ASE 372K Final Project

Gavin Martin

The University of Texas at Austin

December 6th 2018

Contents

List of Figures

List of Tables

1 Introduction

This paper details the creation of modeling and simulation software used to characterize the behavior of attitude determination and control systems (ADCS) on spacecraft, and it directly simulates and discusses a reference mission to demonstrate the viability of a specific suite of sensor, actuator, and controller designs. It breaks down the behavior of the simulation engine into the system's constituent components and explains the mathematical models used to represent them programmatically.

The simulation uses a numeric integrator that propagates the spacecraft's attitude ($\boldsymbol{q}_i^b$), its angular velocity ($\boldsymbol{\omega}_{b/i}^b$), and the angular velocity of its reaction wheels ($\boldsymbol{\omega}_{rxwl}$) over time. At each timestep: (1) imperfect sensors collect estimated angular velocity and direction measurements; (2) the TRIAD algorithm is used to determine the estimated attitude; (3) attitude and attitude rate errors between the desired and estimated states are calculated; (4) a PD controller uses those errors to calculate necessary control torques to stabilize the system; (5) reaction wheels carry out the commanded control torques by accelerating, but with small actuation errors; and (6) the actual torque applied by the actuators is computed and summed with the perturbing torques before being fed into the dynamics equation that drives the integrator. The controller can be optionally turned off to view the evolution of attitude in the absence of actuators, and measurement/actuation noise can be optionally excluded to model perfect estimation and control.

The software developed to carry out the simulation is written in Python and heavily utilizes the SciPy ecosystem. The code created for this project along with instructions for running simulations is available at `https://github.com/gavincmartin/adcs-simulation`.

2 Software Overview

The two key software components of this project are (1) the simulation engine that will execute a simulation given any appropriate set of system parameters and (2) the control script that describes system components and behavior and plots the appropriate results. This structure was built for maximum modularity and reusability and is explained in more detail below.

2.1 Simulation Engine

The core of the simulation engine is found in `simulation.py`. This is where the numerical integration takes place and where sensor measurement, attitude determination, control, and actuation all reside. At each timestep, the following sequence occurs:

1. The time derivative of the quaternion is computed.
2. Sensors on-board the spacecraft collect direction and angular velocity estimates (imperfectly, if sensors are modeled with noise).
3. An estimated attitude is determined by using the TRIAD algorithm with direction measurements.
4. The error and error rate between the estimated state and the desired state are computed.
5. Errors are used by the controller to compute the appropriate control torque.
6. Actuators apply the control torque on the system (imperfectly, if actuators are modeled with noise).
7. The torque applied by actuators is summed with perturbing torques and used to compute the angular acceleration of the body.
8. The quaternion derivative, angular acceleration of the body, and angular acceleration of the reaction wheels (actuators) are fed back into the integrator.

The ODEPACK LSODA integrator implementation in SciPy [1] is used for numerical integration, and a derivatives function (defined in `simulation.py`) carries out the sequence above at each adaptive step. To simplify the derivatives function, a `Spacecraft` class (defined in `spacecraft.py`) was written to house relevant system information:

- Inertia tensor
- A `PDController` object
- A `Gyros` object
- A `Magnetometer` object
- An `EarthHorizonSensor` object
- An `Actuators` object
- Current inertial position
- Current inertial velocity
- Current attitude
- Current angular velocity

The objects that compose the `Spacecraft` class are themselves class abstractions of the relevant spacecraft systems (e.g., the `PDController` class stores the K_p and K_d controller gains and contains a `calculate_control_torques` method). The `Spacecraft` class has many aliases for these methods to simplify the passing of certain variables and improve readability. The behaviors of these subsystems are explained mathematically in the sections to follow.

2.2 Control Script

The simulation engine above relies upon a separate control script that defines system parameters, defines the simulation parameters, and chooses how to display the results. The control script must define:

- A `Spacecraft` object (complete with the composed objects described above and the initial conditions)
- A function describing the nominal attitude and angular velocity of the spacecraft over time
- A function describing the perturbations to be modeled over time
- A function describing the position and velocity of the spacecraft over time
- The start and stop times for the simulation and the user-defined Δt at which results are logged

All of the parameters above are fed as inputs to the simulation engine and are used in the computations at each integrator step.

The control script that specifically defines this final project and reference mission is `project.py`. This script defines the above parameters for three separate scenarios and runs simulations for each of them: (1) the spacecraft with no control, (2) the controlled spacecraft with perfect sensors and actuators, and (3) the controlled spacecraft with imperfect sensors and actuators. The results of all three simulations are separately plotted for clarity on how the addition or modeling of certain components influences overall system behavior.

3 Reference Mission

3.1 Spacecraft

A 6U CubeSat in a circular low Earth orbit was selected for the reference mission.

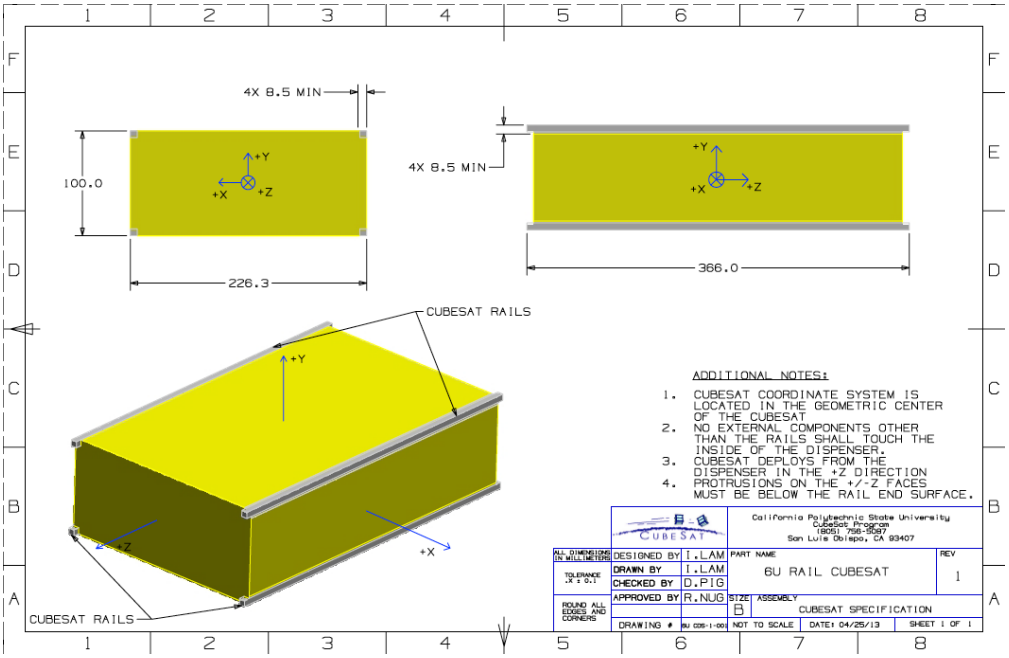


Figure 1: 6U CubeSat Specification [2]

Given the formula for the inertia tensor of a cuboid,

$$\boldsymbol{J}_{cuboid} = \begin{bmatrix} \frac{1}{12}m(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+d^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2+h^2) \end{bmatrix} \tag{1}$$

and an estimated 8 kg of mass (1.33 kg per U), the inertia tensor for the satellite using the body axes shown in Figure 1 (without modeling the rails) is:

$$\boldsymbol{J}^b_{cg} = \begin{bmatrix} 0.09597067 & 0 & 0 \\ 0 & 0.12344513 & 0 \\ 0 & 0 & 0.04080779 \end{bmatrix} kg \cdot m^2 \tag{2}$$

A magnetic dipole model is used to approximate the magnetic field of the spacecraft, and it is assumed that the spacecraft's magnetic field vector acts along the

body-y axis. An estimate for the residual magnetic dipole of a 3U CubeSat collected at the magnetic test facility at NASA Goddard Spaceflight Center is $9 \times 10^{-3} A \cdot m^2$ [3]. This value is doubled to account for twice the volume of electronics present in a 6U CubeSat and is thus modeled as:

$$\boldsymbol{m}^b = \begin{bmatrix} 0 \\ 0.018 \\ 0 \end{bmatrix} A \cdot m^2 \tag{3}$$

3.2 Orbit

The spacecraft orbits the Earth at an altitude of 400 km with an inclination of 45°. With a gravitational parameter (μ) of 398600.4415 $\frac{km^3}{s^2}$, an orbit radius (ρ) of 6,778.1 km, and an orbital angular velocity (ω) of 0.00113138 $\frac{rad}{s}$ ($\omega = \sqrt{\mu/\rho^3}$), the spacecraft's inertial position ($\boldsymbol{r}^i(t)$) and velocity ($\dot{\boldsymbol{r}}^i(t)$) are described by Equations 4 & 5.

$$\boldsymbol{r}^i(t) = \frac{\rho}{\sqrt{2}} \begin{bmatrix} -\cos \omega t \\ \sqrt{2} \sin \omega t \\ \cos \omega t \end{bmatrix} \tag{4}$$

$$\dot{\boldsymbol{r}}^i(t) = \frac{\omega \rho}{\sqrt{2}} \begin{bmatrix} \sin \omega t \\ \sqrt{2} \cos \omega t \\ -\sin \omega t \end{bmatrix} \tag{5}$$

3.3 Attitude

The reference attitude of the spacecraft is such that its body-x axis is pointing towards the center of the Earth and that its body-y axis is pointing in the direction of its velocity vector. Equations 6 - 8 detail what this means mathematically.

$$\boldsymbol{b}_x^i(t) = -\frac{\boldsymbol{r}^i(t)}{\|\boldsymbol{r}^i(t)\|} \tag{6}$$

$$\boldsymbol{b}_y^i(t) = \frac{\dot{\boldsymbol{r}}^i(t)}{\|\dot{\boldsymbol{r}}^i(t)\|} \tag{7}$$

$$\boldsymbol{b}_z^i(t) = \boldsymbol{b}_x^i(t) \times \boldsymbol{b}_y^i(t) \tag{8}$$

From there, the desired initial attitude can be computed as the Direction Cosine Matrix (DCM) from the inertial to body frame.

$$\bar{\boldsymbol{T}}^b_{i,0} = \begin{bmatrix} (\boldsymbol{b}^i_{x,0})^T \\ (\boldsymbol{b}^i_{y,0})^T \\ (\boldsymbol{b}^i_{z,0})^T \end{bmatrix} \tag{9}$$

The angular velocity required to carry out this reference attitude is simply the negative of the orbital angular velocity in the body-z direction.

$$\bar{\boldsymbol{\omega}}^b_{b/i}(t) = \begin{bmatrix} 0 \\ 0 \\ -\omega \end{bmatrix} \tag{10}$$

With this, it is possible to construct the nominal attitude as a function of time.

$$\bar{\boldsymbol{T}}^b_i = \boldsymbol{T}_3(-\omega t)\bar{\boldsymbol{T}}^b_{i,0} \tag{11}$$

3.4 Initial Conditions

To demonstrate the viability of the control system, the spacecraft attitude and angular velocity are slightly off-nominal at the beginning of the simulation: a pitch error of 2° (about the body y-axis) and an angular velocity error of 0.005 rad/s (about the body x-axis).

$$\boldsymbol{q}^b_{i_0} = \begin{bmatrix} 0 \\ \sin\frac{2^\circ}{2} \\ 0 \\ \cos\frac{2^\circ}{2} \end{bmatrix} \otimes \bar{\boldsymbol{q}}^b_{i_0} \tag{12}$$

$$\boldsymbol{\omega}^b_{b/i_0} = \bar{\boldsymbol{\omega}}^b_{b/i_0} \begin{bmatrix} 0.005 \\ 0 \\ 0 \end{bmatrix} \tag{13}$$

4 Dynamics Model

This section details how the evolution of spacecraft attitude and angular velocity are modeled over time. The functions written to model system kinematics, dynamics, and perturbations are located in `kinematics.py`, `dynamics.py`, and `perturbations.py`, respectively.

4.1 Kinematics

The chosen attitude parameterization for purposes of kinematics integration is the quaternion. The quaternion is a singularity-free representation of attitude, and is a 4x1 vector of unit norm constructed from an Euler axis $\boldsymbol{e}$ and angle θ.

$$\boldsymbol{q} = \begin{bmatrix} \boldsymbol{q}_v \\ q_s \end{bmatrix} = \begin{bmatrix} \sin\left(\frac{\theta}{2}\right)\boldsymbol{e} \\ \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \tag{14}$$

The quaternion describing the rotation from the inertial frame to spacecraft body frame evolves over time as a function of itself, $\boldsymbol{q}_i^b$, and the angular velocity of the spacecraft body frame with respect to the inertial frame (in body coordinates), $\boldsymbol{\omega}_{b/i}^b$.

$$\dot{\boldsymbol{q}}_i^b = \frac{1}{2} \begin{bmatrix} \boldsymbol{\omega}_{b/i}^b \\ 0 \end{bmatrix} \otimes \boldsymbol{q}_i^b \tag{15}$$

This kinematic equation governs the torque-free motion of the spacecraft. The influence of external torques on the body is discussed in Section 4.2.

4.2 Dynamics

The dynamic evolution of the spacecraft's attitude over time is a function of its inertia tensor, $\boldsymbol{J}_{cg}^b$, its angular velocity, $\boldsymbol{\omega}_{b/i}^b$, and the sum of external torques on the body, $\boldsymbol{M}_{cg}^b$.

$${}^i\dot{\boldsymbol{\omega}}_{b/i}^b(t) = -(\boldsymbol{J}_{cg}^b)^{-1}\boldsymbol{\omega}_{b/i}^b(t) \times (\boldsymbol{J}_{cg}^b\boldsymbol{\omega}_{b/i}^b(t)) + (\boldsymbol{J}_{cg}^b)^{-1}\boldsymbol{M}_{cg}^b(t) \tag{16}$$

For the simulation carried out in this paper, the external torques modeled are the control torques applied by actuators on the system (as discussed in Section 5), $\boldsymbol{M}_{cg,applied}$, and the sum of the perturbing torques (as discussed in Section 4.3), $\boldsymbol{M}_{cg,pert}$.

$$\boldsymbol{M}_{cg}^b(t) = \boldsymbol{M}_{cg,applied}^b(t) + \boldsymbol{M}_{cg,pert}^b(t) \tag{17}$$

Both Equations 15 & 16 are used in the numeric integrator to propagate the attitude forward in time, while other intermediate equations and math models are used to determine the values of perturbing and control torques at each discrete timestep in the simulation.

4.3 Perturbations

The simulation includes models for both gravity gradient and magnetic field torques that perturb the system over time. These are implemented in `perturbations.py` and are mathematically modeled as follows:

$$\boldsymbol{d}^b = \frac{\boldsymbol{r}^i}{\|\boldsymbol{r}^i\|} \tag{18}$$

$$\omega_{orb} = \frac{\mu}{r^3} \tag{19}$$

$$\boldsymbol{M}^b_{gg}(t) = 3\omega^2_{orb}\boldsymbol{d}^b \times \boldsymbol{J}^b\boldsymbol{d}^b \tag{20}$$

$$\boldsymbol{M}^b_B(t) = \boldsymbol{m}^b \times \boldsymbol{T}^b_i\boldsymbol{B}^i \tag{21}$$

where $\boldsymbol{m}^b$ is the spacecraft magnetic dipole vector from Equation 3 in Section 3.1 and $\boldsymbol{B}^i$ is the magnetic field vector produced by the Earth from Equation 33. A full treatment of how the Earth's magnetic field is modeled and how this value is computed can be found in Section 5.3.

These perturbing torques are summed together as in Equation 22 and are fed into the dynamics model described by Equations 17 & 16.

$$\boldsymbol{M}^b_{cg,pert}(t) = \boldsymbol{M}^b_{gg}(t) + \boldsymbol{M}^b_B(t) \tag{22}$$

4.4 Uncontrolled Simulation Results

For the uncontrolled simulation, the attitude of the spacecraft evolves (Figure 2) simply as a function of the perturbing torques acting on it (Figure 5).

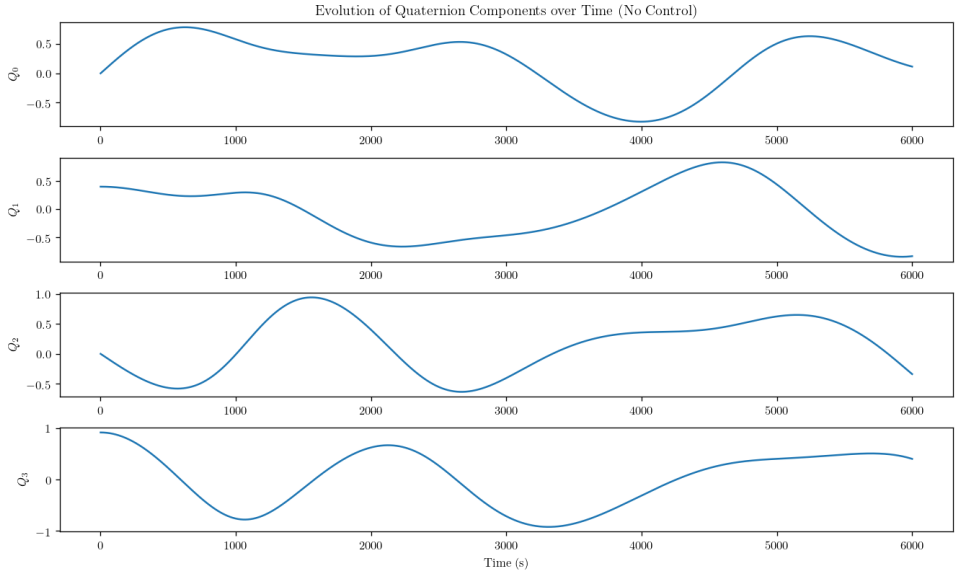


Figure 2: Quaternion evolution in uncontrolled simulation

Figure 3 also clearly shows that the angular velocity of the spacecraft does not converge towards the desired state in any way.

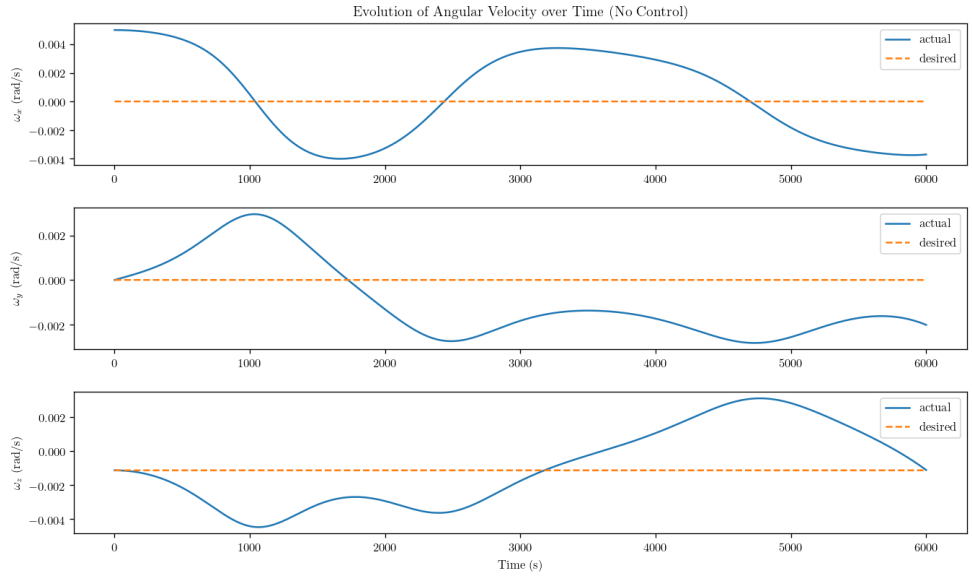


Figure 3: Angular velocity evolution in uncontrolled simulation

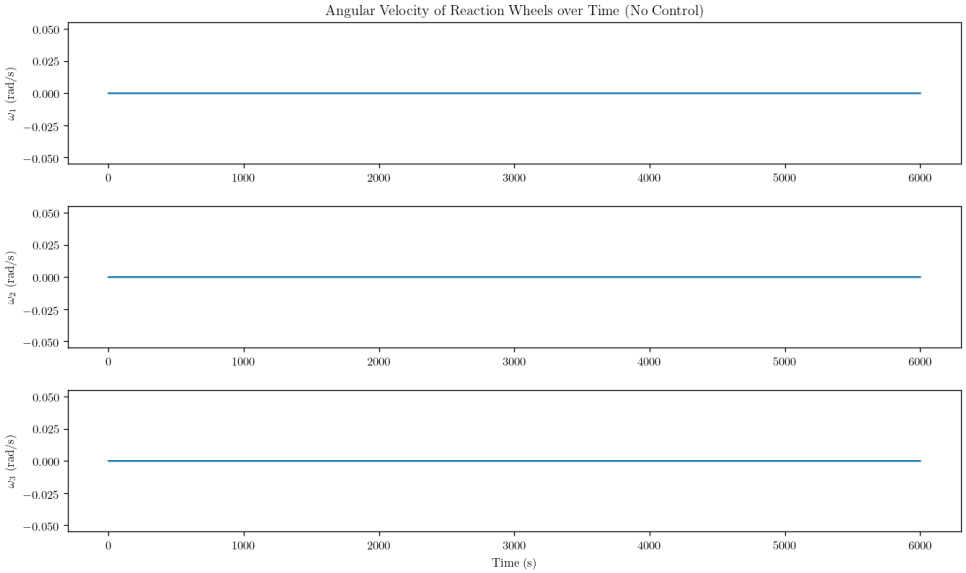


Figure 4: Angular velocity of reaction wheels in uncontrolled simulation

Since the controller is turned off for this simulation, the reaction wheels are never accelerated and remain at zero (as seen in Figure 4).

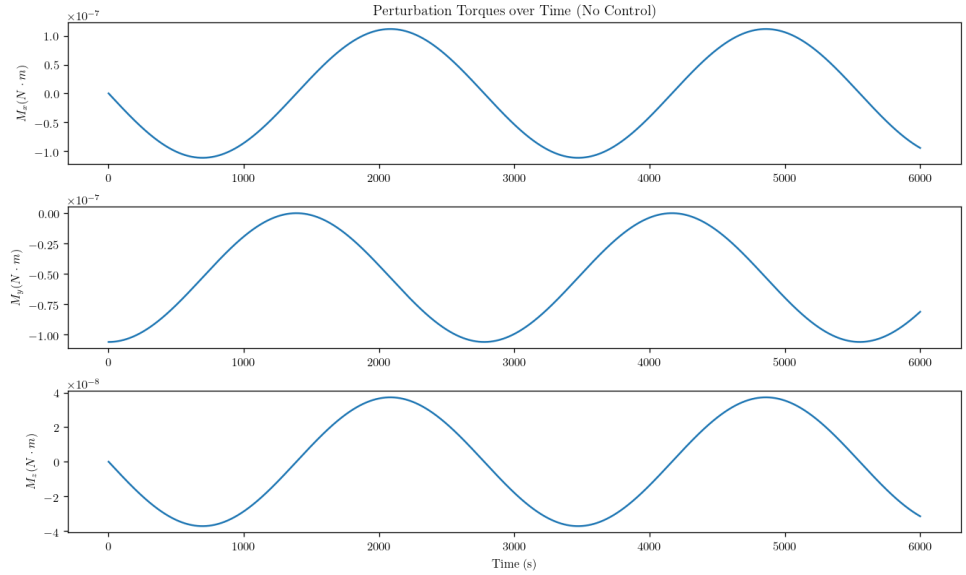


Figure 5: Perturbing torques in uncontrolled simulation

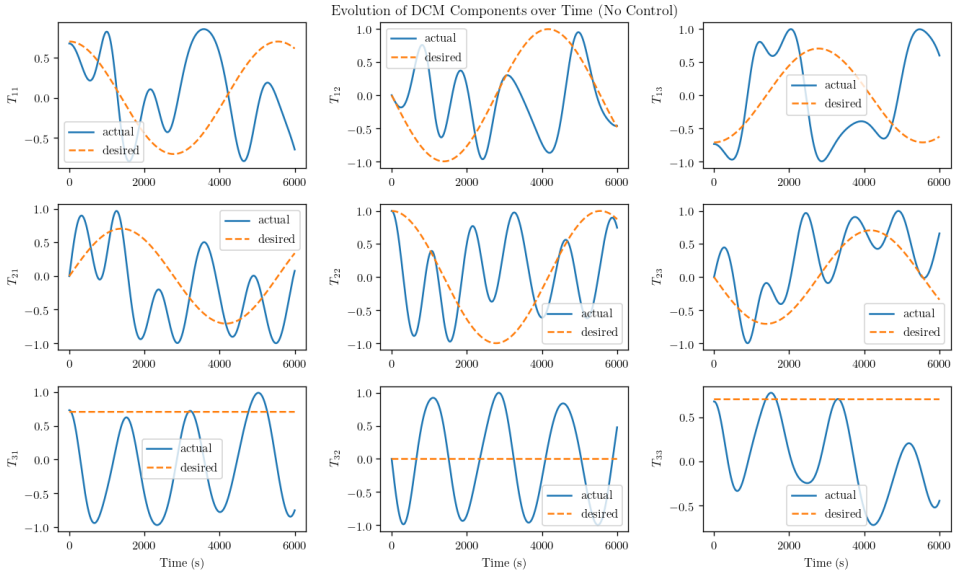


Figure 6: DCM evolution in uncontrolled simulation

Like with angular velocity, it is clear that the attitude (in DCM form in Figure 6) does not approach the nominal state in any way in the absence of control. It is in the presence of a controller (Section 6.2) that attitude values can actually be seen converging towards a desired state.

5 Sensors and Actuators

This section details how the sensors and actuators on-board the spacecraft are modeled, focusing specifically on how measurement estimates are generated and how noise is appropriately applied to the system. The functions written to model sensor and actuator behavior are located in `sensors.py`, `sensor_models.py`, and `actuators.py`.

5.1 Gyroscopes

The Northrop Grumman LN-200S Inertial Measurement Unit (containing 3 gyroscopes and 3 accelerometers) [4] was selected for the simulation in order to estimate the angular velocity of the spacecraft at a given time.

LN-200S Gyros	
Mass (for the full IMU)	748 g
Bias Repeatability (BR)	$1^\circ/hr$
Angular Random Walk (ARW)	$0.07^\circ/\sqrt{hr}$

Table 1: Specifications for the LN-200S IMU gyroscopes

Since the true angular velocity is known, it is necessary to add bias and noise to the measurements to simulate the imperfect estimation of an actual sensor. Equation 23 shows how this is modeled in the simulation.

$$\tilde{\boldsymbol{\omega}}^b_{b/i}(t) = \boldsymbol{\omega}^b_{b/i}(t) + \boldsymbol{b} + \boldsymbol{\nu}_{gyro}(t) \tag{23}$$

The gyro bias, $\boldsymbol{b}$, is randomly drawn from a Gaussian distribution centered at 0 with a σ equal to the bias repeatability (in rad/s). This is repeated for each of the three gyros, and that bias is consistent throughout the duration of the simulation.

The gyro noise, $\boldsymbol{\nu}_{gyro}(t)$, is defined by a Gaussian distribution centered at 0 with a standard deviation described by Equation 24.

$$\sigma = \frac{ARW}{\sqrt{\Delta t}} \tag{24}$$

Noise is randomly drawn from this distribution for each of the three gyros at each Δt (chosen to be 1 second) and is summed with the bias and actual angular velocity to produce an estimate as described in Equation 23.

5.2 Earth Horizon Sensor

The MAI-SES Static Earth Sensor produced by Adcole Maryland Aerospace [5] was selected for the simulation in order to provide a direction vector (to Earth) that could be used for attitude determination at each timestep.

MAI-SES Static Earth Sensor	
Mass	33 g
Accuracy ($\theta_{err,tot}$)	0.25°

Table 2: Specifications for the MAI-SES Static Earth Sensor

The math model describing the inertial direction of Earth, $\boldsymbol{d}_{\oplus}^{i}$ is simply a function of the spacecraft's inertial position (as shown in Equation 25).

$$\boldsymbol{d}_{\oplus}^{i} = -\frac{\boldsymbol{r}^{i}}{\|\boldsymbol{r}\|} \tag{25}$$

Using an approximation of how the angular error in each direction contributes to the accuracy (as described in Equation 26), Equation 27 shows how the overall accuracy of the horizon sensor can be decomposed into its error in the x, y, and z directions.

$$\theta_{err,tot} = \sqrt{(\theta_{err,x})^2 + (\theta_{err,y})^2 + (\theta_{err,z})^2} \tag{26}$$

$$\theta_{err,x} = \sqrt{\frac{(\theta_{err,tot})^2}{3}} \tag{27}$$

At each Δt, angular errors θ_1, θ_2, and θ_3 are each drawn from a Gaussian distribution centered at 0 with a σ equal to $\theta_{err,x}$. These errors are used to construct a direction cosine matrix (DCM) to represent measurement noise using Equations 28, 29, & 30.

$$\theta = \sqrt{(\theta_1)^2 + (\theta_2)^2 + (\theta_3)^2} \tag{28}$$

$$\boldsymbol{e} = \frac{1}{\theta}\begin{bmatrix}\theta_1 \\ \theta_2 \\ \theta_3\end{bmatrix} \tag{29}$$

$$\delta\boldsymbol{T}(\theta_1, \theta_2, \theta_3) = \boldsymbol{I} - \sin\theta[\boldsymbol{e}\times] + (1 - \cos\theta)[\boldsymbol{e}\times]^2 \tag{30}$$

The inertial direction to Earth, $\boldsymbol{d}^i_\oplus$, can be combined with the known attitude (represented as the DCM from the inertial to body frame, $\boldsymbol{T}^b_i$) and the noisy DCM produced in Equation 30, $\delta\boldsymbol{T}(\theta_1, \theta_2, \theta_3)$, to estimate the direction to Earth as measured in body coordinates by the horizon sensor.

$$\tilde{\boldsymbol{d}}^b(t) = \delta\boldsymbol{T}(\theta_1, \theta_2, \theta_3)\boldsymbol{T}^b_i(t)\boldsymbol{d}^i_\oplus \tag{31}$$

5.3 Magnetometer

The DS Magnetometer 6A produced by NewSpace Systems [6] was selected for the simulation in order to provide a direction vector that could be used for attitude determination purposes.

DS Magnetometer 6A	
Mass	85 g
Resolution ($\boldsymbol{\nu}_{mag}$)	10 nT

Table 3: Specifications for the NewSpace Systems DS Magnetometer

A dipole model is used to compute the local magnetic field vector, $\boldsymbol{B}^i$ at each timestep

$$\boldsymbol{B}^n = B_0(\frac{R_\oplus}{\|\boldsymbol{r}^i\|})^3 \begin{bmatrix} \cos\lambda \\ 0 \\ -2\sin\lambda \end{bmatrix} \tag{32}$$

$$\boldsymbol{B}^i = (\boldsymbol{T}^n_i)^T\boldsymbol{B}^n \tag{33}$$

where B_0 is the mean value of the magnetic field at the equator (3.12×10^{-5} Tesla), $R_\oplus$ is the equatorial radius of the Earth (6378.1 km), λ is the latitude of the spacecraft, and $\boldsymbol{T}^n_i$ is the DCM from the inertial to the North-East-Down (NED) frame. Equations 34 - 38 show the intermediate calculations necessary to compute the local magnetic field vector $\boldsymbol{B}^i$.

$$\lambda = \arcsin\frac{r^i_z}{\|\boldsymbol{r}^i\|} \tag{34}$$

$$\boldsymbol{n}^i_z = -\frac{\boldsymbol{r}^i}{\|\boldsymbol{r}^i\|} \tag{35}$$

$$\boldsymbol{n}_y^i = -\frac{\boldsymbol{n}_z^i \times \boldsymbol{i}_z^i}{\|\boldsymbol{n}_z^i \times \boldsymbol{i}_z^i\|} \tag{36}$$

$$\boldsymbol{n}_x^i = \boldsymbol{n}_y^i \times \boldsymbol{n}_z^i \tag{37}$$

$$\boldsymbol{T}_i^n = \begin{bmatrix} (\boldsymbol{n}_x^i)^T \\ (\boldsymbol{n}_y^i)^T \\ (\boldsymbol{n}_z^i)^T \end{bmatrix} \tag{38}$$

The model employed by the simulation for the actual direction vector measured by the sensor is described by Equation 39.

$$\tilde{\boldsymbol{B}}^b(t) = \boldsymbol{T}_i^b(t)\boldsymbol{B}^i(t) + \boldsymbol{\nu}_{mag}(t) \tag{39}$$

The measurement noise, $\boldsymbol{\nu}_{mag}(t)$, is randomly sampled in each direction from a Gaussian distribution with a mean of 0 and a σ equal to the magnetometer resolution at each Δt. This noise is summed with the transformed local magnetic field vector to produce an estimate in body coordinates.

5.4 Reaction Wheels

Three 60 mNms Microsatellite Wheels produced by Sinclair Interplanetary [7] were selected as actuators aligned to spin about the spacecraft's principal axes, applying torques to the body as commanded by the controller.

60 mNms Microsatellite Wheel	
Mass	226 g
Dimensions	77 mm x 65 mm x 38 mm
Peak Momentum	0.18 Nms
Max Torque	± 20 mNm

Table 4: Specifications for the Sinclair Interplanetary 60 mNms Microsatellite Wheel

Given the mass and dimensions in Table 4 (using 65 mm as the reaction wheel radius), the moment of inertia of each wheel about its spin axis is C_w.

$$C_w = \frac{1}{2} m_w r_w^2 = 4.77425 \times 10^{-4} kg \cdot m^2 \tag{40}$$

The angular velocities of the three wheels (ω_1, ω_2, ω_3) are numerically integrated throughout the simulation as the wheels accelerate to actuate the commanded control torques. The desired angular accelerations of the wheels ($\dot{\bar{\omega}}_1$, $\dot{\bar{\omega}}_2$, $\dot{\bar{\omega}}_3$) at each time step can be computed as a function of the angular velocity of the spacecraft ($\boldsymbol{\omega}^b_{b/i}$), the current reaction wheel angular velocities, their moments of inertia, and the torque that the controller wants to produce ($\boldsymbol{M}^b_{cg,ctrl}$).

$$\begin{bmatrix} \dot{\bar{\omega}}_1 \\ \dot{\bar{\omega}}_2 \\ \dot{\bar{\omega}}_3 \end{bmatrix} = -\boldsymbol{\omega}^b_{b/i}(t) \times \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} - \frac{1}{C_w} \boldsymbol{M}^b_{cg,ctrl}(t) \tag{41}$$

There are two limitations to this: (1) reaction wheels cannot produce infinite torque, and (2) reaction wheels become saturated above a certain angular momentum and cannot be further accelerated. This simulation takes both of these into account. To address (1): if the control torque exceeds the maximum torque that the wheels are capable of generating (as shown in Table 4), then the control torque in each direction will be lowered to the maximum torque possible, and the angular accelerations are calculated from this reduced torque. To address (2): if the angular momentum of a given wheel is saturated above the peak momentum (also shown in Table 4) then further angular acceleration will be set to zero, and no control torque will be applied by that wheel.

Additionally, the reaction wheels are imperfect actuators and do not accelerate exactly to the desired values. The math model describing this imperfect actuation in each wheel follows in Equation 42.

$$\dot{\hat{\omega}} = \dot{\bar{\omega}} + \nu_w \tag{42}$$

The noise for each wheel (ν_w) is drawn from a Gaussian distribution centered at the commanded angular acceleration ($\dot{\bar{\omega}}$) with a σ equal to 3% of $\dot{\bar{\omega}}$. The noise is summed with the commanded value to produce the actual angular acceleration ($\dot{\hat{\omega}}$). This means that the angular acceleration in each wheel deviates slightly from the commanded values (usually by a few per cent), and the resulting torque produced is not exactly $\boldsymbol{M}^b_{cg,ctrl}$ but is in fact $\boldsymbol{M}^b_{cg,applied}$. That applied torque can be computed from Equation 43.

$$\boldsymbol{M}^b_{cg,applied}(t) = -C_w \begin{bmatrix} \dot{\hat{\omega}}_1 \\ \dot{\hat{\omega}}_2 \\ \dot{\hat{\omega}}_3 \end{bmatrix} - \boldsymbol{\omega}^b_{b/i}(t) \times C_w \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \tag{43}$$

It is this applied torque that is used in the dynamics models for numerical integration as described in Section 4.2.

6 Attitude Determination and Control System

This section details how the attitude is estimated at each timestep (from sensor measurements explained in Section 5) and how the PD controller calculates errors and control torques to apply to the system. The TRIAD algorithm function is in `math_utils.py`, while the class and functions written to model error computation and controller behavior are located in `errors.py` and `controller.py`.

6.1 Attitude Determination (TRIAD)

The TRIAD algorithm is used to estimate the attitude of the spacecraft at each timestep. It requires two direction vectors measured by sensors in body coordinates ($\tilde{\boldsymbol{d}}_1^b$ and $\tilde{\boldsymbol{d}}_2^b$) and the equivalent vectors ($\boldsymbol{d}_1^i$ and $\boldsymbol{d}_2^i$) produced by math models in inertial coordinates. The algorithm for producing an attitude estimate ($\hat{\boldsymbol{T}}_i^b$) follows in Equations 44 - 50.

$$\boldsymbol{x}^b = \tilde{\boldsymbol{d}}_1^b \tag{44}$$

$$\boldsymbol{z}^b = \frac{\tilde{\boldsymbol{d}}_1^b \times \tilde{\boldsymbol{d}}_2^b}{\|\tilde{\boldsymbol{d}}_1^b \times \tilde{\boldsymbol{d}}_2^b\|} \tag{45}$$

$$\boldsymbol{y}^b = \boldsymbol{z}^b \times \boldsymbol{x}^b \tag{46}$$

$$\boldsymbol{x}^i = \boldsymbol{d}_1^i \tag{47}$$

$$\boldsymbol{z}^i = \frac{\boldsymbol{d}_1^i \times \boldsymbol{d}_2^i}{\|\boldsymbol{d}_1^i \times \boldsymbol{d}_2^i\|} \tag{48}$$

$$\boldsymbol{y}^i = \boldsymbol{z}^i \times \boldsymbol{x}^i \tag{49}$$

$$\hat{\boldsymbol{T}}_i^b = \begin{bmatrix} \boldsymbol{x}^b & \boldsymbol{y}^b & \boldsymbol{z}^b \end{bmatrix} \begin{bmatrix} \boldsymbol{x}^i & \boldsymbol{y}^i & \boldsymbol{z}^i \end{bmatrix}^T \tag{50}$$

The direction vectors used for TRIAD by the simulation carried out in this paper are produced by the Earth horizon sensor and magnetometer measurements as explained in Sections 5.2 & 5.3. In this case, $\tilde{\boldsymbol{d}}_1^b$ is produced by the horizon sensor as described in Equation 31, $\tilde{\boldsymbol{d}}_2^b$ is produced by the magnetometer as described in Equation 39, and their inertial counterparts are produced by Equations 25 & 33, respectively.

6.2 PD Control

A proportional-derivative (PD) controller is used to control the attitude of the spacecraft throughout the duration of the simulation by applying torques to the spacecraft (via the reaction wheels) that minimize the difference between the nominal and estimated states. The proportional and derivative gains used for the controller in this simulation are found in 51 & 52.

$$\boldsymbol{K}_p = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} \tag{51}$$

$$\boldsymbol{K}_d = \begin{bmatrix} 0.01 & 0 & 0 \\ 0 & 0.01 & 0 \\ 0 & 0 & 0.01 \end{bmatrix} \tag{52}$$

At each timestep, attitude and attitude rate errors between the nominal states ($\bar{\boldsymbol{T}}_i^b$ and $\bar{\boldsymbol{\omega}}_{b/i}^b$) and estimated states ($\hat{\boldsymbol{T}}_i^b$ and $\hat{\boldsymbol{\omega}}_{b/i}^b$) are computed using Equations 53 - 56.

$$\delta\boldsymbol{T}(t) = \hat{\boldsymbol{T}}_i^b(t)\bar{\boldsymbol{T}}_i^b(t)^T \tag{53}$$

$$\boldsymbol{\epsilon}(t) \simeq -0.5 \begin{bmatrix} \delta T_{32} - \delta T_{23} \\ \delta T_{13} - \delta T_{31} \\ \delta T_{21} - \delta T_{12} \end{bmatrix} \tag{54}$$

$$\delta\boldsymbol{\omega}^b(t) = \hat{\boldsymbol{\omega}}_{b/i}^b(t) - \bar{\boldsymbol{\omega}}_{b/i}^b(t) \tag{55}$$

$$\dot{\boldsymbol{\epsilon}}(t) = -\hat{\boldsymbol{\omega}}_{b/i}^b(t) \times \boldsymbol{\epsilon}(t) + \delta\boldsymbol{\omega}^b(t) \tag{56}$$

Given these errors, the controller will calculate the necessary control torques that should be applied to the spacecraft, $\boldsymbol{M}_{cg,ctrl}^b(t)$, in order to maintain the nominal attitude.

$$\boldsymbol{u}(t) = -\boldsymbol{K}_d\dot{\boldsymbol{\epsilon}}(t) - \boldsymbol{K}_p\boldsymbol{\epsilon}(t) \tag{57}$$

$$\boldsymbol{M}_{cg,ctrl}^b(t) = \boldsymbol{J}_{cg}^b\boldsymbol{u}(t) \tag{58}$$

This control torque is used to command the reaction wheels to accelerate or decelerate to produce the desired value, but small actuation errors (explained in Section 5.4) instead produce the torque $\boldsymbol{M}^b_{cg,applied}$, a slight deviation from $\boldsymbol{M}^b_{cg,ctrl}$. This is the external torque applied on the spacecraft.

7 Simulation Results

7.1 Perfect Estimation & Control

Using the same parameters as in Section 4.4 but with a PD controller turned on, there is an expected, yet dramatic difference in the behavior of the spacecraft. Figures 7, 8, & 11 clearly show the convergence of attitude and angular velocity towards the nominal values described in Section 3.3. This is confirmed by the acceleration of the reaction wheels (Figure 9).

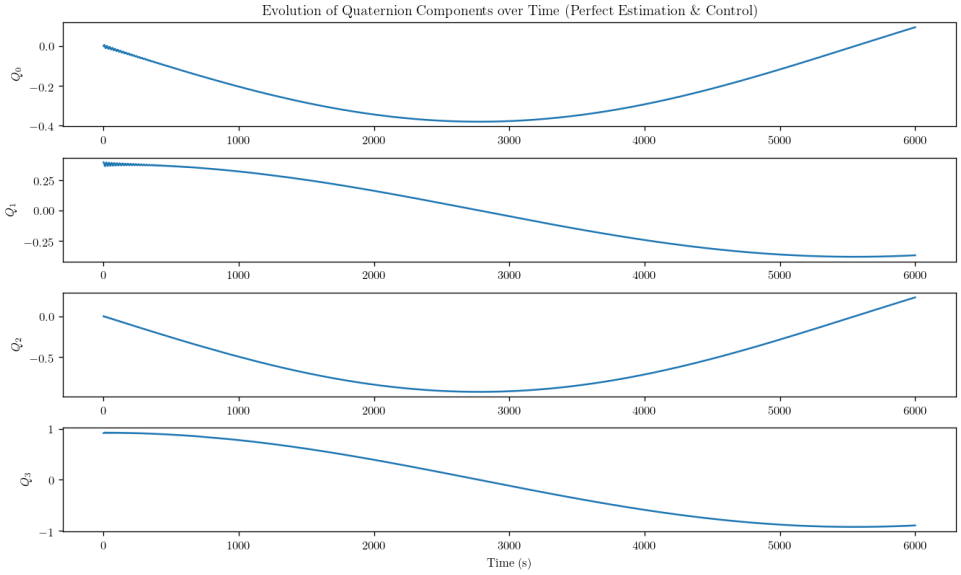


Figure 7: Quaternion evolution in perfect sensor/actuator simulation

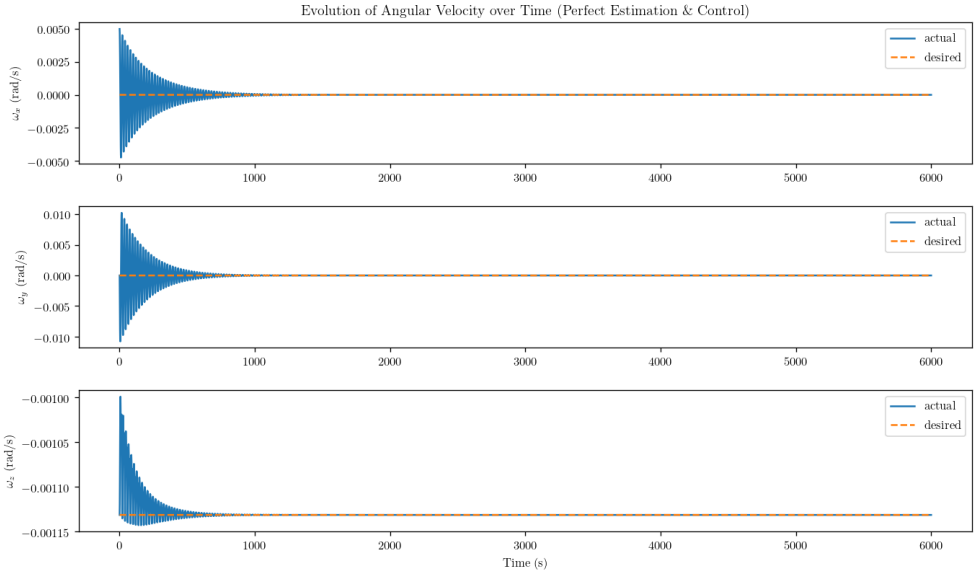


Figure 8: Angular velocity evolution in perfect sensor/actuator simulation

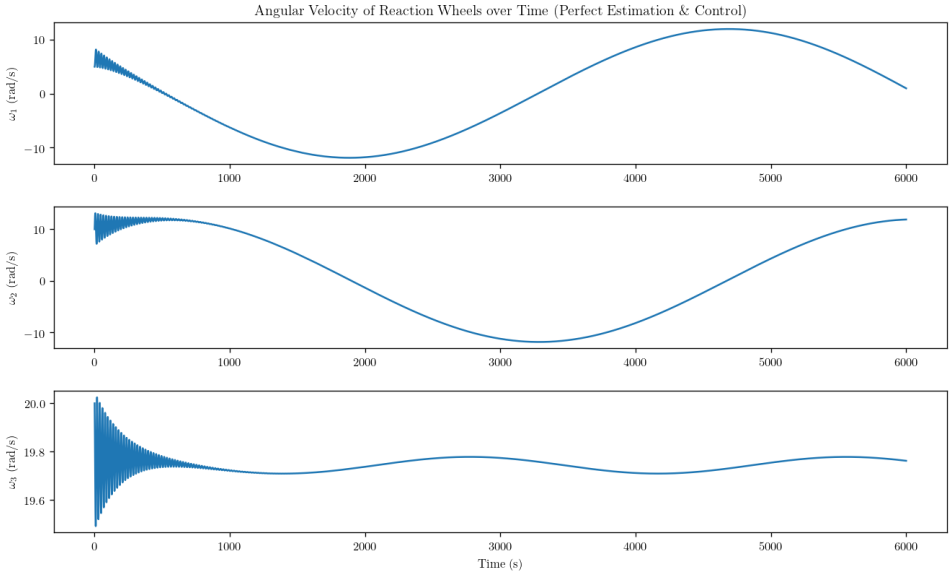


Figure 9: Angular velocity of reaction wheels in perfect sensor/actuator simulation

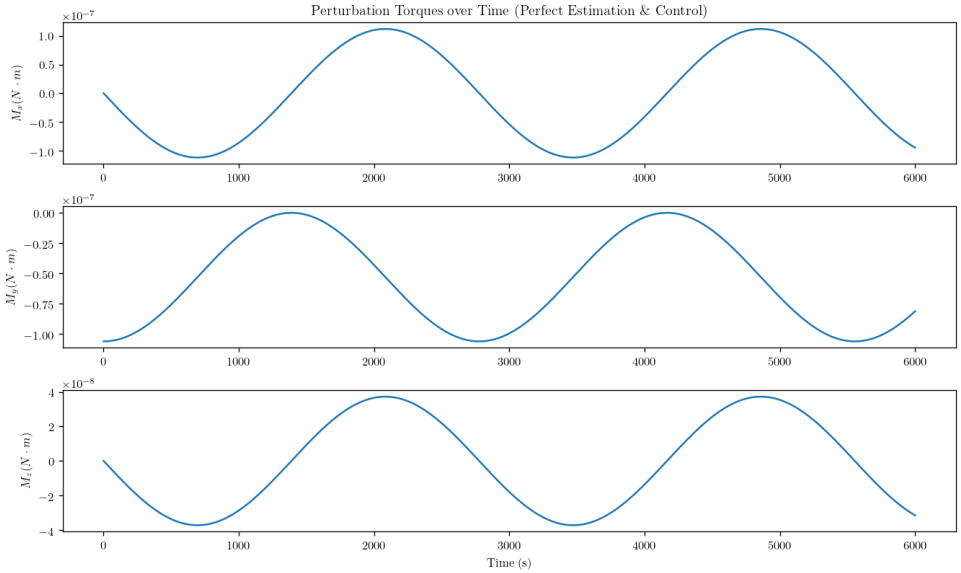


Figure 10: Perturbing torques in perfect sensor/actuator simulation

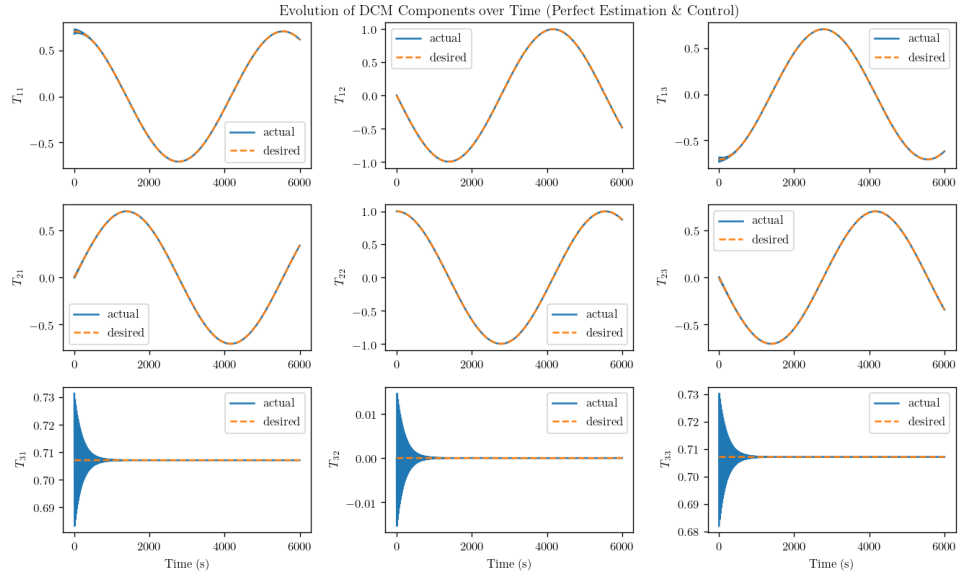


Figure 11: DCM evolution in perfect sensor/actuator simulation

In this simulation run, all sensor noise and biases and all actuator noise (and saturation and maximum torques) were turned off. The result was that the estimated attitude and angular velocity at each timestep matched the actual values exactly.

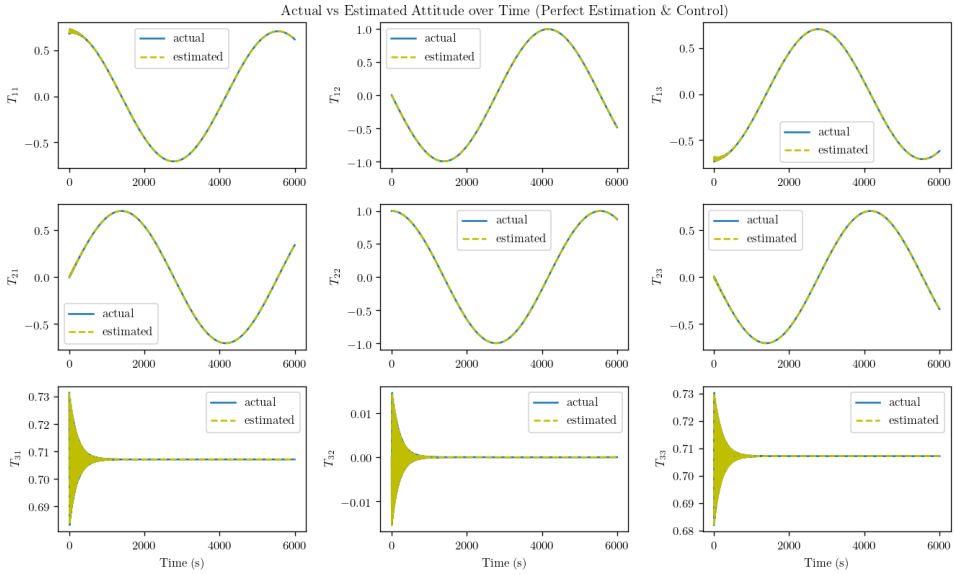


Figure 12: Estimated DCM in perfect sensor/actuator simulation

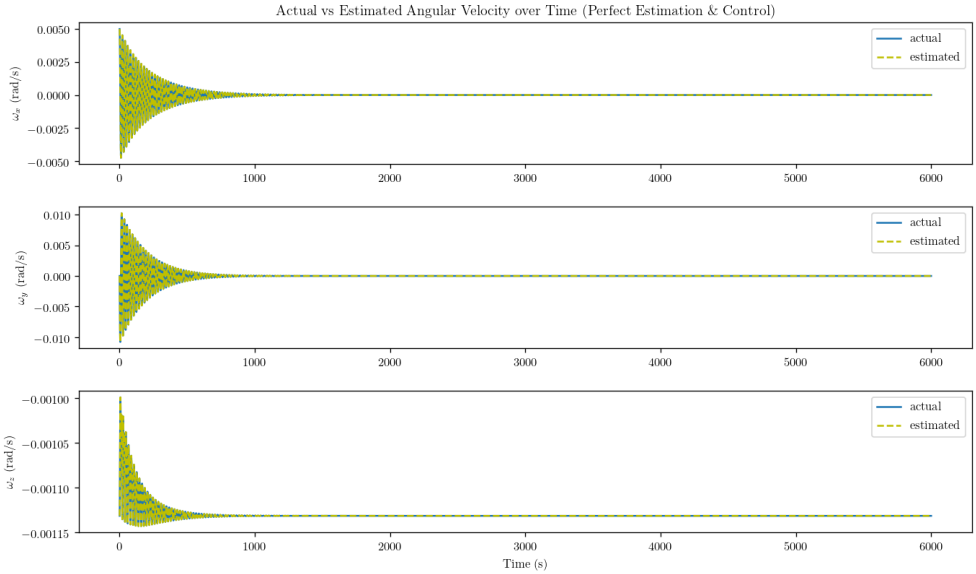


Figure 13: Estimated angular velocity in perfect sensor/actuator simulation

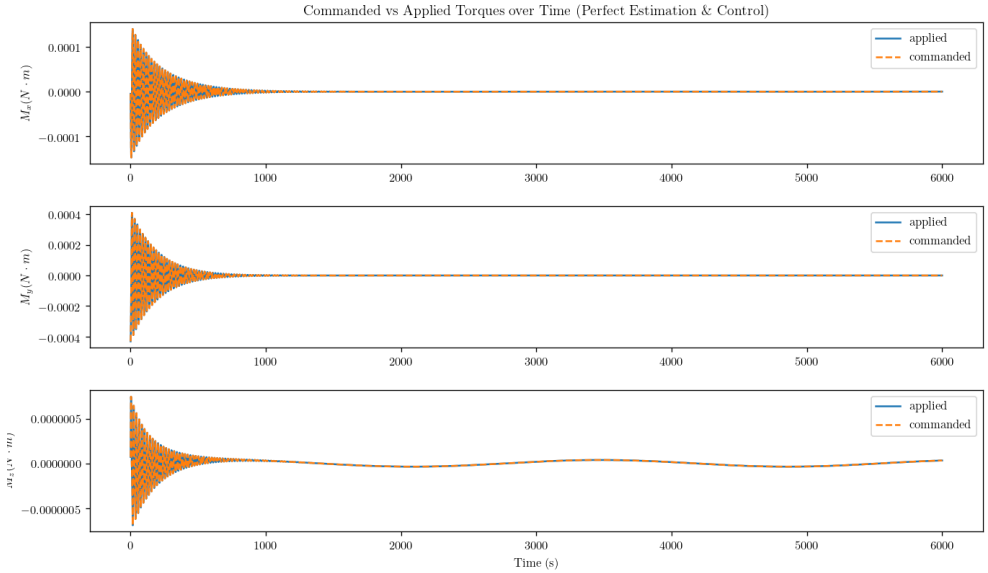


Figure 14: Commanded & applied control torques in perfect sensor/actuator simulation

7.2 Actual Estimation & Control

To model more realistic behavior, sensor and actuator imperfections are added and simulated as described in Section 5. The differences are immediately clear. The curves in the following plots get much noisier (albeit on a small scale), though the attitude and angular velocity still approach their nominal states (Figures 19 & 16). Also of note is how noisy the reaction wheel angular velocities are—indicative of frequent and somewhat jittery actuation.

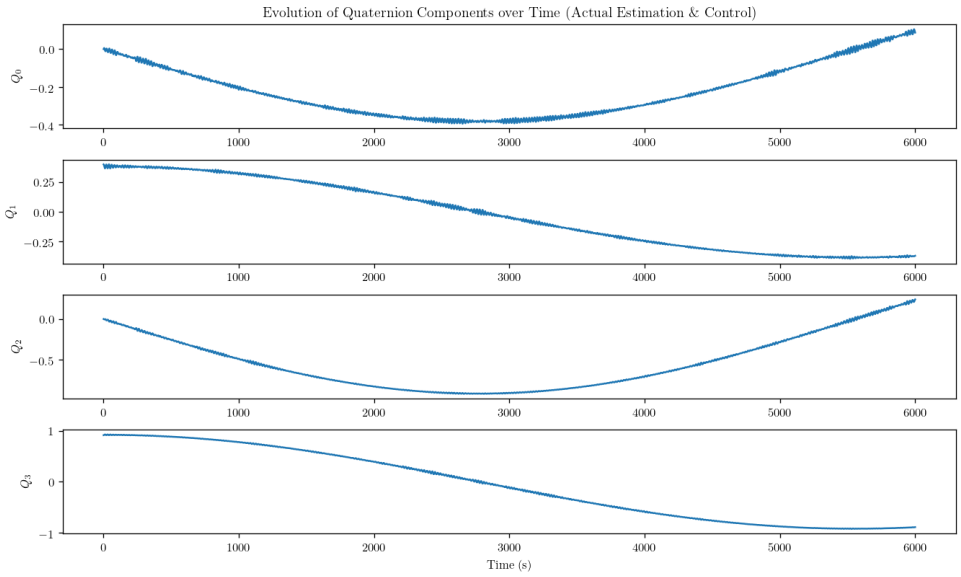


Figure 15: Quaternion evolution in imperfect sensor/actuator simulation

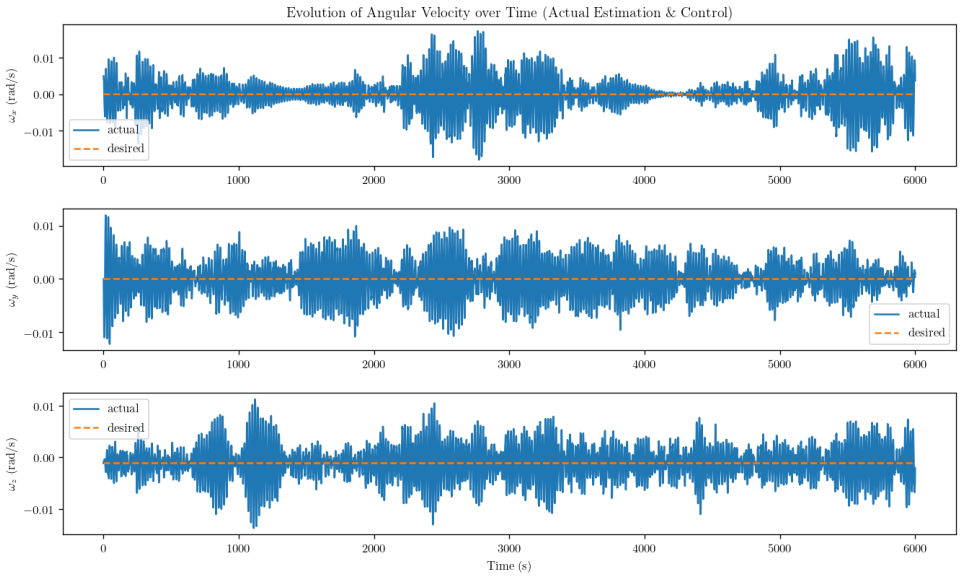


Figure 16: Angular velocity evolution in imperfect sensor/actuator simulation

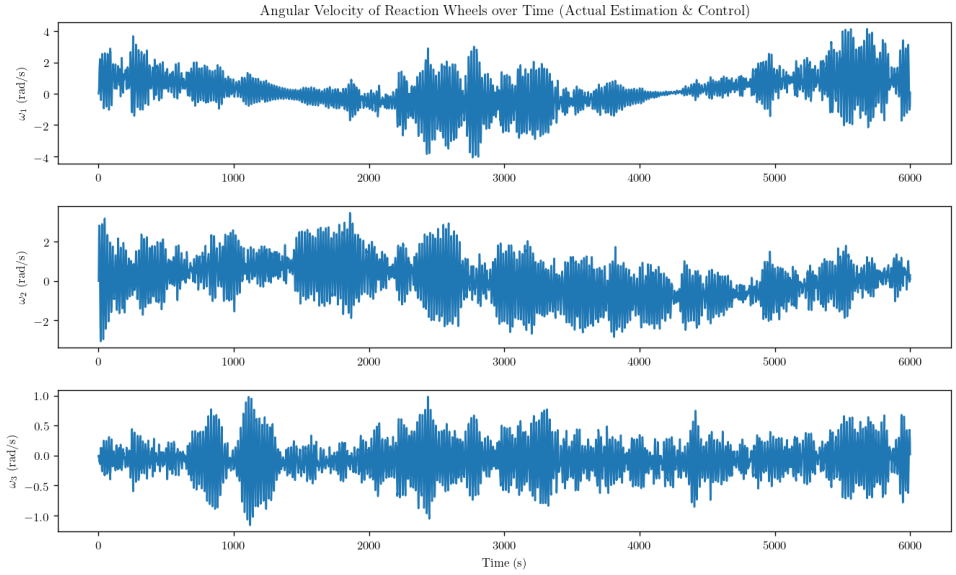


Figure 17: Angular velocity of reaction wheels in imperfect sensor/actuator simulation

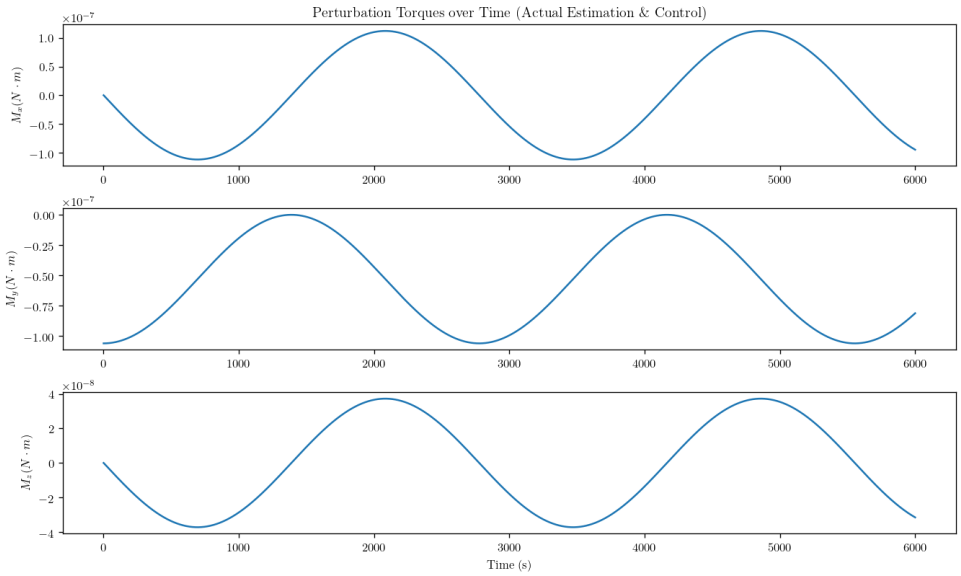


Figure 18: Perturbing torques in imperfect sensor/actuator simulation

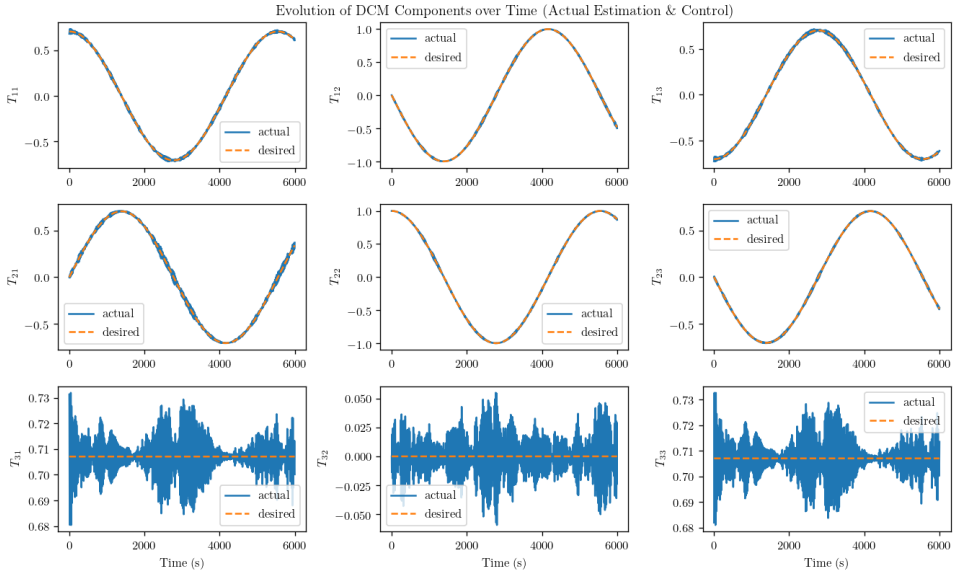


Figure 19: DCM evolution in imperfect sensor/actuator simulation

To highlight not only how system behavior changes with the addition of imperfection but how the measurements and actuation themselves are actually impacted, Figures 20 - 24 show the comparison between actual values and their estimates (for measurement) and commanded values and those actually applied (for actuation). Figures 21 & 24 focus on a smaller timescale to accentuate the differences.

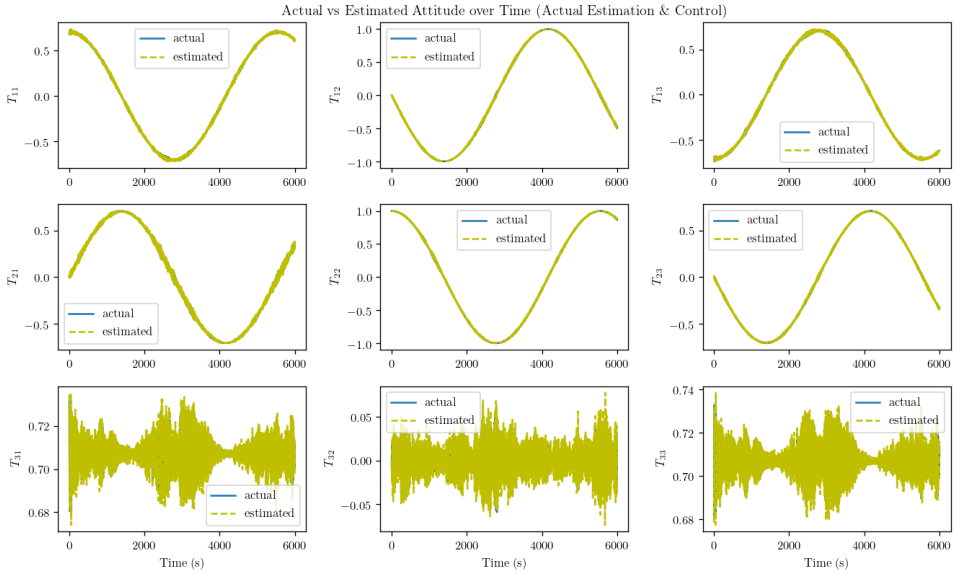


Figure 20: Estimated DCM in imperfect sensor/actuator simulation

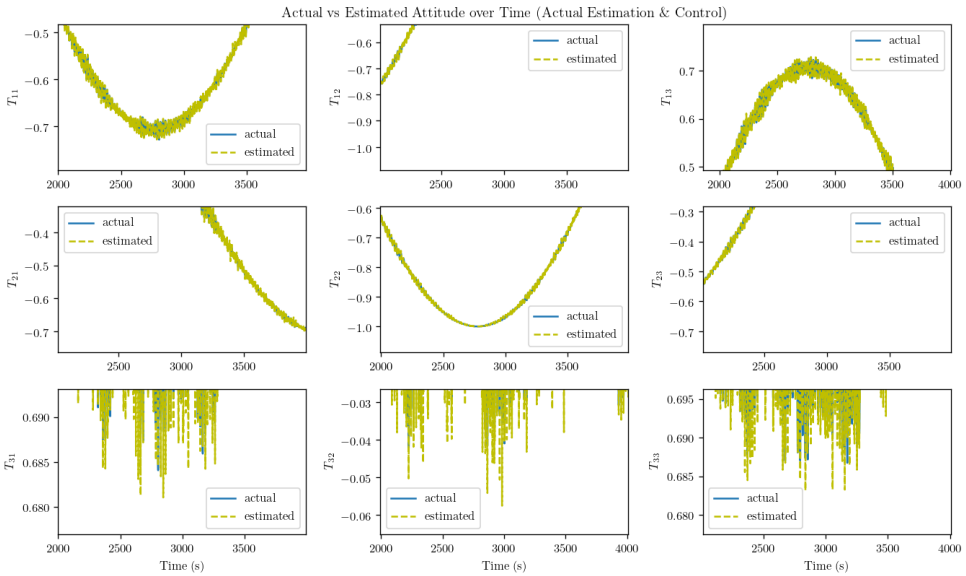


Figure 21: (Zoomed) Estimated DCM in imperfect sensor/actuator simulation

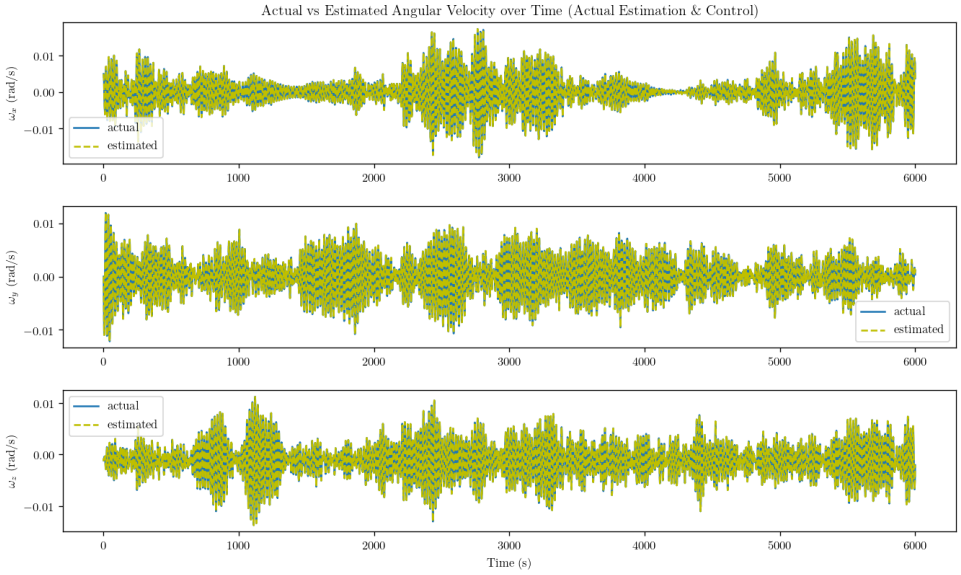


Figure 22: Estimated angular velocity in imperfect sensor/actuator simulation

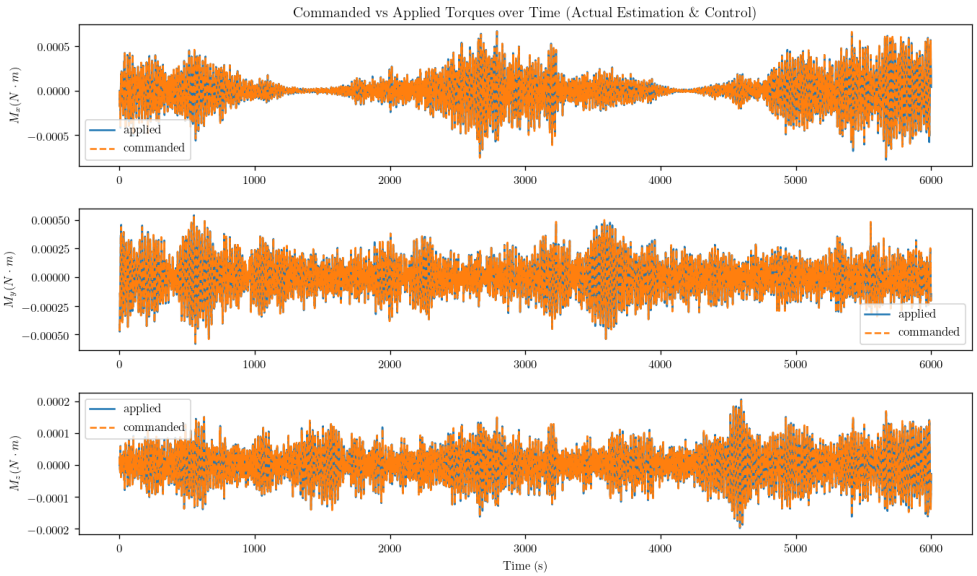


Figure 23: Commanded & applied control torques in imperfect sensor/actuator simulation

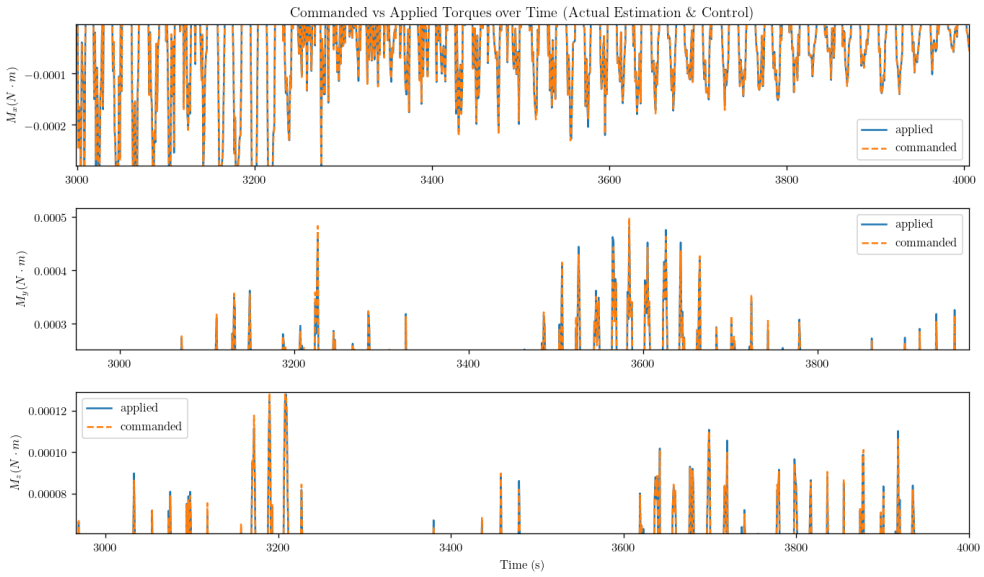


Figure 24: (Zoomed) Commanded & applied control torques in imperfect sensor/actuator simulation

8 Conclusion

This simulation software ultimately performed its purpose well—modeling spacecraft attitude over time given any set of input parameters. The implementation of the mathematical models described in this paper clearly show how adding more layers of abstraction and complexity to the simulation can produce more realistic results.

While the simulation engine is built to be robust to varied system parameters, future improvements could give the user more flexibility in system definition and even include a simpler interface for usage (such as a YAML configuration file rather than a complex control script). Other improvements would be the inclusion of additional types of sensors and actuators to give a system designer a larger toolbox with which to design spacecraft and control systems. This, coupled with the addition of more perturbing torques and more controller options, could make this a much more powerful tool.

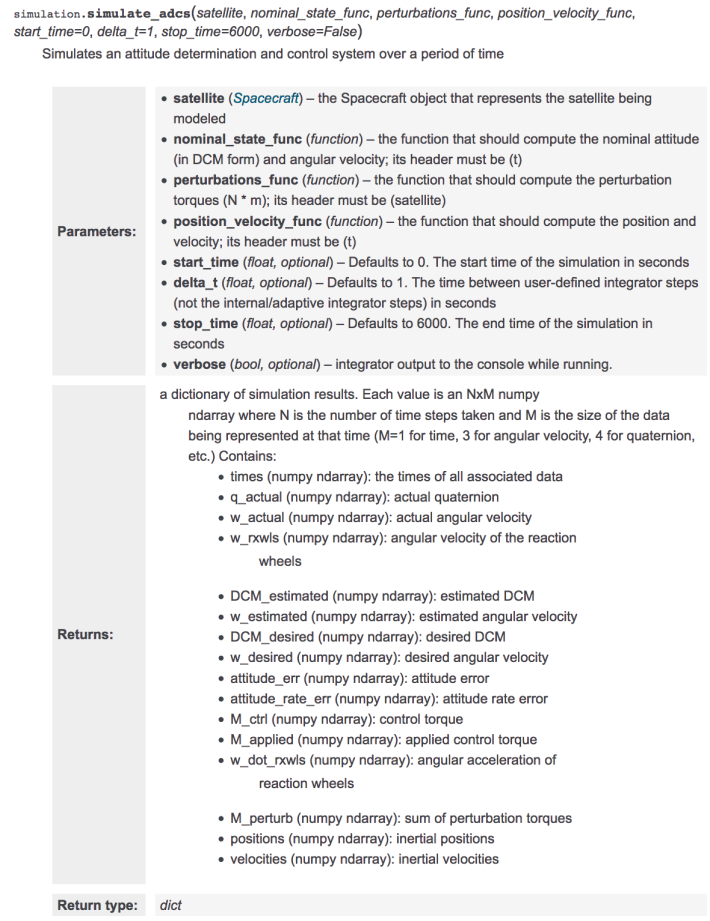
A Appendix: Third-Party Software Modules

A few third-party software modules were used to extend the Python standard library, particularly in the domain of mathematics and computation:

- This project makes heavy use of the NumPy library [8] for storing vectors, matrices, and performing basic linear algebra operations (matrix multiplication, matrix inversion, etc.)
- Two SciPy functions in particular are used in the simulation engine:
  - `scipy.integrate.ode` [1] for numerical integration
  - `scipy.stats.norm` [9] to sample values from Gaussian distributions of the specified sizes for noise and bias calculations
- Algorithms 9.1 & 9.2 from [10] were ported from MATLAB to Python in order to carry out quaternion to direction cosine matrix conversions and their reverse
- Matplotlib [11] was used for plotting the results of the simulations

B Appendix: Software Documentation

The software written for this project is documented according to Google's Python Style Guide [12] for maximum readability. This also enables the use of automated documentation generators to produce HTML-based software documentation. The Sphinx documentation generator [13] was used to produce a full set of documents for this project, and these are hosted online using GitHub Pages at `https://gavincmartin.github.io/adcs-simulation/`. An example of one particular function is shown in Figure 25 below.



simulation.**simulate_adcs**(*satellite*, *nominal_state_func*, *perturbations_func*, *position_velocity_func*, *start_time=0*, *delta_t=1*, *stop_time=6000*, *verbose=False*)

Simulates an attitude determination and control system over a period of time

Parameters:

- **satellite** (*Spacecraft*) – the Spacecraft object that represents the satellite being modeled
- **nominal_state_func** (*function*) – the function that should compute the nominal attitude (in DCM form) and angular velocity; its header must be (t)
- **perturbations_func** (*function*) – the function that should compute the perturbation torques (N * m); its header must be (satellite)
- **position_velocity_func** (*function*) – the function that should compute the position and velocity; its header must be (t)
- **start_time** (*float, optional*) – Defaults to 0. The start time of the simulation in seconds
- **delta_t** (*float, optional*) – Defaults to 1. The time between user-defined integrator steps (not the internal/adaptive integrator steps) in seconds
- **stop_time** (*float, optional*) – Defaults to 6000. The end time of the simulation in seconds
- **verbose** (*bool, optional*) – integrator output to the console while running.

Returns:

a dictionary of simulation results. Each value is an NxM numpy ndarray where N is the number of time steps taken and M is the size of the data being represented at that time (M=1 for time, 3 for angular velocity, 4 for quaternion, etc.) Contains:

- times (numpy ndarray): the times of all associated data
- q_actual (numpy ndarray): actual quaternion
- w_actual (numpy ndarray): actual angular velocity
- w_rxwls (numpy ndarray): angular velocity of the reaction wheels
- DCM_estimated (numpy ndarray): estimated DCM
- w_estimated (numpy ndarray): estimated angular velocity
- DCM_desired (numpy ndarray): desired DCM
- w_desired (numpy ndarray): desired angular velocity
- attitude_err (numpy ndarray): attitude error
- attitude_rate_err (numpy ndarray): attitude rate error
- M_ctrl (numpy ndarray): control torque
- M_applied (numpy ndarray): applied control torque
- w_dot_rxwls (numpy ndarray): angular acceleration of reaction wheels
- M_perturb (numpy ndarray): sum of perturbation torques
- positions (numpy ndarray): inertial positions
- velocities (numpy ndarray): inertial velocities

Return type: *dict*

Figure 25: Example function from documentation

References

[1] SciPy developers. *scipy.integrate.ode*. 2018. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html.

[2] California Polytechnic State University The CubeSat Program. "6U CubeSat Design Specification". In: (June 2018).

[3] John C Springmann, James W Cutler, and Hasan Bahcivan. "Magnetic Sensor Calibration and Residual Dipole Characterization for Application to Nanosatellites". In: (Aug. 2010). DOI: 10.2514/6.2010-7518.

[4] *LN-200S Inertial Measurement Unit*. Northrop Grumman. 21240 Burbank Boulevard Woodland Hills, CA 91367 USA. URL: www.northropgrumman.com/Capabilities/LN200sInertial/Documents/LN200S.pdf.

[5] *MAI-SES Static Earth Sensor*. Adcole Maryland Aerospace. 669 Forest Street, Marlborough, MA 01752. URL: https://www.adcolemai.com/ir-earth-horizon-sensor.

[6] *DS Magnetometer 6A*. NewSpace Systems. 12 Cyclonite Street, The Interchange, Somerset West, South Africa.

[7] *60 mNms Microsatellite Wheel*. Sinclair Interplanetary. URL: http://www.sinclairinterplanetary.com/reactionwheels/60%20mNm-sec%20wheel%202016a.pdf.

[8] NumPy developers. *NumPy*. 2018. URL: http://www.numpy.org/.

[9] SciPy developers. *scipy.stats.norm*. 2018. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html.

[10] H.D. Curtis. *Orbital Mechanics for Engineering Students*. 3rd ed. Oct. 2013, e128–e129.

[11] Matplotlib developers. *Matplotlib*. 2018. URL: https://matplotlib.org/.

[12] Google developers. *Google Python Style Guide*. 2018. URL: http://google.github.io/styleguide/pyguide.html.

[13] Sphinx developers. *Sphinx Python Documentation Generator*. 2018. URL: http://www.sphinx-doc.org/en/master/.