AERO 424 Final Project

Texas A&M University
AERO-424-500 ADCS-101
Andrew Hollister

December 13, 2021

Due: December 14, 2021

Contents

1 Abstract

1.1 Assignment Description

You are provided with 1) a set of ADCS requirements, and 2) a baseline design for the new Hubble Double mission below. Your job in this final project assignment is to verify that you can meet all the requirements with the baseline design. Provide evidence showing that you can indeed meet them. To do that you will have to develop an attitude simulator building upon the work you did in previous assignments. Use a combination of analysis and simulation results and plots to provide the evidence that you can meet the requirements.

This project assignment is somewhat open-ended. If you can't meet all of the requirements, you are asked to modify the design to meet your requirements. As such, there is no single right answer. Use your judgment to apply those changes. If you think those changes are too costly or impractical, you can adjust the requirements as needed as a last resort. In any case, please justify all your changes to either the design or the requirements.

1.2 ADCS Requirements

1. Maneuver 1: Slew 30deg cross-track.
   a. Perform maneuver in less than 30 min (settling time)
   b. Steady state error less than 0.1 deg

2. Hold attitude constant (body frame aligned with inertial frame)
   a. Pointing accuracy better than 0.15 deg

3. Detumbling: From an initial state where omega = [0.1,0.1,0.1], bring the spacecraft to omega_bi = [0,0,0] and attitude = 0,0,0 (body frame = inertial frame)
   a. Perform maneuver in less than 12 hours
   b. Steady state error less than 0.1 deg

4. Momentum dumping: Fire thrusters for enough time to bring saturated reaction wheel to omega = 0
   a. At most one maneuver per wheel per day
   b. Less than a minute

1.3 Additional Questions

1. Verify that you can slew 30deg cross track in less than 30 minutes open loop (no controller). You may consider only one axis. What is the level of torque required?
2. Verify that you can meet the requirement that the steady state error for the 30 -deg slewing maneuver is less than 0.1 deg.
3. Implement a PID controller for each wheel and adjust the gains so that the closed -loop system is stable, and you can meet your time specification requirements and

steady -state requirements (hint: you may want to use the quaternion error rather than the Euler angles - remember the difficulty ode45 has with Euler angles).

4. Verify that you can keep a fixed attitude (body frame aligned with the inertial frame) in the presence of the disturbance torques to within .05 arcsec. (Use closed-loop control)
5. Verify that starting from [0,0,0,1] and 0 angular rates in the body frame, the attitude quaternion (body to orbital) after 10 orbits looks like Fig.1, and starting from [0.5,0.5,0.5,0.5] yields Fig. 2 considering only disturbance torques. (No controller)

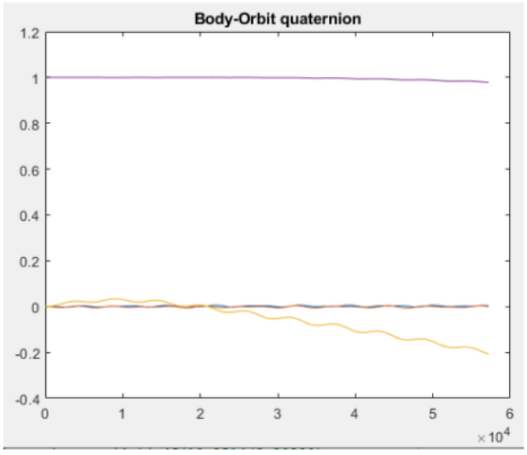


Fig. 1: Starting from [0,0,0,1]

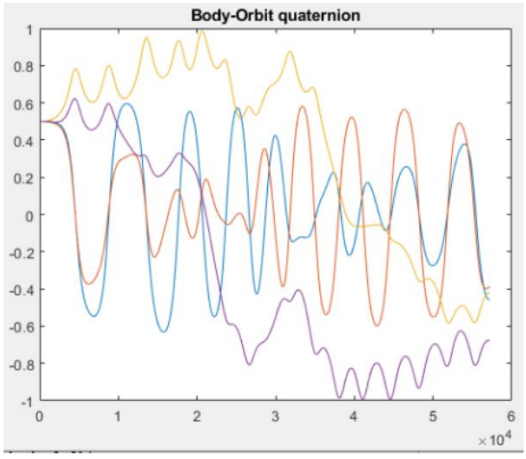


Fig 2: Starting from [0.5,0.5,0.5,0.5].

2 High Accuracy Target Pointing

2.1 Description

Within this pointing mode, the spacecraft was desired to hold am attitude aligned with the inertial frame to within an accuracy of 0.05 arcseconds. The satellite was programmed to do an initial maneuver in order to induce a more realistic degree of error. Additionally, after 5400 seconds, the control algorithm adjusts the maximum torques allowed to 0.1 Nm in order to reduce the jitter experience by the spacecraft.

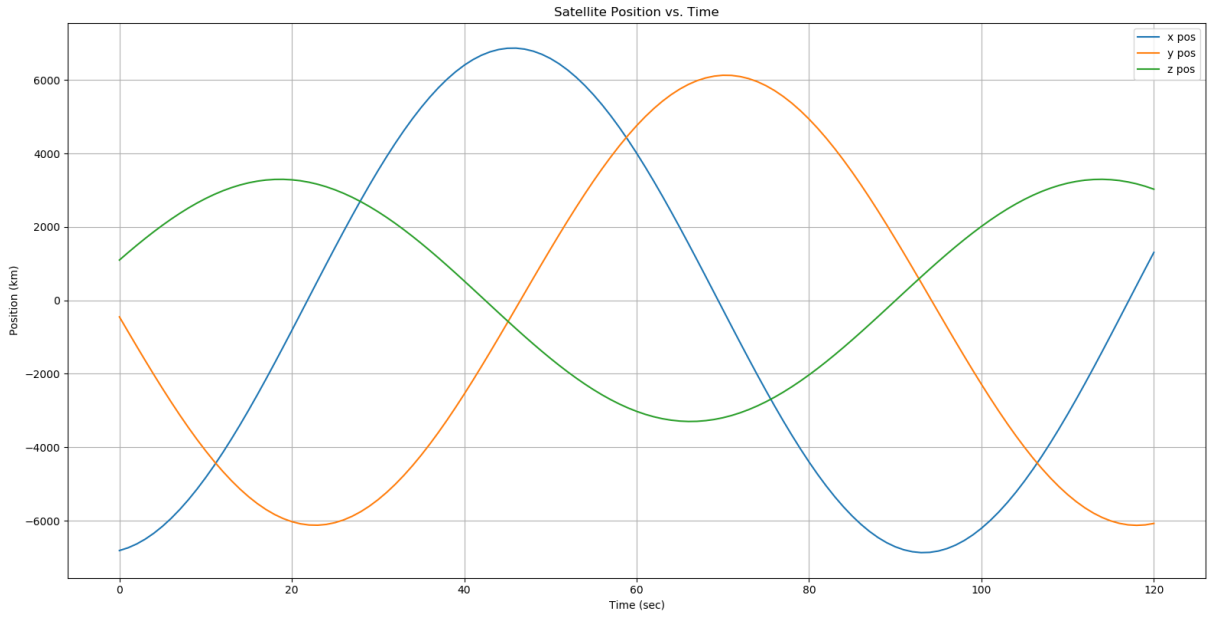
2.2 Modified Inputs/Requirements

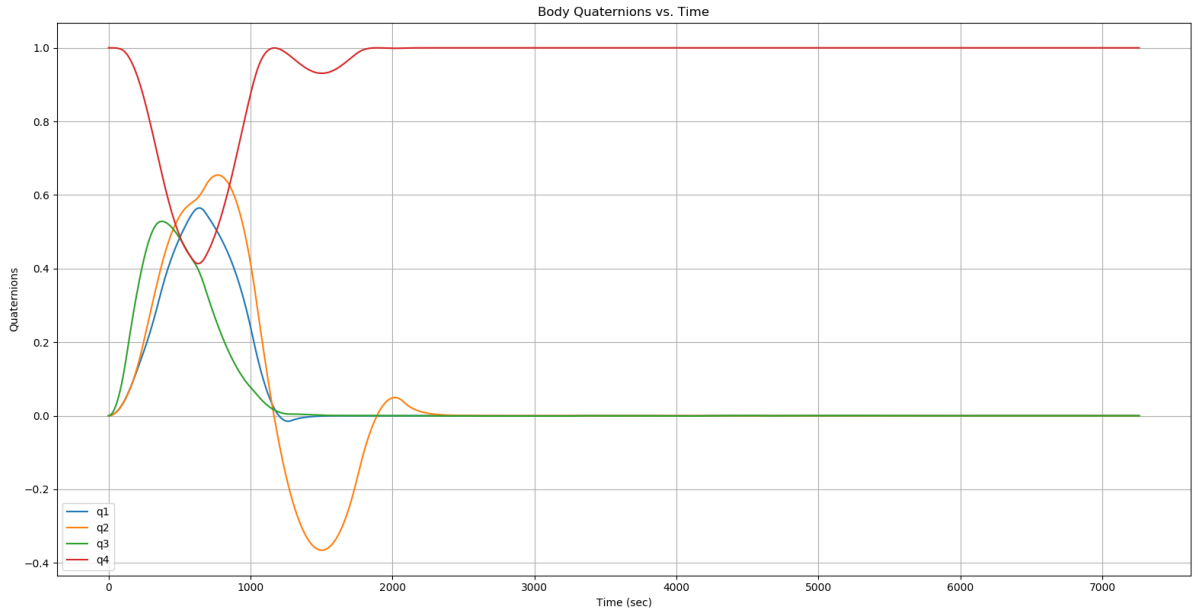
- The reaction wheel moments of inertia were modified to 3 kg*m^2
- The max wheel speed was configured to 700 rad/s (approximately equivalent to ISS CMG angular rates)
- No greater accuracy than 3*10^-5 rad was achievable. The 0.05 arcsec requirement will be modified to 10 arcseconds

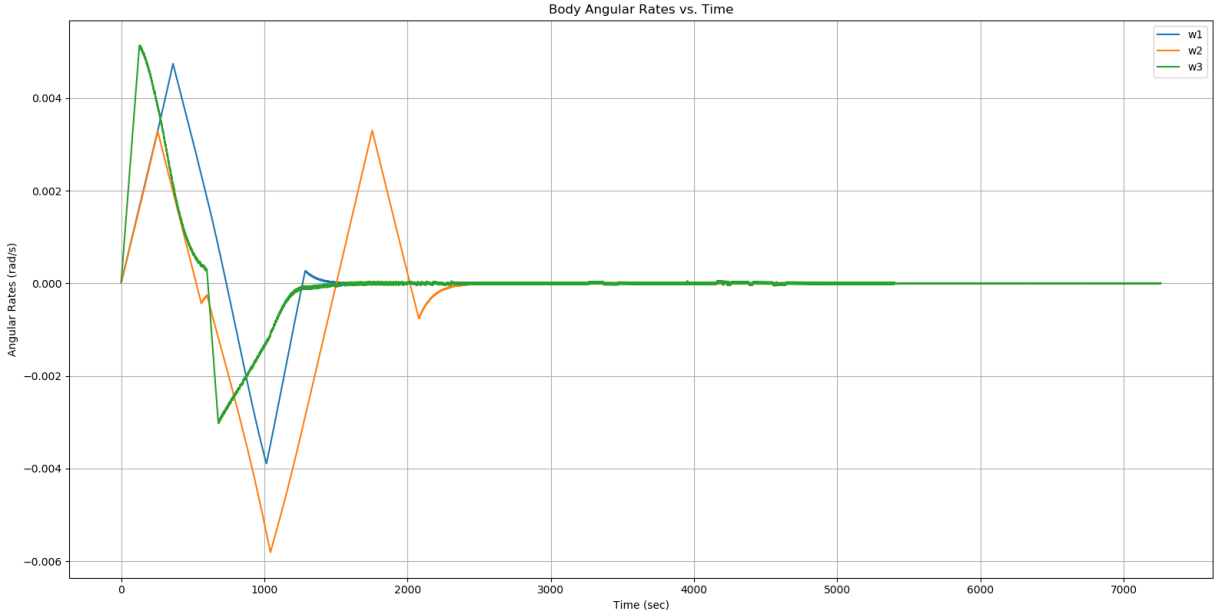
2.3 Control Algorithm Specifications

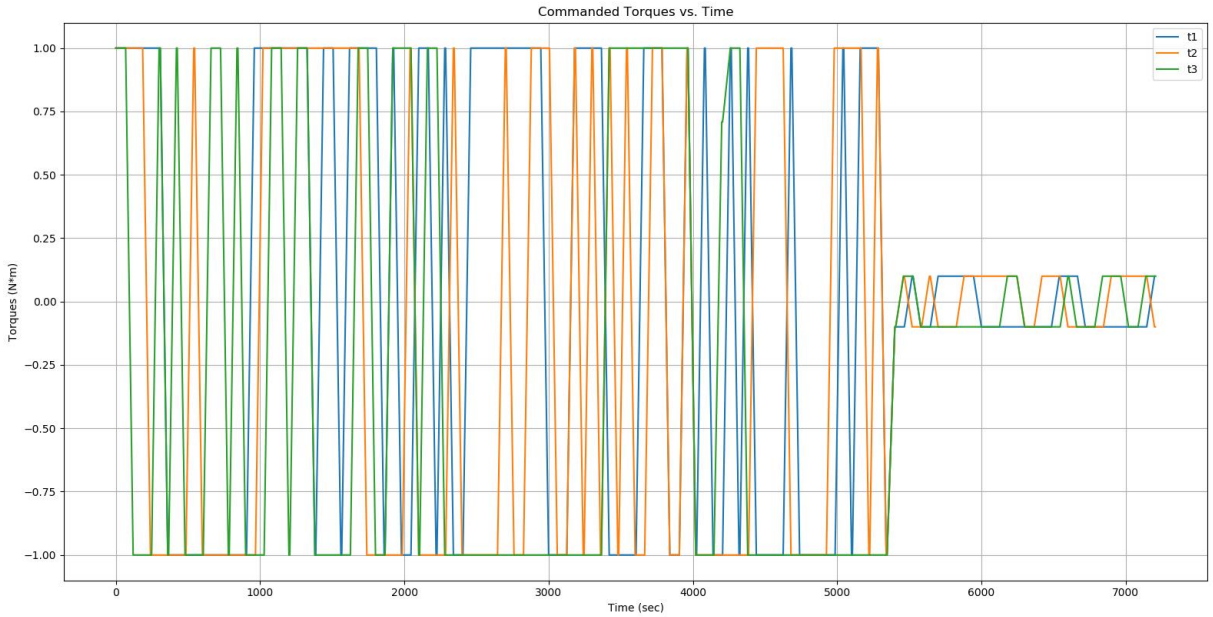
- Proportional Gains: 100000
- Derivative Gains: 10000000
- Max Torques were reduced from 1N*m to 0.1 N*m at 5400 seconds

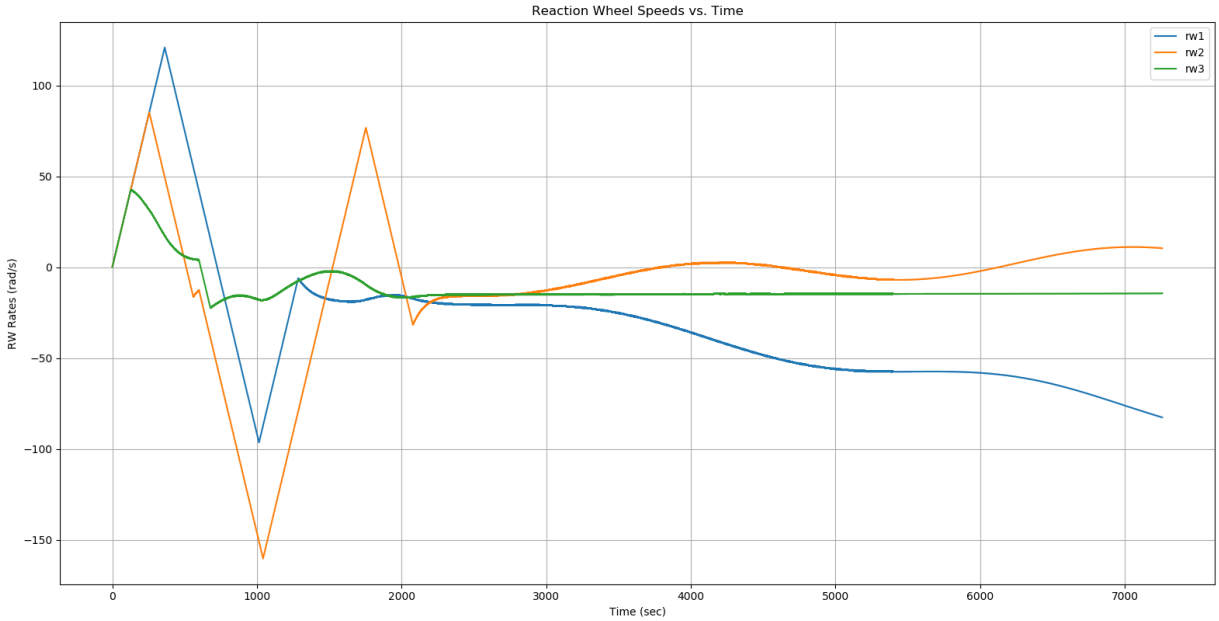
2.4 Results

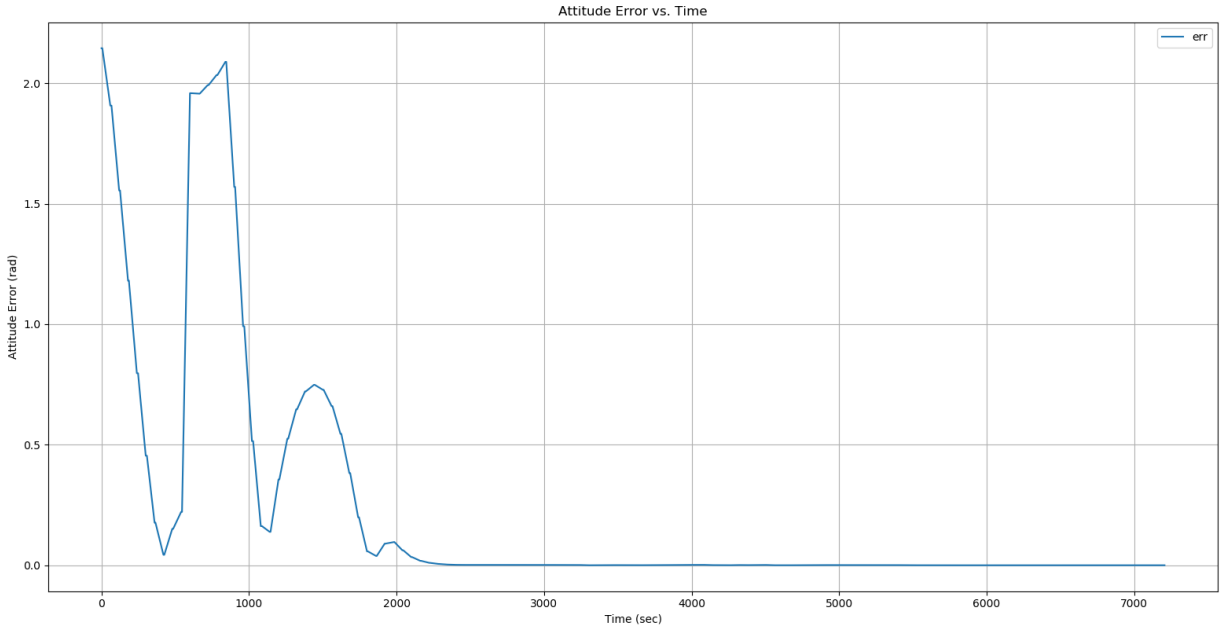


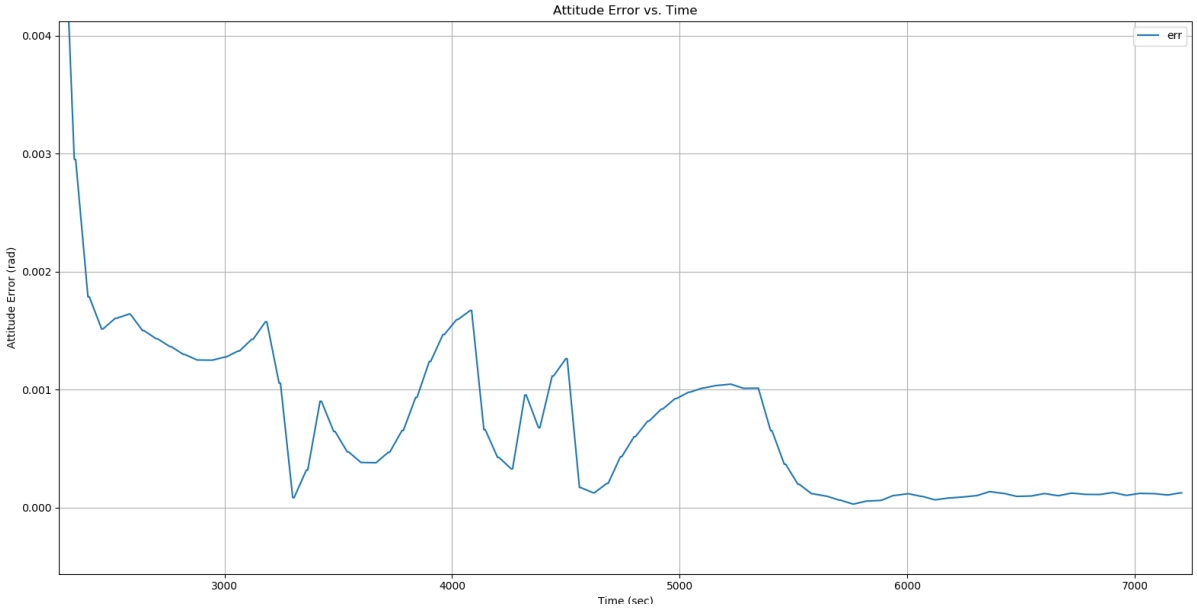

Body Quaternions vs. Time
Quaternions
Time (sec)
q1
q2
q3
q4

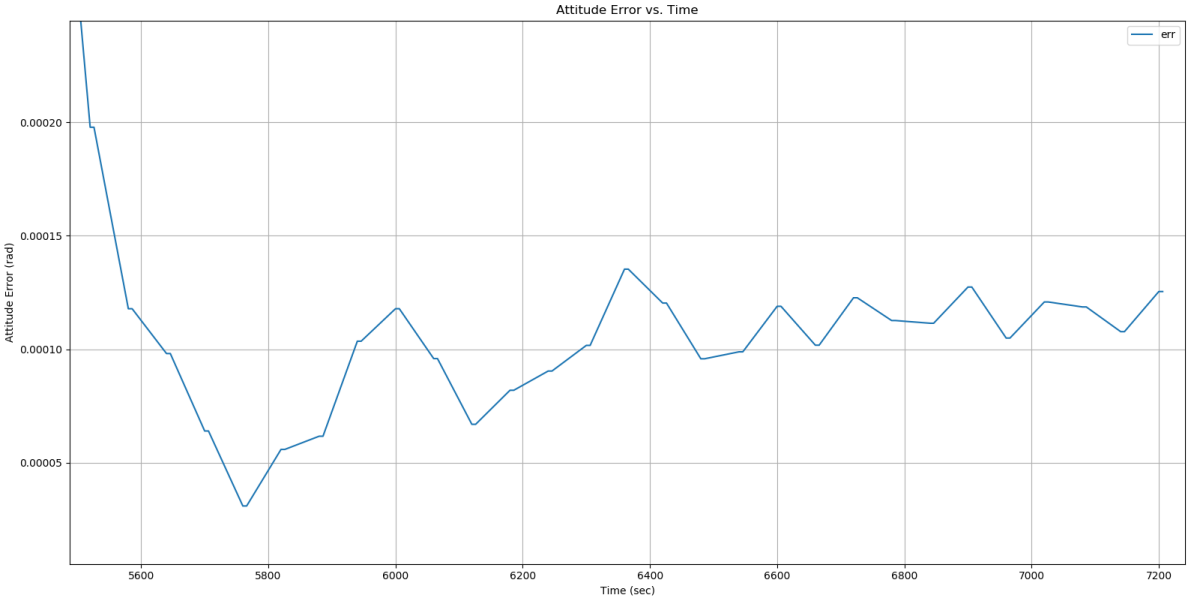

Body Angular Rates vs. Time
Angular Rates (rad/s)
Time (sec)
w1
w2
w3


Commanded Torques vs. Time
t1
t2
t3
Torques (N*m)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1000
2000
3000
4000
5000
6000
7000
Time (sec)


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
100
50
0
−50
−100
−150
0
1000
2000
3000
4000
5000
6000
7000
Time (sec)


Attitude Error vs. Time
err
Attitude Error (rad)
Time (sec)
0.0
0.5
1.0
1.5
2.0
0
1000
2000
3000
4000
5000
6000
7000


Attitude Error vs. Time
err
Attitude Error (rad)
Time (sec)
0.000
0.001
0.002
0.003
0.004
3000
4000
5000
6000
7000



2.5 Discussion

The system was given a maneuver away from alignment with the ECI frame and then back to the ECI frame in order to induce an error, after this maneuver, the accuracy of the pointing was observed. Although the system is unable to achieve the extremely stringent pointing requirements of the original assignment, the ADCS is still capable of extremely well accurate pointing (3e-5 rad). It is worth noting that the accuracy of the system greatly improves after 5400 seconds, when the algorithm reduces its max torques to 0.11N*m.

3 Reaction Wheel Momentum Storage

3.1 Description

For this simulation the spacecraft was commanded to hold an attitude aligned with the inertial frame (q = [0, 0, 0, 1]) in the presence of disturbance torques. It had to be shown within this simulation that the reaction wheels are capable of compensating for the disturbance torques imposed on the spacecraft without the need for frequent reaction wheel desaturations. No more than one desaturation per wheel per day was allowed for this spacecraft.

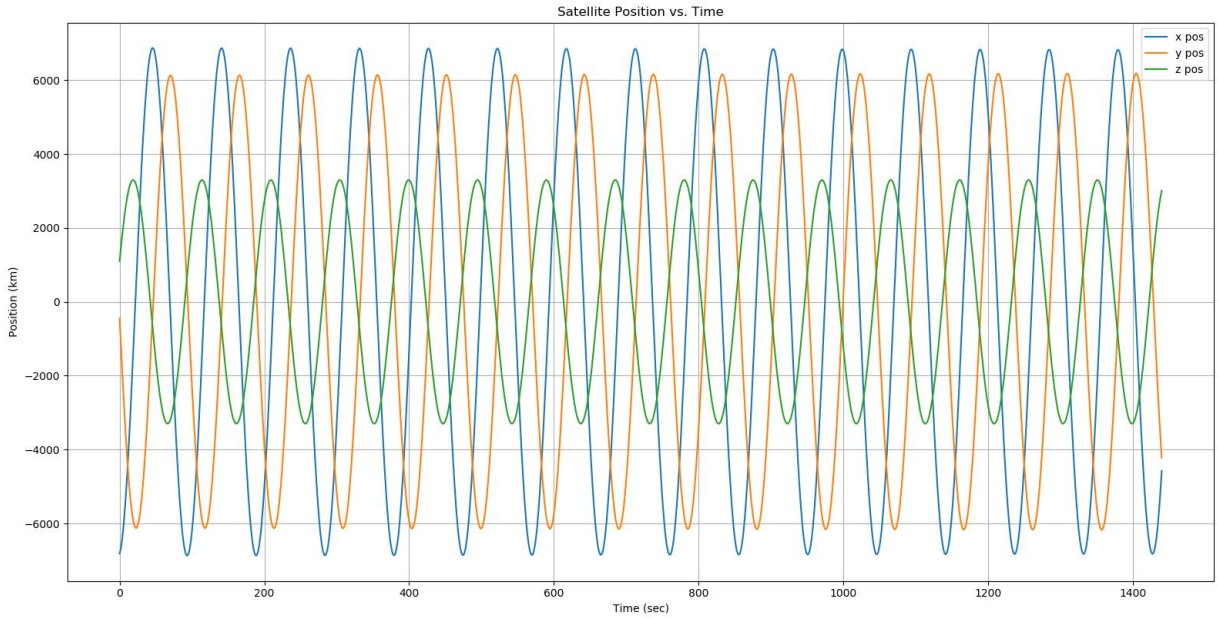
3.2 Modified Inputs/Requirements

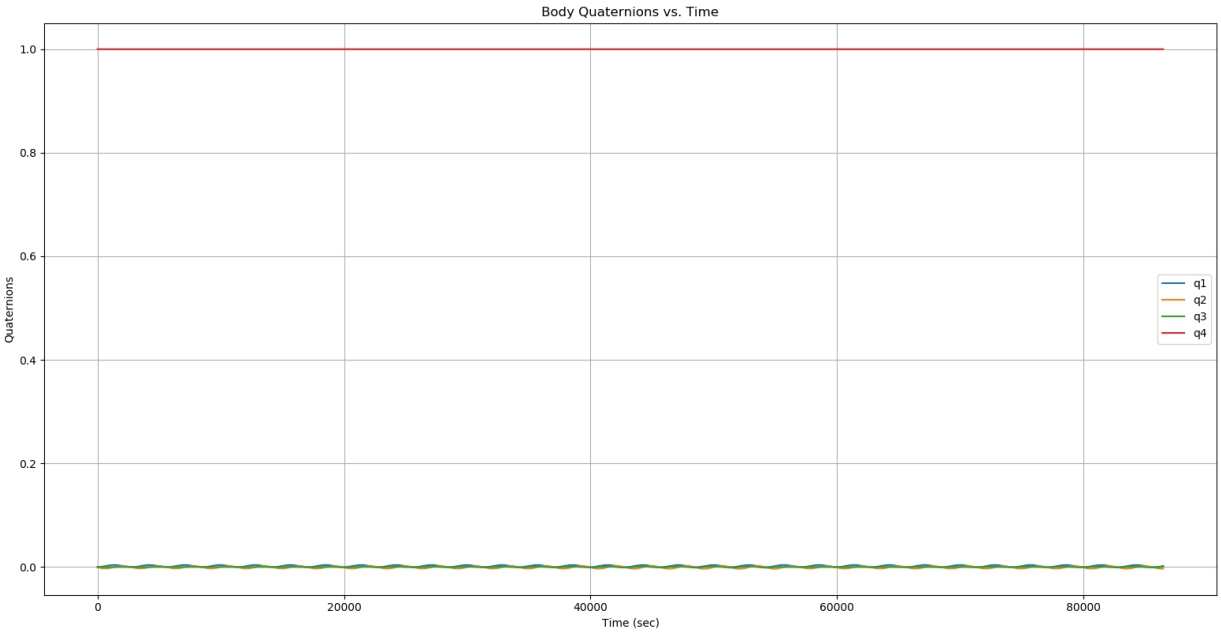
- The reaction wheel moments of inertia were modified to 3 kg*m^2
- The max wheel speed was configured to 700 rad/s (approximately equivalent to ISS CMG angular rates)

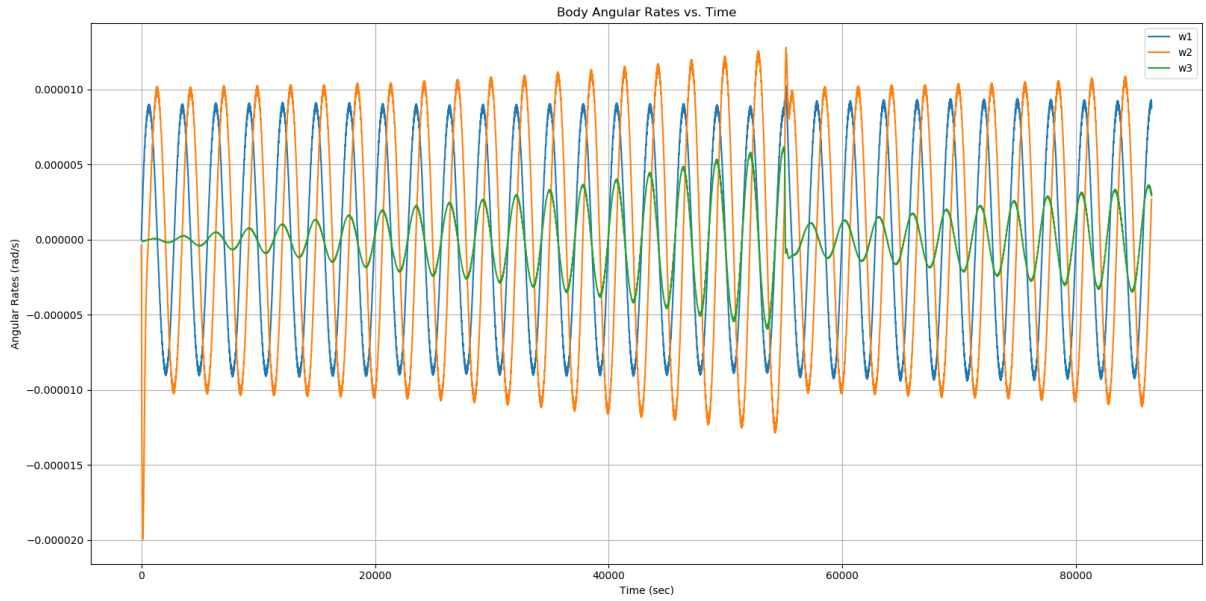
3.3 Control Algorithm Specifications

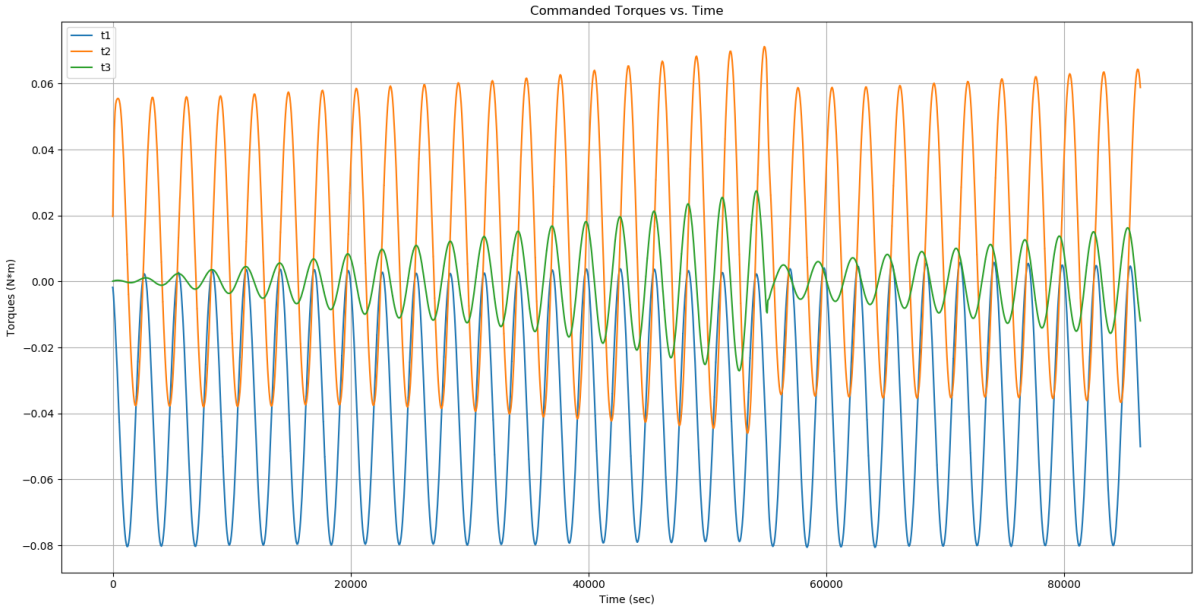
- Proportional Gains: 10
- Derivative Gains: 1000
- Thruster Torques were configured to 35 N*m

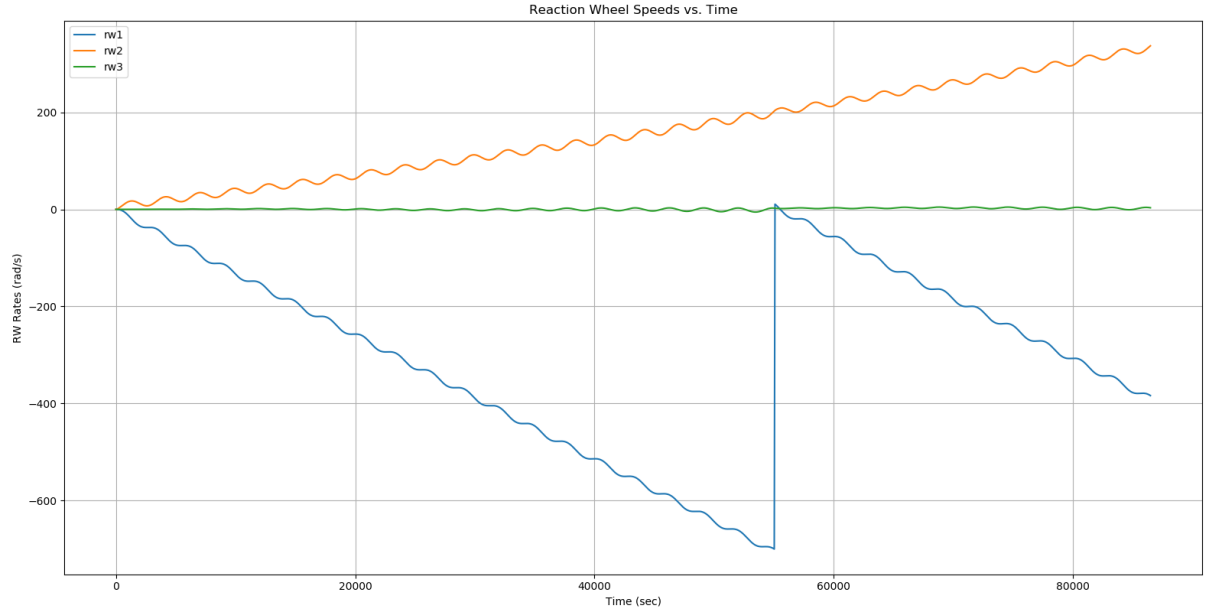
3.4 Results

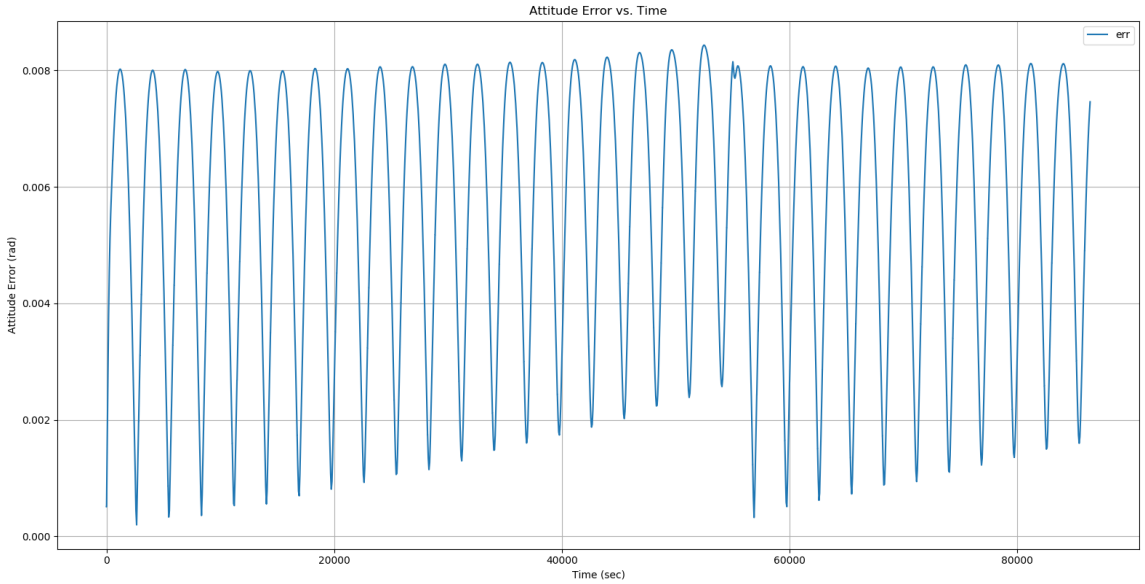



Body Quaternions vs. Time
Quaternions
Time (sec)
q1
q2
q3
q4
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000


Body Angular Rates vs. Time
Angular Rates (rad/s)
Time (sec)
w1
w2
w3
0.000010
0.000005
0.000000
−0.000005
−0.000010
−0.000015
−0.000020
0
20000
40000
60000
80000


Commanded Torques vs. Time
t1
t2
t3
Torques (N*m)
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
0
20000
40000
60000
80000
Time (sec)


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
200
0
−200
−400
−600
0
20000
40000
60000
80000
Time (sec)



3.5 Discussion

The simulation was run for the entire day's worth of data. From the graphs, it is interesting to note that the disturbance torques can be observed in the commanded torques and the reaction wheel speeds. Because this torques occur cyclically, the commanded torques and reaction wheel speeds have a sinusoidal trend. In addition, constant disturbance torques cause the reaction wheel speeds to steadily increase. With the reaction wheel moments of inertia configured to 3 kg*m^2 and the max spin rate configured to 700 rad/s, the system only needs to desaturate the wheels once per day, per wheel.

4 Reaction Wheel Desaturation

4.1 Description

The purpose of this simulation is to demonstrate the thrusters' capability of desaturating the reaction wheels within 60 seconds. For this simulation, the reaction wheels were given an initial spin rate of 701 rad/s. At this angular velocity, the thrusters are triggered to begin a desaturation of the reaction wheels.

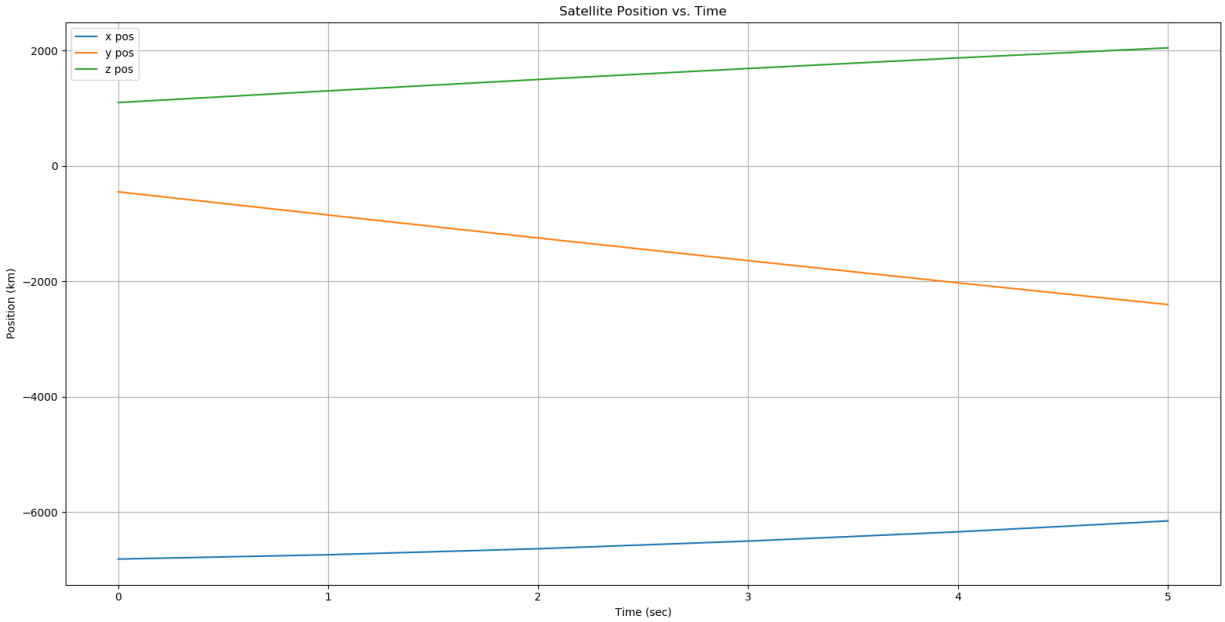
4.2 Modified Inputs/Requirements

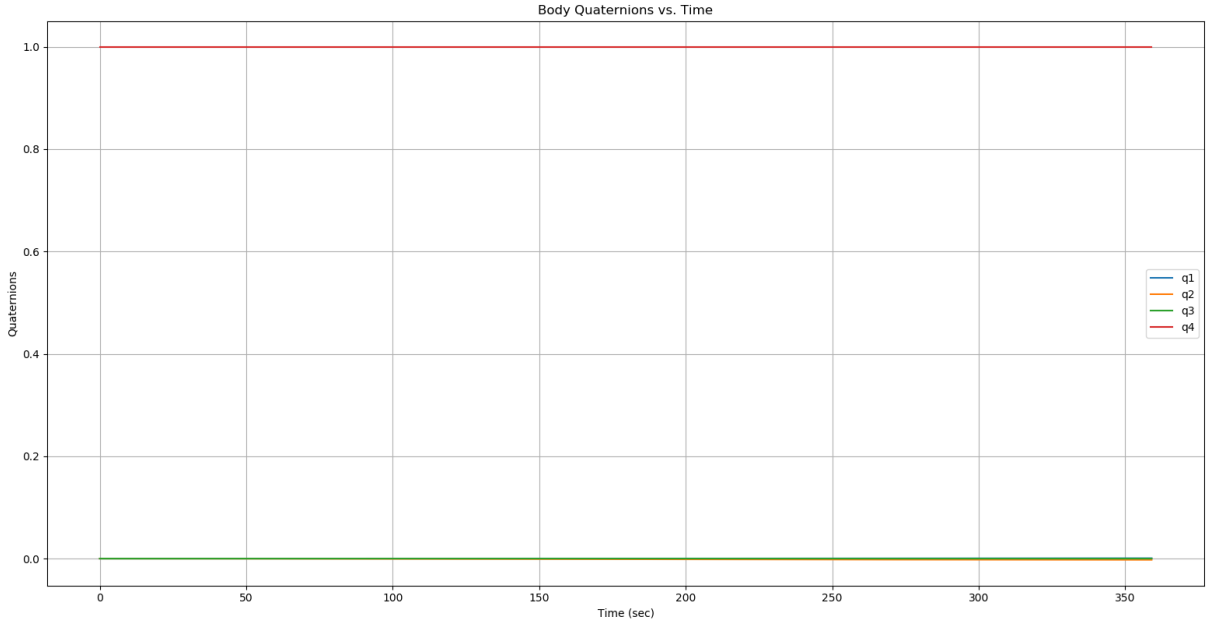
- The reaction wheel moments of inertia were modified to 3 kg*m^2
- The max wheel speed was configured to 700 rad/s (approximately equivalent to ISS CMG angular rates)

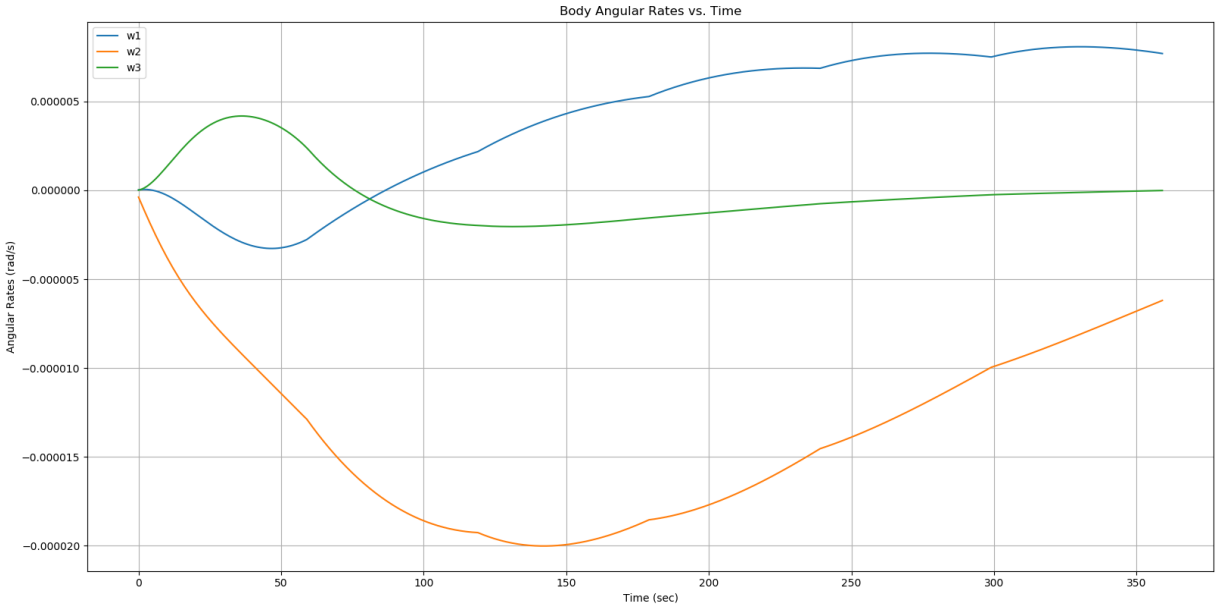
4.3 Control Algorithm Specifications

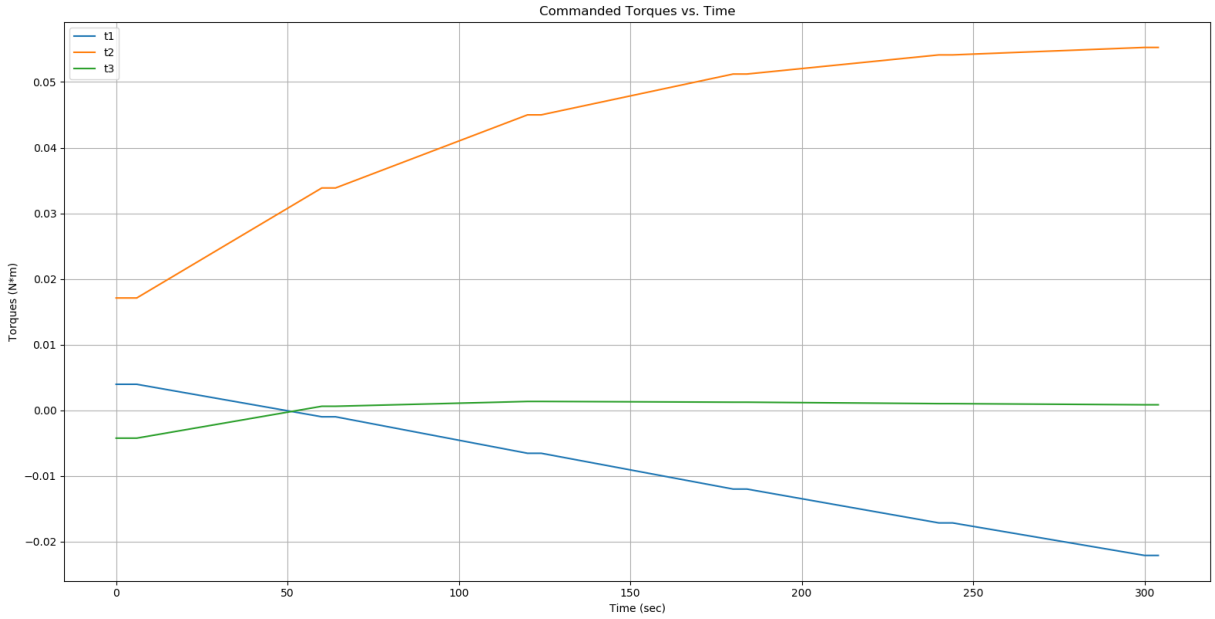
- Proportional Gains: 10
- Derivative Gains: 1000
- Thruster Torques were configured to 35 N*m

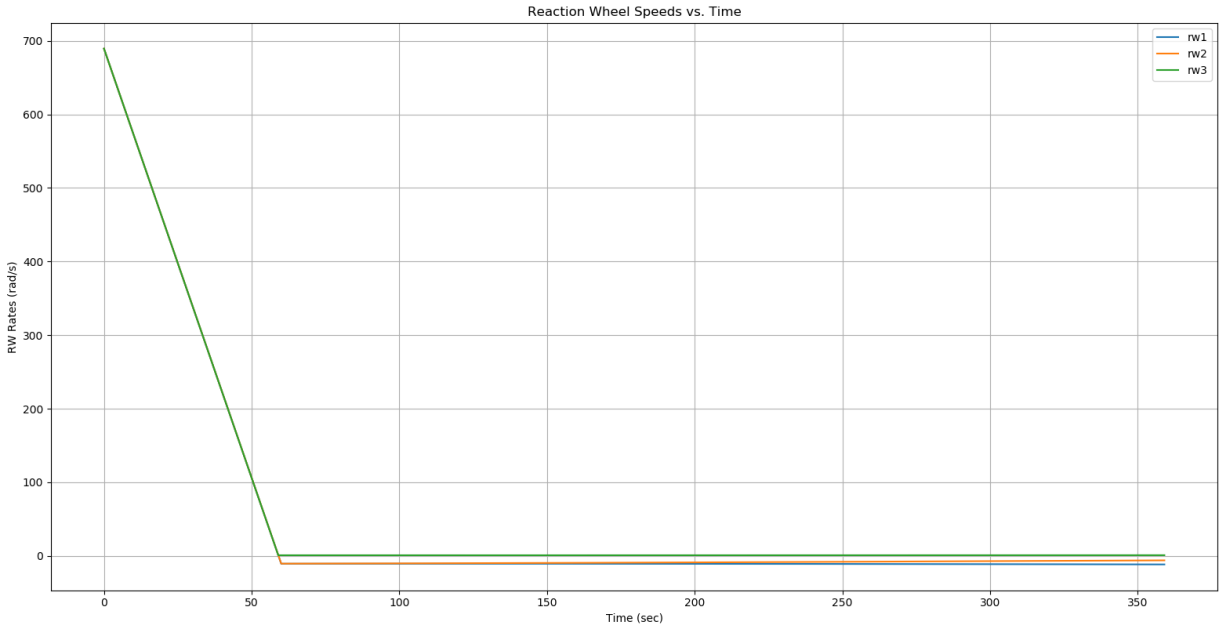
4.4 Results

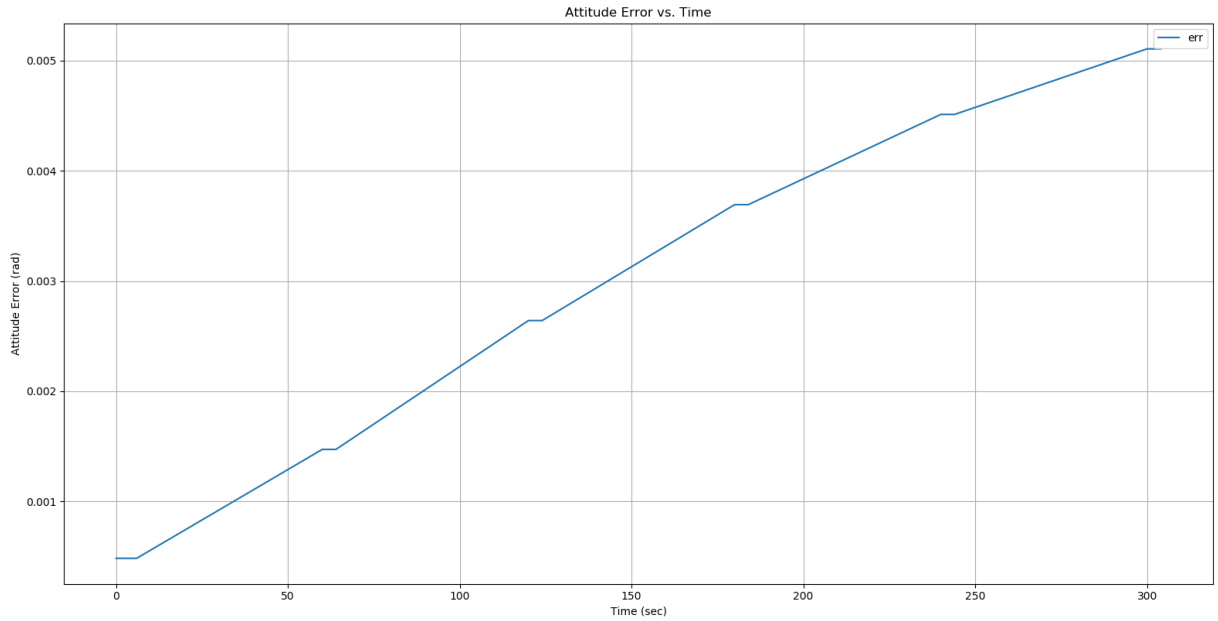



Body Quaternions vs. Time
Quaternions
Time (sec)
q1
q2
q3
q4
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
350


Body Angular Rates vs. Time
Angular Rates (rad/s)
Time (sec)
w1
w2
w3
0.000005
0.000000
−0.000005
−0.000010
−0.000015
−0.000020
0
50
100
150
200
250
300
350


Commanded Torques vs. Time
t1
t2
t3
Torques (N*m)
Time (sec)
0.05
0.04
0.03
0.02
0.01
0.00
−0.01
−0.02
0
50
100
150
200
250
300


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
Time (sec)
700
600
500
400
300
200
100
0
0
50
100
150
200
250
300
350



4.5 Discussion

As can be seen in the reaction wheel angular rate graph, the thrusters can desaturate the reaction wheels in the presence of disturbance torques to within 60 seconds. The required thruster torque for this capability of 35 N*m, which is still reasonable for this size of spacecraft. In addition, the spacecraft is still capable of maintaining attitude control during the desaturation.

5 Detumble

5.1 Description

For this pointing mode the spacecraft was given an initial angular rate of 0.1 rad/s about each axis of rotation and was commanded to detumble and point to align with the inertial reference frame (q = [0, 0, 0, 1]). Requirements specify that the spacecraft must detumble within an hour and hold an attitude error of 0.1 deg with the inertial frame.

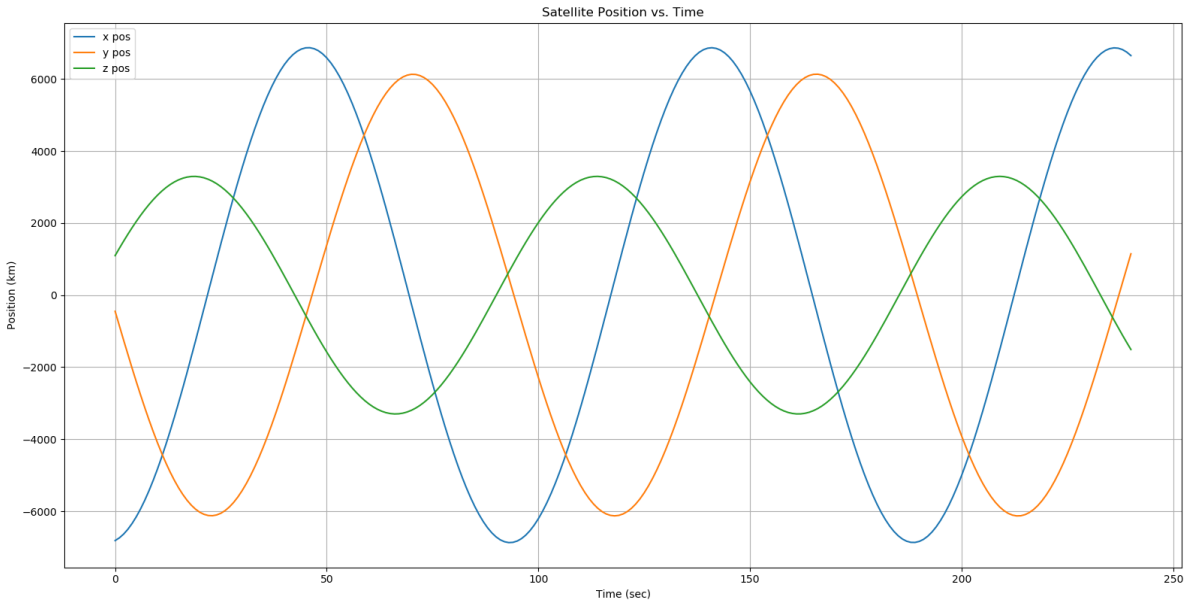
5.2 Modified Inputs/Requirements

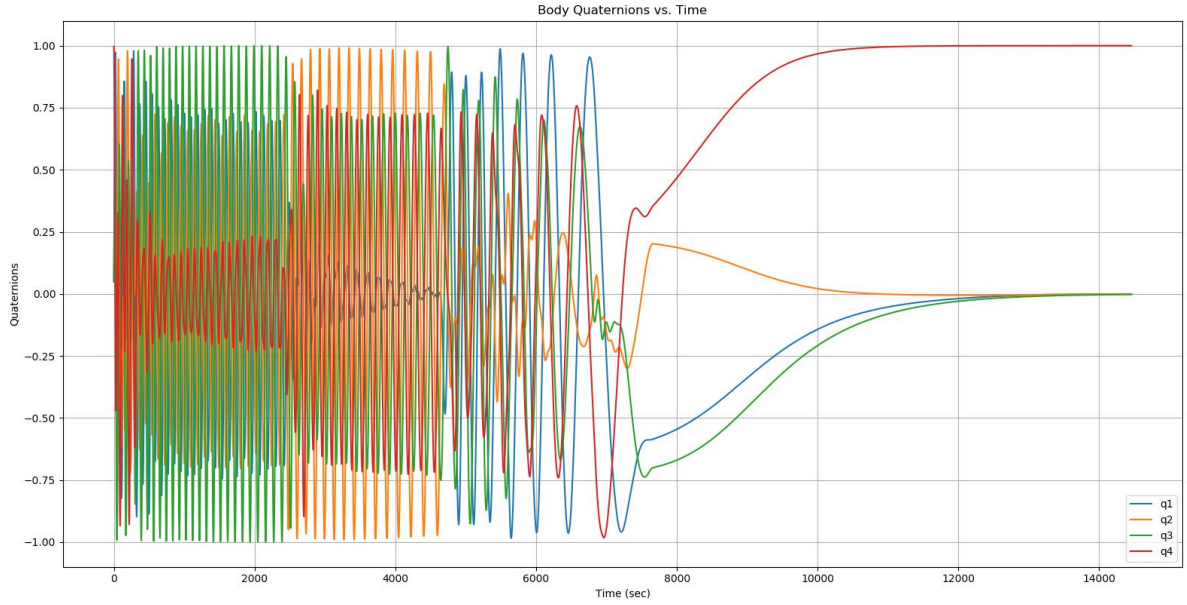
- No requirements were modified
- The reaction wheel moments of inertia were modified to 3 kg*m^2
- The max wheel speed was configured to 700 rad/s (approximately equivalent to ISS CMG angular rates)

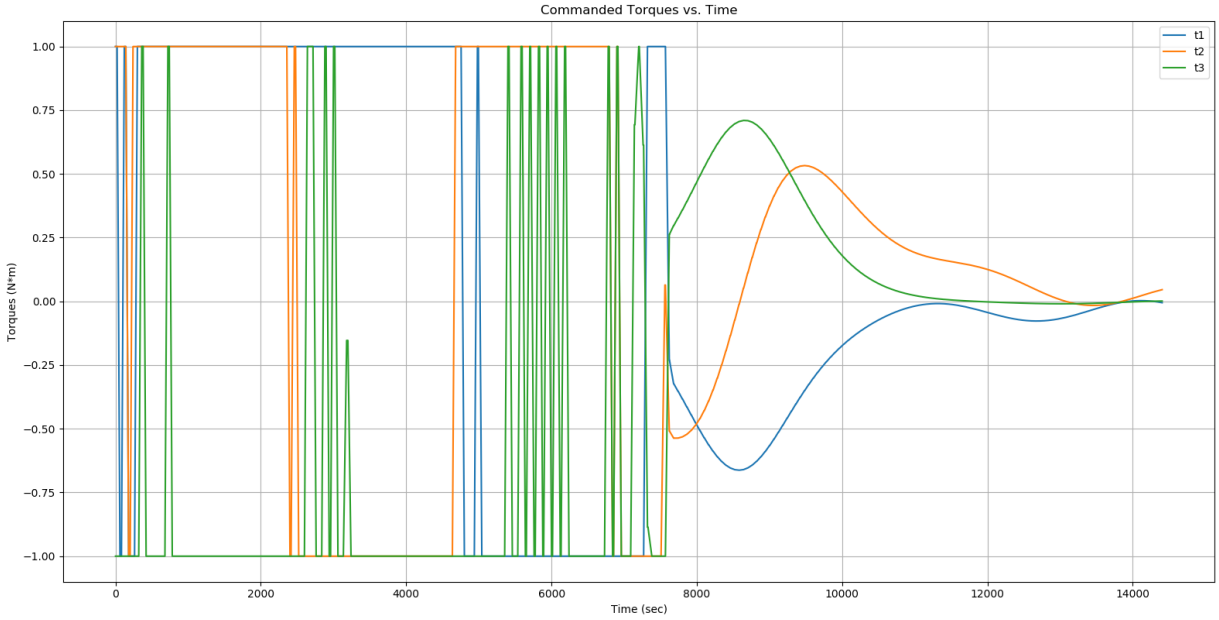
5.3 Control Algorithm Specifications

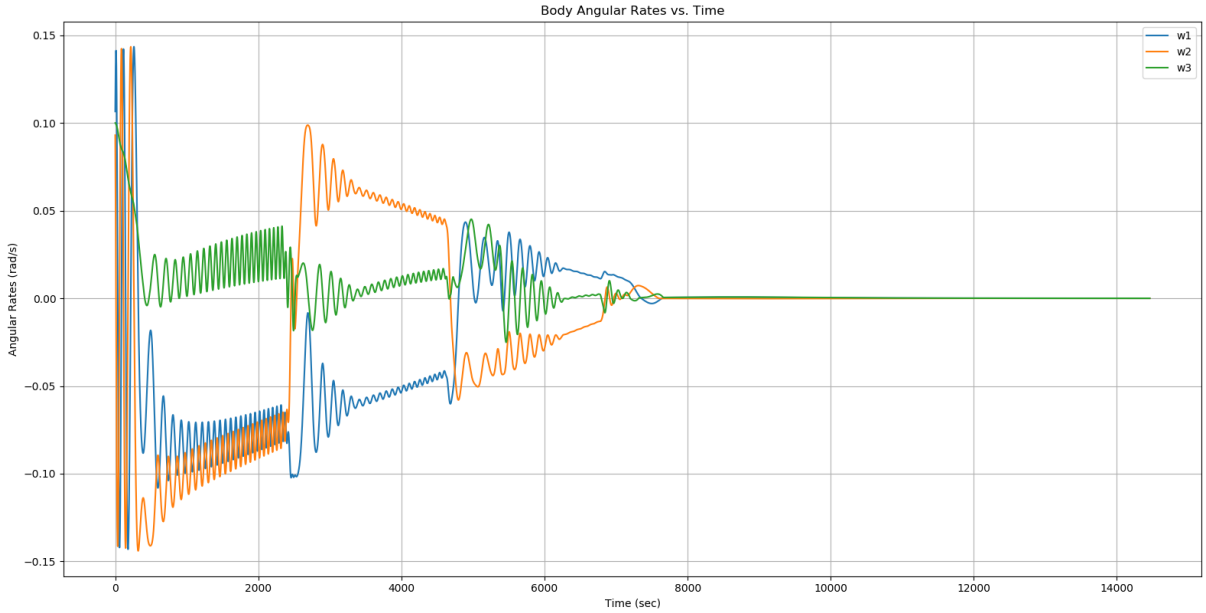
- Proportional Gains: 100
- Derivative Gains: 10000
- Thruster Torques were configured to 35 N*m

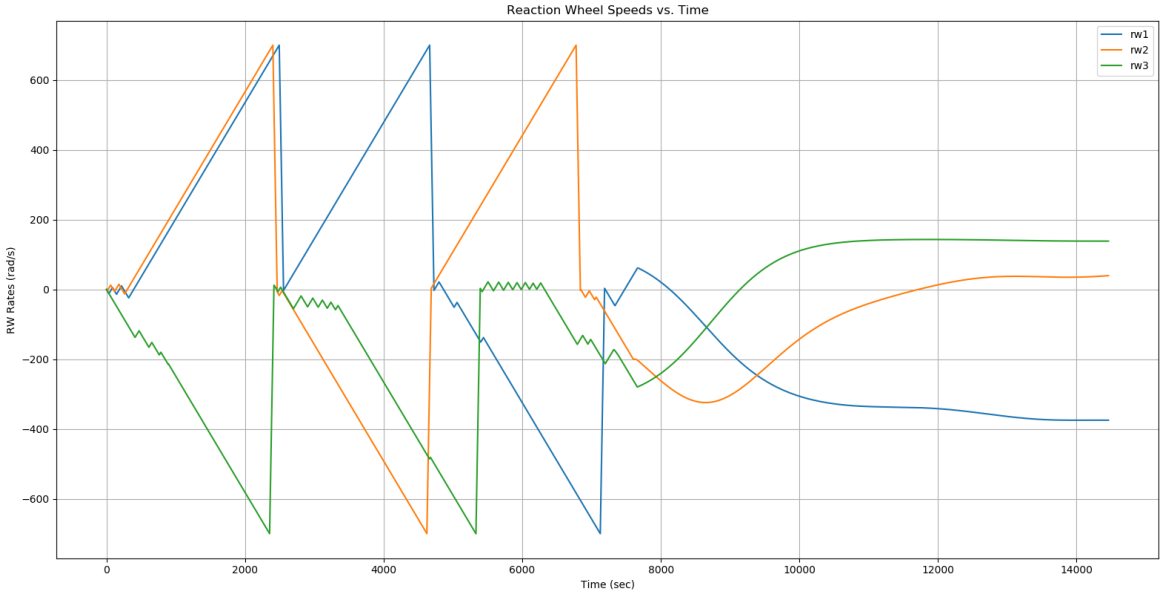
5.4 Results

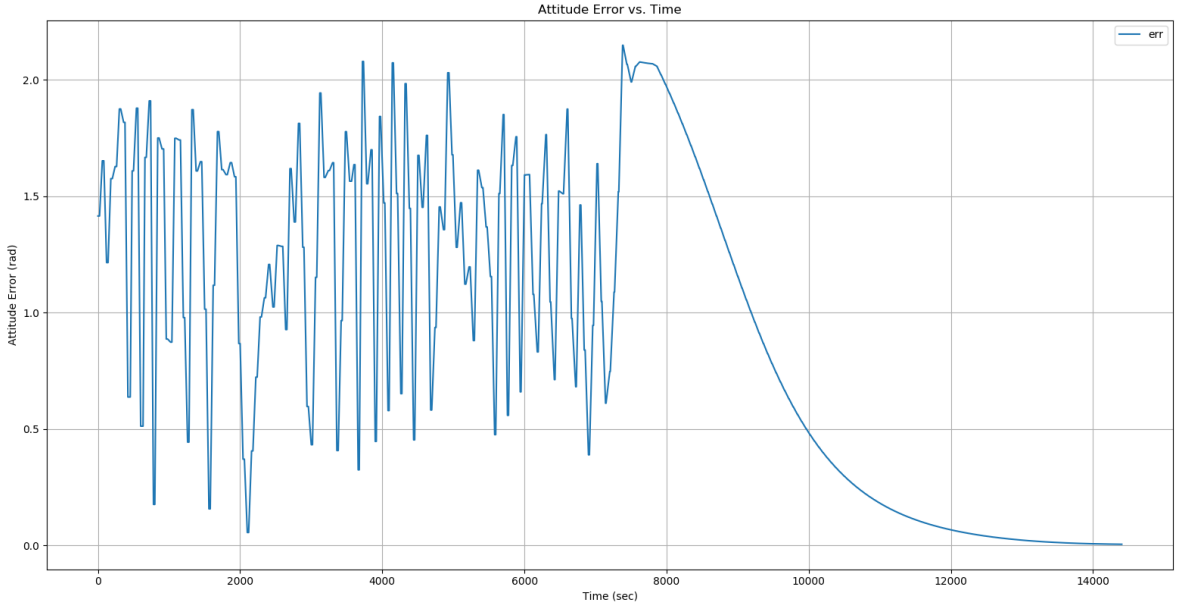



Body Quaternions vs. Time
Quaternions
Time (sec)
q1
q2
q3
q4


Commanded Torques vs. Time
Torques (N*m)
Time (sec)
t1
t2
t3


Body Angular Rates vs. Time
w1
w2
w3
Angular Rates (rad/s)
Time (sec)
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
2000
4000
6000
8000
10000
12000
14000


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
Time (sec)
600
400
200
0
−200
−400
−600
0
2000
4000
6000
8000
10000
12000
14000



5.5 Discussion

As can be seen from the graphs above, it is clear that the spacecraft is capable of detumbling within the 12-hour time imposed by the system requirements. After 6 hours, the spacecraft has successfully detumbled, and has maneuvered to the desired attitude. It is important to note that since the reaction wheels are much less massive than the spacecraft body, that it is impossible for the reaction wheels to completely detumble the spacecraft without the need for desaturating the reaction wheels. In this simulation, it required eight reaction wheel desaturations to dump enough momentum to detumble the satellite.

6 Slew Maneuver

6.1 Description

For this simulation, the satellite was given a 45-degree maneuver in the cross-trek direction. Requirements specify that the spacecraft must be capable of performing this maneuver in less than 30 minutes. In addition, it is required that the satellite maintain an accuracy of less than 0.1 degrees.

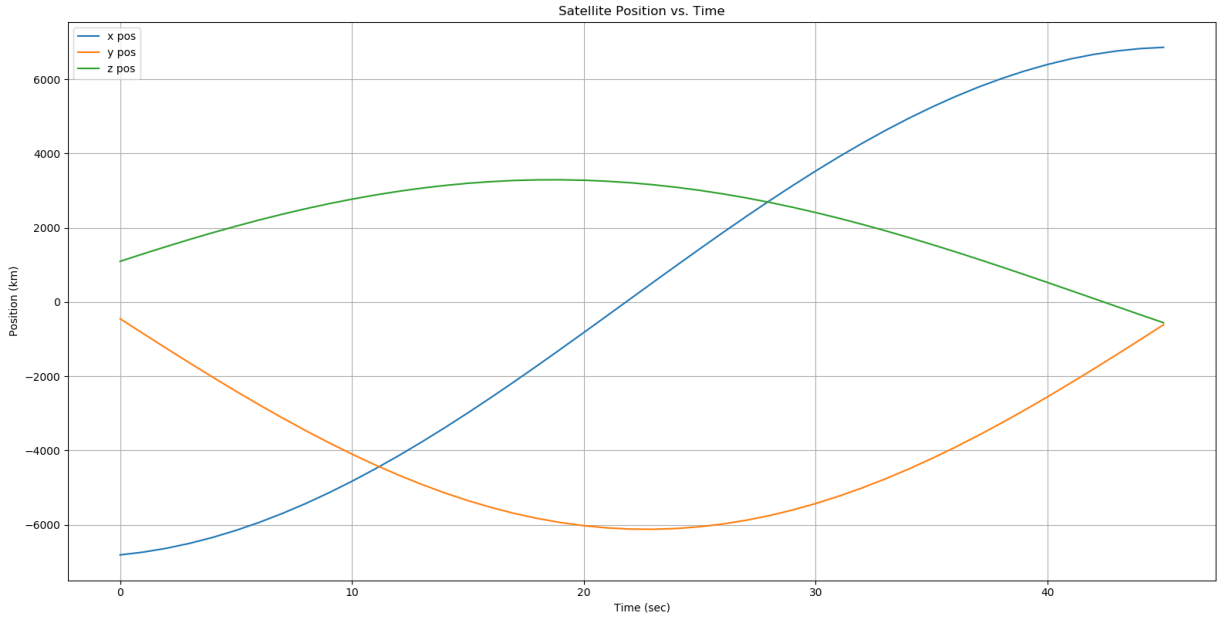
6.2 Modified Inputs/Requirements

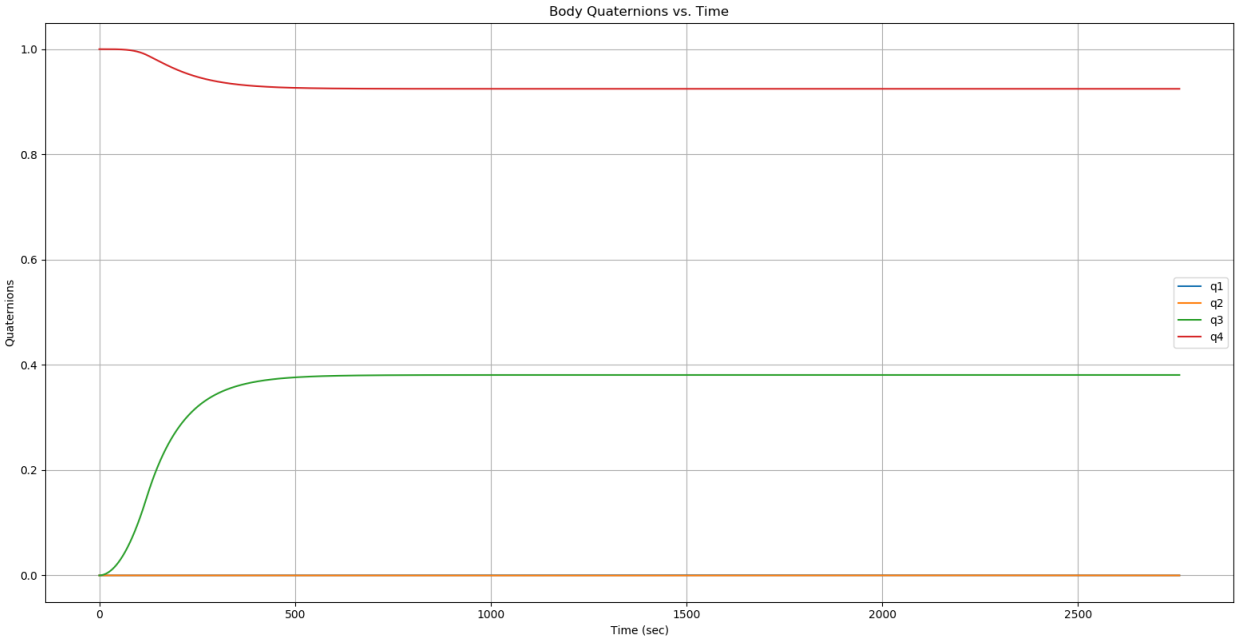
- No requirements were modified
- The reaction wheel moments of inertia were modified to 3 kg*m^2
- The max wheel speed was configured to 700 rad/s (approximately equivalent to ISS CMG angular rates)

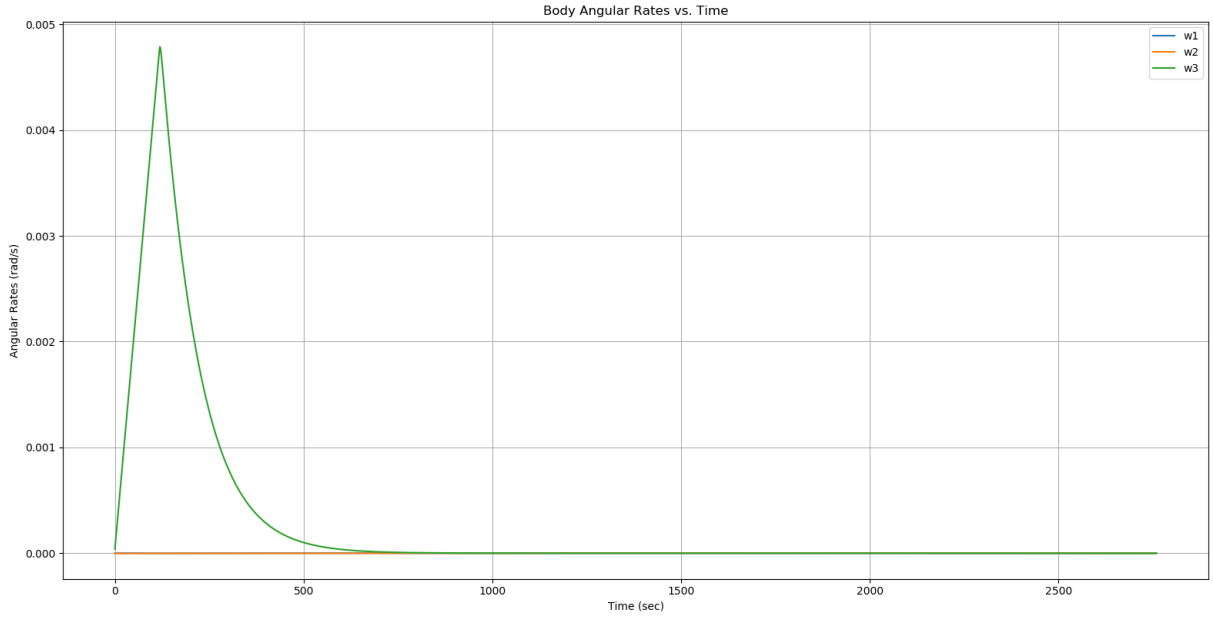
6.3 Control Algorithm Specifications

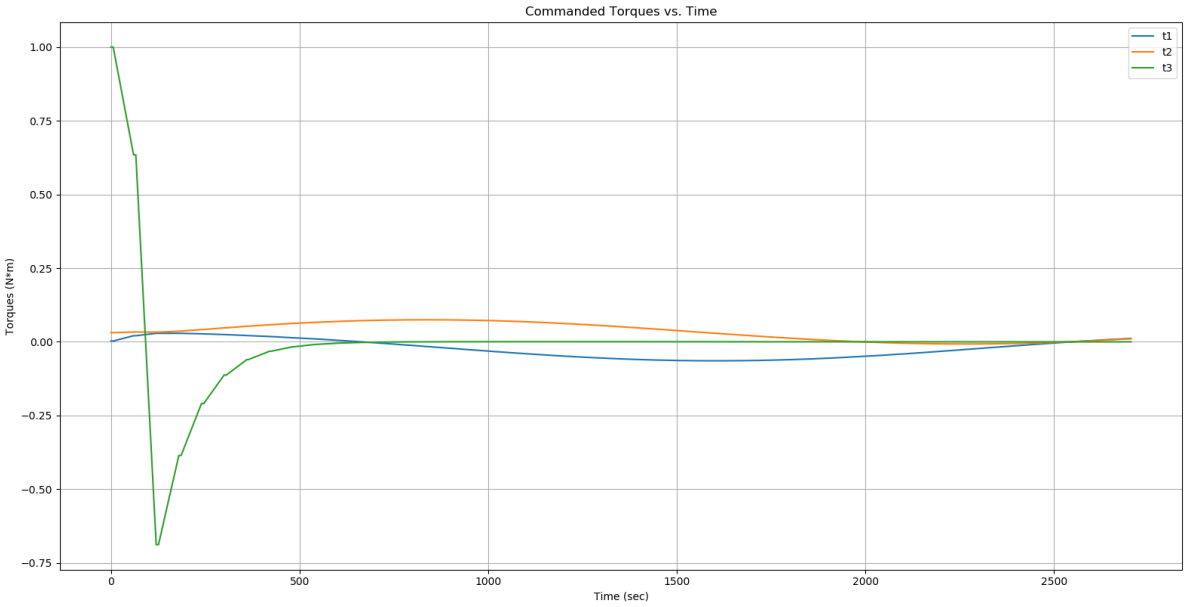
- Proportional Gains: 100
- Derivative Gains: 10000

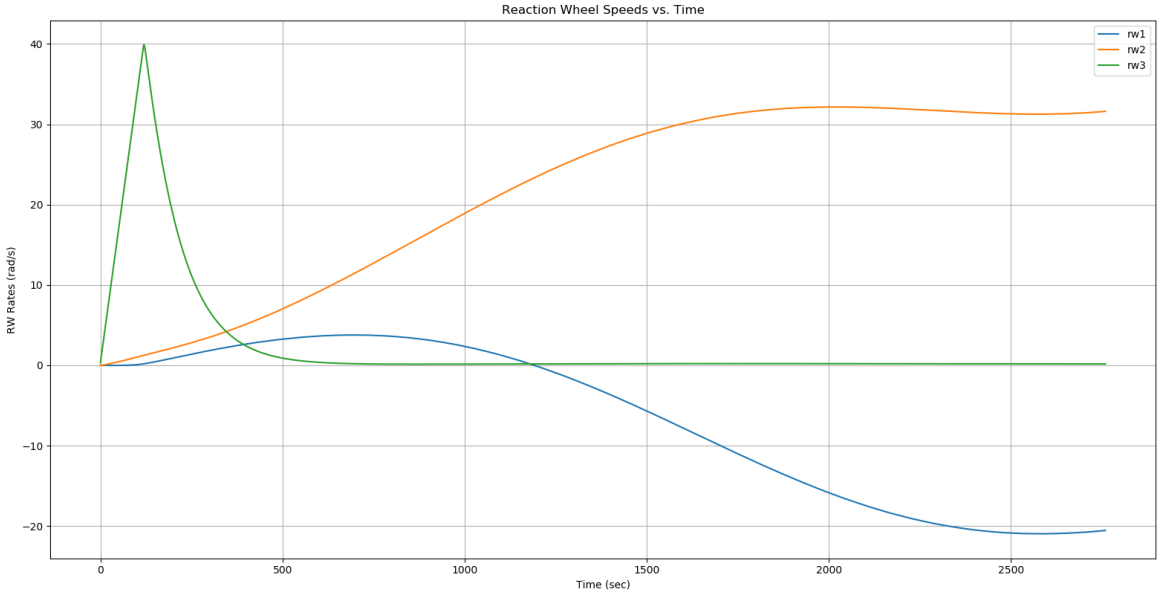
6.4 Results

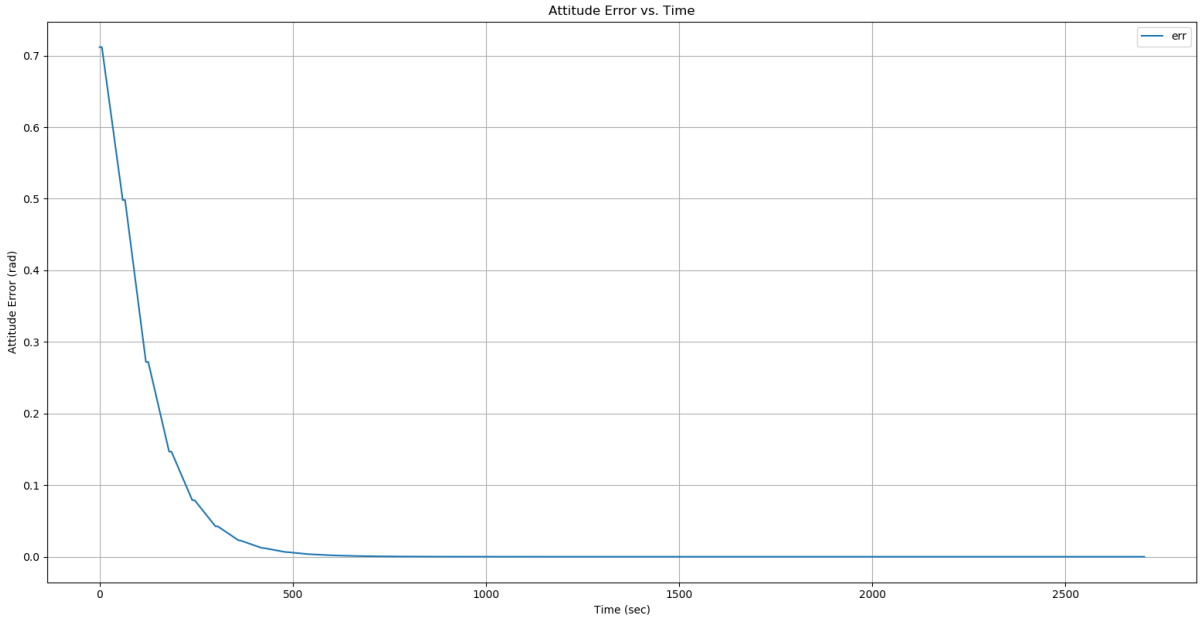



Body Quaternions vs. Time
Quaternions
Time (sec)
q1
q2
q3
q4
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
1500
2000
2500


Body Angular Rates vs. Time
Angular Rates (rad/s)
Time (sec)
w1
w2
w3
0.005
0.004
0.003
0.002
0.001
0.000
0
500
1000
1500
2000
2500


Commanded Torques vs. Time
t1
t2
t3
Torques (N*m)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
500
1000
1500
2000
2500
Time (sec)


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
40
30
20
10
0
−10
−20
0
500
1000
1500
2000
2500
Time (sec)



6.5 Discussion

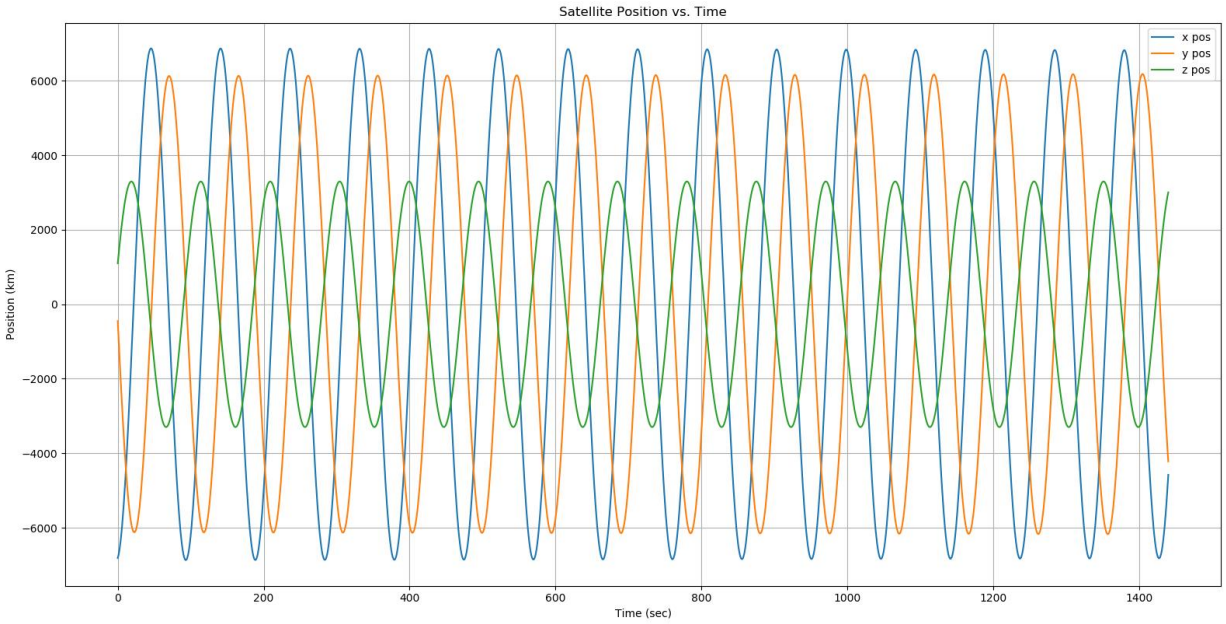
As is demonstrated by the graphs above, the spacecraft is capable of performing the maneuver to within a little of 10 minutes. In addition, the spacecraft is well able to hold an attitude with an error of less than 0.1 degrees. It is also worth noting that the reaction wheels of this spacecraft are capable of performing this maneuver without a large amount of momentum being required.

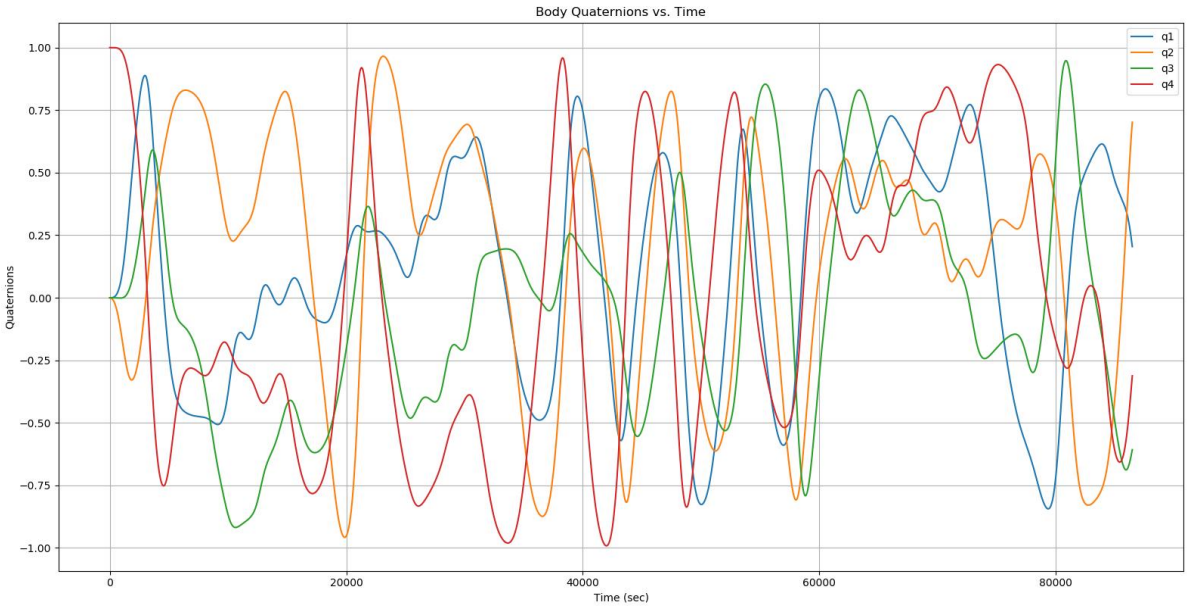
7 Random Tumble [0, 0, 0, 1]

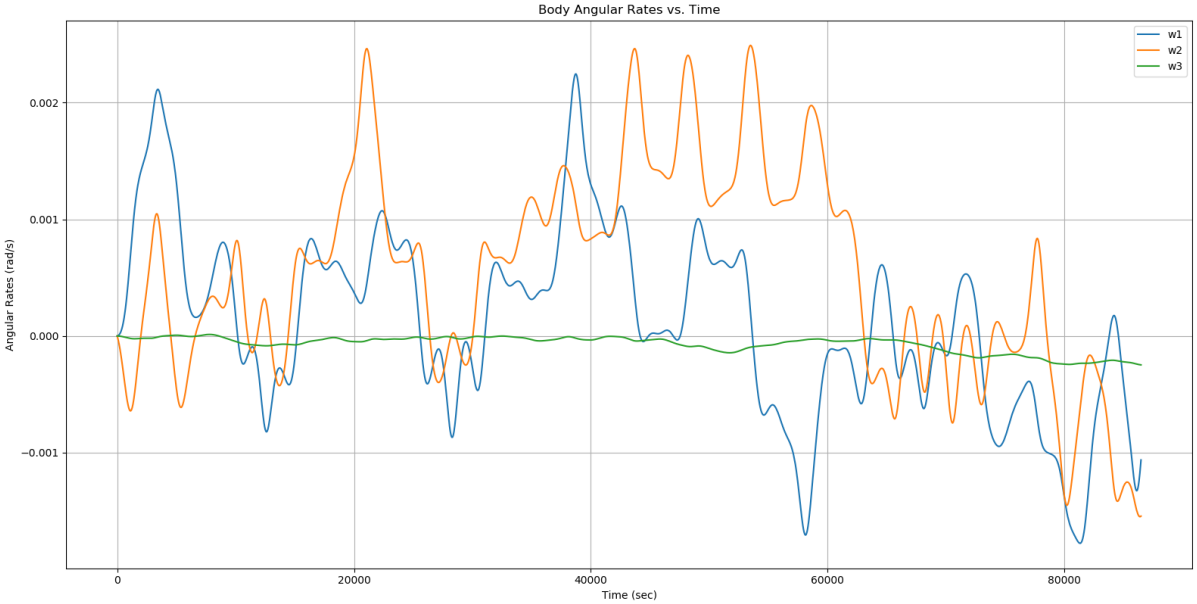
7.1 Description

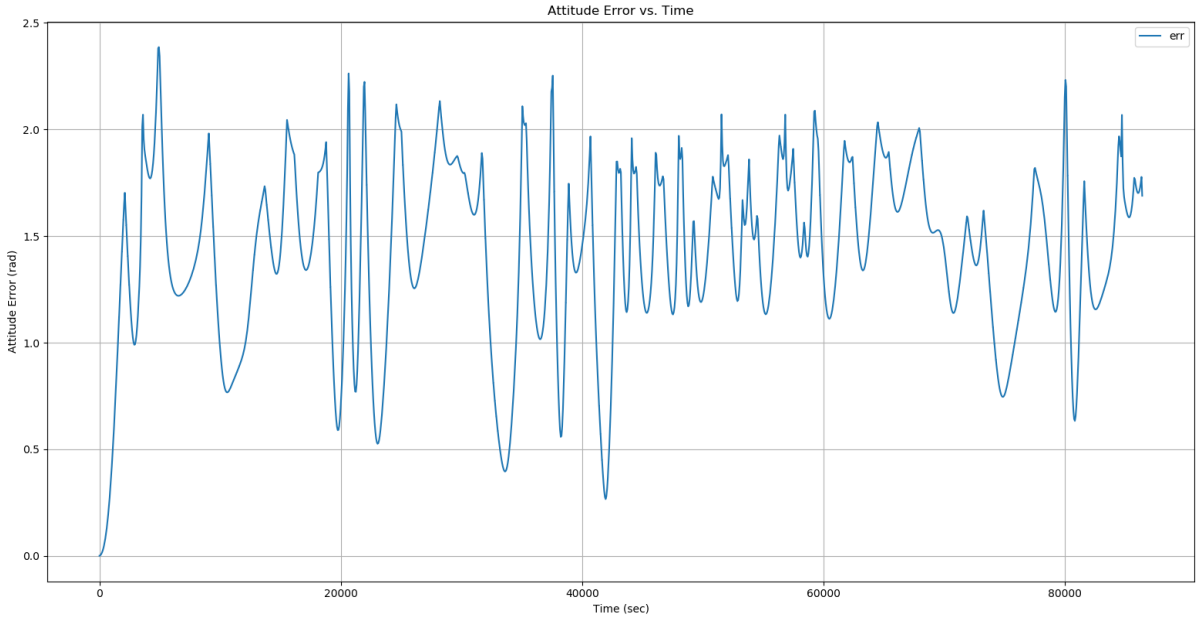
This simulation displays the attitude of the spacecraft without any control torques in place. The satellite is simply allowed to drift and tumble in the presence of disturbance torques. The initial attitude of the spacecraft was configured to q = [0, 0, 0, 1].

7.2 Results






Body Angular Rates vs. Time
w1
w2
w3
0.002
0.001
0.000
−0.001
Angular Rates (rad/s)
0
20000
40000
60000
80000
Time (sec)

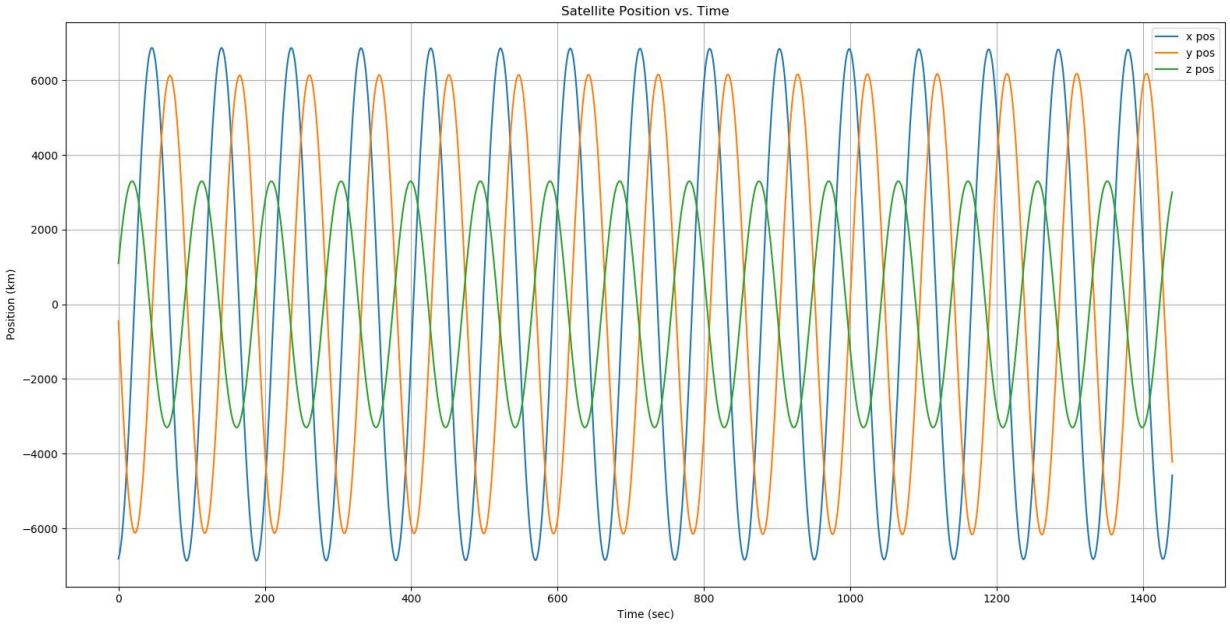

Attitude Error vs. Time
err
2.5
2.0
1.5
1.0
0.5
0.0
Attitude Error (rad)
0
20000
40000
60000
80000
Time (sec)

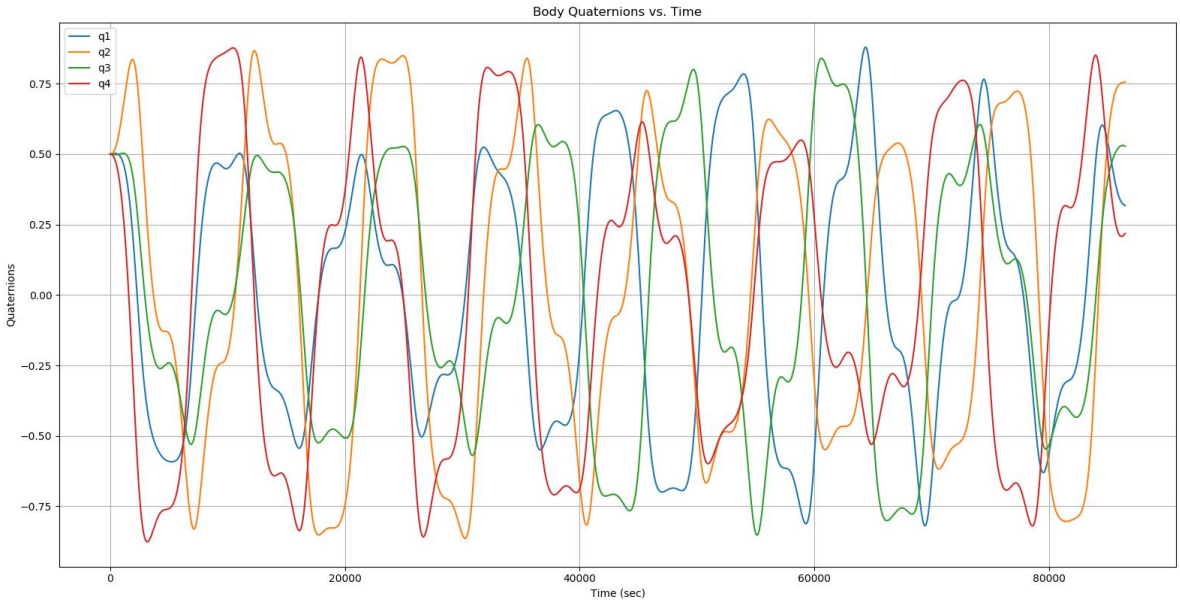
8 Random Tumble [0.5, 0.5, 0.5, 0.5]

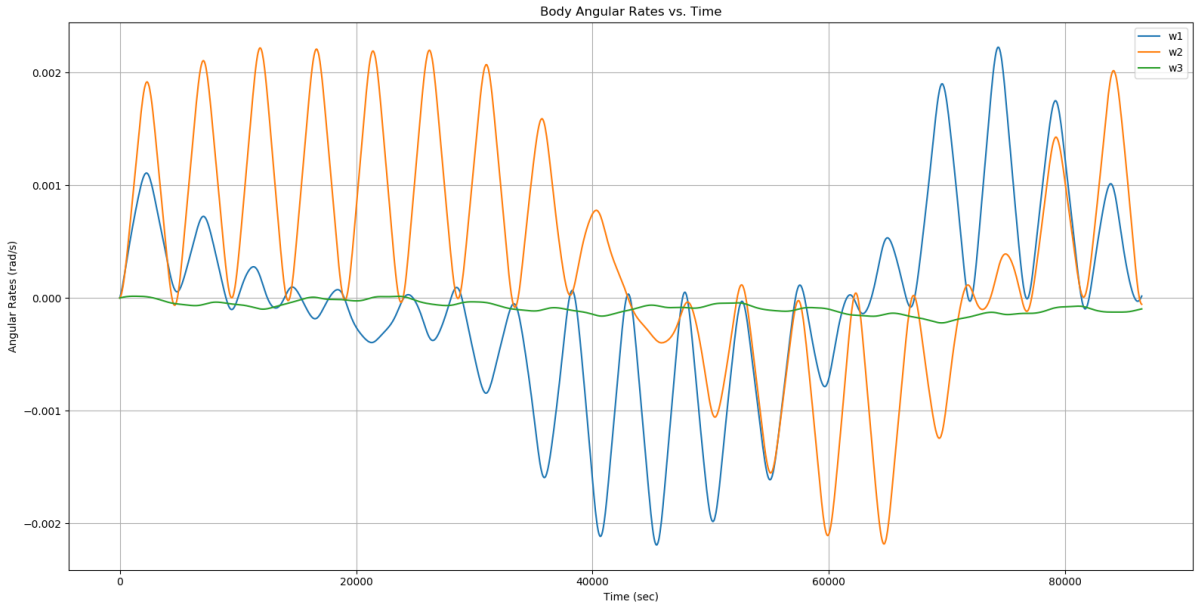
8.1 Description

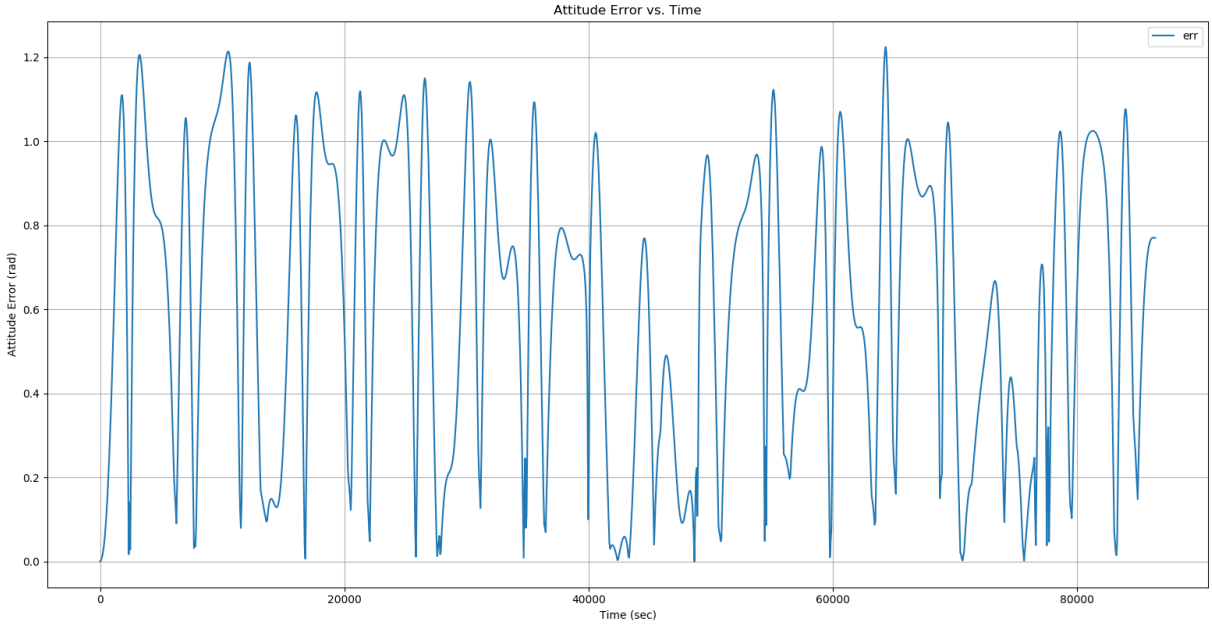
This simulation displays the attitude of the spacecraft without any control torques in place. The satellite is simply allowed to drift and tumble in the presence of disturbance torques. The initial attitude of the spacecraft was configured to q = [0.5, 0.5, 0.5, 0.5].

8.2 Results






Body Angular Rates vs. Time
w1
w2
w3
Angular Rates (rad/s)
0.002
0.001
0.000
−0.001
−0.002
0
20000
40000
60000
80000
Time (sec)

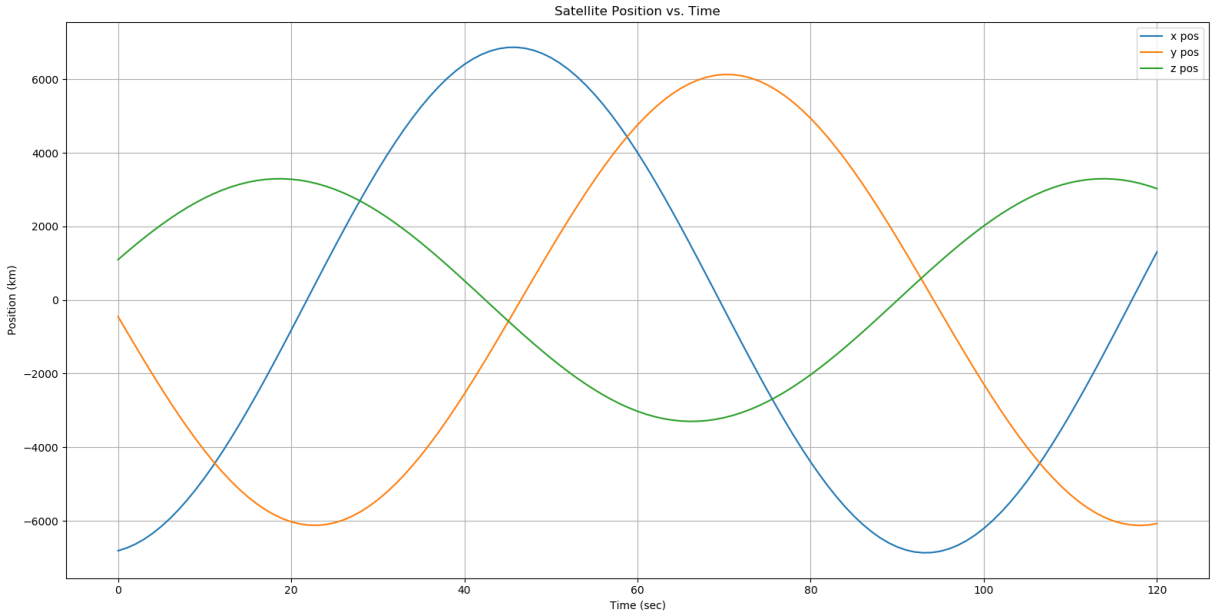

Attitude Error vs. Time
err
Attitude Error (rad)
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
Time (sec)

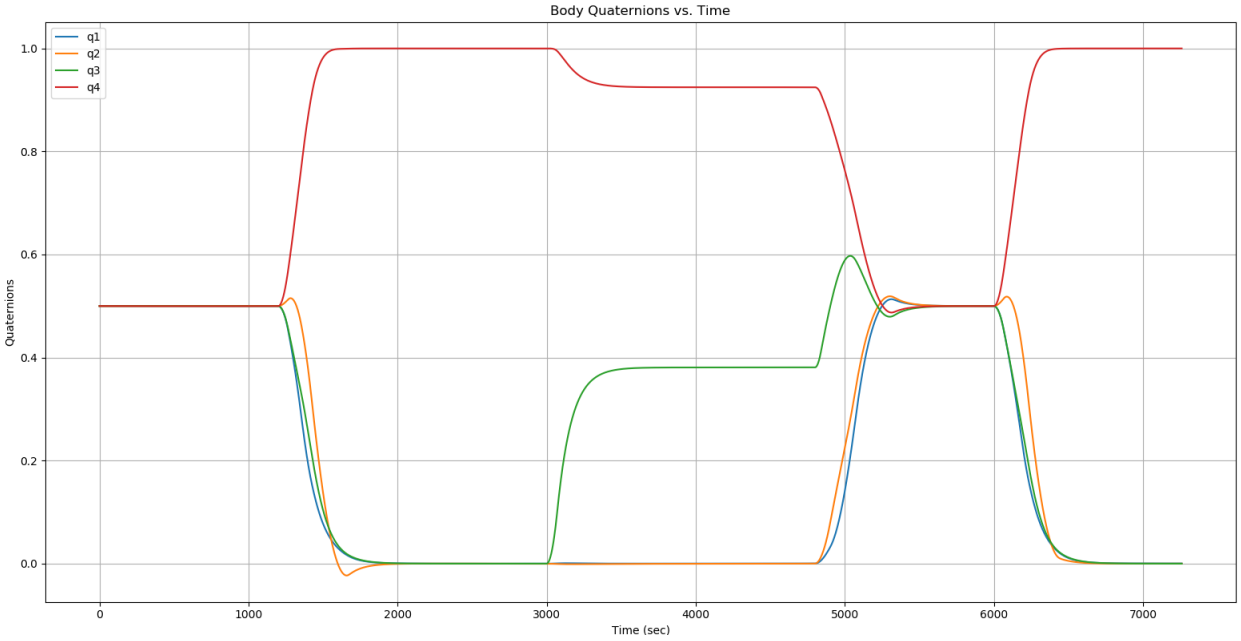
9 Multiple Maneuvers

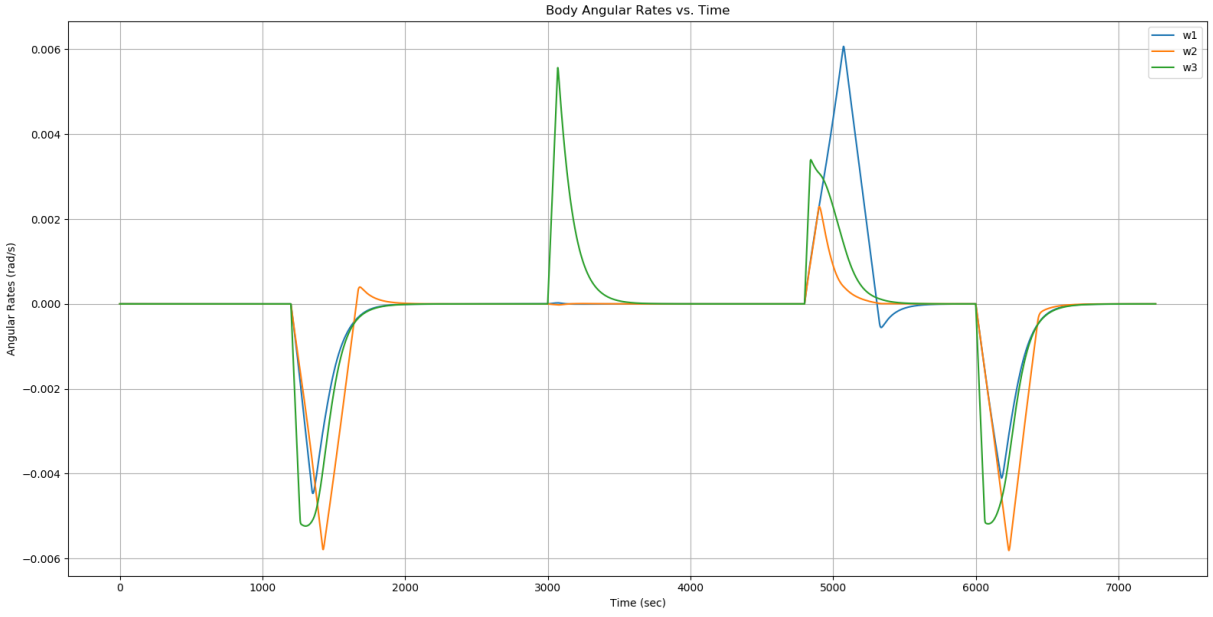
9.1 Description

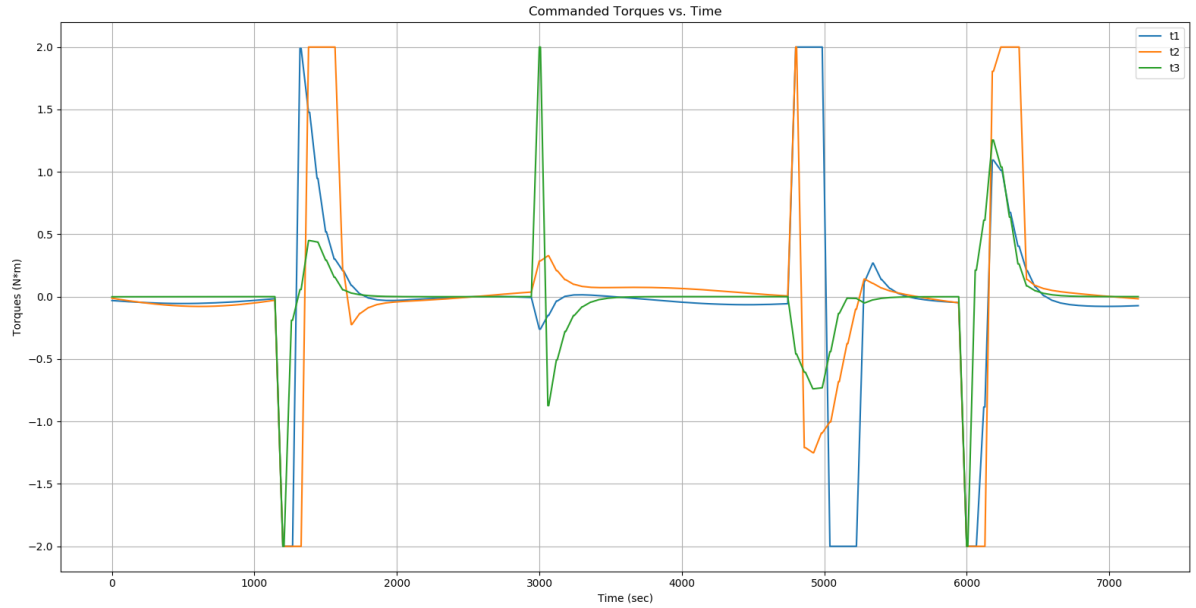
In the interest of exploring the capabilities of this system, the spacecraft was given multiple maneuvers to perform at different times within the simulated day. It was desired to perform this maneuver quickly, and then hold the desired attitude to within 0.1 degrees error.

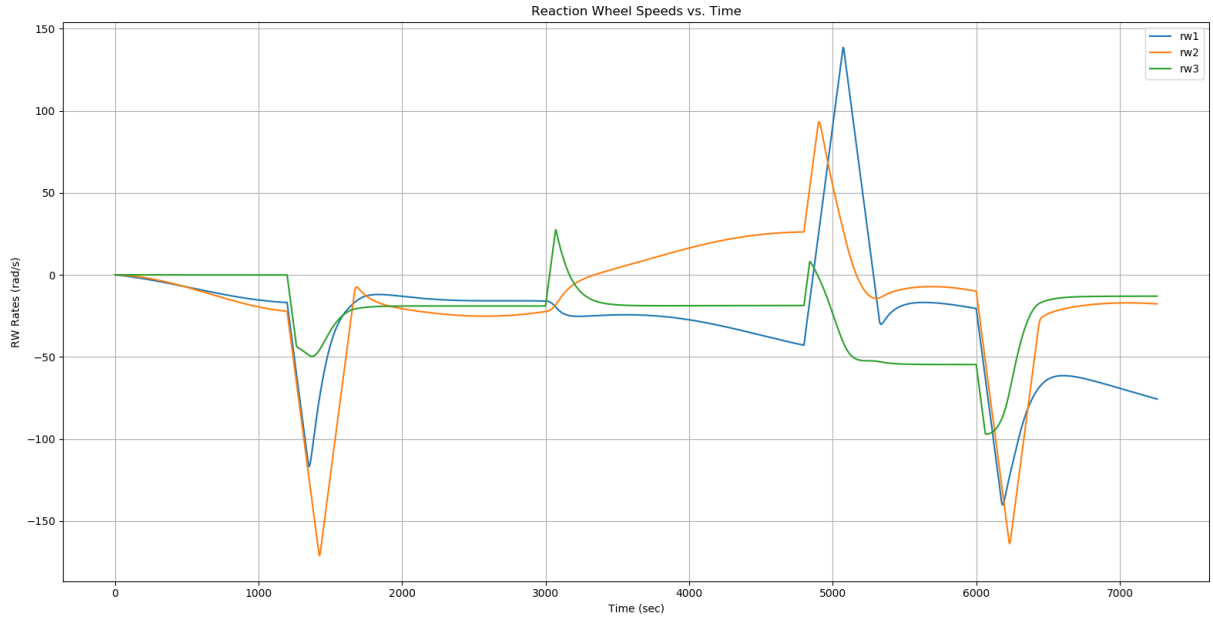
9.2 Results

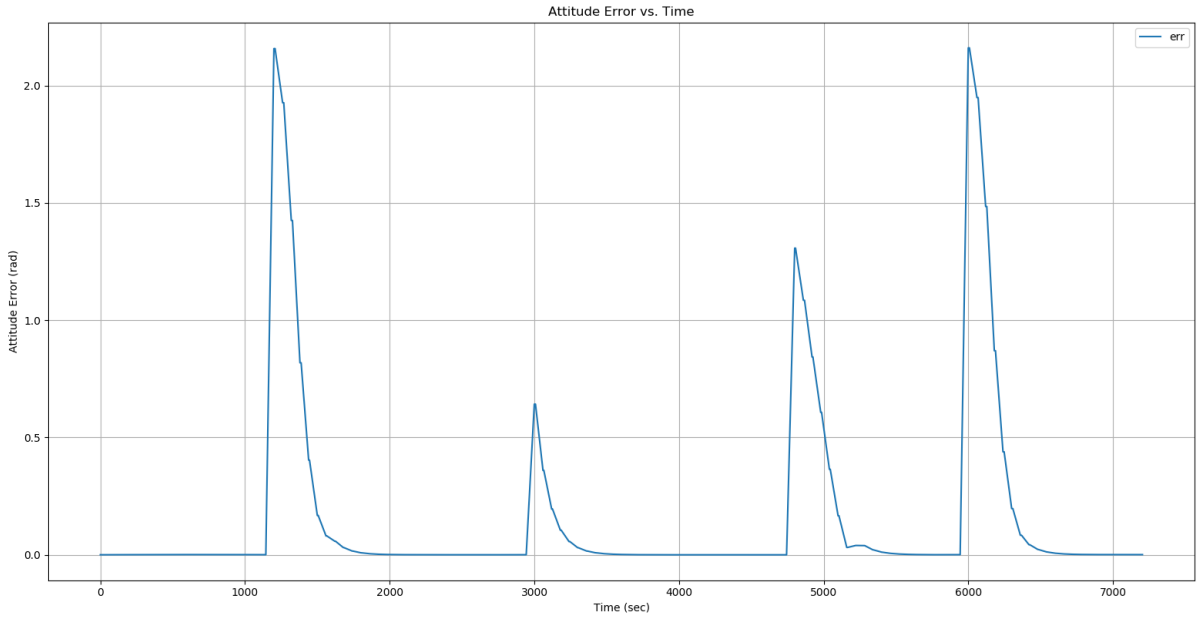



Body Quaternions vs. Time
q1
q2
q3
q4
Quaternions
Time (sec)
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
6000
7000


Body Angular Rates vs. Time
w1
w2
w3
Angular Rates (rad/s)
Time (sec)
0.006
0.004
0.002
0.000
−0.002
−0.004
−0.006
0
1000
2000
3000
4000
5000
6000
7000


Commanded Torques vs. Time
t1
t2
t3
Torques (N*m)
Time (sec)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
1000
2000
3000
4000
5000
6000
7000


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
Time (sec)
150
100
50
0
−50
−100
−150
0
1000
2000
3000
4000
5000
6000
7000



9.3 Discussion

As shown from the data above, the satellite is capable of performing all of these maneuvers in quick succession while in the presence of disturbance torques. The reaction wheels never encountered the need to desaturate, and the attitude error of the spacecraft was kept below 0.1 degrees.

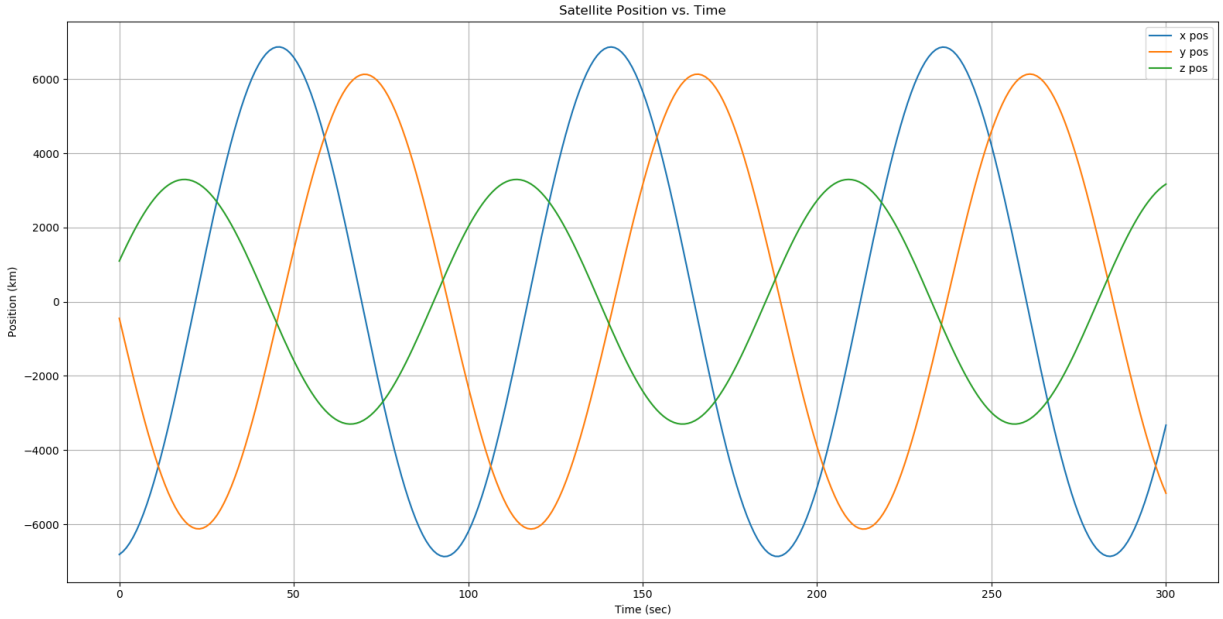
10 Nadir Pointing

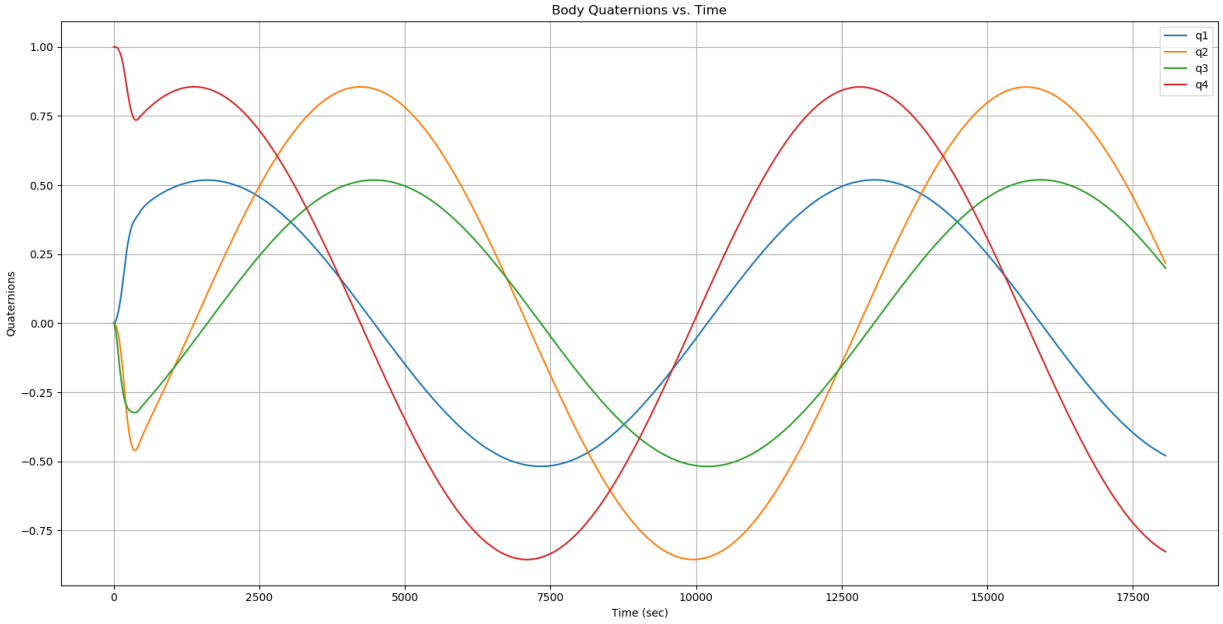
10.1 Description

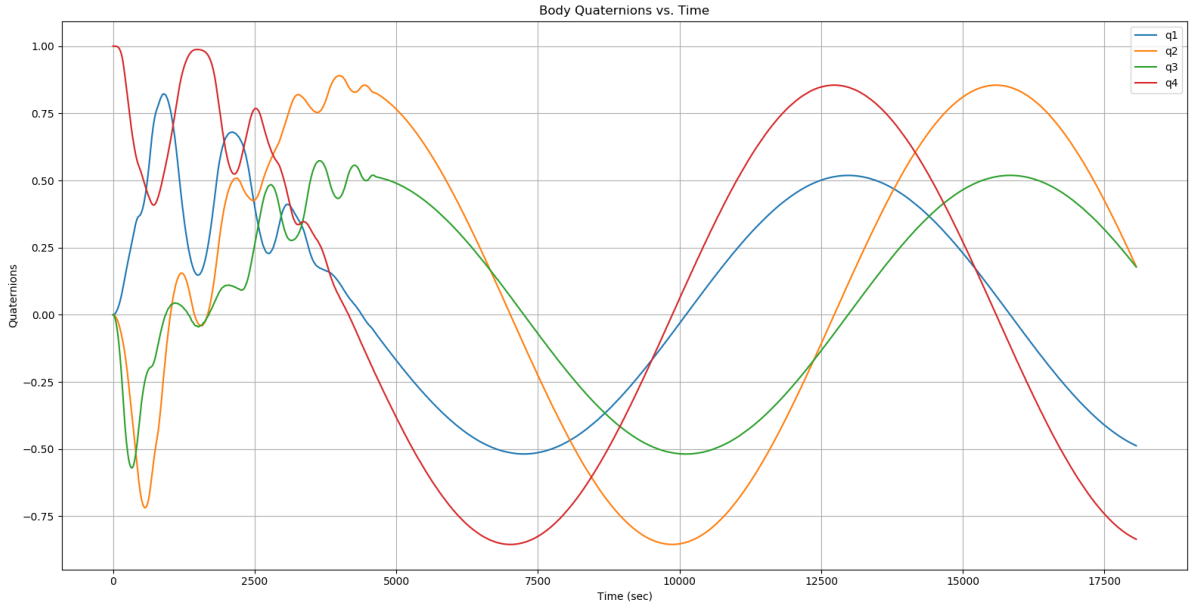
To better understand the capabilities of the spacecraft's ADCS, the spacecraft was commanded to hold a nadir pointing orientation to within an acceptable error. By modifying the proportional and derivative gains of the controller, the attitude error of this pointing mode was able to be minimized.

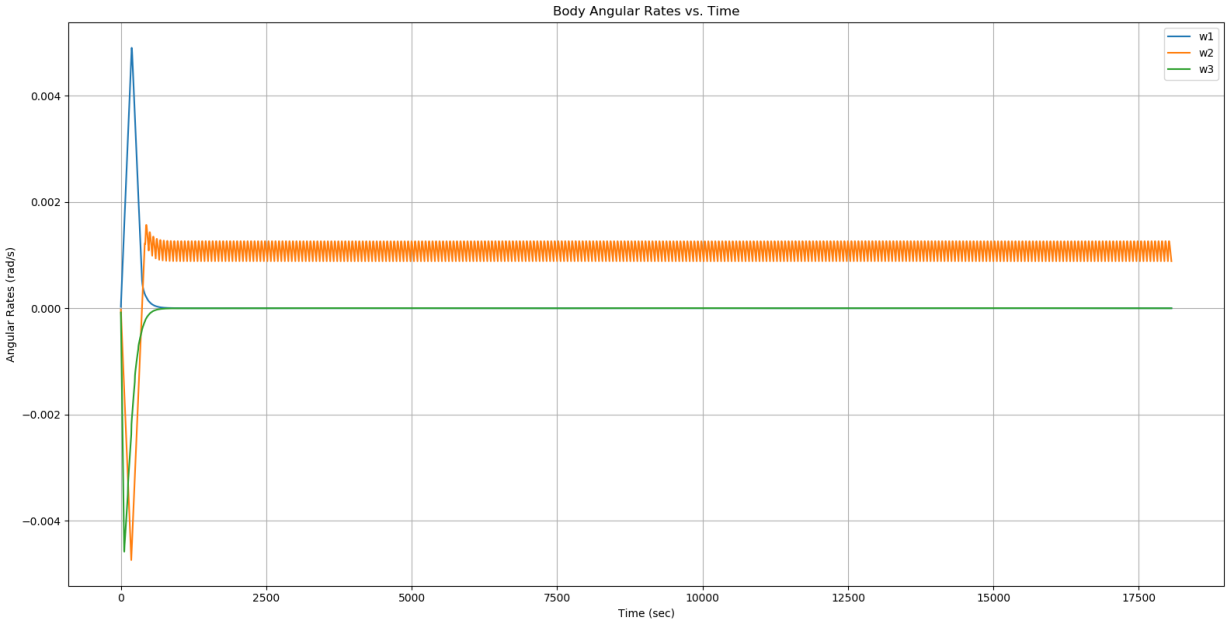
10.2 Results

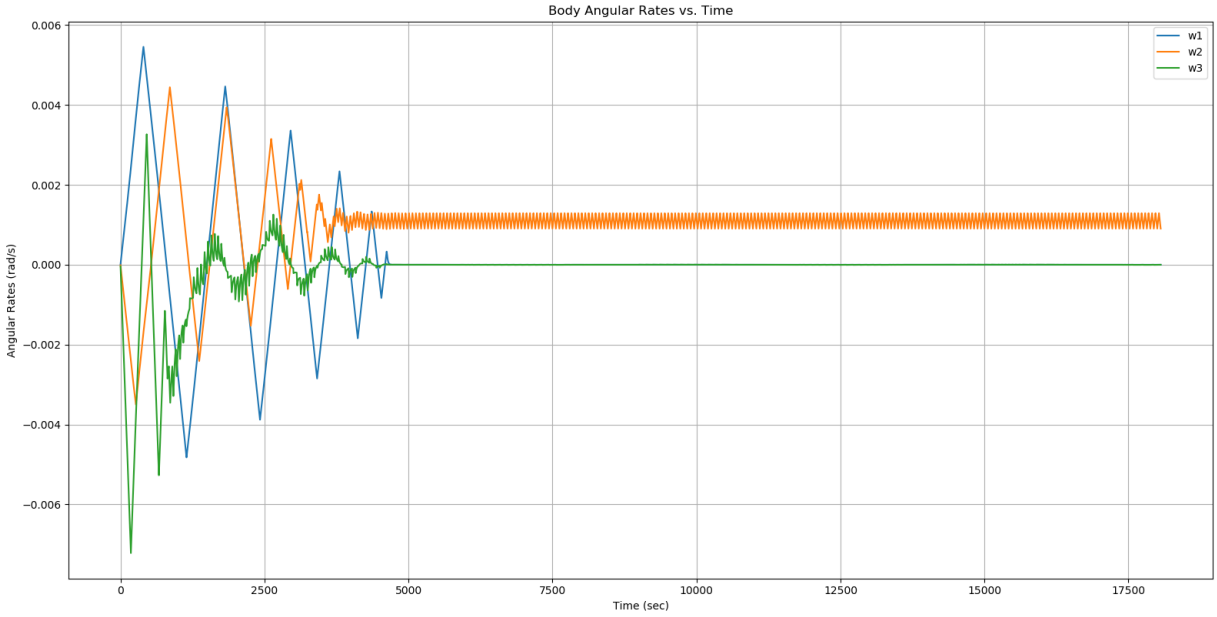
There are two sets of data within this section. The first graph of each type of graph displays the results for the proportional gain set equal to 10, while the second graph shows the data for the proportional gain set equal to 100.

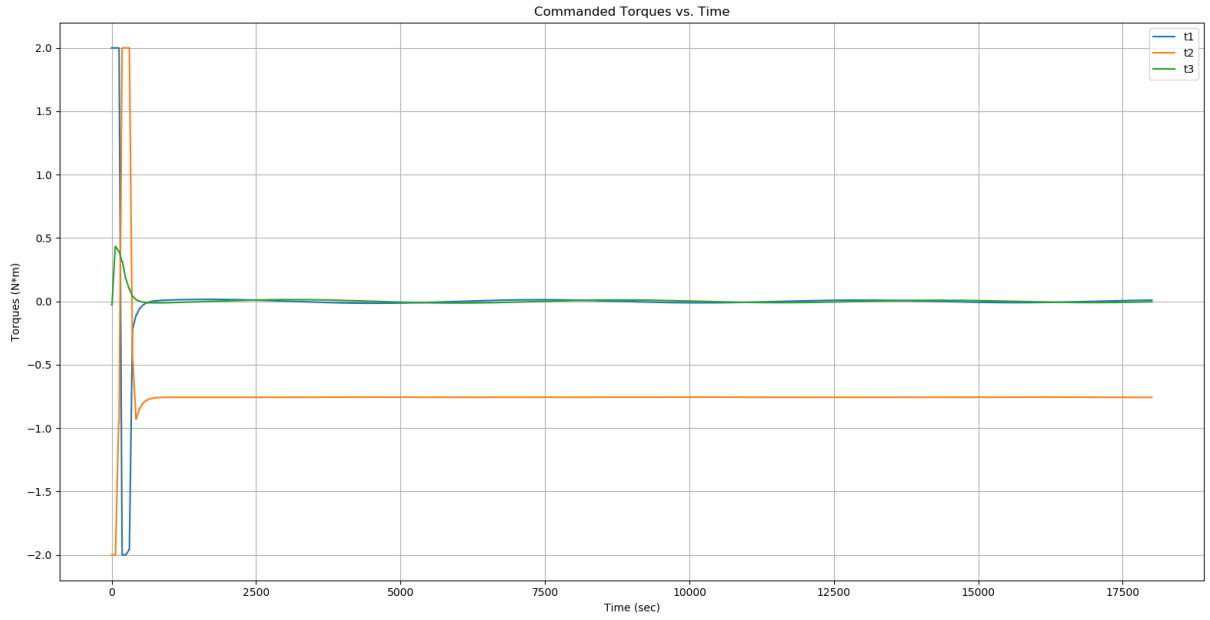


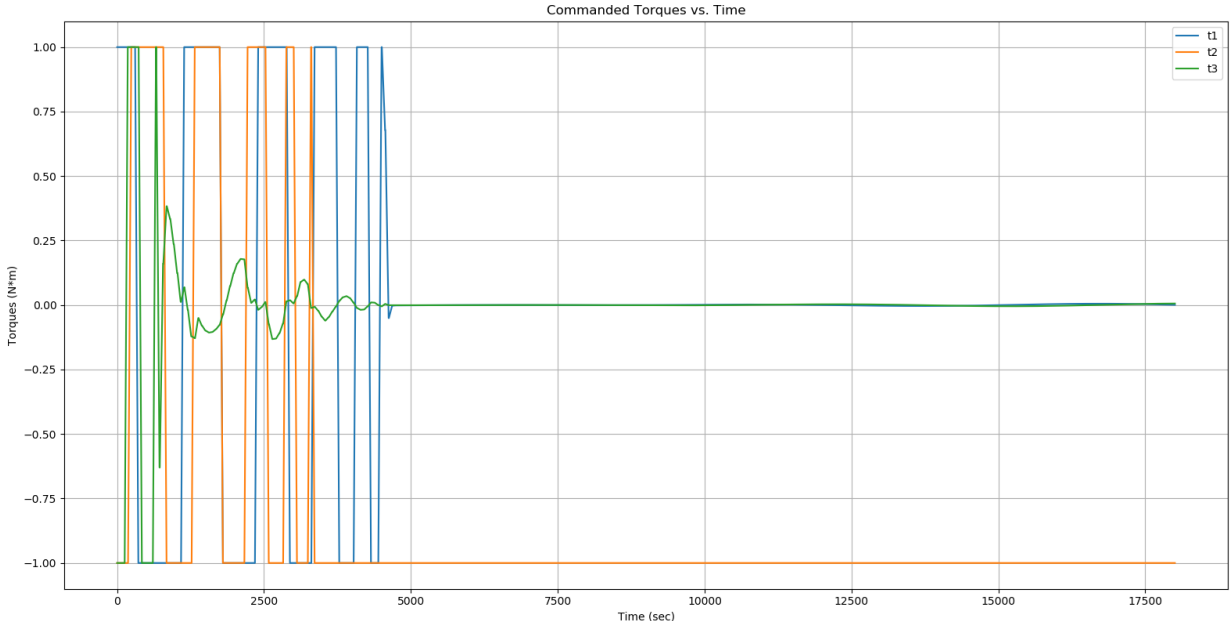

Body Quaternions vs. Time
q1
q2
q3
q4
Quaternions
Time (sec)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
2500
5000
7500
10000
12500
15000
17500

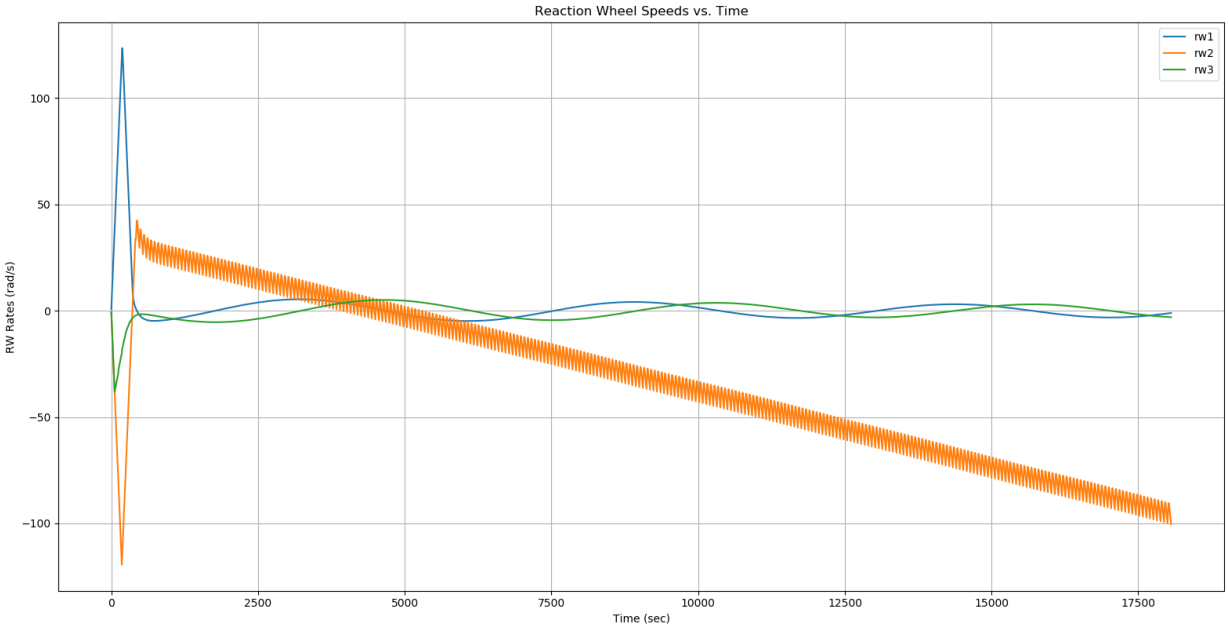

Body Quaternions vs. Time
q1
q2
q3
q4
Quaternions
Time (sec)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
2500
5000
7500
10000
12500
15000
17500

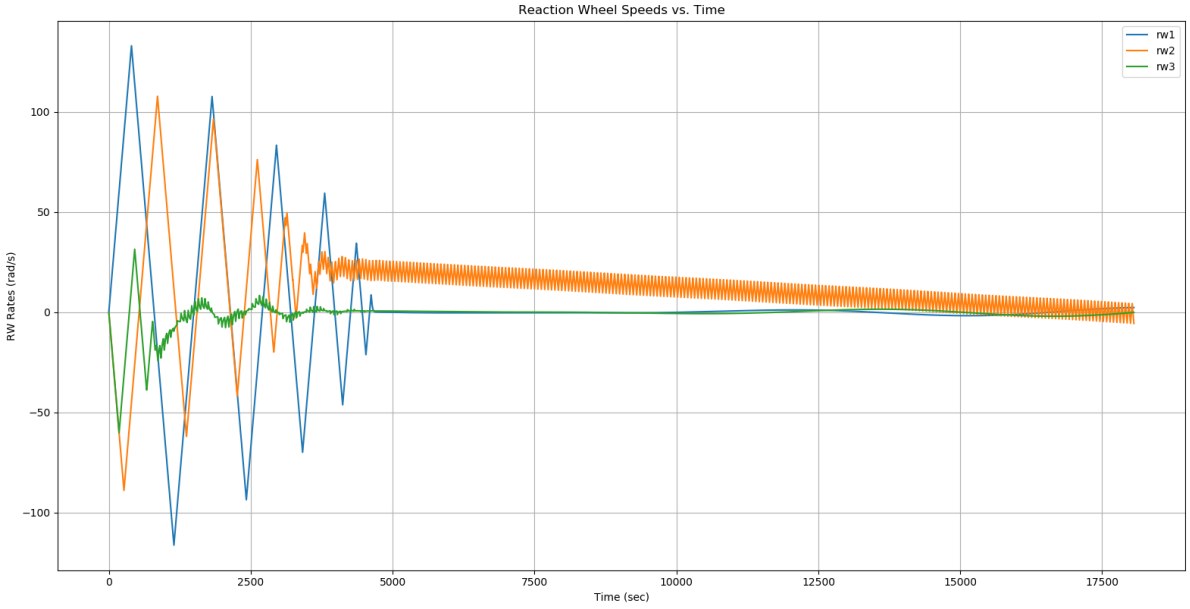

Body Angular Rates vs. Time
w1
w2
w3
Angular Rates (rad/s)
0.004
0.002
0.000
−0.002
−0.004
0
2500
5000
7500
10000
12500
15000
17500
Time (sec)

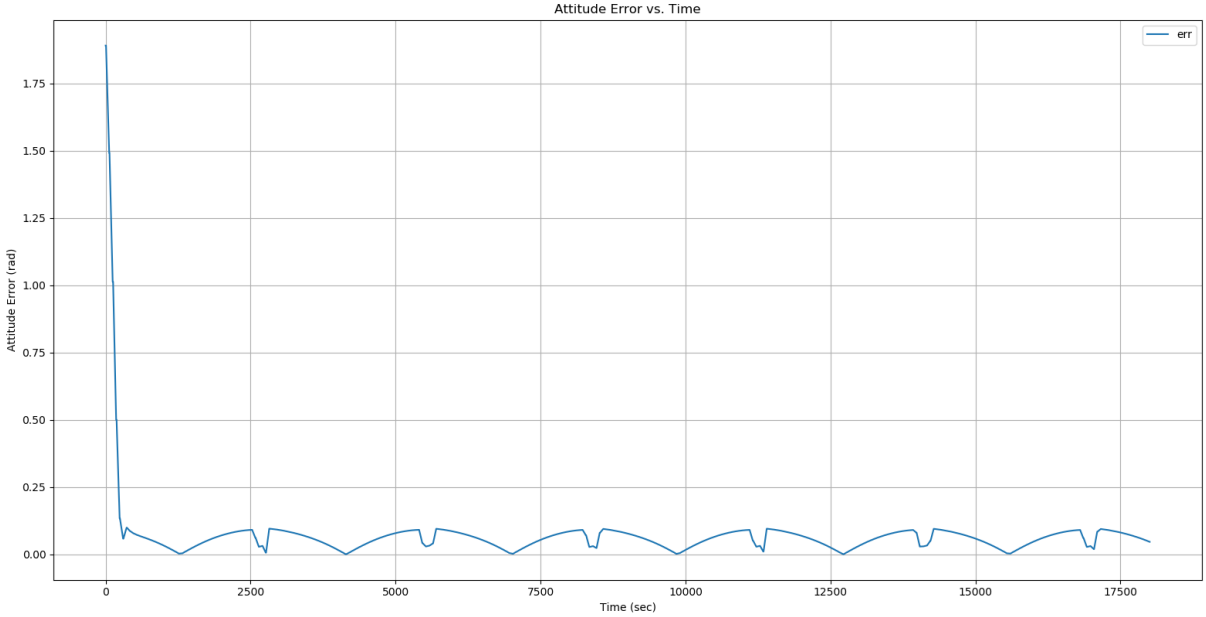

Body Angular Rates vs. Time
w1
w2
w3
Angular Rates (rad/s)
0.006
0.004
0.002
0.000
−0.002
−0.004
−0.006
0
2500
5000
7500
10000
12500
15000
17500
Time (sec)

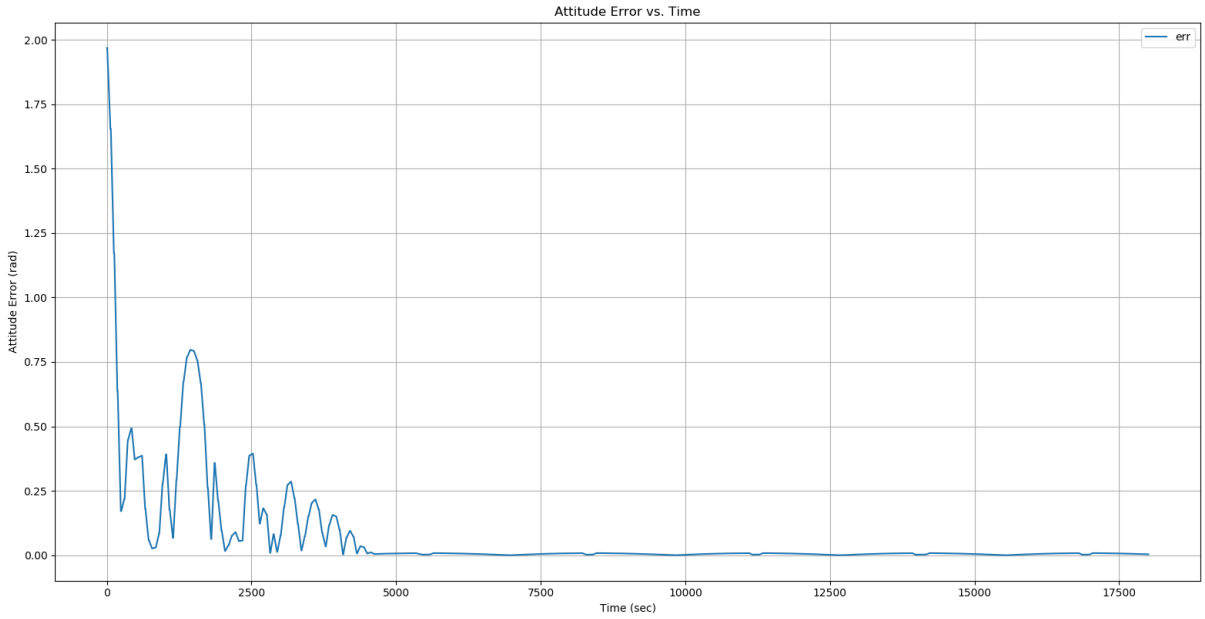

Commanded Torques vs. Time
t1
t2
t3
Torques (N*m)
Time (sec)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
2500
5000
7500
10000
12500
15000
17500


Commanded Torques vs. Time
t1
t2
t3
Torques (N*m)
Time (sec)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2500
5000
7500
10000
12500
15000
17500


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
100
50
0
−50
−100
0
2500
5000
7500
10000
12500
15000
17500
Time (sec)


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
100
50
0
−50
−100
0
2500
5000
7500
10000
12500
15000
17500
Time (sec)





10.3 Discussion

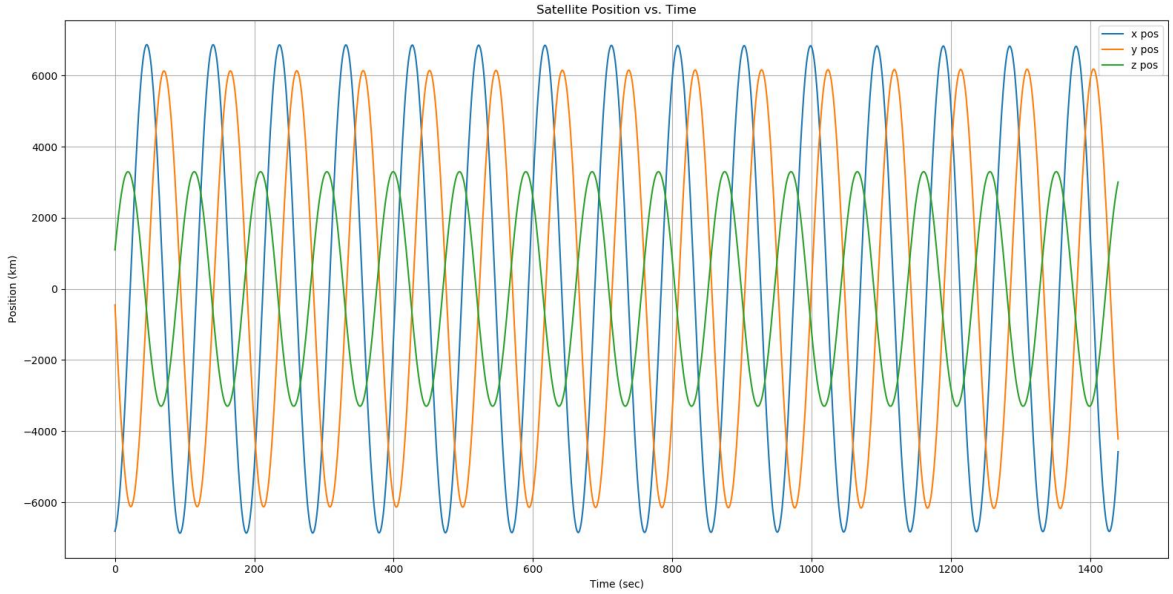
It is interesting to note that, upon increasing the proportional gains of the controller, the initial attitude convergence to nadir pointing took longer. However, the end result of the simulations show that the attitude error performed much better with the higher proportional gains. For the initial maneuver, a different control algorithm that utilizes different gains should be used to perform this maneuver.

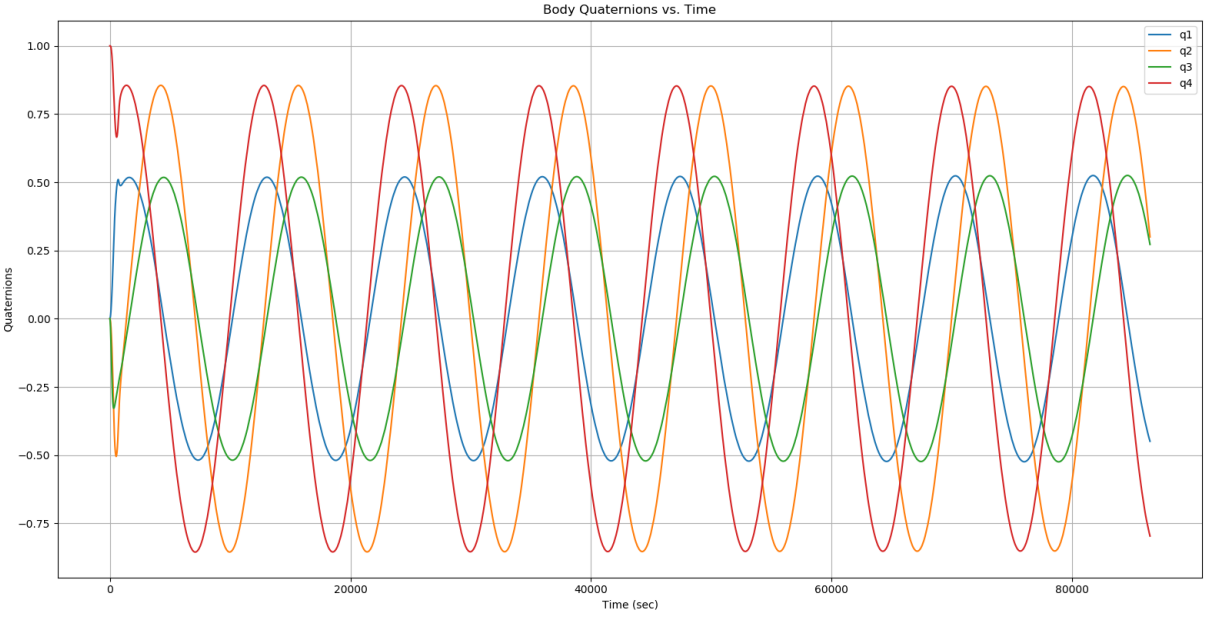
11 Conclusion

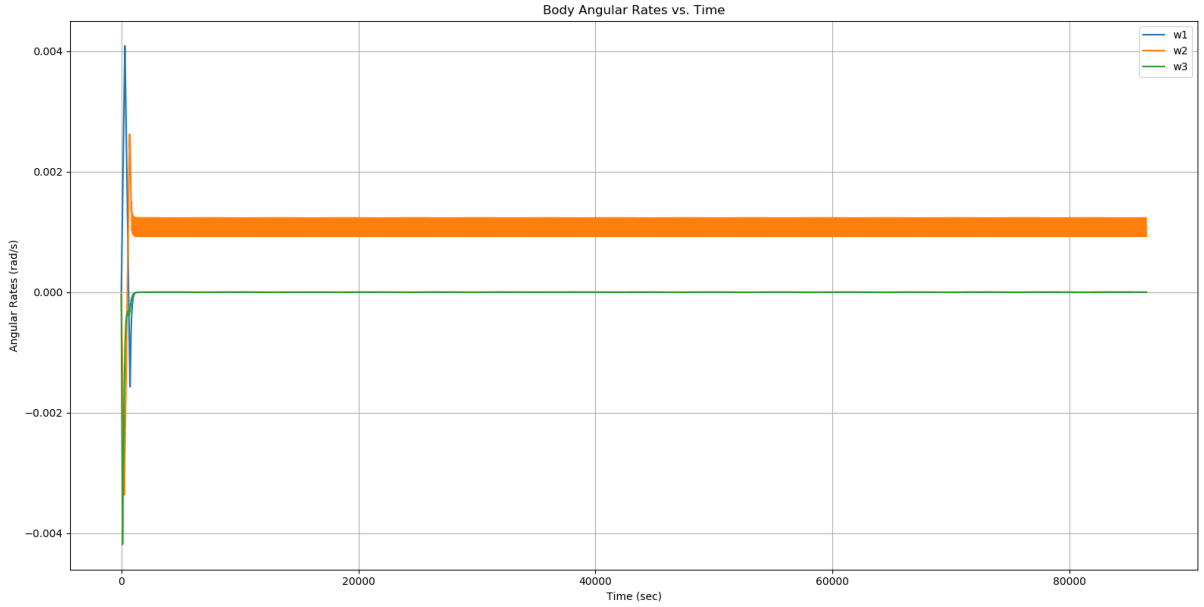
From this report it was found that the ADCS can perform all necessary maneuvers and attitude holds. The spacecraft can detumble with the assistance of thrusters for reaction wheel desaturation, and the reaction wheels can mitigate all disturbance torques while only needing less than one desaturation per wheel per day. One common theme shared amongst each pointing mode was that the controller needed to be modified to satisfy the pointing requirements imposed on each mode of operation. This result is reasonable and can be easily programmed into the ADCS. The final specifications of the spacecraft's actuators are listed below:

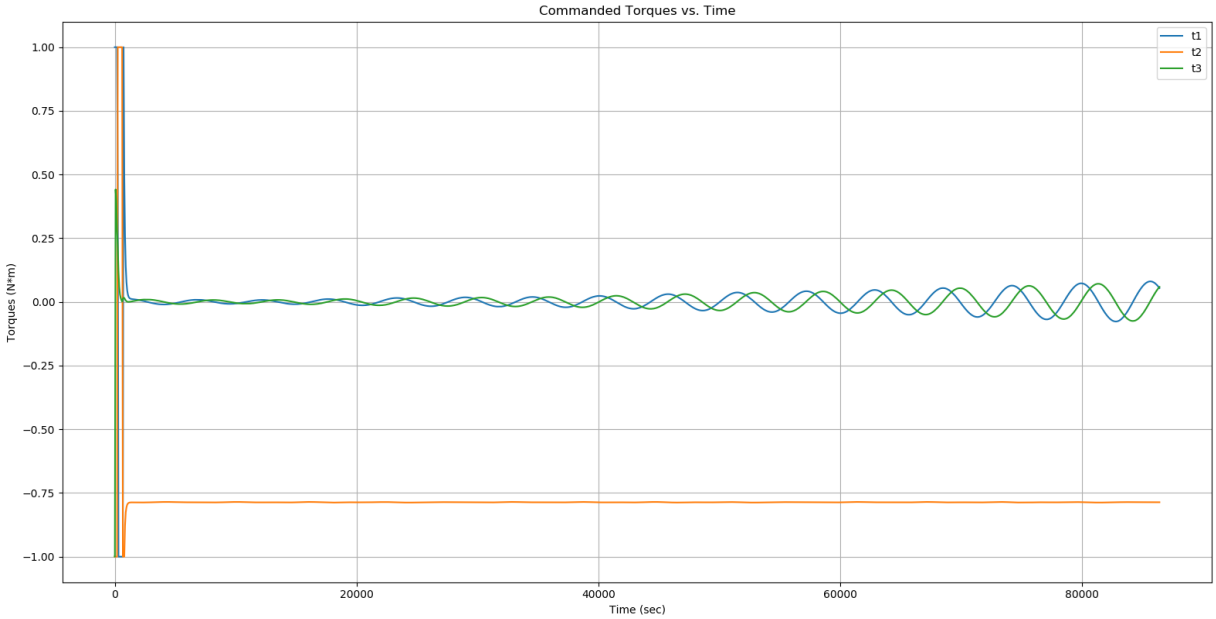
- Reaction wheel moments of inertia: 3 kg*m^2
- Reaction wheel maximum wheel speed: 700 rad/s
- Thruster torques: 1 Nm

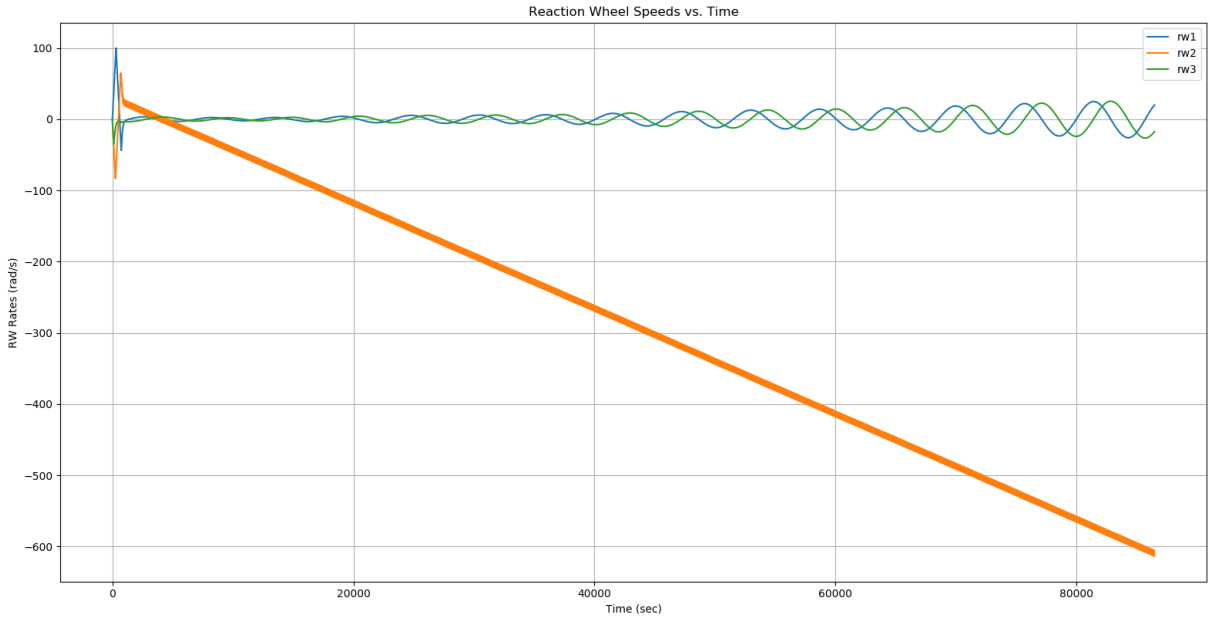
12 Appendix B: Additional Nadir Pointing Data

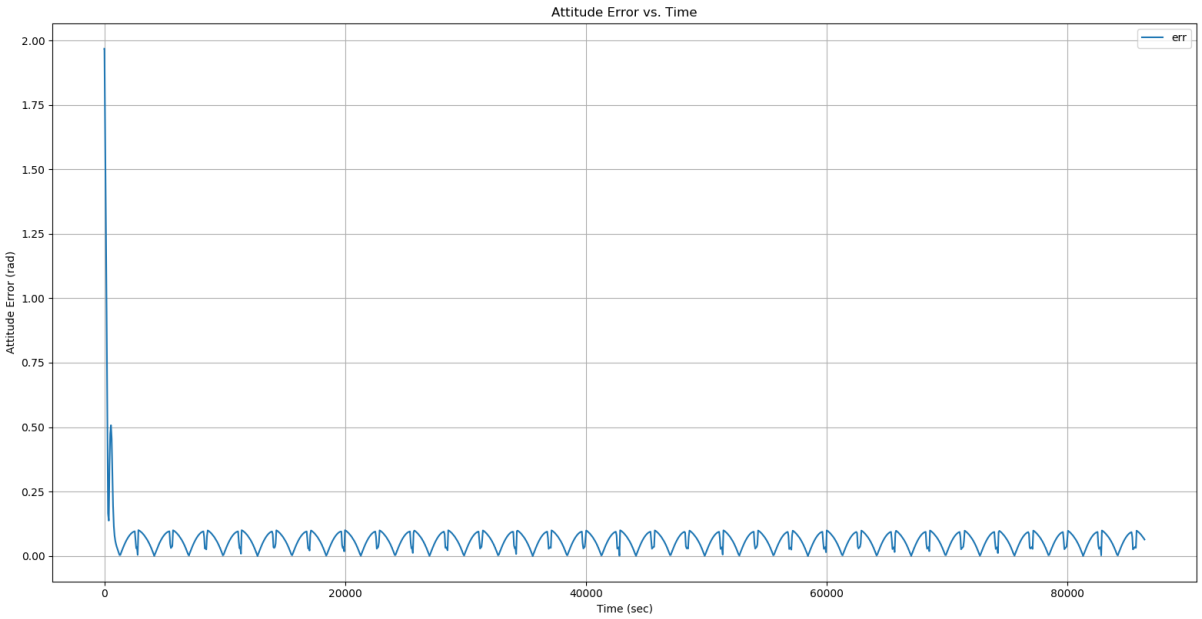





Body Angular Rates vs. Time
w1
w2
w3
0.004
0.002
0.000
−0.002
−0.004
Angular Rates (rad/s)
0
20000
40000
60000
80000
Time (sec)


Commanded Torques vs. Time
t1
t2
t3
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Torques (N*m)
0
20000
40000
60000
80000
Time (sec)


Reaction Wheel Speeds vs. Time
rw1
rw2
rw3
RW Rates (rad/s)
Time (sec)
100
0
−100
−200
−300
−400
−500
−600
0
20000
40000
60000
80000


Attitude Error vs. Time
err
Attitude Error (rad)
Time (sec)
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
20000
40000
60000
80000

13 Appendix B: Code

```
"""
Aero 424 Final Project - ADCS Simulation
Andrew S. Hollitser
UIN: 127008398
"""

import numpy as np
from csv import reader
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from time import perf_counter
import datetime
import math


class DataStructure:

    def __init__(self):
        self.pv_data = []
        self.qw_data = []
        self.cmd_torq_data = []
        self.rw_speed_data = []
        self.att_err_data = []

    # Add data function
    def add_data(self, time_id: float = None, pos: np.array = None, vel: np.array = None,
                 att: np.array = None, w: np.array = None, cmd_torq: np.array = None, rw_sp: np.array = None,
                 att_err_data: np.array = None):

        # Appending pv state data
        self.pv_data.append((time_id, pos, vel))

        # appending attitude related data
        for sec, data in enumerate(zip(att, w)):
            self.qw_data.append((60 * time_id + sec, np.reshape(data[0], (4, 1)), np.reshape(data[1], (3, 1))))

        # Appending Command Torques
        for sec, data in enumerate(cmd_torq):
            self.cmd_torq_data.append((60 * time_id + sec, data))

        # Appending Reaction Wheel Speeds
        for sec, data in enumerate(rw_sp):
            self.rw_speed_data.append((60 * time_id + sec, data))

        # Appending Attitude Error
        for sec, data in enumerate(att_err_data):
            self.att_err_data.append((60 * time_id + sec, data))

    # Returning pv state related data
    def return_all_pv_time(self):
        return [item[0] for item in self.pv_data]

    def return_all_pos(self):
        return [item[1] for item in self.pv_data]

    def return_all_vel(self):
        return [item[2] for item in self.pv_data]

    # Return attitude related data
    def return_all_qw_time(self):
        return [item[0] for item in self.qw_data]

    def return_all_att(self):
        return [item[1] for item in self.qw_data]

    def return_all_w(self):
        return [item[2] for item in self.qw_data]

    # Return Torque Data
    def return_all_ct_times(self):
        return [item[0] for item in self.cmd_torq_data]

    def return_all_cmd_torqs(self):
        return [item[1] for item in self.cmd_torq_data]
```

```
    # Return Reaction Wheel Speed Data
    def return_all_rw_times(self):
        return [item[0] for item in self.rw_speed_data]

    def return_all_rw_speeds(self):
        return [item[1] for item in self.rw_speed_data]

    # Return Attitude Error Data
    def return_all_att_times(self):
        return [item[0] for item in self.att_err_data]

    def return_all_att_err(self):
        return [item[1] for item in self.att_err_data]


class Plate:

    def __init__(self, area: float, normal: np.array, pos: np.array, absorpt: float, spec: float, diffus:
float):
        self.area = area  # m^2
        self.normal = normal  # unit vector | SBF
        self.pos = pos  # m | SBF
        self.absorpt = absorpt  # unitless
        self.spec = spec  # unitless
        self.diffus = diffus  # unitless


class ReactionWheel:

    def __init__(self, jw: float, ws: float, w_dot: float, max_ws: float, min_ws: float, unit: np.array):
        self.jw = jw  # Wheel Moment of Inertia | # kg*m^2
        self.ws = ws  # Wheel Angular Rate | Rad/s
        self.w_dot = w_dot  # Wheel Angular Acceleration | Rad/s^2
        self.max_ws = max_ws  # Wheel Max Angular Rate | Rad/s
        self.min_ws = min_ws  # Wheel Min Angular Rate | Rad/s
        self.unit = unit  # Wheel unit vector

    def return_gyro(self):
        return self.jw*self.ws*self.unit

    def return_torq(self):
        return self.jw * self.w_dot * self.unit


class Thruster:

    def __init__(self, unit: np.array, pos_ori: int, torq: float = 35, desat: bool = 0):
        self.torq = torq  # Torque the thruster is capable of producing
        self.unit = unit  # Unit vector of thrust torque
        self.desat = desat  # boolean to indicate desat in progress
        self.pos_ori = pos_ori  # boolean to indicate positive or negative orientation of thruster

    def thrust(self, ws: float, ws_max: float, ws_min: float):
        self.desat += abs(ws) > ws_max  # Checks for a desat trigger
        self.desat *= np.sign(ws) != self.pos_ori  # Checks to see if the thruster is the properly oriented
thruster
        self.desat = (abs(ws) > ws_min) * bool(self.desat)  # Checks to see if desat is complete
        thrust = bool(self.desat) * self.torq * self.unit
        return thrust


class Satellite:

    def __init__(self,
                 pos: np.array = np.array([[0], [0], [0]]),
                 vel: np.array = np.array([[0], [0], [0]]),
                 att: np.array = np.array([[0], [0], [0], [1]]),
                 des_att: np.array = np.array([[0], [0], [0], [1]]),
                 w: np.array = np.array([[0], [0], [0]]),
                 minute: int = 0):

        # Current Time
        self.minute = minute

        # Satellite Position - ECI Frame
        self.pos = pos  # km

        # Satellite Velocity - ECI Frame
        self.vel = vel  # km/s
```

```
        # Satellite Attitude in Quaternion Form - ECI Frame
        self.att = att

        # Satellite Desired Attitude in Quaternion Form - ECI Frame
        self.des_att = des_att

        # Satellite Angular Velocity - ECI Frame
        self.w = w

        # Atmospheric Density
        self.rho = 3e-18  # kg/m^3

        # Principal Moments of Inertia - SBF
        self.jc = np.diag([77217, 77217, 25000])  # kg*m^2
        self.jc_inv = np.linalg.inv(self.jc)

        # Reaction Wheels' Moments of Inertia - SBF
        self.rw_1 = ReactionWheel(jw=3,
                                  ws=0,
                                  w_dot=0,
                                  max_ws=700,
                                  min_ws=1,
                                  unit=np.array([[1], [0], [0]]))

        self.rw_2 = ReactionWheel(jw=3,
                                  ws=0,
                                  w_dot=0,
                                  max_ws=700,
                                  min_ws=1,
                                  unit=np.array([[0], [1], [0]]))

        self.rw_3 = ReactionWheel(jw=3,
                                  ws=0,
                                  w_dot=0,
                                  max_ws=700,
                                  min_ws=1,
                                  unit=np.array([[0], [0], [1]]))

        self.jw_inv = np.linalg.inv(np.diag([self.rw_1.jw, self.rw_2.jw, self.rw_3.jw]))

        # Thrusters of the Satellite
        self.thruster1 = Thruster(unit=np.array([[1], [0], [0]]), pos_ori=1)
        self.thruster2 = Thruster(unit=np.array([[-1], [0], [0]]), pos_ori=-1)
        self.thruster3 = Thruster(unit=np.array([[0], [1], [0]]), pos_ori=1)
        self.thruster4 = Thruster(unit=np.array([[0], [-1], [0]]), pos_ori=-1)
        self.thruster5 = Thruster(unit=np.array([[0], [0], [1]]), pos_ori=1)
        self.thruster6 = Thruster(unit=np.array([[0], [0], [-1]]), pos_ori=-1)

        # Satellite Magnetic Residual - SBF
        self.mag_res = np.array([[20], [20], [20]])  # Am^2

        # Satellite Plates
        self.plate_1 = Plate(area=23,
                             normal=np.array([[1], [0], [0]]),
                             pos=np.array([[1], [0], [0]]),
                             absorpt=0.1,
                             spec=0.8,
                             diffus=0.1)

        self.plate_2 = Plate(area=18,
                             normal=np.array([[np.cos(np.deg2rad(30))], [np.sin(np.deg2rad(30))], [0]]),
                             pos=np.array([[0], [0.6], [0]]),
                             absorpt=0.2,
                             spec=0.75,
                             diffus=0.05)

        self.plate_3 = Plate(area=23,
                             normal=np.array([[np.cos(np.deg2rad(30))], [np.sin(np.deg2rad(30))], [0]]),
                             pos=np.array([[0], [-0.6], [0]]),
                             absorpt=0.2,
                             spec=0.75,
                             diffus=0.05)

    def propagate(self, pos: np.array, vel: np.array):

        # Updating Position and Velocity
        self.pos = pos
        self.vel = vel
```

```
        # Propagating Attitude and Angular Rates
        def odes(y, t):
            q_1, q_2, q_3, q_4, w_1, w_2, w_3, rw_1, rw_2, rw_3 = y

            # Angular rate terms
            w = np.array([[w_1], [w_2], [w_3]])

            # Disturbance torques
            l_grav = grav_grad(self.pos, self.jc, self.att)
            l_mag = mag_torq(self.pos, self.att, self.mag_res)
            l_aero = aero_torq([self.plate_1, self.plate_2, self.plate_3], self.pos, self.vel, self.att,
self.rho)
            l_srp = srp_torq([self.plate_1, self.plate_2, self.plate_3], self.pos, self.att)
            l_dist = l_srp+l_grav+l_mag+l_aero

            # Control Torque
            ctrl_t = controller(self.des_att, self.att, self.w)
            # ctrl_t = np.array([[0], [0], [0]])

            # Capping Control Torques - Keeps wheels from over saturating
            ctrl_t[0][0] = ctrl_t[0][0] * (abs(self.rw_1.ws) < self.rw_1.max_ws
                                           or np.sign(ctrl_t[0][0]) != np.sign(self.rw_1.ws))
            ctrl_t[1][0] = ctrl_t[1][0] * (abs(self.rw_2.ws) < self.rw_2.max_ws
                                           or np.sign(ctrl_t[1][0]) != np.sign(self.rw_2.ws))
            ctrl_t[2][0] = ctrl_t[2][0] * (abs(self.rw_3.ws) < self.rw_3.max_ws
                                           or np.sign(ctrl_t[2][0]) != np.sign(self.rw_3.ws))

            # Thruster Torques
            t1 = self.thruster1.thrust(self.rw_1.ws, self.rw_1.max_ws, self.rw_1.min_ws)
            t2 = self.thruster2.thrust(self.rw_1.ws, self.rw_1.max_ws, self.rw_1.min_ws)
            t3 = self.thruster3.thrust(self.rw_2.ws, self.rw_2.max_ws, self.rw_2.min_ws)
            t4 = self.thruster4.thrust(self.rw_2.ws, self.rw_2.max_ws, self.rw_2.min_ws)
            t5 = self.thruster5.thrust(self.rw_3.ws, self.rw_3.max_ws, self.rw_3.min_ws)
            t6 = self.thruster6.thrust(self.rw_3.ws, self.rw_3.max_ws, self.rw_3.min_ws)
            desat_t = sum([t1, t2, t3, t4, t5, t6])
            # desat_t = np.array([[0], [0], [0]])

            # Wheel Dynamics
            gyro_rw = np.cross(w, self.rw_1.return_gyro() + self.rw_2.return_gyro() + self.rw_3.return_gyro(),
axis=0)
            rw_dot = np.matmul(self.jw_inv, desat_t+ctrl_t)

            # Gyroscopic terms
            gyro = np.cross(-w, np.matmul(self.jc, w), axis=0)
            w_dot = np.matmul(self.jc_inv, l_dist+ctrl_t+gyro+gyro_rw)

            # Recording Commanded Torques
            cmd_torqs.append(ctrl_t)

            # Recording Quaternion Error
            att_err_list.append(att_err(self.att, self.des_att))

            # Attitude Terms
            q = np.array([[q_1], [q_2], [q_3], [q_4]])
            q_dot_b = q_dot(w, q)

            return [q_dot_b[0][0],
                    q_dot_b[1][0],
                    q_dot_b[2][0],
                    q_dot_b[3][0],
                    w_dot[0][0],
                    w_dot[1][0],
                    w_dot[2][0],
                    rw_dot[0][0],
                    rw_dot[1][0],
                    rw_dot[2][0]]

        # Performing Numerical Integration
        t = np.linspace(0, 1, 2)
        att_list = []
        w_list = []
        cmd_torqs = []
        rw_list = []
        att_err_list = []

        for step in range(60):
            # Command Torque Data
            cmd_torqs = []
```

```
            att_err_list = []

            # Updating time
            global time
            time = self.minute*60 + step

            y_0 = [self.att[0][0], self.att[1][0], self.att[2][0], self.att[3][0],
                   self.w[0][0], self.w[1][0], self.w[2][0],
                   self.rw_1.ws, self.rw_2.ws, self.rw_3.ws]

            sol = odeint(odes, y_0, t)

            q1 = sol[:, 0]
            q2 = sol[:, 1]
            q3 = sol[:, 2]
            q4 = sol[:, 3]
            w1 = sol[:, 4]
            w2 = sol[:, 5]
            w3 = sol[:, 6]
            rw1 = sol[:, 7]
            rw2 = sol[:, 8]
            rw3 = sol[:, 9]

            prop_q = np.transpose(([q1, q2, q3, q4]))
            prop_w = np.transpose(([w1, w2, w3]))

            self.att = np.reshape(prop_q[-1], (4, 1))
            self.w = np.reshape(prop_w[-1], (3, 1))
            self.rw_1.ws = rw1[-1]
            self.rw_2.ws = rw2[-1]
            self.rw_3.ws = rw3[-1]

            att_list.append(self.att)
            w_list.append(self.w)
            rw_list.append(np.array([[self.rw_1.ws], [self.rw_2.ws], [self.rw_3.ws]]))

        return att_list, w_list, cmd_torqs, rw_list, att_err_list


"""
PID Controller
"""


def controller(des_att: np.array, curr_att: np.array, curr_w: np.array):

    # Unpacking data elements for convenience
    # Desired quaternion
    qc1 = curr_att[0][0]
    qc2 = curr_att[1][0]
    qc3 = curr_att[2][0]
    qc4 = curr_att[3][0]

    # Current quaternion
    qd1 = des_att[0][0]
    qd2 = des_att[1][0]
    qd3 = des_att[2][0]
    qd4 = des_att[3][0]

    # Current angular rates
    p = curr_w[0][0]
    q = curr_w[1][0]
    r = curr_w[2][0]

    # Deriving error quaternion
    qd_err = np.array([[qc4, qc3, -qc2, qc1],
                       [-qc3, qc4, qc1, qc2],
                       [qc2, -qc1, qc4, qc3],
                       [-qc1, -qc2, -qc3, qc4]])
    qc_err = np.array([[-qd1], [-qd2], [-qd3], [qd4]])
    q_err = np.matmul(qd_err, qc_err)

    # Proportional gains
    kx = 10
    ky = 10
    kz = 10

    # Derivative gains
    kdx = 10000
```

```
    kdy = 10000
    kdz = 10000

    # Calculating control torque
    tx = -2*kx*q_err[0][0]*q_err[3][0] - kdx*p
    ty = -2*ky*q_err[1][0]*q_err[3][0] - kdy*q
    tz = -2*kz*q_err[2][0]*q_err[3][0] - kdz*r

    # Max Torque
    max_t = 1
    # if time > 60*90:
    #     max_t = 10e-2

    # Capping control torques at 1 Nm
    tx = np.sign(tx)*min(max_t, abs(tx))
    ty = np.sign(ty)*min(max_t, abs(ty))
    tz = np.sign(tz)*min(max_t, abs(tz))

    # Formatting Torque
    ctrl_t = np.array([[tx], [ty], [tz]])

    return ctrl_t


"""
Disturbance Torques
"""


def grav_grad(pos: np.array, jb: np.array, att: np.array):
    mu = 398600  # km^3/s^2

    # Transforming attitude into a DCM
    a_body_eci = q_to_dcm(att)

    # Expressing Position in Body Frame
    pos = np.matmul(a_body_eci, pos)

    # Calculating the gravity gradient torque
    g_torq = 3*mu/np.linalg.norm(pos)**5*np.cross(pos, np.matmul(jb, pos), axis=0)

    return g_torq


def mag_torq(pos, att, res):

    # Defining Magnetic Spherical Reference Radius
    a = 6371.2e3  # m

    # Defining Additional Magnetic Constants
    g_1_0 = -29554.63e-9
    g_1_1 = -1669.05e-9
    h_1_1 = 5077.99e-9

    # Earth Magnetic Dipole Moment in ECEF Frame
    m = a**3*np.array([[g_1_1], [h_1_1], [g_1_0]])

    # ECEF Transformation Frame
    a_ecef_eci = pos_ecef_trans()

    # Position vector expressed in ecef frame
    pos_ecef = np.matmul(a_ecef_eci, 1e3*pos)

    # Magnetic field in ecef frame
    b = (3*np.matmul(np.transpose(m), pos_ecef)[0]*pos_ecef - np.linalg.norm(pos_ecef)**2*m) / \
        np.linalg.norm(pos_ecef)**5

    # Magnetic torque in ecef frame
    l_mag = np.cross(res, b, axis=0)

    # Creating ecef to body transformation
    a_body_eci = q_to_dcm(att)
    a_body_ecef = np.matmul(a_body_eci, np.transpose(a_ecef_eci))

    # Expressing magnetic torque in body frame
    l_mag_bod = np.matmul(a_body_ecef, l_mag)

    return l_mag_bod
```

```
def aero_torq(plates: list, pos: np.array, v_eci: np.array, att: np.array, rho: float):

    # Calculation of aerodynamic torque for all plates
    l_aero_sum = np.array([[0], [0], [0]])
    for index, plate in enumerate(plates):

        # Defining earth rotation in eci frame
        we_eci = np.array([[0], [0], [2 * np.pi / (23 * 3600 + 56 * 60 + 4)]])

        # Defining plate relative velocity in the body frame
        v_rel_eci = v_eci + np.cross(we_eci, pos, axis=0)
        a_body_eci = q_to_dcm(att)
        v_rel_body = np.matmul(a_body_eci, v_rel_eci)

        # Inclination of the plate with respect to the relative velocity
        theta_aero = np.matmul(np.transpose(plate.normal), v_rel_body)/np.linalg.norm(v_rel_body)

        # Approximate drag coefficient
        cd = 2.25

        # Aero dynamic force
        f_aero = -0.5*rho*cd*np.linalg.norm(v_rel_body)*v_rel_body*plate.area*max(np.cos(theta_aero), 0)*1e6

        # Position of the plate in the body frame
        ri = plate.pos*plate.normal

        # Aerodynamic torque in body frame
        l_aero = np.cross(ri, f_aero, axis=0)
        l_aero_sum = np.add(l_aero_sum, l_aero)

    return l_aero_sum


def srp_torq(plates, pos, att):

    # Getting time
    year = 2021
    month = 11
    day = 12 + time // 86400
    h = time // 3600 % 24
    m = time // 60 % 60
    s = time % 60

    # Determining angles based on date
    date = datetime.datetime(year, month, day, h, m, s)
    t_ut1 = get_julian_datetime(date)
    m_sun = np.deg2rad(357.5277233 + 35999.05034 * t_ut1)
    epsilon = np.deg2rad(23.439291 - 0.0130042 * t_ut1)
    phi_sun = 280.46 + 36000.771 * t_ut1
    phi_ecliptic = np.deg2rad(phi_sun + 1.914666471 * np.sin(m_sun) + 0.019994643 * np.sin(2 * m_sun))

    # Sun to satellite vector
    dist = (1.000140612-0.016708617*np.cos(m_sun)-0.000139589*np.cos(2*m_sun))
    r_earth_sun = dist * np.array([[np.cos(phi_ecliptic)],
                                   [np.cos(epsilon) * np.sin(phi_ecliptic)],
                                   [np.sin(epsilon) * np.sin(phi_ecliptic)]])

    # Sun to satellite vector
    r_sat_sun = r_earth_sun * 149597870.7 - pos

    # Inertial to body transformation
    a_body_eci = q_to_dcm(att)

    # Sun to satellite unit vector
    r_sat_sun_unit = np.matmul(a_body_eci, r_sat_sun / np.linalg.norm(r_sat_sun))

    # Sun to satellite vector in AU
    r_sat_sun_au = r_sat_sun / 149597870.7

    # Solar radiation pressure
    srp = 1361 / (299792458 * np.linalg.norm(r_sat_sun_au) ** 2)

    # Calculating torques on each plate
    l_srp_sum = np.array([[0], [0], [0]])
    for index, plate in enumerate(plates):

        # Angle between plate normal and sun unit vector
        theta_i = np.cos(np.matmul(np.transpose(plate.normal), r_sat_sun_unit))
```

```
        # Reflectivity of plate
        r_plate = (2 * (plate.diffus / 3 + plate.spec * np.cos(theta_i)) * plate.normal +
                   (1 - plate.spec) * r_sat_sun_unit)

        # Force on plate
        f_srp_i = (srp * plate.area * r_plate * max(np.cos(theta_i), 0))

        # Adding torque due to plate to summation
        l_srp_sum = np.add(l_srp_sum, np.cross(plate.pos, f_srp_i, axis=0))

    return l_srp_sum


"""
Operators and useful functions
"""


# Data Loading Update Statements
def percent_complete(iterations, total, update_rate):
    if not iterations % update_rate:
        print('Data generation is '+str(iterations/total*100)[:4]+' % complete')


def lvlh_frame(pos, vel):

    # Creating ECI to LVLH DCM
    z = pos / np.linalg.norm(pos)
    y = np.cross(pos, vel, axis=0) / np.linalg.norm(np.cross(pos, vel, axis=0))
    x = np.cross(y, z, axis=0)

    a_lvlh_eci = np.array([[x[0][0], x[1][0], x[2][0]],
                           [y[0][0], y[1][0], y[2][0]],
                           [z[0][0], z[1][0], z[2][0]]])

    return a_lvlh_eci


def pos_ecef_trans():

    year = 2021
    month = 11
    day = 12 + time // 86400
    h = time // 3600
    m = time // 60
    s = time

    t_0 = (1721013.5 + 367 * year - int((7 / 4) * (year + int((month + 9) / 12)))
           + int(275 * month / 9) + day - 2451545) / 36525

    theta = 24110.54841 + 8640184.812866*t_0+0.093194*t_0**2-6.2*10**-
6*t_0**3+1.002737909350795*(3600*h+60*m+s)

    while theta > 86400:
        theta = theta-86400

    theta = theta/240*np.pi/180

    dcm = np.array([[np.cos(theta), np.sin(theta), 0],
                    [-np.sin(theta), np.cos(theta), 0],
                    [0, 0, 1]])

    return dcm


def get_julian_datetime(date):

    # Perform the calculation
    julian_datetime = 367 * date.year - int((7 * (date.year + int((date.month + 9) / 12.0))) / 4.0) + int(
        (275 * date.month) / 9.0) + date.day + 1721013.5 + (
                          date.hour + date.minute / 60.0 + date.second / math.pow(60,
                                                                                  2)) / 24.0 - 0.5 *
math.copysign(
        1, 100 * date.year + date.month - 190002.5) + 0.5

    return julian_datetime
```

```
def q_dot(w, q):
    q1 = q[0][0]
    q2 = q[1][0]
    q3 = q[2][0]
    q4 = q[3][0]
    return 0.5 * np.matmul(np.array([[q4, -q3, q2],
                                     [q3, q4, -q1],
                                     [-q2, q1, q4],
                                     [-q1, -q2, -q3]]), w)


def q_to_dcm(att):
    q1 = att[0][0]
    q2 = att[1][0]
    q3 = att[2][0]
    q4 = att[3][0]

    return np.array([[q4 ** 2 + q1 ** 2 - q2 ** 2 - q3 ** 2, 2 * (q1 * q2 + q3 * q4), 2 * (q1 * q3 - q2 * q4)],
                     [2 * (q1 * q2 - q3 * q4), q4 ** 2 - q1 ** 2 + q2 ** 2 - q3 ** 2, 2 * (q2 * q3 + q1 * q4)],
                     [2 * (q1 * q3 + q2 * q4), 2 * (q2 * q3 - q1 * q4), q4 ** 2 - q1 ** 2 - q2 ** 2 + q3 **
2]])


def att_err(curr, des):
    des = q_to_dcm(des)
    des = np.array([[np.arctan(des[0][1] / des[0][0])], [-np.arcsin(des[0][2])],
                    [np.arctan(des[1][2] / des[2][2])]])
    des = np.linalg.norm(des)

    curr = q_to_dcm(curr)
    curr = np.array([[np.arctan(curr[0][1] / curr[0][0])], [-np.arcsin(curr[0][2])],
                     [np.arctan(curr[1][2] / curr[2][2])]])
    curr = np.linalg.norm(curr)

    err = abs(curr-des)

    return err


def dcm_to_q(a):    # Computes a quaternion from the attitude matrix
    trace_a = np.trace(a)

    if a[0][0] > a[1][1] and a[0][0] > a[2][2] and a[0][0] > trace_a:
        q1 = np.array([[1 + 2 * a[0][0] - trace_a],
                       [a[0][1] + a[1][0]],
                       [a[0][2] + a[2][0]],
                       [a[1][2] - a[2][1]]])
        q1 = q1 / np.linalg.norm(q1)
        return q1

    elif a[1][1] > a[0][0] and a[1][1] > a[2][2] and a[1][1] > trace_a:
        q2 = np.array([[a[1][0] + a[0][1]],
                       [1 + 2 * a[1][1] - trace_a],
                       [a[1][2] + a[2][1]],
                       [a[2][0] - a[0][2]]])
        q2 = q2 / np.linalg.norm(q2)
        return q2

    elif a[2][2] > a[0][0] and a[2][2] > a[1][1] and a[2][2] > trace_a:
        q3 = np.array([[a[2][0] + a[0][2]],
                       [a[2][1] + a[1][2]],
                       [1 + 2 * a[2][2] - trace_a],
                       [a[0][1] - a[1][0]]])
        q3 = q3 / np.linalg.norm(q3)
        return q3

    else:
        q4 = np.array([[a[1][2] - a[2][1]],
                       [a[2][0] - a[0][2]],
                       [a[0][1] - a[1][0]],
                       [1 + trace_a]])
        q4 = q4 / np.linalg.norm(q4)
        return q4


"""
Defining Main Function
"""
```

```
def main():

    # Iteration Break Time (min)
    break_time = 1500

    # Global variable
    global time
    time = 0

    # Defining Position and Velocity Data File
    rv_file = "rv_vec_HSV.csv"

    # Initiating Satellite
    sat = Satellite(w=np.array([[0], [0], [0]]),
                    att=np.array([[0], [0], [0], [1]]),
                    des_att=np.array([[0], [0], [0], [1]]))

    # Initializing Data Structure
    data = DataStructure()

    # Opening Position and Velocity Data File
    with open(rv_file, 'r') as rv_file:

        # Reading in data
        rv_data = reader(rv_file)

        # Skipping Header
        next(rv_data)

        for minute, line in enumerate(rv_data):

            # Updating Time
            sat.minute = minute

            # Converts Data to Floats and Removes Timestamp
            line = [float(item) for item in line if item is not line[0]]

            # Extracting and organizing data
            x, y, z, vx, vy, vz = line
            pos = np.array([[x], [y], [z]])
            vel = np.array([[vx], [vy], [vz]])

            # Nadir Pointing
            # sat.des_att = dcm_to_q(lvlh_frame(pos, vel))

            # Updating Satellite Position and Velocity
            att, w, cmd_t, rw_s, q_err = sat.propagate(pos, vel)

            # Recording Data to Data Structure
            data.add_data(minute, pos, vel, att, w, cmd_t, rw_s, q_err)

            # Percent Complete
            percent_complete(minute, break_time, 10)

            # FOR DEBUGGING | NEEDS TO BE ULTIMATELY REMOVED
            if minute == break_time:
                print(f'\nData generation stopped at {break_time} minutes\n')
                break

    return data


def post_process(data):

    # Creating x - axis for all plots
    x_vals = data.return_all_pv_time()

    # Returning position data
    rx_vals = [item[0][0] for item in data.return_all_pos()]
    ry_vals = [item[1][0] for item in data.return_all_pos()]
    rz_vals = [item[2][0] for item in data.return_all_pos()]
    plt.plot(x_vals, rx_vals, label='x pos')
    plt.plot(x_vals, ry_vals, label='y pos')
    plt.plot(x_vals, rz_vals, label='z pos')
    plt.title('Satellite Position vs. Time')
    plt.xlabel('Time (sec)')
    plt.ylabel('Position (km)')
    plt.grid()
```

```
    plt.legend()
    plt.show()

    # Returning attitude data
    qx_vals = data.return_all_qw_time()
    q1_vals = [item[0][0] for item in data.return_all_att()]
    q2_vals = [item[1][0] for item in data.return_all_att()]
    q3_vals = [item[2][0] for item in data.return_all_att()]
    q4_vals = [item[3][0] for item in data.return_all_att()]

    # Plotting attitude data
    plt.plot(qx_vals, q1_vals, label='q1')
    plt.plot(qx_vals, q2_vals, label='q2')
    plt.plot(qx_vals, q3_vals, label='q3')
    plt.plot(qx_vals, q4_vals, label='q4')
    plt.title('Body Quaternions vs. Time')
    plt.ylabel('Quaternions')
    plt.xlabel('Time (sec)')
    plt.grid()
    plt.legend()
    plt.show()

    # Returning rate data
    wx_vals = data.return_all_qw_time()
    w1_vals = [item[0][0] for item in data.return_all_w()]
    w2_vals = [item[1][0] for item in data.return_all_w()]
    w3_vals = [item[2][0] for item in data.return_all_w()]

    # Plotting rate data
    plt.plot(wx_vals, w1_vals, label='w1')
    plt.plot(wx_vals, w2_vals, label='w2')
    plt.plot(wx_vals, w3_vals, label='w3')
    plt.title('Body Angular Rates vs. Time')
    plt.ylabel('Angular Rates (rad/s)')
    plt.xlabel('Time (sec)')
    plt.grid()
    plt.legend()
    plt.show()

    # Plotting Commanded Torques
    tx_vals = data.return_all_ct_times()
    t1_vals = [item[0][0] for item in data.return_all_cmd_torqs()]
    t2_vals = [item[1][0] for item in data.return_all_cmd_torqs()]
    t3_vals = [item[2][0] for item in data.return_all_cmd_torqs()]

    # Plotting rate data
    plt.plot(tx_vals, t1_vals, label='t1')
    plt.plot(tx_vals, t2_vals, label='t2')
    plt.plot(tx_vals, t3_vals, label='t3')
    plt.title('Commanded Torques vs. Time')
    plt.ylabel('Torques (N*m)')
    plt.xlabel('Time (sec)')
    plt.grid()
    plt.legend()
    plt.show()

    # Plotting Reaction Wheel Speeds
    rwx_vals = data.return_all_rw_times()
    rw1_vals = [item[0][0] for item in data.return_all_rw_speeds()]
    rw2_vals = [item[1][0] for item in data.return_all_rw_speeds()]
    rw3_vals = [item[2][0] for item in data.return_all_rw_speeds()]

    # Plotting rate data
    plt.plot(rwx_vals, rw1_vals, label='rw1')
    plt.plot(rwx_vals, rw2_vals, label='rw2')
    plt.plot(rwx_vals, rw3_vals, label='rw3')
    plt.title('Reaction Wheel Speeds vs. Time')
    plt.ylabel('RW Rates (rad/s)')
    plt.xlabel('Time (sec)')
    plt.grid()
    plt.legend()
    plt.show()

    # Plotting Attitude Error
    err_x_vals = data.return_all_att_times()
    err_1_vals = data.return_all_att_err()

    # Plotting error data
    plt.plot(err_x_vals, err_1_vals, label='err')
```

```
    plt.title('Attitude Error vs. Time')
    plt.ylabel('Attitude Error (rad)')
    plt.xlabel('Time (sec)')
    plt.grid()
    plt.legend()
    plt.show()


if __name__ == '__main__':

    # Data Generation
    start = perf_counter()
    sim = main()
    end = perf_counter()
    print(f'\nTime to generate data: {end-start} seconds')

    # Post Processing
    post_process(sim)
```