}_1$ and $\hat{\boldsymbol{n}}_3$, and vector $\hat{\boldsymbol{m}}_3$ is found by calculating $\hat{\boldsymbol{m}}_1 \times \hat{\boldsymbol{m}}_2$.

Reference frame $\mathcal{R}$ is used throughout the report to indicate a generic reference frame which could apply to $\mathcal{S}$, $\mathcal{G}$, or $\mathcal{M}$.

### A. Translation Between $\mathcal{N}$ and $\mathcal{O}$

The three angles described above to describe the spacecrafts' orbit planes relative to Mars's equatorial plane, $\Omega$, $i$, and $\theta$, form a 3-1-3 Euler angle set. The first two angles are constant assuming perfect two-body dynamics with no environmental perturbations. True anomaly, however, necessarily changes with time, and so the direction cosine matrix (DCM) which maps from frame $\mathcal{N}$ to frame $\mathcal{O}$ can be written as

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

DCMs are orthogonal, so the inverse is simply the transpose, or $[\mathcal{NO}] = [\mathcal{ON}]^\top$.

As both spacecraft travel in perfectly circular orbits, the orbit mean motion is also the exact rate of change of their true anomaly. At any point, true anomaly can be calculated as

$$\theta(t) = \theta_0 + nt \tag{2}$$

with the answer typically normalized by 360°. Using this, the DCM can be calculated at any time relative to epoch $t_0$.

Since the orbits are perfectly circular and unperturbed, the position and velocity vectors of each spacecraft are unchanging within the orbit frames. The position vector can be represented in orbit frame coordinates at all times as just ${}^{O}\boldsymbol{r} = r\hat{\boldsymbol{i}}_r$. Inertial velocity, too, can be expressed in orbit frame coordinates with the following expression

$$\dot{\boldsymbol{r}} = {}^{O}\frac{d\boldsymbol{r}}{dt} + \boldsymbol{\omega}_{O/\mathcal{N}} \times \boldsymbol{r} \tag{3}$$

Circular orbits see no change in orbit radius, so the orbit frame derivative is simply $\mathbf{0}$. Due to only change in true anomaly, the angular velocity between the two frames is $\boldsymbol{\omega}_{O/\mathcal{N}} = n\hat{\boldsymbol{i}}_h$. This results in the inertial derivative of the position vector being only the transport velocity term, or $\dot{\boldsymbol{r}} = n\hat{\boldsymbol{i}}_\theta \times r\hat{\boldsymbol{i}}_r = rn\hat{\boldsymbol{i}}_\theta$.

With the orbit frame position and velocity vectors defined, inertial frame descriptions of the two vectors are calculated by

$${}^{\mathcal{N}}\boldsymbol{r}(t) = [\mathcal{N}O](t)\,{}^{O}\boldsymbol{r} \tag{4}$$

$${}^{\mathcal{N}}\dot{\boldsymbol{r}}(t) = [\mathcal{N}O](t)\,{}^{O}\dot{\boldsymbol{r}} \tag{5}$$

### B. Translation Between $\mathcal{N}$ and $\mathcal{S}$

The Sun-pointing reference frame $\mathcal{S}$ is invariant with time and has basis vectors that are either co-aligned or anti-aligned with the inertial frame basis vectors. In this frame, $\hat{\boldsymbol{s}}_1 = -\hat{\boldsymbol{n}}_1$, and $\hat{\boldsymbol{s}}_3 = \hat{\boldsymbol{n}}_2$. The final basis vector is found by $\hat{\boldsymbol{s}}_2 = \hat{\boldsymbol{s}}_3 \times \hat{\boldsymbol{s}}_1 = \hat{\boldsymbol{n}}_2 \times -\hat{\boldsymbol{n}}_1 = \hat{\boldsymbol{n}}_3$. When converted to a DCM this is represented as

$$[\mathcal{S}\mathcal{N}] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{6}$$

This DCM does not change with time, so the angular rate between the two frames is $\boldsymbol{\omega}_{\mathcal{S}/\mathcal{N}} = \mathbf{0}$.

### C. Translation Between $\mathcal{N}$ and $\mathcal{G}$

The nadir-pointing reference frame $\mathcal{G}$ is just a modified version of the orbit frame $O$: $\hat{\boldsymbol{g}}_1 = -\hat{\boldsymbol{i}}_r$, $\hat{\boldsymbol{g}}_2 = \hat{\boldsymbol{i}}_\theta$, and thus $\hat{\boldsymbol{g}}_3 = \hat{\boldsymbol{g}}_1 \times \hat{\boldsymbol{g}}_2 = -\hat{\boldsymbol{i}}_r \times \hat{\boldsymbol{i}}_\theta = -\hat{\boldsymbol{i}}_h$. The equivalent DCM is

$$[\mathcal{G}O] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{7}$$

In order to map from inertial coordinates to the nadir-pointing reference frame the appropriate DCMs must be multiplied in succession.

$$[\mathcal{G}\mathcal{N}] = [\mathcal{G}O][O\mathcal{N}] = \begin{bmatrix} -\cos\theta(t)\cos\Omega + \sin\theta(t)\cos i\sin\Omega & -\cos\theta(t)\sin\Omega - \sin\theta(t)\cos i\cos\Omega & -\sin\theta(t)\sin i \\ -\sin\theta(t)\cos\Omega - \cos\theta(t)\cos i\sin\Omega & -\sin\theta(t)\sin\Omega + \cos\theta(t)\cos i\cos\Omega & \cos\theta(t)\sin i \\ -\sin i\sin\Omega & \sin i\cos\Omega & -\cos i \end{bmatrix} \tag{8}$$

The angular rate between the two frames is the sum of the following terms

$$\boldsymbol{\omega}_{\mathcal{G}/\mathcal{N}} = \boldsymbol{\omega}_{\mathcal{G}/O} + \boldsymbol{\omega}_{O/\mathcal{N}} \tag{9}$$

However, the orientation between $\mathcal{G}$ and $O$ does not change over time, and so the corresponding angular rate is $\mathbf{0}$. Thus, $\boldsymbol{\omega}_{\mathcal{G}/\mathcal{N}} = \boldsymbol{\omega}_{O/\mathcal{N}}$. Representing ${}^{\mathcal{N}}\boldsymbol{\omega}_{\mathcal{G}/\mathcal{N}}$ is accomplished by pre-multiplying ${}^{O}\boldsymbol{\omega}_{O/\mathcal{N}}$ by $[\mathcal{N}O](t)$, as is done for position and velocity in Eqs. (4) and (5).

**D. Translation Between $\mathcal{N}$ and $\mathcal{M}$**

The GMO-pointing reference frame $\mathcal{M}$ is meant to point the communication payload of the LMO spacecraft, which has its boresight along -$\hat{\boldsymbol{b}}_1$, at the GMO spacecraft. The first axis is defined as

$$\hat{\boldsymbol{m}}_1 = -\Delta\hat{\boldsymbol{r}} = -\frac{\boldsymbol{r}_{GMO} - \boldsymbol{r}_{LMO}}{|\boldsymbol{r}_{GMO} - \boldsymbol{r}_{LMO}|} \tag{10}$$

The second axis is defined as

$$\hat{\boldsymbol{m}}_2 = \frac{\Delta\boldsymbol{r} \times \hat{\boldsymbol{n}}_3}{|\Delta\boldsymbol{r} \times \hat{\boldsymbol{n}}_3|} \tag{11}$$

Finally, the third axis is orthogonal two the first two as $\hat{\boldsymbol{m}}_3 = \hat{\boldsymbol{m}}_1 \times \hat{\boldsymbol{m}}_2$. From this, the DCM is constructed as

$$[\mathcal{MN}] = \begin{bmatrix} \hat{\boldsymbol{m}}_1^\intercal \\ \hat{\boldsymbol{m}}_2^\intercal \\ \hat{\boldsymbol{m}}_3^\intercal \end{bmatrix} \tag{12}$$

The angular velocity between $\mathcal{M}$ and $\mathcal{N}$ is not easily determined analytically. Instead, the instantaneous angular rate is approximated using the differential kinematic equation for DCMs

$$[\dot{\mathcal{MN}}] = -[\tilde{\omega}][\mathcal{MN}] \tag{13}$$

which is rearranged to yield

$$[\tilde{\omega}] = -[\dot{\mathcal{MN}}][\mathcal{MN}]^\intercal \tag{14}$$

To estimate $\dot{\mathcal{MN}}$ at a given time $t$, a linear approximation centered on $t$ is given by

$$[\dot{\mathcal{MN}}](t) \approx \frac{[\mathcal{MN}](t + \delta t) - [\mathcal{MN}](t - \delta t)}{2\delta t} \tag{15}$$

A value of $\delta t = 0.001s$ is sufficiently small enough to yield accurate results. Once the DCM and its instantaneous derivative are calculated, the components of $\boldsymbol{\omega}$ can be extracted from $[\tilde{\omega}]$.

## III. State Errors

In order to control attitude and rates, whether for regulation or tracking purposes, the error in both states with respect to a reference frame must be calculated for each time step. The integrator as described in Section IV propagates attitudes and rates with respect to the inertial frame, i.e. $\boldsymbol{\sigma}_{B/N}$ and $\boldsymbol{\omega}_{B/N}$. However, the control torque as described in Section V is calculated in terms of $\boldsymbol{\sigma}_{B/R}$ and $\boldsymbol{\omega}_{B/R}$.

The translation of attitude with respect to inertial to attitude with respect to reference can be performed with the right DCMs. The DCM for a given body-to-inertial MRP set can be found by

$$[\mathcal{BN}] = [I_{3x3}] + \frac{8[\tilde{\boldsymbol{\sigma}}_{B/N}]^2 - 4(1 - \sigma_{B/N}^2)[\tilde{\boldsymbol{\sigma}}_{B/N}]}{(1 + \sigma_{B/N}^2)^2} \tag{16}$$

with $\sigma_{B/N}^2 = \boldsymbol{\sigma}_{B/N}^\intercal \boldsymbol{\sigma}_{B/N}$.

The reference frame DCMs can be found as described in Section II. This allows the body-to-reference DCM to be found by $[\mathcal{BR}] = [\mathcal{BN}][\mathcal{RN}]$. The body-to-reference MRPs are then extracted using

$$\boldsymbol{\sigma}_{B/R} = \frac{1}{\xi(\xi + 2)} \begin{bmatrix} [\mathcal{BR}]_{23} - [\mathcal{BR}]_{32} \\ [\mathcal{BR}]_{31} - [\mathcal{BR}]_{13} \\ [\mathcal{BR}]_{12} - [\mathcal{BR}]_{21} \end{bmatrix} \tag{17}$$

where

$$\xi = \sqrt{tr([\mathcal{BR}]) + 1} \tag{18}$$

Note that $\xi$ goes to 0 at a 180° rotation, resulting in a singularity for (17). To avoid this, Sheppard's method can be used to find a quaternion set from the DCM, and then the MRP set can be calculated as

$$\sigma_i = \frac{\beta_i}{1+\beta_0} \tag{19}$$

For body-to-reference rate errors, the body-to-inertial DCM is used to convert the reference frame angular rate to body frame coordinates, and then body-to-reference rates can be found through vector subtraction

$${}^{\mathcal{B}}\boldsymbol{\omega}_{\mathcal{B}/\mathcal{R}} = {}^{\mathcal{B}}\boldsymbol{\omega}_{\mathcal{B}/N} - [\mathcal{BN}]\,{}^{N}\boldsymbol{\omega}_{\mathcal{R}/N} \tag{20}$$

## IV. Numerical Integrator

In order to perform attitude and rate tracking and regulation, the spacecraft state at any given time is represented as

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

and the state update is represented simply as

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

The differential kinematic equation for MRPs is

$$\dot{\boldsymbol{\sigma}} = \frac{1}{4}\left[(1-\sigma^2)[I_{3x3}] + 2[\tilde{\boldsymbol{\sigma}}] + 2\boldsymbol{\sigma}\boldsymbol{\sigma}^{\mathsf{T}}\right]\boldsymbol{\omega} \tag{21}$$

The spacecraft is assumed to be a rigid body which obeys Euler's equation of motion for rotation

$$[I]\dot{\boldsymbol{\omega}}_{B/N} = -[\tilde{\boldsymbol{\omega}_{B/N}}][I]\boldsymbol{\omega}_{B/N} + \boldsymbol{L} + \boldsymbol{u} \tag{22}$$

with $\boldsymbol{L}$ representing the net environmental torque acting on the body and $\boldsymbol{u}$ representing the net control torque acting on the body. Unless specified, $\boldsymbol{L}$ is assumed to be $\boldsymbol{0}$.

It is clear to see that attitude change is dependent on angular rates. However, the control law as described in Section V is dependent on attitude, and thus angular rate change is dependent on attitude. Since both state variables are coupled to each other, any integrator used to solve these differential equations must calculate both states in tandem.

Due to its accuracy at relatively low computation cost, a fourth-order Runge-Kutta (RK4) integrator is used to propagate spacecraft state over time. For a given time-step (assumed to be 1 second for the rest of this report), the integrator performs the following steps:

1) Determine the desired reference frame (based on calculations outlined in Section VI)
2) Calculate the DCM and angular rate of the desired reference frame with respect to the inertial frame $[\mathcal{RN}], \boldsymbol{\omega}_{R/N}$
3) Calculate attitude and rate errors with respect to the desired reference frame $\boldsymbol{\sigma}_{B/R}, \boldsymbol{\omega}_{B/R}$
4) Calculate environmental torque $\boldsymbol{L}$ based on the current state $\boldsymbol{X}$
5) Calculate control torque $\boldsymbol{u}$ based on state errors $\boldsymbol{\sigma}_{B/R}, \boldsymbol{\omega}_{B/R}$
6) Propagate the state using
    1) $k_1 = \Delta t f\left(\boldsymbol{X}, t, \boldsymbol{L}, \boldsymbol{u}\right)$
    2) $k_2 = \Delta t f\left(\boldsymbol{X} + {}^{k_1}/_{2}, t + {}^{\Delta t}/_{2}, \boldsymbol{L}, \boldsymbol{u}\right)$
    3) $k_3 = \Delta t f\left(\boldsymbol{X} + {}^{k_2}/_{2}, t + {}^{\Delta t}/_{2}, \boldsymbol{L}, \boldsymbol{u}\right)$
    4) $k_2 = \Delta t f\left(\boldsymbol{X} + k_3, t + \Delta t, \boldsymbol{L}, \boldsymbol{u}\right)$
    5) $\boldsymbol{X}(t+\Delta t) = \boldsymbol{X} + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$
7) Switch next timestep MRP to shadow set if $|\boldsymbol{\sigma}_{B/N}| > 1$

For the above, $f(\boldsymbol{X}, \ldots)$ is Eqn. (21) for attitude and Eqn. (22) for attitude rates.

### A. Torque-Free Motion

For the initial tumbling conditions outlined in Section I, and with no external torques, the spacecraft state propagates as follows in Fig. 1over 500 seconds.

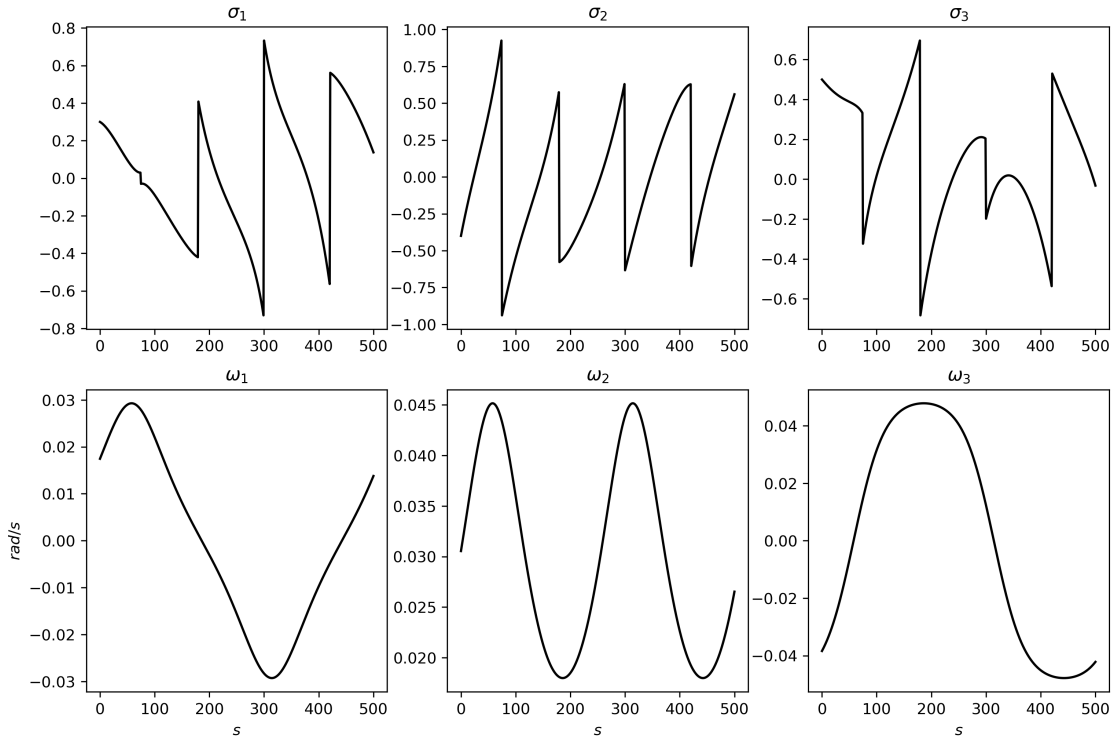


**Fig. 1 Torque-Free Tumbling for 500 Seconds**

### B. Motion with Constant External Torque

Assuming the same initial conditions but instead applying a constant environmental torque of $\boldsymbol{L} = [0.01 \; -0.01 \; 0.02]^{\intercal} Nm$, the spacecraft state evolves quite differently over 500 seconds as shown in Fig 2.

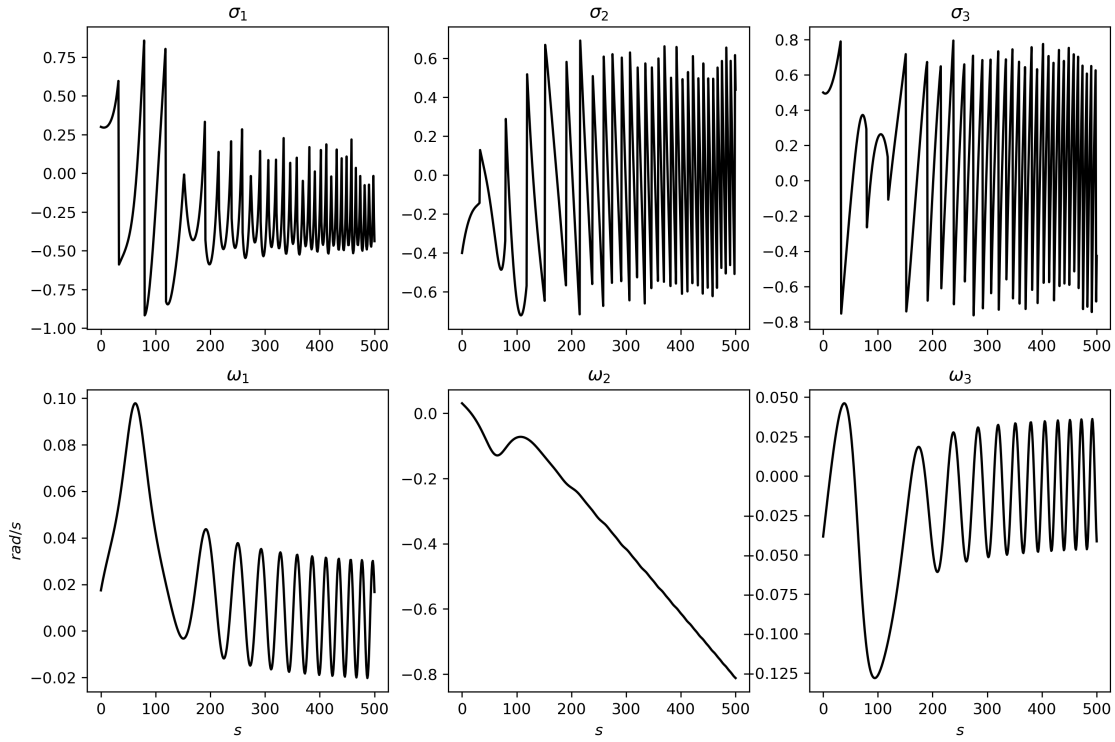


**Fig. 2 Tumbling with Constant External Torque for 500 Seconds**

This YORP-like effect would quickly cause the spacecraft to disintegrate.

## V. Spacecraft Control

The control law used for pointing is dependent on both attitude and rate errors for the $\mathcal{B}$ frame with respect to one of the three reference frames

$$^{\mathcal{B}}\boldsymbol{u} = -K\boldsymbol{\sigma}_{B/R} - P\,{}^{\mathcal{B}}\boldsymbol{\omega}_{B/R} \tag{23}$$

where $K$ and $P$ are constant linear gains. The selected value for $P$ was driven by the requirement that the slowest spacecraft mode has a settling time constant, the time at which tracking errors are reduced to $1/e \approx 36.79\%$ of initial errors, be 120 seconds. For a given mode, the time constant is calculated as

$$\tau_i = \frac{2I_i}{P} \tag{24}$$

The value $I_i$ represents the primary inertia for axis $i$. For a value of $P = 1/6$, the time constants are $\tau_1 = 120s$, $\tau_2 = 60s$, and $\tau_3 = 90s$.

A second requirement is that one mode be critically damped with the others underdamped. The damping ratio is found with

$$\zeta_i = \frac{P}{\sqrt{KI_i}} \tag{25}$$

With $P = 1/6$ and $K = 1/180$, the damping ratios of the different modes are $\zeta_1 = 985/1393$, $\zeta_2 = 1$, and $\zeta_3 = 881/1079$.

### A. Sun-Pointing Performance

From the initial tumbling conditions provided in Section I, the calculated control law behaves as in Figs. 3 and 4, with 750 seconds of propagation shown.

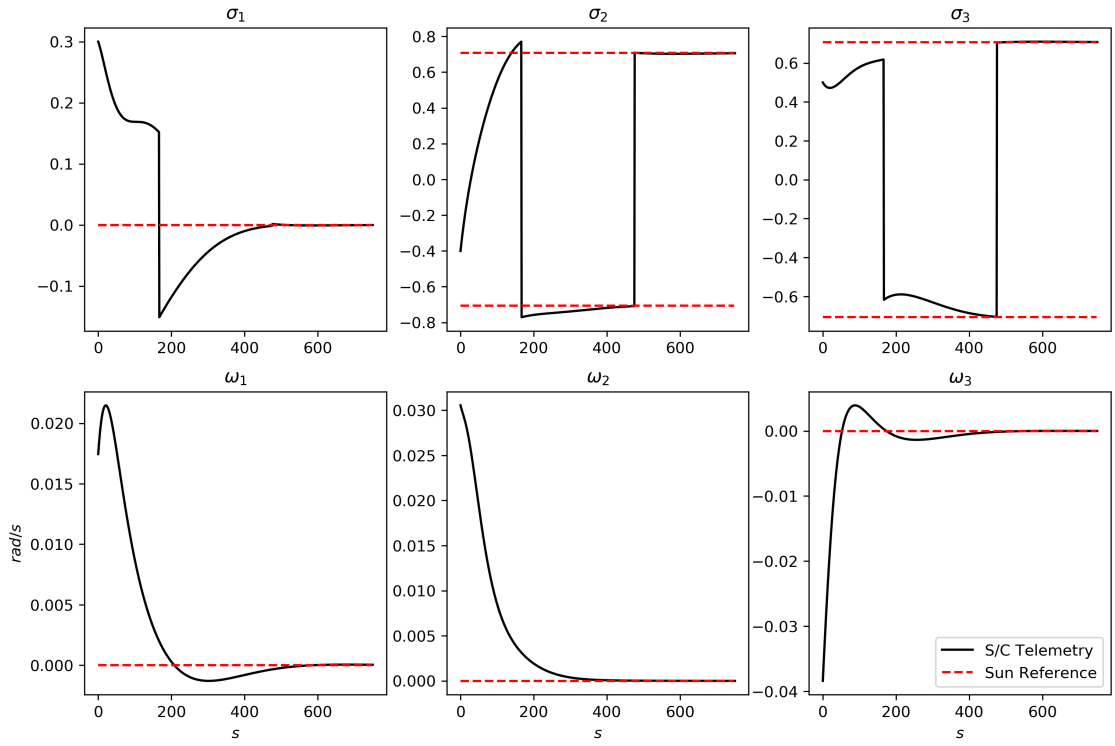


**Fig. 3 Sun-Track Mode for 750 Seconds, Spacecraft States with Reference Frame Included**

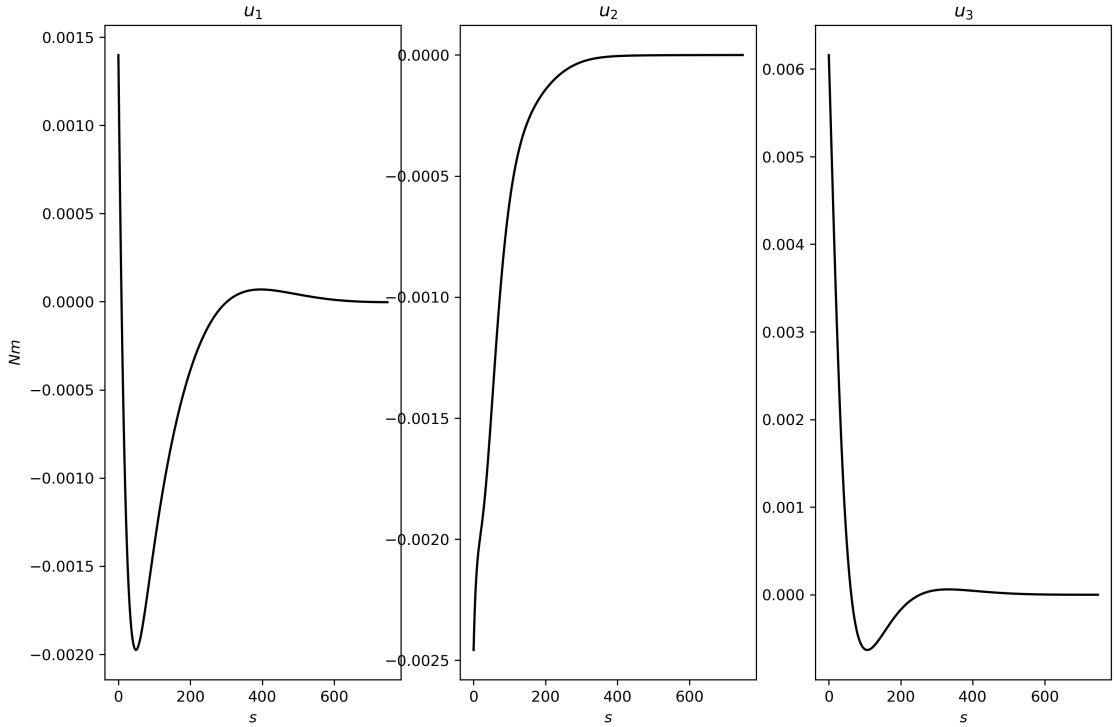


**Fig. 4 Sun-Track Mode for 750 Seconds, Control Torques**

## B. Nadir-Pointing Performance

The same initial conditions and propagation time were tested for nadir-pointing control, shown in Figs. 5 and 6.

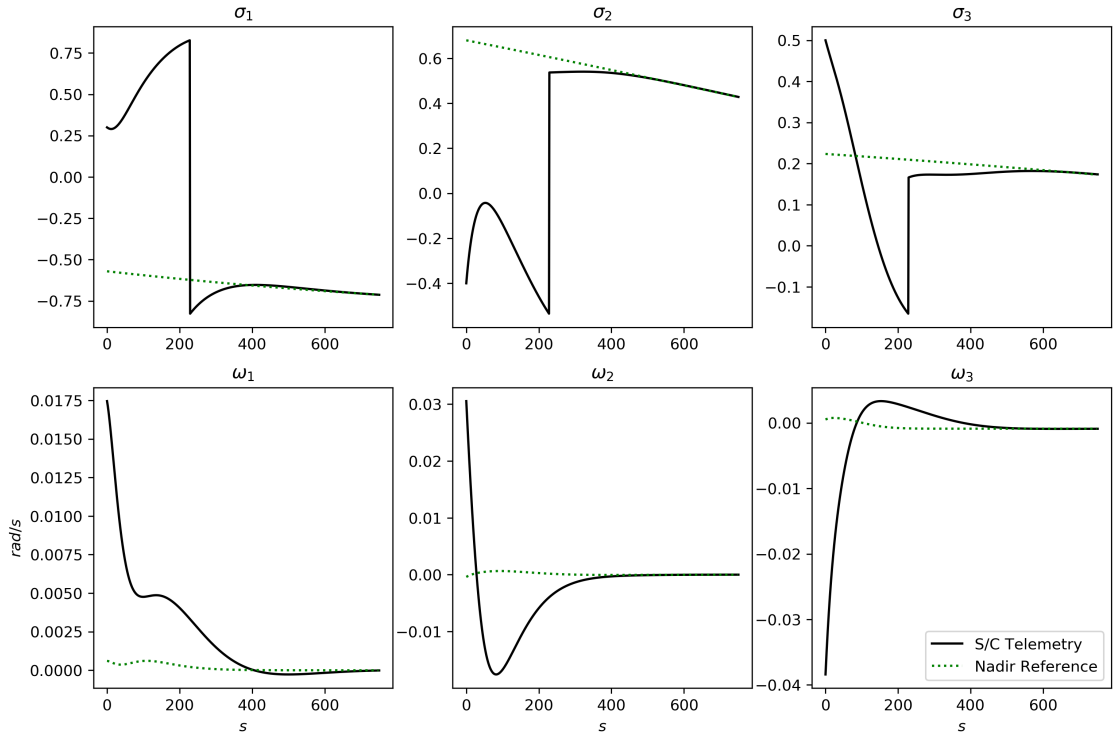


**Fig. 5 Nadir-Track Mode for 750 Seconds, Spacecraft States with Reference Frame Included**

## C. GMO-Pointing Performance

The performance of the GMO-pointing control given the same initial conditions and time is displayed in Figs. 7 and 8.

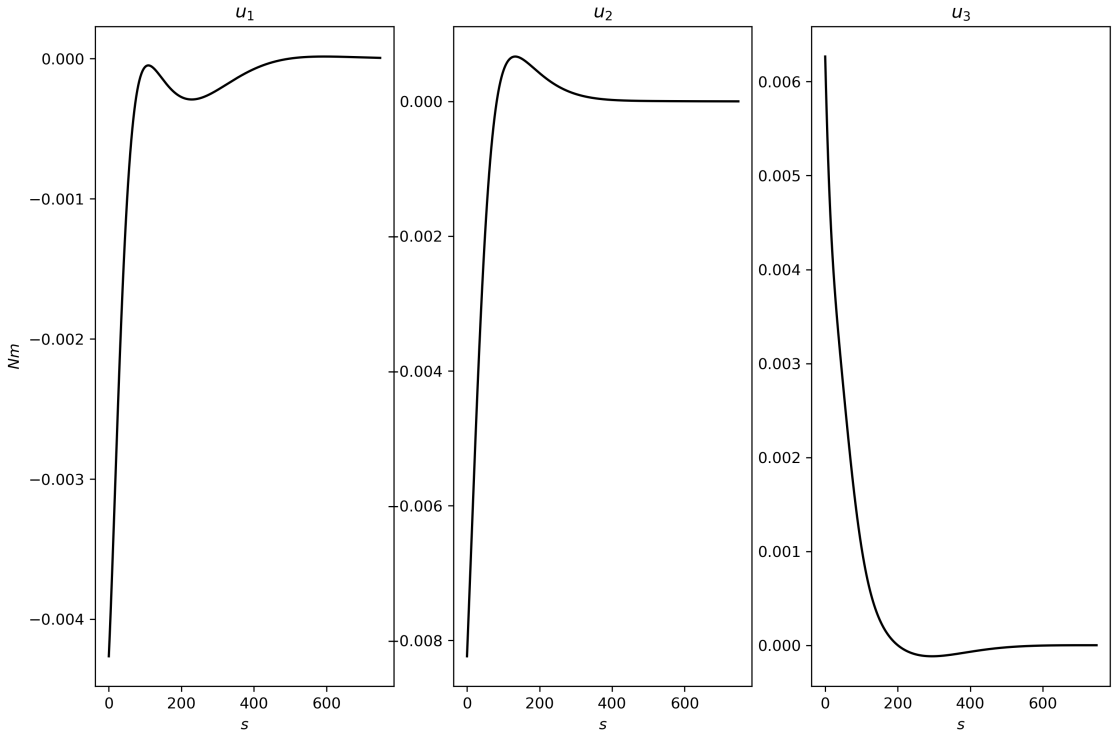


**Fig. 6 Nadir-Track Mode for 750 Seconds, Control Torques**

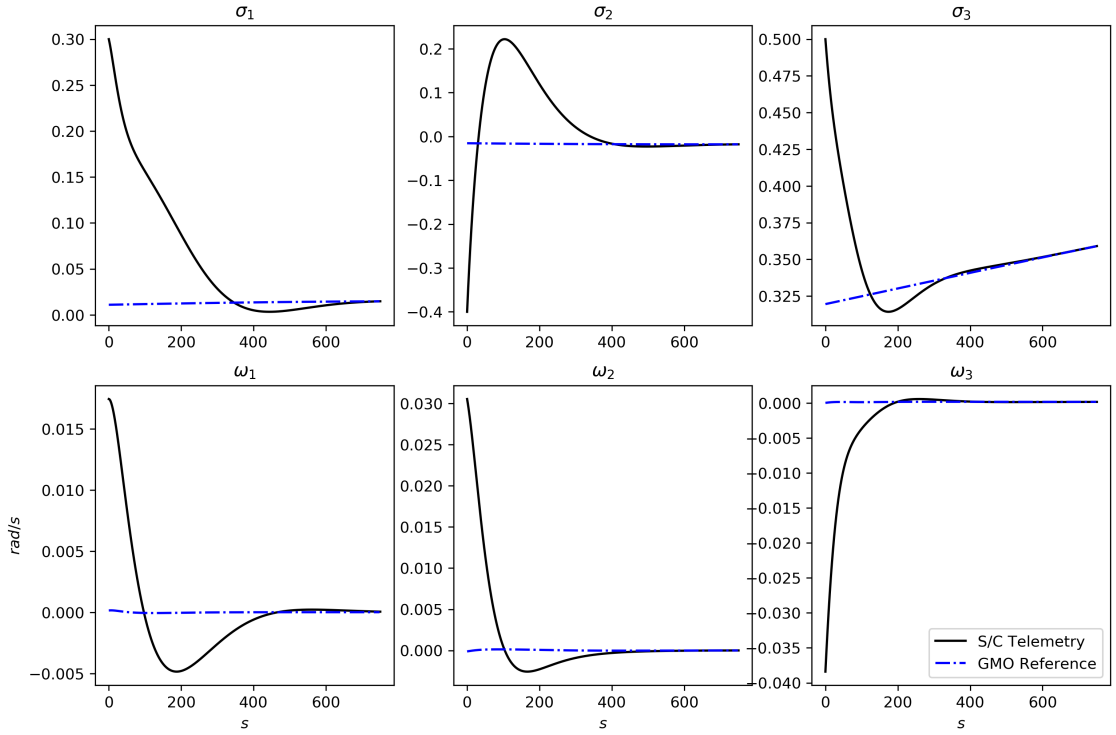


**Fig. 7 GMO-Track Mode for 750 Seconds, Spacecraft States with Reference Frame Included**

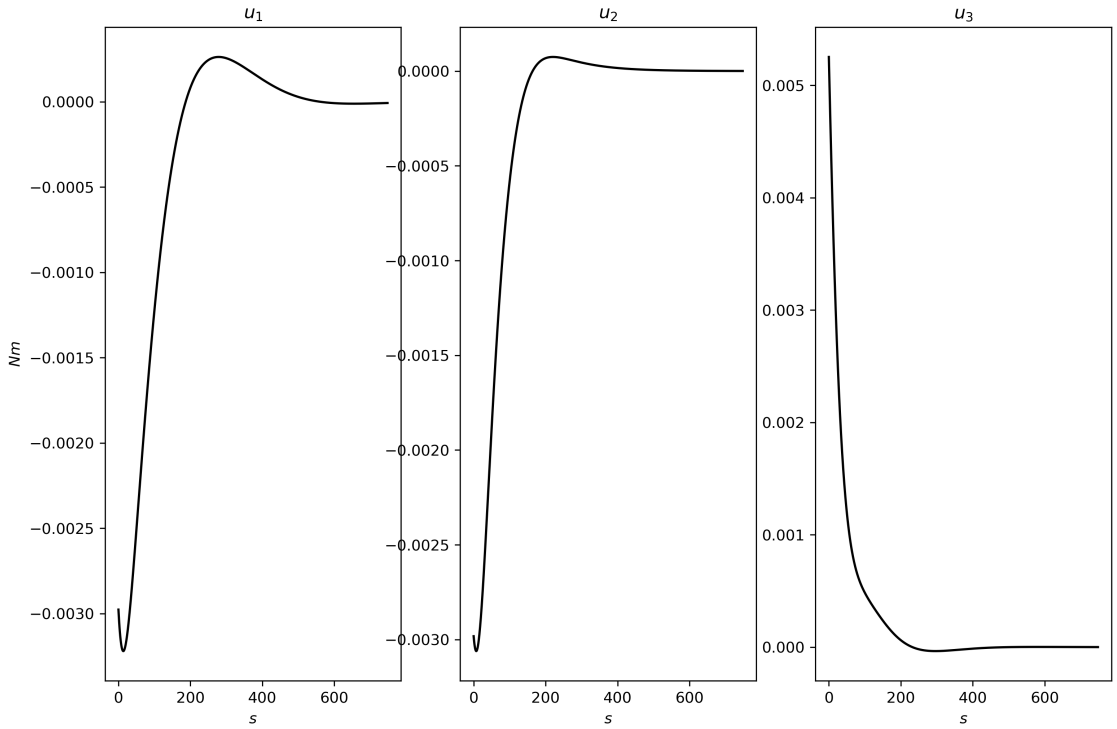


**Fig. 8 GMO-Track Mode for 750 Seconds, Control Torques**

## VI. Mission Simulation

As mentioned in Section I, the LMO spacecraft must be in sun-pointing mode at all times in sunlight in order to meet power requirements. At each time step the integrator checks this by calculating

$$^{\mathcal{N}}\boldsymbol{r}_{LMO} \cdot \hat{\boldsymbol{n}}_2 > 0$$

If not in sun-pointing mode, the next highest priority is the GMO-pointing mode. The communication link is guaranteed when the angle between the two spacecraft as measured from the center of Mars is less than 35°, which is calculated at each time step as

$$acos\left(\frac{^{\mathcal{N}}\boldsymbol{r}_{LMO} \cdot {}^{\mathcal{N}}\boldsymbol{r}_{GMO}}{|\,^{\mathcal{N}}\boldsymbol{r}_{LMO}||\,^{\mathcal{N}}\boldsymbol{r}_{GMO}|}\right) < 35°$$

If neither of the above apply, the nadir-pointing mode is selected.

The integrator with the initial tumbling conditions was propagated for 10,000 seconds to gauge performance. The spacecraft attitude and rates with respect to the $\mathcal{N}$ frame (with the latter expressed in $\mathcal{B}$ frame coordinates) are illustrated below in Fig. 9. Mode transitions are easily spotted by the spikes in angular velocities as the control law actuates the spacecraft to a new reference frame.

Something to note is to how the attitude set switches between nominal and shadow sets fairly regularly during sun-pointing mode. This is due to the reference frame being exactly 180° away from the $\mathcal{N}$ frame. Rounding errors in the integrator push the magnitude of $\boldsymbol{\omega}_{B/N}$ slightly above 1, causing the opposite set to be used. However, this does not have any impact on performance or invoke any control torques, since the value $\boldsymbol{\sigma}_{B/S}$ remains at 0. A "cleaner" look at how the spacecraft is switching modes can be found by having the propagator only switch between MRP sets at $\boldsymbol{\omega}_{B/N} > 1.1$. The results of a 10,000 second simulation are shown in Fig. 10. As control torque is dependent attitude error this could result in slightly different torque profiles and thus mission performance, but this exercise is purely for illustrative purposes.

For reference, the control torques for the first 10,000 second simulation are shown in Fig. 11.

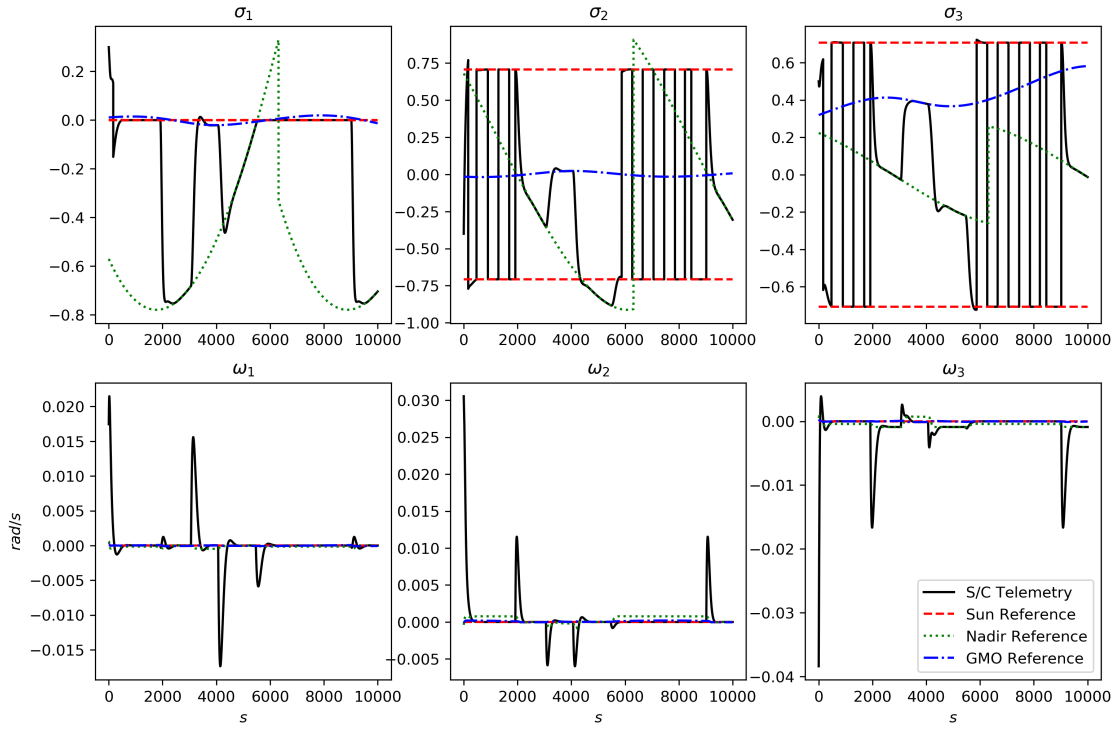


**Fig. 9 Mission Simulation for 10,000 Seconds, Spacecraft States with Reference Frames Included**

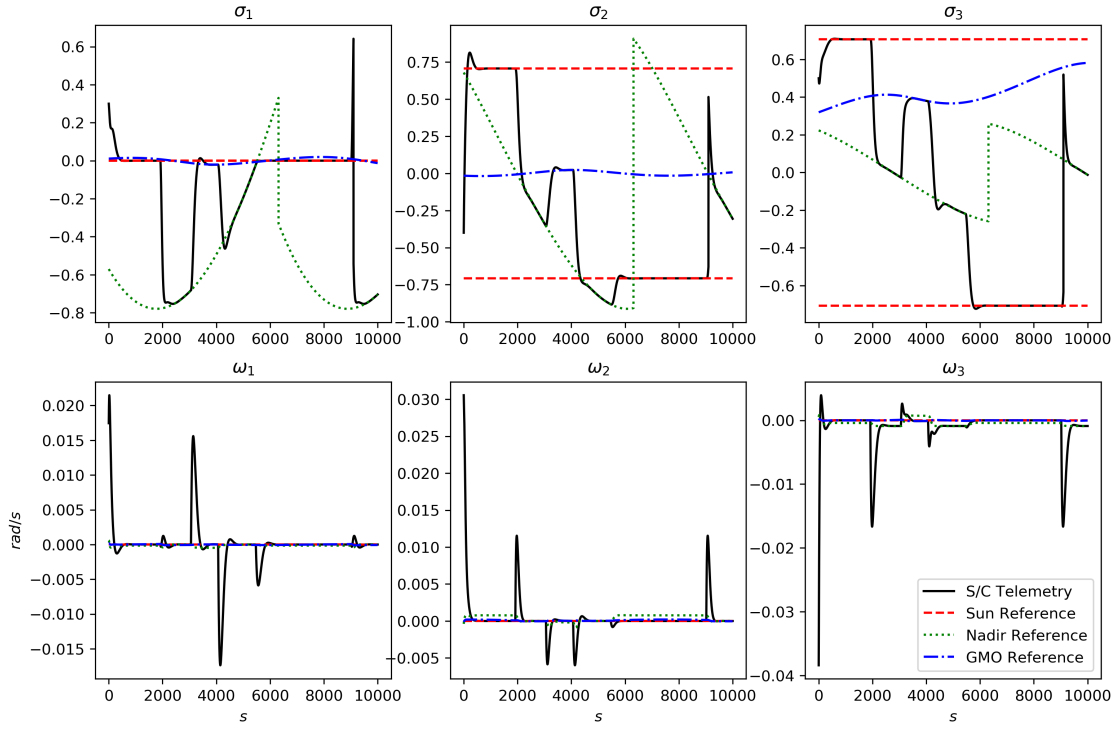


**Fig. 10 Mission Simulation for 10,000 Seconds, Spacecraft States with Reference Frames Included, Reduced MRP Switching**

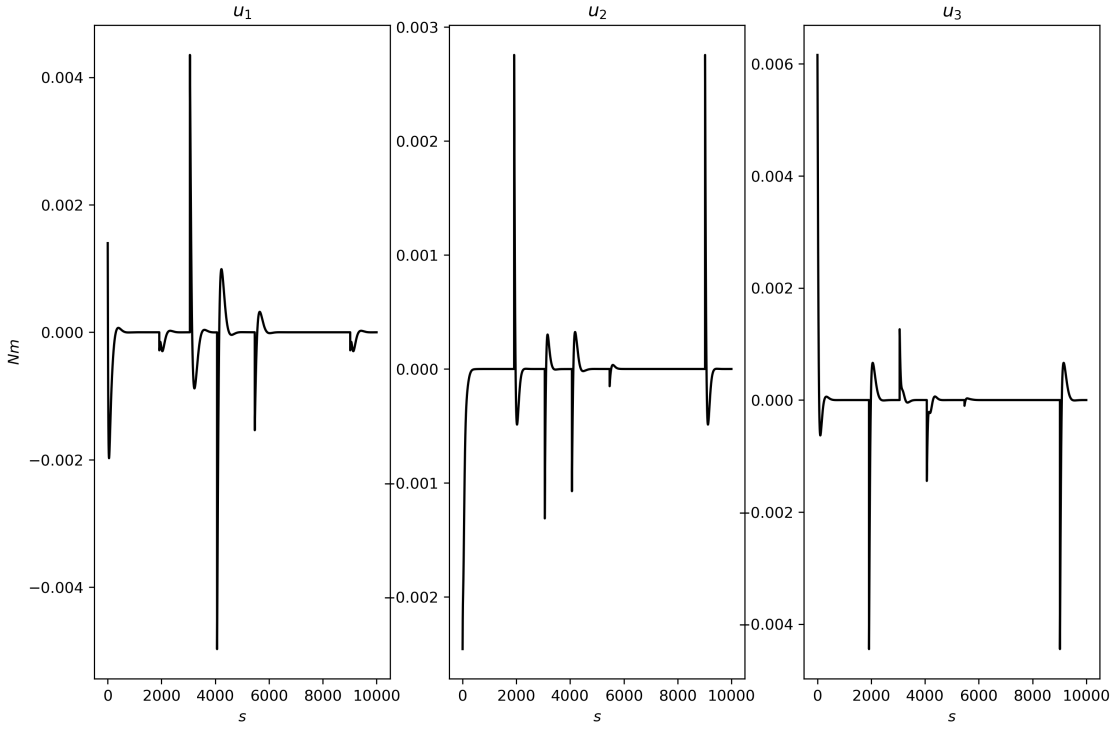


**Fig. 11 Mission Simulation for 10,000 Seconds, Control Torques**

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 19 18:36:29 2020

@author: Tim Russell

Calculates the DCMs to convert from the inertial frame to reference frames.
"""

def dcm_orbit(RAAN, inc, TA0, n, t):
    import numpy as np
    import math
    from calc_TA import calc_TA

    TA = calc_TA(TA0, n, t)

    c1 = math.cos(RAAN * math.pi/180)
    s1 = math.sin(RAAN * math.pi/180)
    c2 = math.cos(inc * math.pi/180)
    s2 = math.sin(inc * math.pi/180)
    c3 = math.cos(TA * math.pi/180)
    s3 = math.sin(TA * math.pi/180)

    DCM_ON = np.array([[c3*c1-s3*c2*s1, c3*s1+s3*c2*c1, s3*s2],
                       [-s3*c1-c3*c2*s1, -s3*s1+c3*c2*c1, c3*s2],
                       [s2*s1, -s2*c1, c2]])

    return DCM_ON


def dcm_sun():
    import numpy as np

    DCM_SN = np.array([[-1, 0, 0],
                       [0, 0, 1],
                       [0, 1, 0]])

    return DCM_SN


def dcm_nadir(RAAN, inc, TA0, n, t):
    import numpy as np

    DCM_ON = dcm_orbit(RAAN, inc, TA0, n, t)

    DCM_GO = np.array([[-1, 0, 0],
                       [0, 1, 0],
                       [0, 0, -1]])

    DCM_GN = DCM_GO @ DCM_ON

    return DCM_GN


def dcm_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
            RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO, t):
    import numpy as np
```

```
    from numpy.linalg import norm
    from cross_matrix import cross_matrix

    O_r_LMO = np.array([[r_LMO],[0],[0]])
    DCM_NO_LMO = np.transpose(dcm_orbit(RAAN_LMO, inc_LMO, TA0_LMO, n_LMO, t))
    N_r_LMO = DCM_NO_LMO @ O_r_LMO

    O_r_GMO = np.array([[r_GMO],[0],[0]])
    DCM_NO_GMO = np.transpose(dcm_orbit(RAAN_GMO, inc_GMO, TA0_GMO, n_GMO, t))
    N_r_GMO = DCM_NO_GMO @ O_r_GMO

    N_delr = N_r_GMO - N_r_LMO

    m1 = -N_delr / norm(N_delr)

    m2 = cross_matrix(N_delr) @ np.array([[0],[0],[1]])
    m2 = m2 / norm(m2)

    m3 = cross_matrix(m1) @ m2
    m3 = m3 / norm(m3)

    DCM_MN = np.transpose(m1)
    DCM_MN = np.append(DCM_MN, np.transpose(m2), axis=0)
    DCM_MN = np.append(DCM_MN, np.transpose(m3), axis=0)

    return DCM_MN


def dcm_body(sigma_BN):
    import numpy as np
    from numpy.linalg import norm
    from cross_matrix import cross_matrix

    sigma_tilde_BN = cross_matrix(sigma_BN)

    DCM_BN = np.identity(3) + (8*sigma_tilde_BN@sigma_tilde_BN - \
                               4*(1-norm(sigma_BN)**2)*sigma_tilde_BN) / \
                              (1+norm(sigma_BN)**2)**2

    return DCM_BN
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Apr  4 14:50:52 2020

@author: Tim Russell

Calculates the true anomaly of a circular orbit given initial true anomaly
(deg), mean motion (rad/s), and time elapsed (s). Normalizes to [0, 360) deg.
"""

def calc_TA(TA0, n, t):
    import math
    n = n * 180/math.pi
    TA = (TA0 + n*t) % 360

    return TA
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 19 18:17:32 2020

@author: Tim Russell

Calculates the angular rates of reference frames with respect to the inertial
frame.

"""

def angrate_orbit(RAAN, inc, TA0, n, t):
    import numpy as np
    from project_dcms import dcm_orbit

    O_omega_ON = np.array([[0],[0],[n]])
    DCM_NO = np.transpose(dcm_orbit(RAAN, inc, TA0, n, t))
    N_omega_ON = DCM_NO @ O_omega_ON

    return N_omega_ON

def angrate_sun():
    import numpy as np

    N_omega_SN = np.array([[0],[0],[0]])

    return N_omega_SN

def angrate_nadir(RAAN, inc, TA0, n, t):
    """
    Calculate the angular rate of the nadir-pointing reference frame with
    respect to the inertial frame. Note that this orbit is circular and
    unperturbed.

    Parameters
    ----------
    RAAN : float (deg)
        Right ascension of the ascending node of the spacecraft orbit.
    inc : float (deg)
        Inclination of the spacecraft orbit.
    TA0 : float (deg)
        True anomaly of the spacecraft in its orbit at epoch t=0.
    n : float (rad/s)
        Mean motion of the spacecraft orbit.
    t : float (s)
        Time elapsed since t=0.

    Returns
    -------
    ang_rate : float (rad/s)
        Angular rate of the nadir-pointing reference frame with respect to the
        inertial frame. Represented in inertial frame coordinates.

    """
    import numpy as np
    from project_dcms import dcm_nadir
```

```
    G_omega_GN = np.array([[0],[0],[-n]])
    DCM_NG = np.transpose(dcm_nadir(RAAN, inc, TA0, n, t))

    N_omega_GN = DCM_NG @ G_omega_GN

    return N_omega_GN


def angrate_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
                RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO, t):
    """
    Calculate the angular rate of the GMO-pointing reference frame with
    respect to the inertial frame. Note that the orbits of the two spacecraft
    are circular and unperturbed.

    Parameters
    ----------
    RAAN_LMO : float (deg)
        Right ascension of the ascending node of the LMO spacecraft orbit.
    inc_LMO : float (deg)
        Inclination of the LMO spacecraft orbit.
    TA0_LMO : float (deg)
        True anomaly of the LMO spacecraft in its orbit at epoch t=0.
    r_LMO : float (km)
        Raidus of the LMO spacecraft orbit.
    n_LMO : float (rad/s)
        Mean motion of the LMO spacecraft orbit.
    RAAN_GMO : float (deg)
        Right ascension of the ascending node of the GMO spacecraft orbit.
    inc_GMO : float (deg)
        Inclination of the GMO spacecraft orbit.
    TA0_GMO : float (deg)
        True anomaly of the GMO spacecraft in its orbit at epoch t=0.
    r_GMO : float (km)
        Raidus of the GMO spacecraft orbit.
    n_GMO : float (rad/s)
        Mean motion of the GMO spacecraft orbit.
    t : float (s)
        Time elapsed since t=0.

    Returns
    -------
    ang_rate : float (rad/s)
        Angular rate of the GMO-pointing reference frame with respect to the
        inertial frame. Represented in inertial frame coordinates.

    """
    import numpy as np
    from project_dcms import dcm_gmo

    # Calculate [MN](t)
    DCM_MN = dcm_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
                     RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO, t)
    # Calculate [MN](t-dt)
    mDCM_MN = dcm_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
                      RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO, t-0.001)
    # Calculate [MN](t+dt)
```

```
pDCM_MN = dcm_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
                  RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO, t+0.001)
# Calculate d[MN]/dt(t)
DCMdot_MN = (pDCM_MN - mDCM_MN)/0.002

# Calculate w_tilde = -d[MN]/dt*[MN]'
omega_tilde_MN = -DCMdot_MN @ np.transpose(DCM_MN)

# Calculate w in M frame
M_omega_MN = np.array([[omega_tilde_MN[2][1]],[omega_tilde_MN[0][2]],
                       [omega_tilde_MN[1][0]]])
# Calculate w in N frame
N_omega_MN = np.transpose(DCM_MN) @ M_omega_MN

return N_omega_MN
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 19 20:55:54 2020

@author: Tim Russell

Calculate attitude and rate errors of a body frame with respect to a reference
frame.

"""

def attitude_error(sigma_BN, DCM_RN):
    import numpy as np
    from numpy.linalg import norm
    import math
    from cross_matrix import cross_matrix

    sigma_tilde_BN = cross_matrix(sigma_BN)

    DCM_BN = np.identity(3) + (8*sigma_tilde_BN@sigma_tilde_BN - \
                               4*(1-norm(sigma_BN)**2)*sigma_tilde_BN) / \
                              (1+norm(sigma_BN)**2)**2

    DCM_BR = DCM_BN @ np.transpose(DCM_RN)

    xi = math.sqrt(np.trace(DCM_BR)+1)

    sigma_BR = 1/(xi*(xi+2))*np.array([[DCM_BR[1][2]-DCM_BR[2][1]],
                                       [DCM_BR[2][0]-DCM_BR[0][2]],
                                       [DCM_BR[0][1]-DCM_BR[1][0]]])

    return sigma_BR


def rate_error(sigma_BN, B_omega_BN, DCM_RN, N_omega_RN):
    import numpy as np
    from numpy.linalg import norm
    from cross_matrix import cross_matrix

    sigma_BN_tilde = cross_matrix(sigma_BN)

    DCM_BN = np.identity(3) + (8*sigma_BN_tilde@sigma_BN_tilde - \
                               4*(1-norm(sigma_BN)**2)*sigma_BN_tilde) / \
                              (1+norm(sigma_BN)**2)**2

    B_omega_RN = DCM_BN @ N_omega_RN
    B_omega_BR = B_omega_BN - B_omega_RN

    return B_omega_BR
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 19 21:19:33 2020

@author: Tim Russell

"""

def torque_free_integrator(X0, tspan, tstep, I):
    import numpy as np
    from numpy.linalg import norm
    from mrp_dke import mrp_dke
    from euler_eom import euler_eom
    from mrp_shadow_set import mrp_shadow_set
    import matplotlib.pyplot as plt

    steps = int(tspan/tstep)

    X = np.zeros((6, steps+1))
    t = np.zeros(steps+1)

    X[:,0] = X0.T
    t[0] = 0

    for i in range(0, steps):
        # Current state
        sigma = np.array([[X[0,i]],[X[1,i]],[X[2,i]]])
        omega = np.array([[X[3,i]],[X[4,i]],[X[5,i]]])

        # Uncomment the following if you want to ensure angular momentum and
        # kinetic energy preserved at each step
        H = norm(I@omega)
        T = 0.5*(I[0,0]*omega[0,0]**2 + I[1,1]*omega[1,0]**2 +
                 I[2,2]*omega[2,0]**2)
        print(f'H = {H}, T = {T}\n')

        # k1
        omega1 = omega
        omega_dot1 = euler_eom(I, omega1, 0)
        k1_omega = tstep*omega_dot1
        sigma1 = sigma
        sigma_dot1 = mrp_dke(sigma1, omega1)
        k1_sigma = tstep*sigma_dot1

        # k2
        omega2 = omega + 0.5*k1_omega
        omega_dot2 = euler_eom(I, omega2, 0)
        k2_omega = tstep*omega_dot2
        sigma2 = sigma + 0.5*k1_sigma
        sigma_dot2 = mrp_dke(sigma2, omega2)
        k2_sigma = tstep*sigma_dot2

        # k3
        omega3 = omega + 0.5*k2_omega
        omega_dot3 = euler_eom(I, omega3, 0)
        k3_omega = tstep*omega_dot3
        sigma3 = sigma + 0.5*k2_sigma
```

```
        sigma_dot3 = mrp_dke(sigma3, omega3)
        k3_sigma = tstep*sigma_dot3

        # k4
        omega4 = omega + k3_omega
        omega_dot4 = euler_eom(I, omega4, 0)
        k4_omega = tstep*omega_dot4
        sigma4 = sigma + k3_sigma
        sigma_dot4 = mrp_dke(sigma4, omega4)
        k4_sigma = tstep*sigma_dot4

        # Next state
        sigma_next = sigma + 1/6*(k1_sigma + 2*k2_sigma +
                                  2*k3_sigma + k4_sigma)
        omega_next = omega + 1/6*(k1_omega + 2*k2_omega +
                                  2*k3_omega + k4_omega)

        # Shadow set conversion if necessary
        if norm(sigma_next) > 1:
            sigma_next = mrp_shadow_set(sigma_next)

        X[:,i+1] = np.append(sigma_next, omega_next).T
        t[i+1] = tstep*(i+1)

    plt.figure(num=None, figsize=(12, 8), dpi=300, facecolor='w', edgecolor='k')

    plt.subplot(2, 3, 1)
    plt.plot(t, X[0,:], '-k')
    plt.title('$\sigma_1$')

    plt.subplot(2, 3, 2)
    plt.plot(t, X[1,:], '-k')
    plt.title('$\sigma_2$')

    plt.subplot(2, 3, 3)
    plt.plot(t, X[2,:], '-k')
    plt.title('$\sigma_3$')

    plt.subplot(2, 3, 4)
    plt.plot(t, X[3,:], '-k')
    plt.title('$\omega_1$')
    plt.xlabel('$s$')
    plt.ylabel('$rad/s$')

    plt.subplot(2, 3, 5)
    plt.plot(t, X[4,:], '-k')
    plt.title('$\omega_2$')
    plt.xlabel('$s$')

    plt.subplot(2, 3, 6)
    plt.plot(t, X[5,:], '-k')
    plt.title('$\omega_3$')
    plt.xlabel('$s$')

    return X, t

def constant_torque_integrator(X0, tspan, tstep, I):
```

```
import numpy as np
from numpy.linalg import norm
from mrp_dke import mrp_dke
from euler_eom import euler_eom
from mrp_shadow_set import mrp_shadow_set
import matplotlib.pyplot as plt

steps = int(tspan/tstep)

X = np.zeros((6, steps+1))
t = np.zeros(steps+1)
u = np.array([[0.01],[-0.01],[0.02]])

X[:,0] = X0.T
t[0] = 0

for i in range(0, steps):
    # Current state
    sigma = np.array([[X[0,i]],[X[1,i]],[X[2,i]]])
    omega = np.array([[X[3,i]],[X[4,i]],[X[5,i]]])

    # Uncomment the following if you want to ensure angular momentum and
    # kinetic energy preserved at each step
    H = norm(I@omega)
    T = 0.5*(I[0,0]*omega[0,0]**2 + I[1,1]*omega[1,0]**2 +
             I[2,2]*omega[2,0]**2)
    print(f'H = {H}, T = {T}\n')

    # k1
    omega1 = omega
    omega_dot1 = euler_eom(I, omega1, u)
    k1_omega = tstep*omega_dot1
    sigma1 = sigma
    sigma_dot1 = mrp_dke(sigma1, omega1)
    k1_sigma = tstep*sigma_dot1

    # k2
    omega2 = omega + 0.5*k1_omega
    omega_dot2 = euler_eom(I, omega2, u)
    k2_omega = tstep*omega_dot2
    sigma2 = sigma + 0.5*k1_sigma
    sigma_dot2 = mrp_dke(sigma2, omega2)
    k2_sigma = tstep*sigma_dot2

    # k3
    omega3 = omega + 0.5*k2_omega
    omega_dot3 = euler_eom(I, omega3, u)
    k3_omega = tstep*omega_dot3
    sigma3 = sigma + 0.5*k2_sigma
    sigma_dot3 = mrp_dke(sigma3, omega3)
    k3_sigma = tstep*sigma_dot3

    # k4
    omega4 = omega + k3_omega
    omega_dot4 = euler_eom(I, omega4, u)
    k4_omega = tstep*omega_dot4
    sigma4 = sigma + k3_sigma
```

```
        sigma_dot4 = mrp_dke(sigma4, omega4)
        k4_sigma = tstep*sigma_dot4

        # Next state
        sigma_next = sigma + 1/6*(k1_sigma + 2*k2_sigma +
                                  2*k3_sigma + k4_sigma)
        omega_next = omega + 1/6*(k1_omega + 2*k2_omega +
                                  2*k3_omega + k4_omega)

        # Shadow set conversion if necessary
        if norm(sigma_next) > 1:
            sigma_next = mrp_shadow_set(sigma_next)

        X[:,i+1] = np.append(sigma_next, omega_next).T
        t[i+1] = tstep*(i+1)

    plt.figure(num=None, figsize=(12, 8), dpi=300, facecolor='w', edgecolor='k')

    plt.subplot(2, 3, 1)
    plt.plot(t, X[0,:], '-k')
    plt.title('$\sigma_1$')

    plt.subplot(2, 3, 2)
    plt.plot(t, X[1,:], '-k')
    plt.title('$\sigma_2$')

    plt.subplot(2, 3, 3)
    plt.plot(t, X[2,:], '-k')
    plt.title('$\sigma_3$')

    plt.subplot(2, 3, 4)
    plt.plot(t, X[3,:], '-k')
    plt.title('$\omega_1$')
    plt.xlabel('$s$')
    plt.ylabel('$rad/s$')

    plt.subplot(2, 3, 5)
    plt.plot(t, X[4,:], '-k')
    plt.title('$\omega_2$')
    plt.xlabel('$s$')

    plt.subplot(2, 3, 6)
    plt.plot(t, X[5,:], '-k')
    plt.title('$\omega_3$')
    plt.xlabel('$s$')

    return X, t

def pd_feedback_integrator(X0, tspan, tstep, I, K, P, RAAN_LMO, inc_LMO,
                           TA0_LMO, n_LMO, r_LMO, RAAN_GMO, inc_GMO, TA0_GMO,
                           n_GMO, r_GMO):
    import numpy as np
    from numpy.linalg import norm
    from mrp_dke import mrp_dke
    from euler_eom import euler_eom
    from mrp_shadow_set import mrp_shadow_set
    from project_dcms import dcm_orbit, dcm_sun, dcm_nadir, dcm_gmo, dcm_body
```

```
from project_angular_rates import angrate_sun, angrate_nadir, angrate_gmo
from project_errors import attitude_error, rate_error
import math
import matplotlib.pyplot as plt

steps = int(tspan/tstep)

X = np.zeros((6, steps+1))
X[:,0] = X0.T

t = np.zeros(steps+1)

u = np.zeros((3, steps+1))

sigma_SN = np.zeros((3, steps+1))
omega_SN = np.zeros((3, steps+1))

sigma_GN = np.zeros((3, steps+1))
omega_GN = np.zeros((3, steps+1))

sigma_MN = np.zeros((3, steps+1))
omega_MN = np.zeros((3, steps+1))

O_r_LMO = np.array([[r_LMO],[0],[0]])
O_r_GMO = np.array([[r_GMO],[0],[0]])

for i in range(0, steps):

    # Current state
    sigma = np.array([[X[0,i]],[X[1,i]],[X[2,i]]])
    omega = np.array([[X[3,i]],[X[4,i]],[X[5,i]]])

    # Determine inertial positions of each spacecraft
    DCM_ON_LMO = dcm_orbit(RAAN_LMO, inc_LMO, TA0_LMO, n_LMO, t[i])
    N_r_LMO = np.transpose(DCM_ON_LMO) @ O_r_LMO
    DCM_ON_GMO = dcm_orbit(RAAN_GMO, inc_GMO, TA0_GMO, n_GMO, t[i])
    N_r_GMO = np.transpose(DCM_ON_GMO) @ O_r_GMO

    # Determine if LMO spacecraft is on shaded side
    sun_mode = bool((np.transpose(N_r_LMO) @ np.array([[0],[1],[0]])
                     > 0))

    # Determine if LMO and GMO comm link closed
    gmo_mode = bool(math.acos((np.transpose(N_r_LMO) @ N_r_GMO)/(r_LMO*r_GMO))
                    < 35*math.pi/180)

    # Feedback control torque
    if sun_mode:
        DCM_SN = dcm_sun()
        sigma_BS = attitude_error(sigma, DCM_SN)
        angrate_SN = angrate_sun()
        omega_BS = rate_error(sigma, omega, DCM_SN, angrate_SN)
        torque = -K*sigma_BS - P*omega_BS
    elif gmo_mode:
        DCM_MN = dcm_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
                         RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO, t[i])
        sigma_BM = attitude_error(sigma, DCM_MN)
```

```
    angrate_MN = angrate_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
                             RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO,
                             t[i])
    omega_BM = rate_error(sigma, omega, DCM_MN, angrate_MN)
    torque = -K*sigma_BM - P*omega_BM
else:
    DCM_GN = dcm_nadir(RAAN_LMO, inc_LMO, TA0_LMO, n_LMO, t[i])
    sigma_BG = attitude_error(sigma, DCM_GN)
    angrate_GN = angrate_nadir(RAAN_LMO, inc_LMO, TA0_LMO, n_LMO, t[i])
    omega_BG = rate_error(sigma, omega, DCM_GN, angrate_GN)
    torque = -K*sigma_BG - P*omega_BG

# Uncomment the following if you want to ensure angular momentum and
# kinetic energy preserved at each step
# H = norm(I@omega)
# T = 0.5*(I[0,0]*omega[0,0]**2 + I[1,1]*omega[1,0]**2 +
#           I[2,2]*omega[2,0]**2)
# print(f'H = {H}, T = {T}\n, u = {norm(u)}\n')

# k1
omega1 = omega
omega_dot1 = euler_eom(I, omega1, torque)
k1_omega = tstep*omega_dot1
sigma1 = sigma
sigma_dot1 = mrp_dke(sigma1, omega1)
k1_sigma = tstep*sigma_dot1

# k2
omega2 = omega + 0.5*k1_omega
omega_dot2 = euler_eom(I, omega2, torque)
k2_omega = tstep*omega_dot2
sigma2 = sigma + 0.5*k1_sigma
sigma_dot2 = mrp_dke(sigma2, omega2)
k2_sigma = tstep*sigma_dot2

# k3
omega3 = omega + 0.5*k2_omega
omega_dot3 = euler_eom(I, omega3, torque)
k3_omega = tstep*omega_dot3
sigma3 = sigma + 0.5*k2_sigma
sigma_dot3 = mrp_dke(sigma3, omega3)
k3_sigma = tstep*sigma_dot3

# k4
omega4 = omega + k3_omega
omega_dot4 = euler_eom(I, omega4, torque)
k4_omega = tstep*omega_dot4
sigma4 = sigma + k3_sigma
sigma_dot4 = mrp_dke(sigma4, omega4)
k4_sigma = tstep*sigma_dot4

# Next state
sigma_next = sigma + 1/6*(k1_sigma + 2*k2_sigma +
                          2*k3_sigma + k4_sigma)
omega_next = omega + 1/6*(k1_omega + 2*k2_omega +
                          2*k3_omega + k4_omega)
```

```
# Shadow set conversion if necessary
if norm(sigma_next) > 1:
    sigma_next = mrp_shadow_set(sigma_next)

# Populate next step state
X[:,i+1] = np.append(sigma_next, omega_next).T
t[i+1] = tstep*(i+1)

# Calculate non-state variables for making cool* plots
# *unofficial opinion of author

# Control torque
u[0,i] = torque[0]
u[1,i] = torque[1]
u[2,i] = torque[2]

# Body-Inertial DCM
DCM_BN = dcm_body(sigma)

# Sun frame attitude and rates
sigma_SN[0,i] = 0
sigma_SN[1,i] = 0.7071
sigma_SN[2,i] = 0.7071

N_omega_SN = angrate_sun()
B_omega_SN = DCM_BN @ N_omega_SN
omega_SN[0,i] = B_omega_SN[0]
omega_SN[1,i] = B_omega_SN[1]
omega_SN[2,i] = B_omega_SN[2]

# Nadir frame attitude and rates
DCM_GN = dcm_nadir(RAAN_LMO, inc_LMO, TA0_LMO, n_LMO, t[i])
xi = math.sqrt(np.trace(DCM_GN)+1)
MRP_GN = 1/(xi*(xi+2))*np.array([[DCM_GN[1][2]-DCM_GN[2][1]],
                                 [DCM_GN[2][0]-DCM_GN[0][2]],
                                 [DCM_GN[0][1]-DCM_GN[1][0]]])
sigma_GN[0,i] = MRP_GN[0]
sigma_GN[1,i] = MRP_GN[1]
sigma_GN[2,i] = MRP_GN[2]

N_omega_GN = angrate_nadir(RAAN_LMO, inc_LMO, TA0_LMO, n_LMO, t[i])
B_omega_GN = DCM_BN @ N_omega_GN
omega_GN[0,i] = B_omega_GN[0]
omega_GN[1,i] = B_omega_GN[1]
omega_GN[2,i] = B_omega_GN[2]

# GMO frame attitude and rates
DCM_MN = dcm_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
                 RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO, t[i])
xi = math.sqrt(np.trace(DCM_MN)+1)
MRP_MN = 1/(xi*(xi+2))*np.array([[DCM_MN[1][2]-DCM_MN[2][1]],
                                 [DCM_MN[2][0]-DCM_MN[0][2]],
                                 [DCM_MN[0][1]-DCM_MN[1][0]]])
sigma_MN[0,i] = MRP_MN[0]
sigma_MN[1,i] = MRP_MN[1]
sigma_MN[2,i] = MRP_MN[2]
```

```
    N_omega_MN = angrate_gmo(RAAN_LMO, inc_LMO, TA0_LMO, r_LMO, n_LMO,
                             RAAN_GMO, inc_GMO, TA0_GMO, r_GMO, n_GMO, t[i])
    B_omega_MN = DCM_BN @ N_omega_MN
    omega_MN[0,i] = B_omega_MN[0]
    omega_MN[1,i] = B_omega_MN[1]
    omega_MN[2,i] = B_omega_MN[2]

plt.figure(num=None, figsize=(12, 8), dpi=300, facecolor='w', edgecolor='k')
plt.title('Spacecraft Attitude and Rates (w/r/t Inertial Frame)')

plt.subplot(2, 3, 1)
plt.plot(t, X[0,:], '-k')
plt.plot(t[0:-2], sigma_SN[0,0:-2], '--r')
plt.plot(t[0:-2], -sigma_SN[0,0:-2], '--r')
plt.plot(t[0:-2], sigma_GN[0,0:-2], ':g')
plt.plot(t[0:-2], sigma_MN[0,0:-2], '-.b')
plt.title('$\sigma_1$')

plt.subplot(2, 3, 2)
plt.plot(t, X[1,:], '-k')
plt.plot(t[0:-2], sigma_SN[1,0:-2], '--r')
plt.plot(t[0:-2], -sigma_SN[1,0:-2], '--r')
plt.plot(t[0:-2], sigma_GN[1,0:-2], ':g')
plt.plot(t[0:-2], sigma_MN[1,0:-2], '-.b')
plt.title('$\sigma_2$')

plt.subplot(2, 3, 3)
plt.plot(t, X[2,:], '-k')
plt.plot(t[0:-2], sigma_SN[2,0:-2], '--r')
plt.plot(t[0:-2], -sigma_SN[2,0:-2], '--r')
plt.plot(t[0:-2], sigma_GN[2,0:-2], ':g')
plt.plot(t[0:-2], sigma_MN[2,0:-2], '-.b')
plt.title('$\sigma_3$')

plt.subplot(2, 3, 4)
plt.plot(t, X[3,:], '-k')
plt.plot(t[0:-2], omega_SN[0,0:-2], '--r')
plt.plot(t[0:-2], omega_GN[0,0:-2], ':g')
plt.plot(t[0:-2], omega_MN[0,0:-2], '-.b')
plt.title('$\omega_1$')
plt.xlabel('$s$')
plt.ylabel('$rad/s$')

plt.subplot(2, 3, 5)
plt.plot(t, X[4,:], '-k')
plt.plot(t[0:-2], omega_SN[1,0:-2], '--r')
plt.plot(t[0:-2], omega_GN[1,0:-2], ':g')
plt.plot(t[0:-2], omega_MN[1,0:-2], '-.b')
plt.title('$\omega_2$')
plt.xlabel('$s$')

ax = plt.subplot(2, 3, 6)
ax.plot(t, X[5,:], '-k', label='S/C Telemetry')
ax.plot(t[0:-2], omega_SN[2,0:-2], '--r', label='Sun Reference')
ax.plot(t[0:-2], omega_GN[2,0:-2], ':g', label='Nadir Reference')
ax.plot(t[0:-2], omega_MN[2,0:-2], '-.b', label='GMO Reference')
plt.title('$\omega_3$')
```

```
plt.xlabel('$s$')
ax.legend()

plt.figure(num=None, figsize=(12, 8), dpi=300, facecolor='w', edgecolor='k')
plt.title('Control Torque (Body Frame)')

plt.subplot(1, 3, 1)
plt.plot(t[0:-2], u[0,0:-2], '-k')
plt.title('$u_1$')
plt.xlabel('$s$')
plt.ylabel('$Nm$')

plt.subplot(1, 3, 2)
plt.plot(t[0:-2], u[1,0:-2], '-k')
plt.title('$u_2$')
plt.xlabel('$s$')

plt.subplot(1, 3, 3)
plt.plot(t[0:-2], u[2,0:-2], '-k')
plt.title('$u_3$')
plt.xlabel('$s$')

return X, t, u, sigma_SN, omega_SN, sigma_GN, omega_GN, sigma_MN, omega_MN
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 29 13:32:06 2020

@author: Tim Russell

Calculates the instantaneous attitude rates in terms of MRPs. Inputs are an
MRP attitude description and an attitude rate vector (in rad/s).
"""


def mrp_dke(sigma, omega):
    import numpy as np
    from numpy.linalg import norm
    from cross_matrix import cross_matrix

    sigma_tilde = cross_matrix(sigma)
    B_sigma = (1 - norm(sigma)**2)*np.identity(3) + 2*sigma_tilde + \
              2*sigma@sigma.T
    sigma_dot = 0.25*B_sigma@omega

    return sigma_dot
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 29 14:20:32 2020

@author: Tim Russell
"""


def mrp_shadow_set(sigma):
    from numpy.linalg import norm

    sigma_s = -sigma/norm(sigma)**2

    return sigma_s
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 29 13:46:00 2020

@author: Tim Russell

Uses Euler's equation of motion for rotating bodies to calculate the
instantaneous attitude rate derivative (rad/s^2) given a body inertia tensor
(kg-m^2/s), an attitude rate vector (rad/s), and a net external torque vector
(kg-m^2/s^2)
"""


def euler_eom(I, omega, L):
    from numpy.linalg import inv
    from cross_matrix import cross_matrix

    omega_tilde = cross_matrix(omega)

    omega_dot = inv(I) @ (-omega_tilde @ I @ omega + L)

    return omega_dot
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 22 11:38:51 2020

@author: Tim Russell

Converts a 3x1 vector into a skew-symmetric matrix equivalent to the cross
product operator.
"""

def cross_matrix(vector):
    import numpy as np
    a=vector[0,0]
    b=vector[1,0]
    c=vector[2,0]

    matrix = np.array([[0, -c, b],
                       [c, 0, -a],
                       [-b, a, 0]])
    return matrix
```