Modeling, Controlling, and Flight Testing of a Small Quadcopter

David Li

davidlxl@umich.edu

June 24, 2022

Abstract

Unmanned aerial vehicles are an increasingly important technology that supports people's daily life and military activities. In my capstone project, I worked on the control aspect of a quadcopter (Parrot Mambo) from around February 2021 to June 2022. This report documents what I completed in this time frame and serves as a technical reference guide for future researchers. In particular, in this report, I will provide detailed instructions on how to use MATLAB and Simulink to model the quadcopter and connect it to the computer, demonstrate how control theories can be successfully applied to improve the quadcopter performance in simulation, and discuss the challenges and possible steps for improving the quadcopter performance in practice. This video presentation is a high-level overview and is roughly parallel to the document.

Contents

1 Introduction **1**

2 Problem **2**

3 General Strategy **2**

4 Modeling **2**

4.1 Getting Started . . . 3

4.1.1 Installing Add-Ons . . . 3

4.1.2 Locating the Project Directory . . . 3

4.1.3 Opening the Project . . . 4

4.2 Configuring Input and Output . . . 4

4.2.1 Input Configuration . . . 4

4.2.2 Output Configuration . . . 6

4.3 Making Two Copies . . . 7

5 Simulating and Controlling **8**

5.1 Performing a Simulation and Plotting the Outputs . . . 9
5.1.1 Simulating . . . 9
5.1.2 Plotting . . . 9
5.2 Understanding the Flight Controller . . . 15
5.3 Understanding How to Convert z-y-p-r Values into Motor Commands . . . 16
5.4 Controlling the Altitude . . . 17
5.4.1 Getting Rid of Two Advanced Mechanisms . . . 17
5.4.2 Creating the *Constant-Altitude Scenario* . . . 18
5.4.3 Simulating with the Default Controller . . . 19
5.4.4 Designing the Altitude Controller for the Simplified Model . . . 20
5.4.5 Upgrading the Altitude Controller in the Original Model . . . 22
5.5 Controlling the Yaw . . . 24
5.5.1 Creating the *Constant-Altitude-and-Then-Constant-Yaw Scenario* . . . 24
5.5.2 Simulating with the Default Controller . . . 25
5.5.3 Upgrading the Yaw Controller . . . 26
5.6 Controlling the Pitch and Roll . . . 27
5.6.1 Creating Two Scenarios . . . 27
5.6.2 Extending the x and y Limits . . . 28
5.6.3 Simulating with the Default Controllers . . . 29
5.6.4 Upgrading the Pitch and Roll Controllers . . . 31
5.7 Controlling the Horizontal Movements . . . 33
5.7.1 Creating Two Scenarios . . . 33
5.7.2 Simulating with the Default Controllers . . . 34
5.7.3 Upgrading the x and y Controllers . . . 36
5.8 Testing with the *Constant-Altitude-and-Then-Alternate-x-y-Yaw Scenario* . . . 38

6 Flight Testing **41**

6.1 Understanding How to Make Bluetooth Connections . . . 41
6.2 Procedure for Flight Testing and Postflight Analysis . . . 41
6.2.1 Setting the Scenario of Interest . . . 41
6.2.2 Configuring the Models for Flight Code Generation . . . 42

6.2.3 Generating Flight Code 42
6.2.4 Completing the Flight Test 43
6.2.5 Making Plots for Postflight Analysis 43
6.3 Potential Sources for Performance Improvement 45

7 Summary and Conclusion **46**

A Clarification of the Meaning of *Active Signal* **48**

B Preventing Unintentional Live Script Simulation Output **48**

C More Information on the Firmware of Parrot Mambo **49**

D List of Add-Ons Used for the Project **50**

1 Introduction

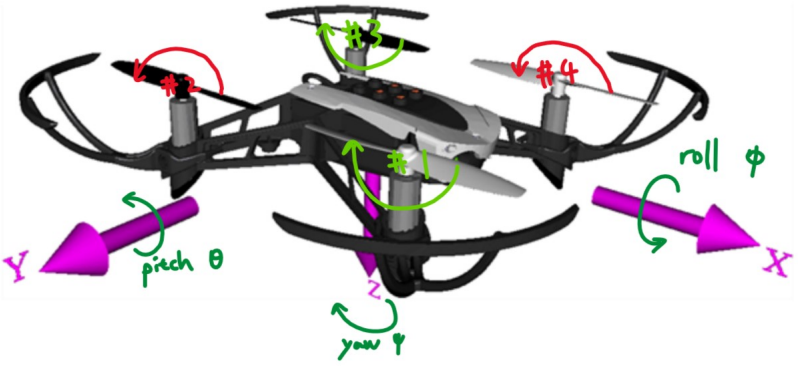


Figure 1: Illustration of a quadcopter, definition of the body frame, and definition of the Euler angles.

An illustration of the Parrot Mambo quadcopter is shown in figure 1. For ease of reference, the rotor on the front right is referred to as rotor #1, the one on the back right is referred to as rotor #2, the one on the back left is referred to as rotor #3, and the one on the front left is referred to as rotor #4. When viewed from above, rotors #1 and #3 spin clockwise, whereas the other two spin counterclockwise. The x-axis of the body frame points to the front, the y-axis of the body frame points to the right, and the z-axis of the body frame points downwards. The associated Euler angles (roll, pitch, and yaw) of the three axes have their positive senses defined according to the right-hand rule, as indicated in figure 1. The six state values of particular interest are x-coordinate, y-coordinate, z-coordinate, roll, pitch, and yaw. Note that x-, y-, and z-coordinates are coordinates in the *world reference frame* (the frame where the origin is fixed and located at the original take-off position of the quadcopter), whereas roll, pitch, and yaw are quantities in the body reference frame.

Consider the following thought exercise. We first place a stationary Parrot Mambo quadcopter on the ground. Before taking off, none of its four rotors is spinning, and all six state values are equal to zero. If we would like to have the quadcopter take off and fly straight up to some altitude, we would increase the speeds of all four rotors simultaneously (keeping the four rotor speeds the same), which would decrease the z-coordinate from zero to some negative value. Then, while the quadcopter is hovering in the air, if we would like to have the quadcopter move forward, we would increase the speeds of rotors #2 and #3 (by the same amount) and/or decrease the speeds of rotors #1 and #4 (by the same amount), which would decrease the pitch and consequently increase the x-coordinate. Similarly, if we would like to have the quadcopter move to the right instead, we would increase the speeds of rotors #3 and #4 (by the same amount) and/or decrease the speeds of rotors #1 and #2 (by the same amount), which would increase the roll and consequently increase the y-coordinate. In this manner (by changing the roll and

pitch appropriately), we could get the quadcopter to go to any arbitrary position specified by a pair of x- and y-coordinates. Alternatively, we could also change the yaw and then change either the roll or pitch. To increase the yaw, for example, we would increase the speeds of rotors #1 and #3 (by the same amount) and/or decrease the speeds of rotors #2 and #4 (by the same amount).

For the full derivation of the above qualitative description of how the motion of a quadcopter is determined by its rotors speeds, see *Drone Simulation and Control, Part 1: Setting Up the Control Problem*.

2 Problem

For the Parrot Mambo quadcopter, we are interested in controlling the speeds of the four rotors over time to achieve any desired flight trajectory.

3 General Strategy

As explained in *Drone Simulation and Control, Part 4: How to Build a Model for Simulation*, our general strategy for achieving the desired flight trajectory consists of the following steps:

1) build/obtain a computer model for the quadcopter that can
    a) be used for simulations
    b) communicate with the quadcopter (that is, we can build flight code with the model and send it to the physical quadcopter)
2) perform simulations with the model and modify the model and/or control logic appropriately
3) send flight code from the model to the quadcopter

Step 1 can be thought of as the initial, one-time step, whereas steps 2 and 3 are inherently iterative.

4 Modeling

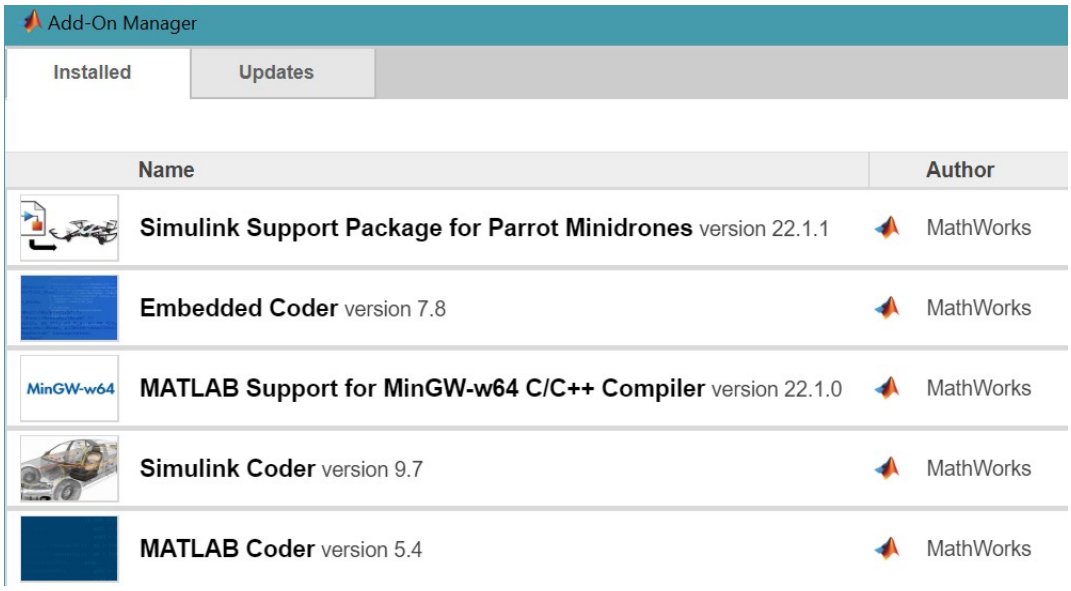
The qualitative model for how the rotor speeds affect the motion of a quadcopter is given in section 1. The quantitative model, on the other hand, is much more involved. *Modeling Vehicle Dynamics – Quadcopter Equations of Motion* covers the vehicle dynamics for such a quantitative model. Fortunately, we do not need to build a computer model from scratch, as MATLAB's Aerospace Blockset provides the model of the Parrot Mambo that serves exactly the purposes of simulation and communication.

4.1 Getting Started

Note: all the technical procedures in this document have been tested with MATLAB R2022a, and slight, possibly trivial differences may be observed with a different version of MATLAB.

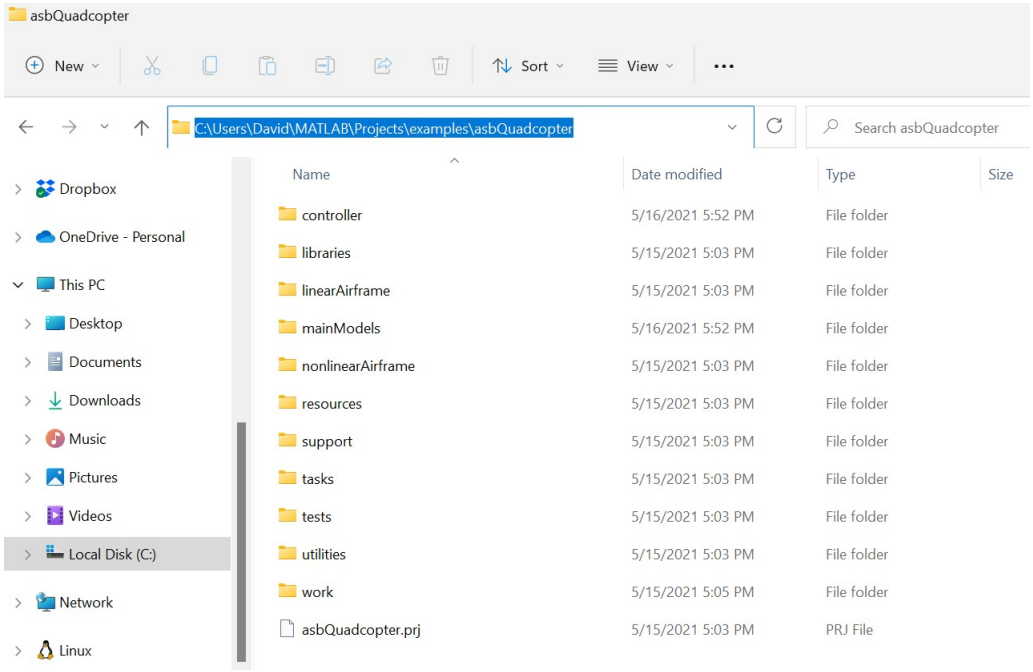
4.1.1 Installing Add-Ons

There is an official documentation page for the quadcopter model I used. This page, however, contains more information than needed for us to get started. To start the process of setting up the project, we first install Simulink and the six required toolboxes (as well as the Aerospace Toolbox, Control System Toolbox, and Image Processing Toolbox) on that page. Now, at this point, if we were only interested in simulations and not interested in flight testing in practice, we could simply go ahead and run the `asbQuadcopterStart` command in MATLAB. However, in my experience, if we wanted to eventually test the physical quadcopter, we should also install the following add-ons and only then would we run the `asbQuadcopterStart` command:



4.1.2 Locating the Project Directory

The `asbQuadcopterStart` command will initiate a MATLAB project. All the files for this project are stored in a particular directory. For PC, the directory should look like the one in the following figure:



4.1.3 Opening the Project

For ease of access, I moved the entire project folder to a more convenient, personal directory. Afterwards, the easiest way to open the project is to run the `openProject('<directory>')` command in the Command Window (not live script; see appendix B):

```
Command Window
>> openProject('C:\Users\David\OneDrive\Documents\Part 11 - Fall 2021\Honors Capstone\Technical Work\asbQuadcopter');
```

4.2 Configuring Input and Output

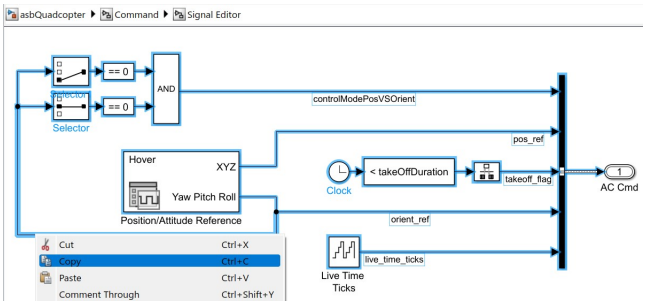
We refer to the Simulink model immediately after setting up the project as the *prebuilt model*. Now we need to configure the input and output. We define the input as the reference flight trajectory and output as the actual (simulated) flight trajectory as well as other simulation outputs. A flight trajectory can be completely described by how x-coordinate, y-coordinate, z-coordinate, and yaw change over time. However, as explained in section 1, roll and pitch are in some sense more "fundamental" than x- and y-coordinates from the control perspective, so we are interested in all six state values for a flight simulation: x-, y-, and z-coordinates, as well as roll, pitch, and yaw.

4.2.1 Input Configuration

The input configuration for this model is very tricky. In my actual experience of going through the simulation phase (step 2 of our general strategy in section 3), I worked exclusively with the Signal Editor inside the Command subsystem (what I mean by "Command subsystem" will be clear shortly)

to configure the input. However, as discussed in *Drone Simulation and Control, Part 3: How to Build the Flight Code*, only the "flightControlSystem" model (again, what I mean by "'flightControlSystem' model" will be clear shortly) is used for building the flight code and deploying it to the physical quadcopter. It took me a lot of effort during the flight testing phase (step 3 of our general strategy in section 3) to figure out a way to configure input inside the "flightControlSystem" model. The best way that I found is as follows.

First, copy everything except the "AC Cmd" Outport block inside the Signal Editor inside the Command subsystem (it should now be clear what I mean by "Command subsystem"; see the figure immediately below):



Next, paste what was copied into the "landing logic" subsystem of the "flightControlSystem" model (it should now be clear what I mean by "'flightControlSystem' model"; see the figure immediately below) and then make the following three changes: connect the "serverCmd" Inport block to a Terminator block; replace the Clock, Compare To Constant, and Rate Transition blocks with a Counter Free-Running block and the modified Compare To Constant block; and connect the output of the Bus Creator block to input 3 of the Switch block. This is shown in the figure below:

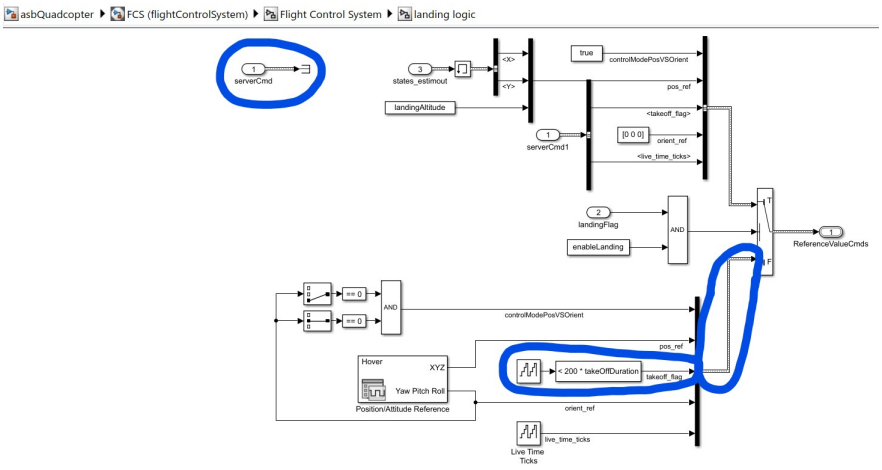
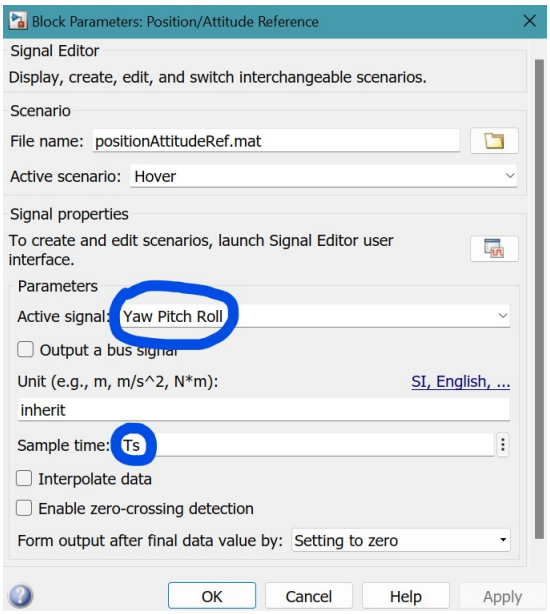


Figure 2: Subsystem for input configuration.

Note that the new Counter Free-Running block can be made by simply copying and pasting the

existing ("Live Time Ticks") one (or we can manually set the number of bits for the new block to be 32) and that we multiply the `takeOffDuration` constant by 200 in order to account for the factor-of-200 correspondence between a *count* and a *simulation second* (1 simulation second = 200 counts because the sample time of this entire model is 5 ms).

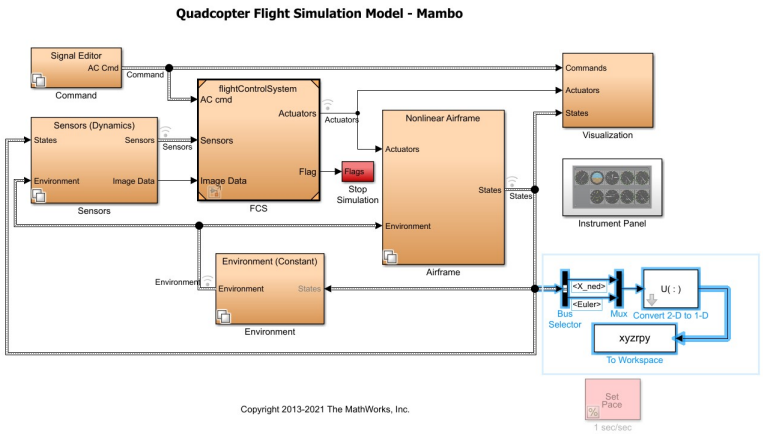
Finally, open the "Position/Attitude Reference" Signal Editor block, set the *active signal* (the meaning of which is explained in appendix A) to "Yaw Pitch Roll", and set the sample time to "Ts" (which is a constant equal to 0.005). This step is necessary because the prebuilt model has the sample time corresponding to the "Yaw Pitch Roll" active signal equal to 0, which would result in a compilation error due to the inconsistency of sample time (hence why I consider this step to be one that fixes an error in the prebuilt model).



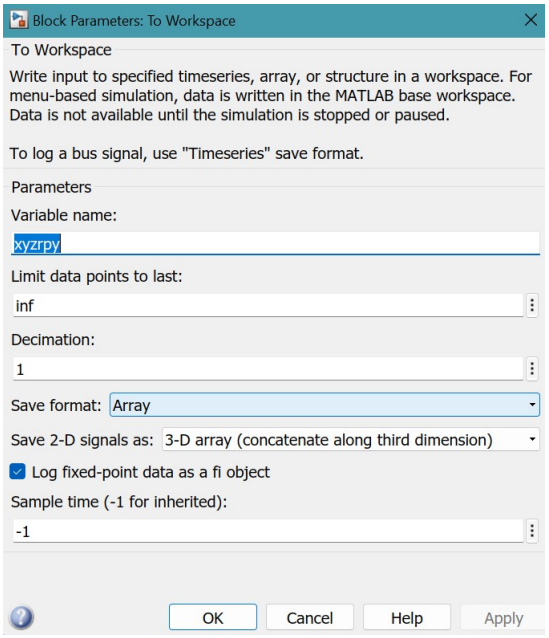
4.2.2 Output Configuration

My method of output configuration makes use of a particular Simulink block, and in order to use this block, we need to first install the DSP System Toolbox.

Now, MATLAB has made life easier for us by including the vast majority of our desired outputs into the prebuilt model. These preconfigured outputs include estimated state values, input references, motor commands, and sensor measurements. However, the "true" state values ("true" in the context of the simulation versus the actual flight test with the physical drone) are not included, and I thought it would be interesting to include and compare them with the estimated state values during the simulation phase (step 2 of our general strategy as discussed in section 3). To output the relevant true state values, simply add and connect the following four highlighted blocks:



A note is in order regarding the To Workspace block above: I named the variable "xyzrpy" (which corresponds to x-coordinate, y-coordinate, z-coordinate, row, pitch, and yaw) and I set the format of the variable to "Array":

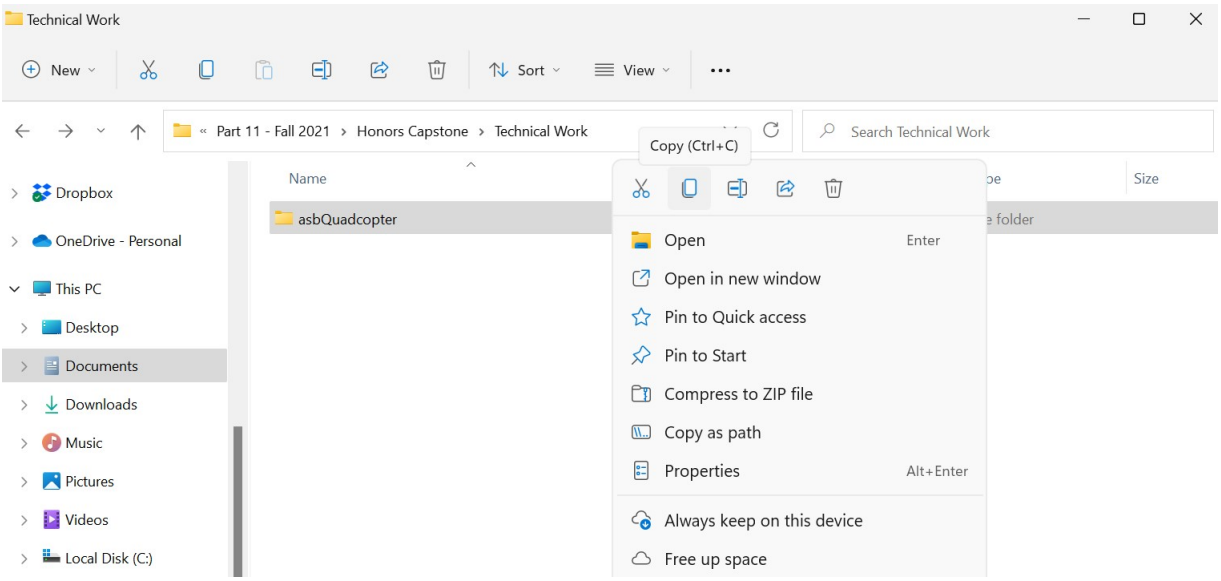


The exact meanings of the configured outputs and how to plot them will be covered in subsection 5.1.

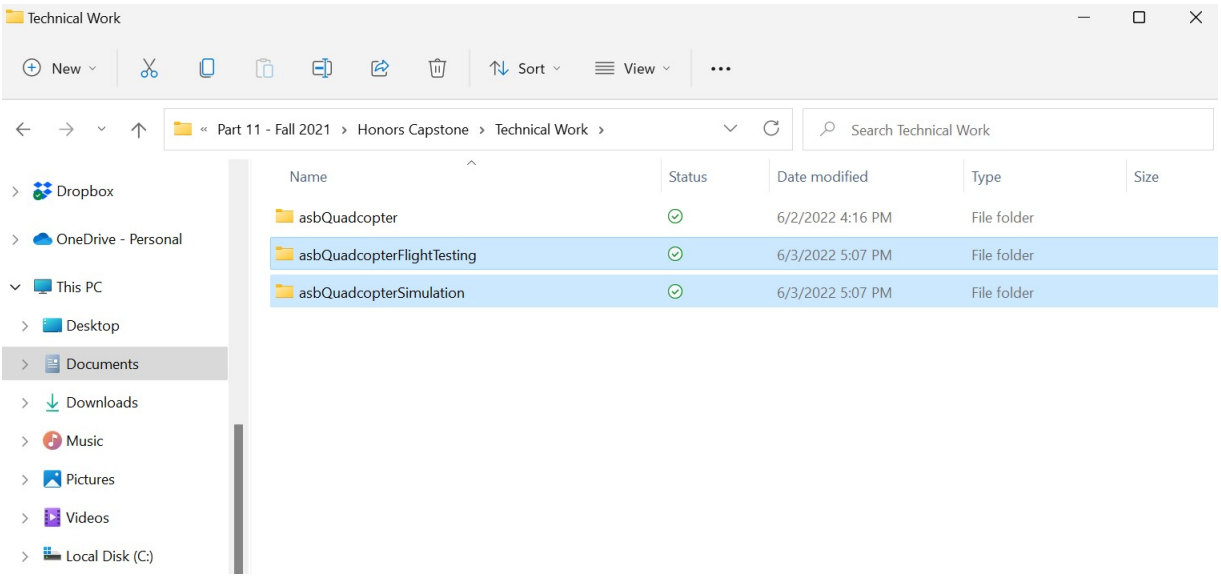
4.3 Making Two Copies

The current state of the model is a great starting point for both the simulation phase (step 2 of the general strategy in section 3) and the flight testing phase (step 3 of the general strategy in section 3). In my experience, the model resulting directly from completing the simulation phase could easily lead to complete disasters during the flight testing phase. Therefore, I recommend treating the two phases somewhat independently and starting afresh for both of them with the model we have just obtained after completing the input and output configurations.

To make a copy of the project, simply go to the parent directory of the project, copy the project folder, and paste into the same directory:



For ease of reference, I named one duplicate folder "asbQuadcopterSimulation" and the other "asbQuadcopterFlightTesting":



5 Simulating and Controlling

This section is about the simulation phase and corresponds to step 2 of our general strategy in section 3. Throughout this section, we will work with the model inside the "asbQuadcopterSimulation" folder:

```
Command Window
>> openProject('C:\Users\David\OneDrive\Documents\Part 11 - Fall 2021\Honors Capstone\Technical Work\asbQuadcopterSimulation');
```

5.1 Performing a Simulation and Plotting the Outputs

In this subsection, I will demonstrate a general technique of programmatically running a simulation and plotting the outputs. I will also clarify the meanings of the output variables.

5.1.1 Simulating

At the current stage of the project, the Signal Editor block in the "landing logic" subsystem (see sub-subsection 4.2.1) has two scenarios: "Hover" and "LandingSearch". I came up with the following code to efficiently run a simulation corresponding to an arbitrary scenario and stop time (the start time is always assumed to be 0):

```
1  scenarios = {'Hover'; 'LandingSearch'};
2  TFinals = [10; 15];
3
4  in = Simulink.SimulationInput.empty(size(scenarios, 1), 0);
5  for i = 1 : size(scenarios, 1)
6      in(i) = Simulink.SimulationInput('asbQuadcopter');
7      in(i) = in(i).setBlockParameter(['flightControlSystem/Flight Control System/' ...
8              'landing logic/Position//Attitude Reference'], ...
9              'ActiveScenario', scenarios{i});
10     in(i) = in(i).setVariable('TFinal', TFinals(i), 'Workspace', 'asbQuadcopter');
11 end
12
13 out = sim(in(1)); % simulate with the "Hover" scenario for 10 seconds
```

In the code above, lines 1-2 define the scenario names and the corresponding stop times, and line 13 runs the simulation with the chosen scenario. For example, if we wanted to simulate with the "LandingSearch" scenario for 15 seconds instead, we would simply change line 13 to `out = sim(in(2));`. Also, as we design new scenarios in the Signal Editor block, we would simply modify lines 1, 2, and 13 appropriately to perform the new simulations.

A minor note: when the code is run in live script, an additional unintentional figure output may show up; see appendix B for the solution.

5.1.2 Plotting

All the output information that we are interested in is stored in the variable `out`. To simplify the subsequent code, we first extract the individual output variables from the `out` variable:

```
1  t = out.posref.time;
2  xyzrpy = out.xyzrpy;
```

```
estim = out.estim.signals.values;
posref = out.posref.signals.values;
motor = out.motor.signals.values;
sensor = out.sensor.signals.values;
```

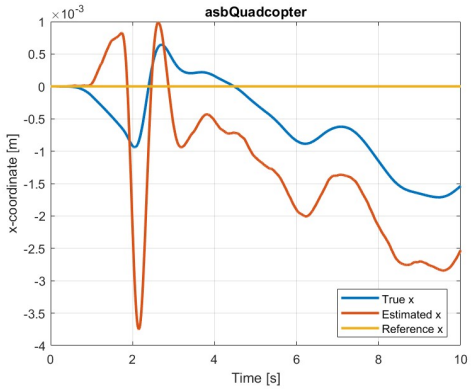
Note that in line 1, we could have written `t = (0 : Ts : <simulation stop time>)';` instead. For example, to be consistent with the provided code in subsubsection 5.1.1, we could have written `t = 0 : Ts : TFinals(1);`. The two lines of code are equivalent for all intents and purposes, even though the `isequal()` function would return a value of `false` (due to roundoff errors) when we test the two for equality.

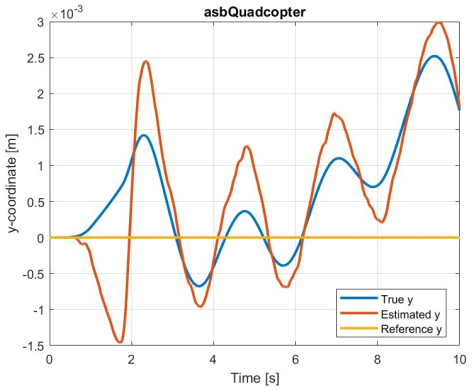
We can now plot the output variables mentioned in subsubsection 4.2.2 one at a time.

1) x- and y-coordinates

Since we define the origin of the world reference frame to be at the place where the quadcopter takes off, we need to subtract the initial x- and y-coordinates from the data.

```
plot(t, xyzrpy(:, 1) - xyzrpy(1, 1) * ones(size(t)), t, estim(:, 1), t, posref(:, 1), ...
    'LineWidth', 2);
legend('True x', 'Estimated x', 'Reference x', 'Location', 'best');
xlabel('Time [s]');
ylabel('x-coordinate [m]');
title('asbQuadcopter');
grid on;

plot(t, xyzrpy(:, 2) - xyzrpy(1, 2) * ones(size(t)), t, estim(:, 2), t, posref(:, 2), ...
    'LineWidth', 2);
legend('True y', 'Estimated y', 'Reference y', 'Location', 'best');
xlabel('Time [s]');
ylabel('y-coordinate [m]');
title('asbQuadcopter');
grid on;
```





The plots above show that in this simulation, for both x- and y-coordinates, the true and estimated (based on sensor measurements) values agree well with each other and are essentially at the reference level (less than 5 mm of difference). (This shows the x- and y-performances of the default controller appear to be largely satisfactory, but, as shown later in section 5.7, when the reference x (or y) is not constant-zero, the true and estimated x (or y) do not track the reference as well.)

2) z-coordinate

```
plot(t, xyzrpy(:, 3), t, estim(:, 3), t, posref(:, 3), 'LineWidth', 2);
legend('True z', 'Estimated z', 'Reference z', 'Location', 'best');
xlabel('Time [s]');
ylabel('z-coordinate [m]');
title('asbQuadcopter');
grid on;
```

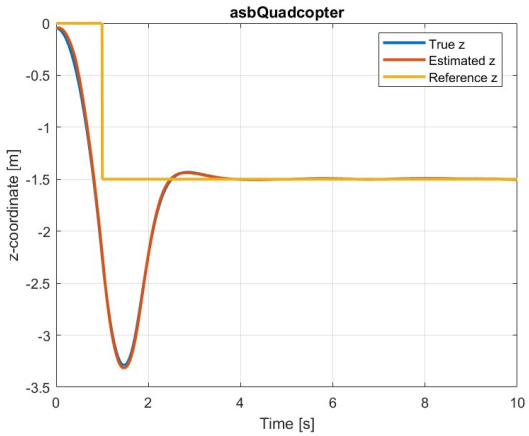


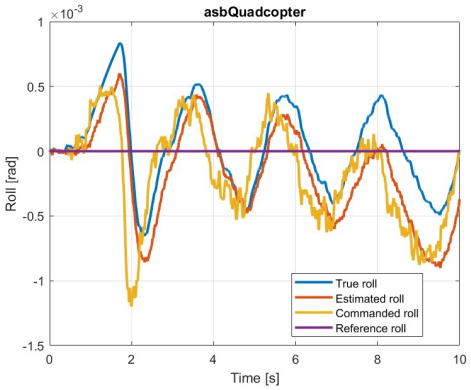
Figure 3: Simulated altitude profile for the "Hover" scenario.

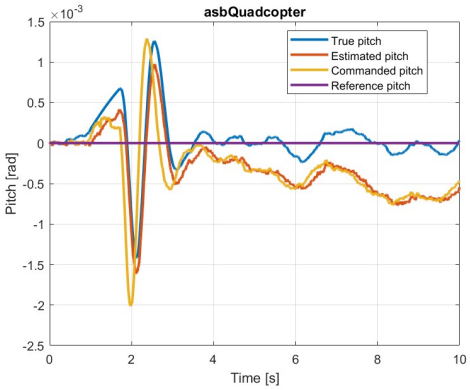
Again, the true and estimated values agree extremely well with each other. The take-off duration of one second (the value of `takeOffDuration` being `1`) explains the transient behavior of this altitude performance. Recall from section 1 that the z-axis points downwards, so the actual altitude of the quadcopter is the exact opposite of the data (`-xyzrpy(:, 3)`).

3) roll and pitch

```
plot(t, xyzrpy(:, 4), t, estim(:, 6), t, posref(:, 8), t, posref(:, 6), 'LineWidth', 2);
legend('True roll', 'Estimated roll', 'Commanded roll', 'Reference roll', ...
       'Location', 'best');
xlabel('Time [s]');
ylabel('Roll [rad]');
title('asbQuadcopter');
grid on;
```

```
8
9  plot(t, xyzrpy(:, 5), t, estim(:, 5), t, posref(:, 7), t, posref(:, 5), 'LineWidth', 2);
10 legend('True pitch', 'Estimated pitch', 'Commanded pitch', 'Reference pitch', ...
11         'Location', 'best');
12 xlabel('Time [s]');
13 ylabel('Pitch [rad]');
14 title('asbQuadcopter');
15 grid on;
```





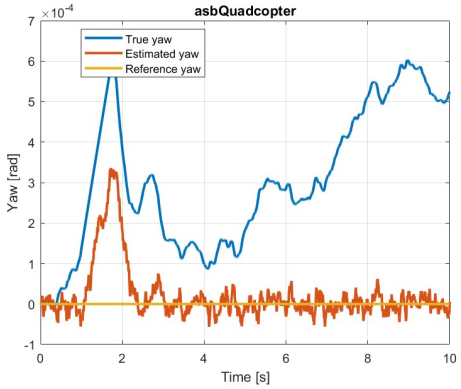
A clarification is needed here regarding the definitions of the terms *commanded roll*, *commanded pitch*, *reference roll*, and *reference pitch* (the plots above are a great way to introduce this clarification). Recall from section 1 that a quadcopter needs to change its roll and pitch values first in order to achieve horizontal movements. Therefore, in the flight controller of the model (see figure 4 in the next subsection), there is a subsystem that converts the x- and y- reference values into roll and pitch commands. These resulting commands are defined as the *commanded roll* and *commanded pitch*.

Now, recall from subsubsection 4.2.1 that the Signal Editor block outputs the "Yaw Pitch Roll" signal, which contains the roll and pitch reference signals. We refer to these two signals as the *reference roll* and *reference pitch*. As shown in figure 2, when both reference signals are zero, the "controlModePosVSOrient" signal is set equal to `true`. When this happens, as shown in figure 4 in the next subsection, the reference roll and pitch are essentially invisible to the controller and that the roll and pitch performances are governed by the commanded roll and pitch instead.

4) yaw

```
1 plot(t, xyzrpy(:, 6), t, estim(:, 4), t, posref(:, 4), 'LineWidth', 2);
2 legend('True yaw', 'Estimated yaw', 'Reference yaw', 'Location', 'best');
3 xlabel('Time [s]');
4 ylabel('Yaw [rad]');
5 title('asbQuadcopter');
```

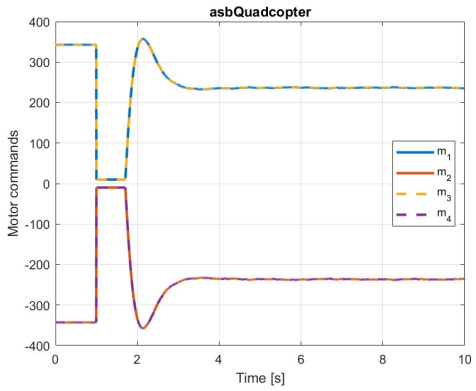
```
grid on;
```



The true and estimated yaw agree well with each other and are both essentially at the reference level (less than 0.8 mrad of difference).

5) Motor commands

```
plot(t, motor(:, 1), t, motor(:, 2), t, motor(:, 3), '--', t, motor(:, 4), '--', ...
    'LineWidth', 2);
legend('m_1', 'm_2', 'm_3', 'm_4', 'Location', 'best');
xlabel('Time [s]');
ylabel('Motor commands');
title('asbQuadcopter');
grid on;
```



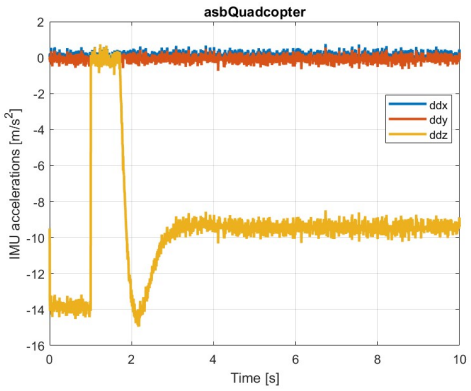
The plot above motivates the clarification on the relation between rotors and motors: The four motor commands control the speeds of the four rotors in figure 1. Two rotors having the same motor command magnitude would spin at the same speed. A positive (or negative) motor command would spin the corresponding rotor counterclockwise (or clockwise) when viewed from above. However, the motor and rotor numbers do not correspond to each other in the natural order; for example, the speed of rotor #1 is actually not proportional to motor command #1 but rather proportional to motor command #2.

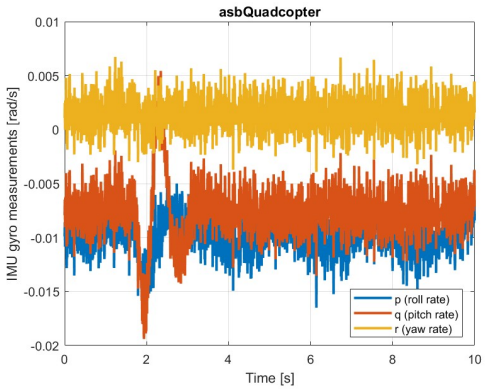
The precise correspondence between the rotors and motors will be presented in subsection 5.3. For now, recall from section 1 that theoretically, if we wanted the quadcopter to go straight up or down without roll, pitch, or yaw motion, we would increase or decrease the rotor speeds simultaneously by the same amount, which is seen in the plot above.

6) IMU measurements

The quadcopter has an inertial measurement unit (IMU), which consists of accelerometers and gyroscopes.

```
plot(t, sensor(:, 1), t, sensor(:, 2), t, sensor(:, 3), 'LineWidth', 2);
legend('ddx', 'ddy', 'ddz', 'Location', 'best');
xlabel('Time [s]');
ylabel('IMU accelerations [m/s^2]');
title('asbQuadcopter');
grid on;

plot(t, sensor(:, 4), t, sensor(:, 5), t, sensor(:, 6), 'LineWidth', 2);
legend('p (roll rate)', 'q (pitch rate)', 'r (yaw rate)', 'Location', 'best');
xlabel('Time [s]');
ylabel('IMU gyro measurements [rad/s]');
title('asbQuadcopter');
grid on;
```



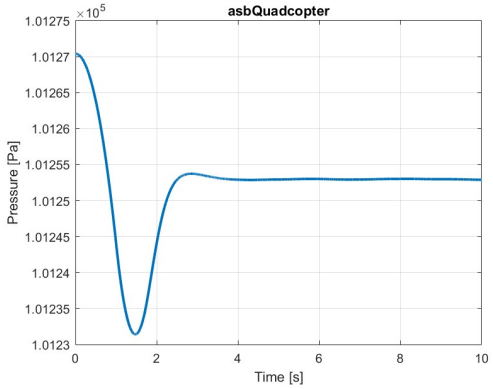


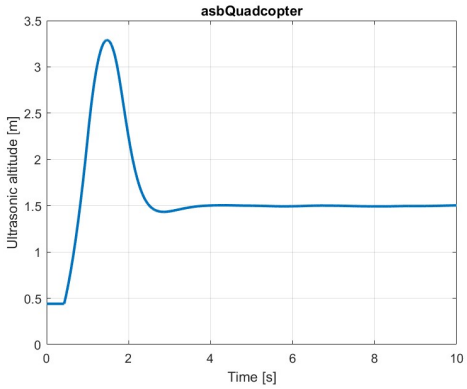
The noisy measurements produced by these sensors are filtered and used for state estimation in the flight controller.

7) altitude-related sensor measurements

The quadcopter also has a pressure sensor and an ultrasonic sensor, which are responsible for altitude estimation.

```
plot(t, sensor(:, 8), 'LineWidth', 2);
xlabel('Time [s]');
ylabel('Pressure [Pa]');
title('asbQuadcopter');
grid on;

plot(t, sensor(:, 7), 'LineWidth', 2);
xlabel('Time [s]');
ylabel('Ultrasonic altitude [m]');
title('asbQuadcopter');
grid on;
```





Note that in our simulation model, these measurements are significantly less noisy compared to the other sensor measurements. Also, the ultrasonic altitude profile very much resembles the actual altitude profile (`-xyzrpy(:, 3)`); in fact, the altitude during a flight is estimated using an ultrasonic-altitude-based Kalman filter that uses the pressure sensor measurements only to control updates.

To conclude, general techniques are presented in this subsubsection for plotting the outputs configured in subsubsection 4.2.2. The code provided is repeatedly used throughout the project to evaluate the controllers' performances for various scenarios.

5.2 Understanding the Flight Controller

As explained in *Drone Simulation and Control, Part 2: How Do You Get a Drone to Hover?*, our general control strategy is to use six controllers (for the six states of interest) that ultimately output only four values: z-coordinate (z) thrust, yaw (y) torque, pitch (p) torque, and roll (r) torque. These four values are then converted into motor commands, which will then govern the motion of the quadcopter. The figure below contains the high-level diagram for the flight controller and the relevant definitions that are used throughout this document.

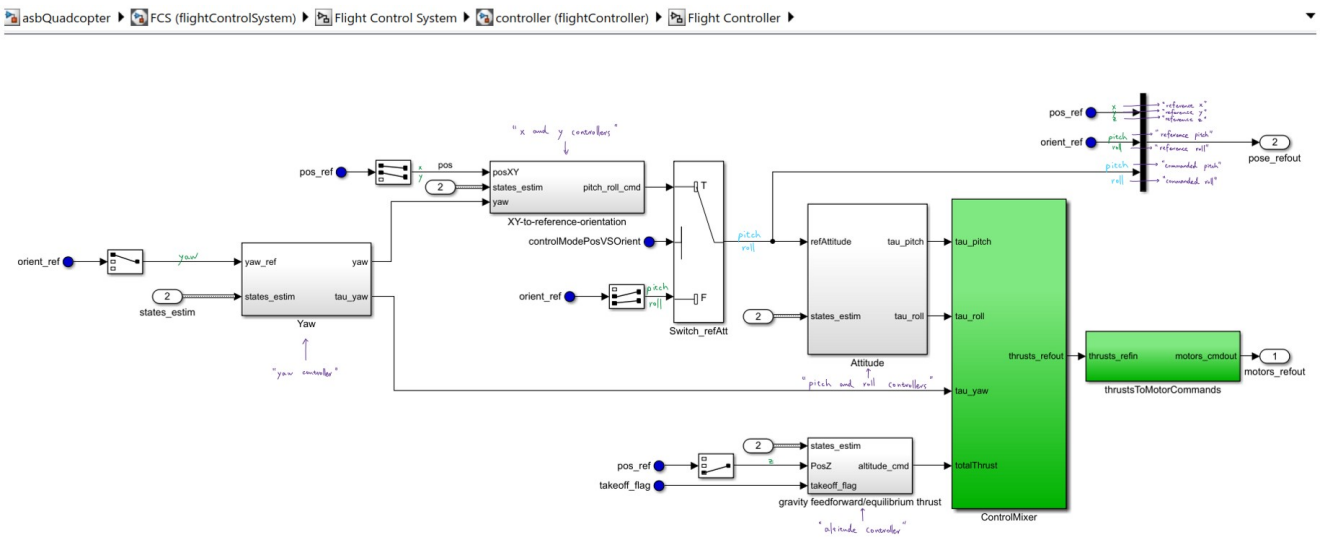


Figure 4: Definitions of controllers, reference signals, and commanded signals.

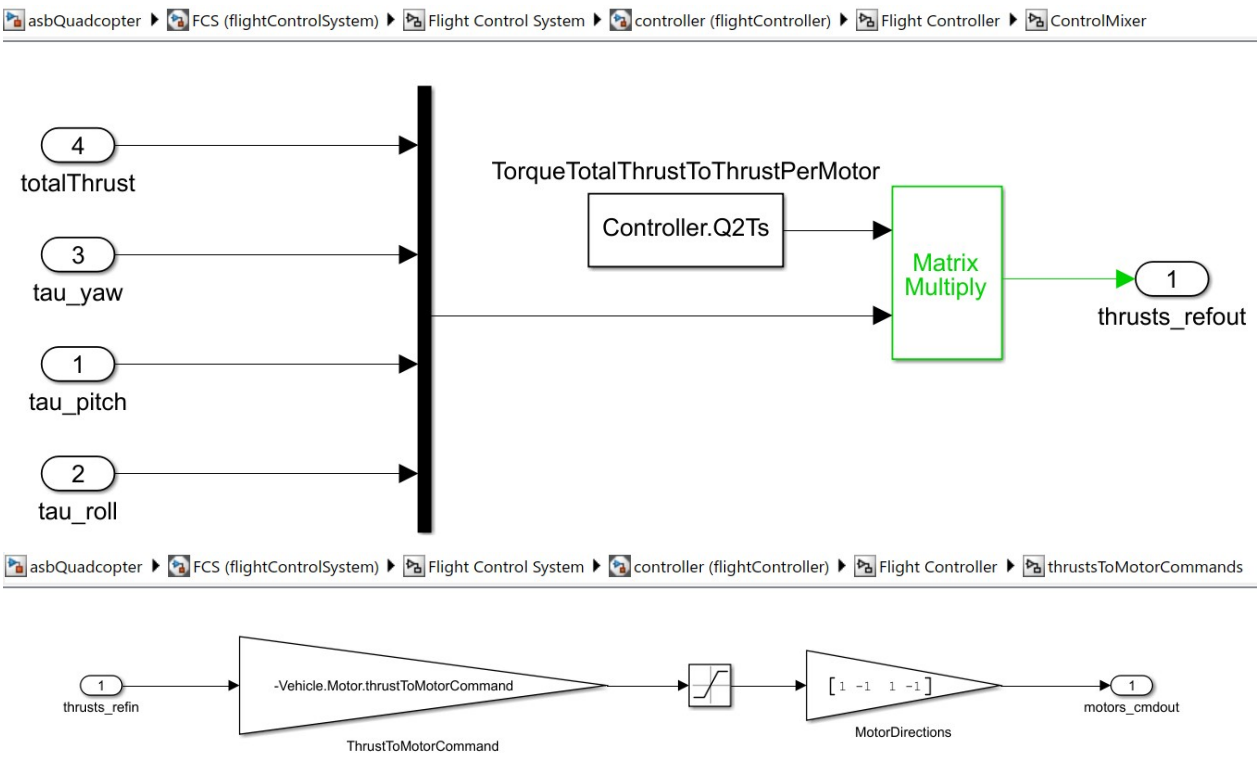
5.3 Understanding How to Convert z-y-p-r Values into Motor Commands

In this project, I mostly assumed the correctness of the model for using the z-y-p-r values to simulate the motion of the quadcopter, so I mostly worked with the six controllers instead. However, I did make the effort at the beginning of the project to understand how the z-y-p-r values are converted into motor commands precisely. The derivation process is not trivial and involves a lot of logical reasoning. In the interest of space, I will only present the final result in this document.

Let r_1, r_2, r_3, and r_4 be the motor commands responsible for controlling the speeds of rotors #1, #2, #3, and #4 (the rotor numbers are defined in section 1), respectively. Let m_1, m_2, m_3, and m_4 denote the four motor commands corresponding to those in the plot in subsubsection 5.1.2. Let $zypr \in \mathbb{R}^4$ be a column vector with the first, second, third, and fourth entries storing the z, yaw, pitch, and roll values, respectively. Then

$$\begin{bmatrix} r_4 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} A(zypr),$$

where $A \in \mathbb{R}^{4\times4}$ is equal to `Controller.Q2Ts * (-Vehicle.Motor.thrustToMotorCommand)`. The global variables `Controller.Q2Ts` and `(-Vehicle.Motor.thrustToMotorCommand)` are used in "ControlMixer" and "thrustsToMotorCommands" subsystems, respectively, as shown below:



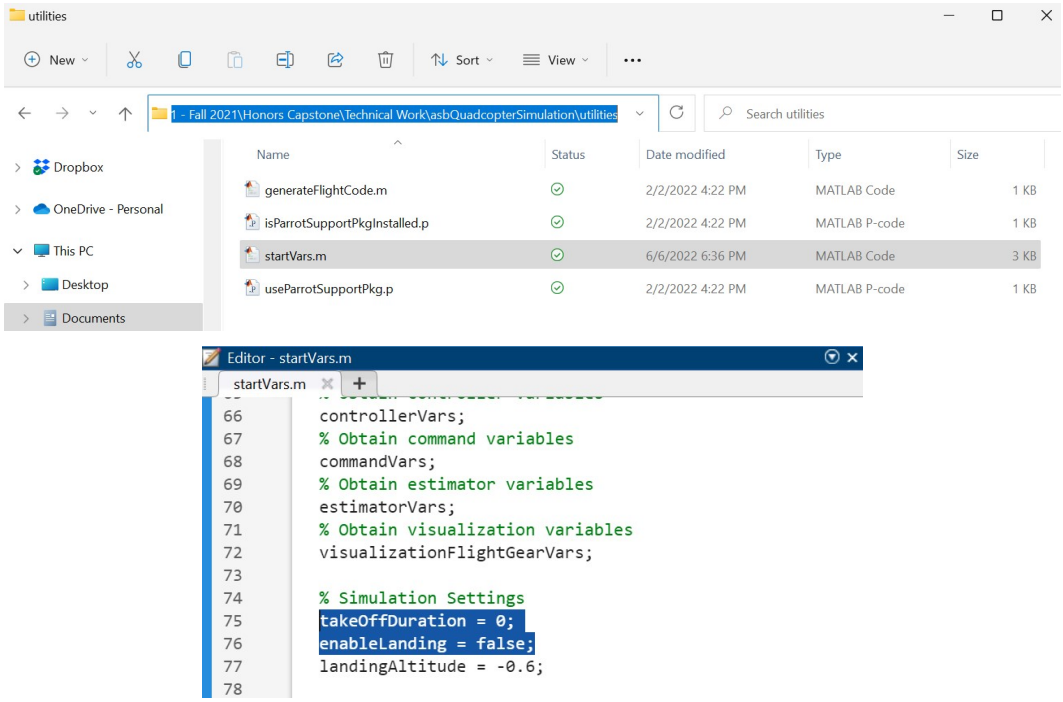
5.4 Controlling the Altitude

Altitude is the simplest yet also the most foundational state for quadcopter control. In order for a quadcopter to do any sort of maneuver in a flight, it has to first fly up and hover. In this subsection, I discuss the steps I took to improve the altitude performance in simulation as much as I could.

5.4.1 Getting Rid of Two Advanced Mechanisms

There are two advanced mechanisms in our current model that are not particularly relevant to controller tuning: the take-off mechanism and the landing mechanism. The former refers to having the altitude controller of the quadcopter behave differently during the initial take-off period and the period afterwards, while the latter refers to allowing the quadcopter to suddenly land in the middle of a flight when some condition is met (for example, when it detects a red object below). These two mechanisms may be useful during the flight testing phase (step 3 of our general strategy in section 3), but they are not necessary throughout the current simulation phase (step 2 of our general strategy in section 3).

The simplest way to disable these two mechanisms is to set the `takeOffDuration` variable equal to `0` and the `enableLanding` variable equal to `false` inside the "startVars.m" file:



Note that the "startVars.m" file is responsible for initializing all the variables when the project is open, so after we modify the values of the two variables above, we need to close and reopen the project for the changes to take effect.

5.4.2 Creating the *Constant-Altitude Scenario*

At the current stage, the Signal Editor block in the "landing logic" subsystem (see subsubsection 4.2.1) still only has two scenarios: "Hover" and "LandingSearch". The "Hover" scenario assumes an initial take-off period (see the reference signal in figure 3), while the "LandingSearch" scenario asks the quadcopter to do a complicated maneuver (search for red objects on the ground). In this subsubsection, we create a much simpler scenario that effectively serves the purpose of aiding us with altitude control.

Recall that the Signal Editor block in the "landing logic" subsystem outputs two signals, one storing the x, y, and z references and the other storing the yaw, pitch, and roll references. For the purpose of altitude control, the simplest scenario possible is one where throughout the simulation, the z reference is set equal to a nonzero constant and everything else is set equal to zero. We refer to such a scenario as a *constant-altitude scenario*. For this subsubsection, we will create the constant-altitude scenario with `-1` being the constant z reference value and `5` being the total simulation time.

The Signal Editor block is intuitive to use, so I will not go over the exact implementation of creating the scenario. I recommend duplicating the "Hover" scenario, renaming the duplicate scenario "ConstantAltitude", and then editing the data points of the "ConstantAltitude" scenario (these data points are consistent with reference z signal in the plots in sections 5.4.3 and 5.4.5). The final data points

should look like the following:

ConstantAltitude.XYZ

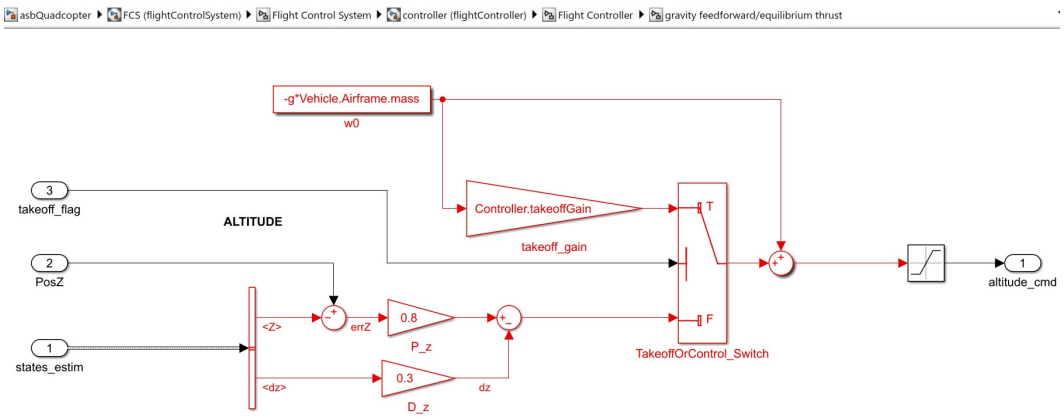
TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	-1
5	0	0	-1

ConstantAltitude.Yaw Pitch Roll

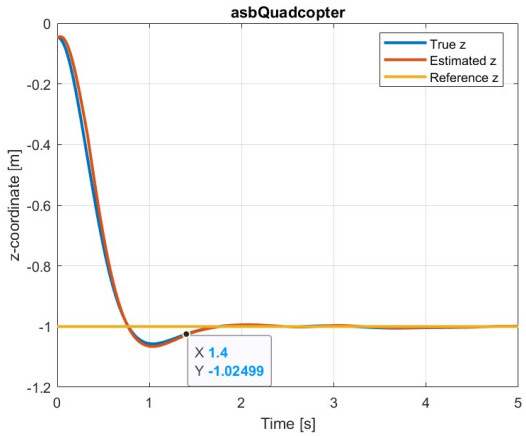
TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	0
5	0	0	0

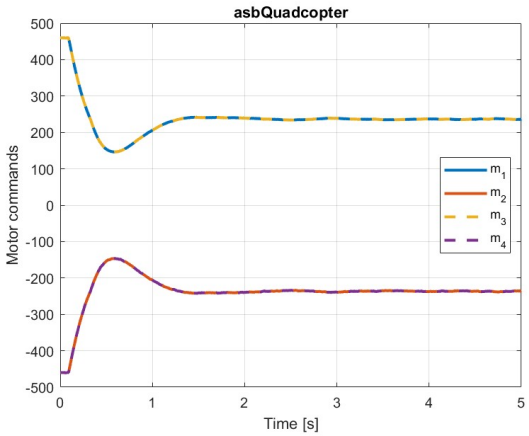
5.4.3 Simulating with the Default Controller

The default controller (the one in the prebuilt model) for altitude is a PD controller (which is why I upgraded it to a PID controller in sections 5.4.4 and 5.4.5) with a proportional gain of `0.8` and a derivative gain of `0.3`:



Using the technique presented in subsection 5.1, we can simulate the performance of the quadcopter with the newly created "ConstantAltitude" scenario and obtain the following plots:



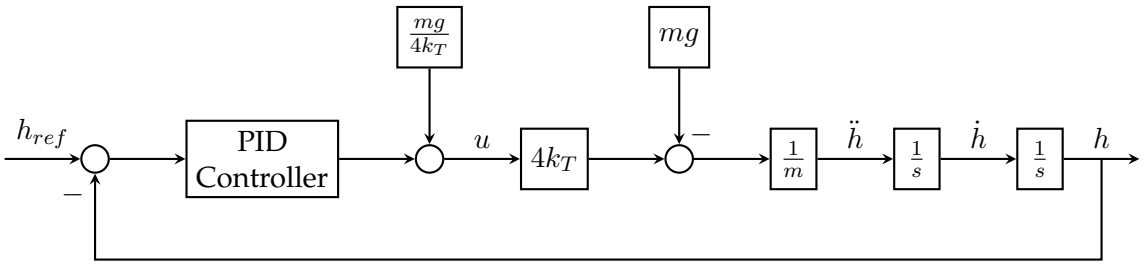


The altitude response with the default controller is characterized by a noticeable overshoot, slow speed of response, and unsaturated motor commands. The plot on the left shows that the altitude response with the default controller is underdamped with an overshoot of less than 10%. The 97.5%

settling time is about 1.4 s. The plot on the right shows that the motor commands remain unsaturated throughout the simulation (the saturation lower and upper limits are 10 and 500, respectively). Sections 5.4.4 and 5.4.5 show altitude performances with a smaller overshoot, faster speed of response, but also more motor saturation.

5.4.4 Designing the Altitude Controller for the Simplified Model

I would like to upgrade the altitude controller from PD control to PID control. To facilitate altitude controller tuning, I worked with the much simpler linear model (used in EECS 460) for the altitude dynamics (based on Newton's second law of motion): $m\ddot{h}(t) = 4k_T u(t) - mg$, where m is the quadcopter mass, h is the quadcopter altitude, k_T is the thrust coefficient, u is the control output (a single motor command), g is the gravitational constant, and t is time. Below is the block diagram for this simplified model.



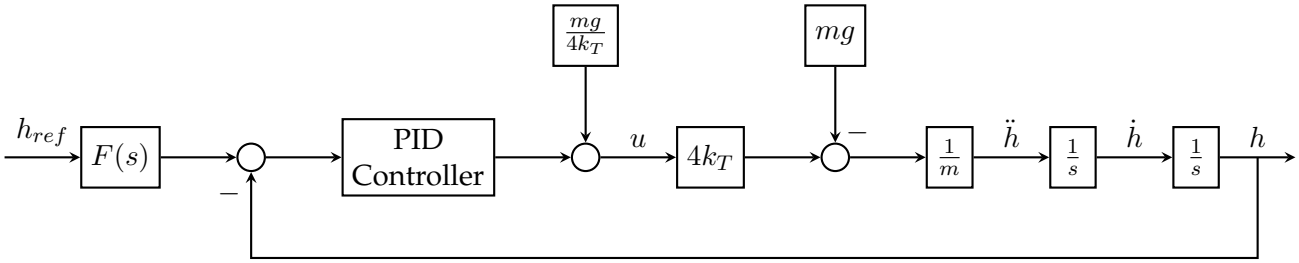
A couple of comments are in order before we proceed. First, note that there is no saturation block in the simplified model; throughout the simulation phase (step 2 of our general strategy in section 3), we strive to achieve the best theoretical performance, and it turns out that the absence of saturation blocks in the simplified model does not lead to any significant unexpected behavior when simulating with the original model (which contains saturation blocks). Second, m can be set equal to `Vehicle.Airframe.mass` (which is `0.063`) and g corresponds to `g` in the original model (which is `9.81`). In EECS 460, k_T is experimentally derived to be `5.276e-4`, but in my experience, the exact value of k_T does not seem to have any significant impact on controller tuning and the altitude performance, as long as it is sufficiently large (for example, greater than or equal to `4e-4`).

Now, for the simplified model, the PID control law I considered takes the following form:

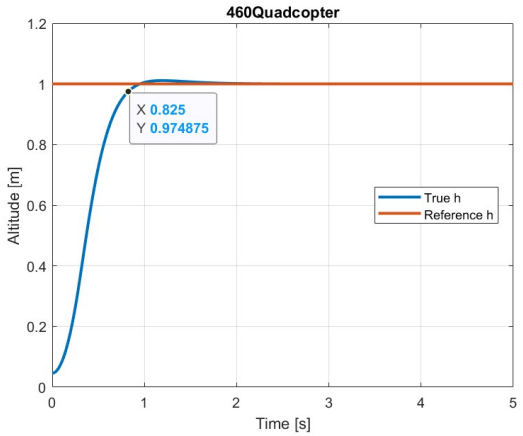
$$u(t) = K_i \int_0^t \left[h_{ref}(\tau) - h(\tau)\right] d\tau + K_p \left[h_{ref}(t) - h(t)\right] - K_d \dot{h}(t) + \frac{mg}{4k_T},$$

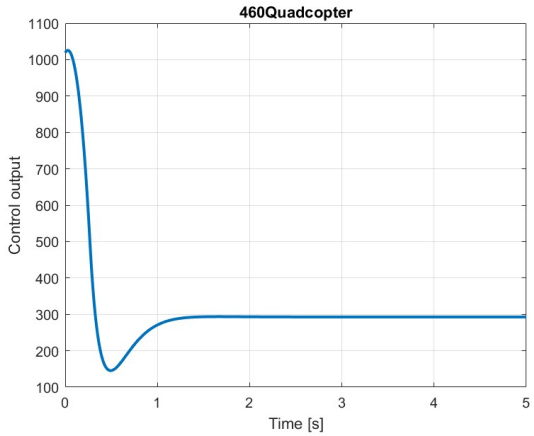
where h_{ref} is the reference altitude and K_p, K_i, and K_d are the proportional, integral, and derivative gains to be determined.

I derived the continuous-time closed-loop transfer function $\frac{K_p s+K_i}{\frac{m}{4k_T}s^3+K_d s^2+K_p s+K_i}$ and hand-tuned the PID controllers by placing the poles at arbitrary locations. However, when I simulated the resulting system (in a separate Simulink model I created), the altitude and control-input performances would resemble sinusoidal signals with exponentially growing amplitudes. This was due to the left-half-plane zero (at $-\frac{K_i}{K_p}$) of the resulting closed-loop transfer function. In order to solve the issue, we could introduce a precompensator $F(s)$, which would result in the following model:



The precompensator $F(s)$ can be designed to either get rid of the left-half-plane zero or shift the left-half-plane zero to the left (i.e. make it more negative). I experimented with both approaches. The former approach would produce a strictly overdamped altitude response, while the latter would produce a slightly underdamped response. I eventually decided on the latter approach, because the slightly underdamped response has an overshoot of only about 1% but a speed (measured by the 97.5% settling time) about 43% faster than that of the overdamped response. For this approach, the precompensator is $F(s) = \frac{K_i}{K_p s+K_i}\left(1-\frac{s}{z^*}\right)$, where z^* is the desired location of the zero. This would result in the closed-loop transfer function $T_{h_{ref}\to h}(s) = \frac{K_i\left(1-\frac{s}{z^*}\right)}{\frac{m}{4k_T}s^3+K_d s^2+K_p s+K_i}$. Through trial and error, I found that the controller gains $K_p = \frac{75m}{4k_T}$, $K_i = \frac{125m}{4k_T}$, and $K_d = \frac{15m}{4k_T}$ (which would result in a triple pole at -5), together with the new zero placed at $z^* = -4.5$, would yield approximately the best performance (in terms of the speed of response and amount of overshoot) for the simplified altitude model:





Compared to the performance in section 5.4.3, this new altitude performances has a smaller over-

shoot, faster speed of response, but also more motor saturation. As shown in the plot on the left, the 97.5% settling time is about 0.825 s. The overshoot is roughly 1%. On the other hand, the control output u is above 500 for almost 0.3 s, but this does not lead to any significant unexpected behavior when simulating with the original model (which contains saturation blocks).

5.4.5 Upgrading the Altitude Controller in the Original Model

The first step of upgrading the altitude controller in the original model is to determine the controller gains. Through careful comparison and contrast between the simplified and original models for the altitude dynamics, I found the relation between the controller gains of the two models for altitude control. It turns out that if we set the controller gains in the original model equal to the corresponding controller gains in the simplified model multiplied by $4k_T$, then the two models would yield approximately the same altitude performance. Thus, the proportional, integral, and derivative gains in the original model would be `K_p = 75 * Vehicle.Airframe.mass`, `K_i = 125 * Vehicle.Airframe.mass`, and `K_d = 15 * Vehicle.Airframe.mass`, respectively.

There is one more obstacle in the way of upgrading the altitude controller in the original model. The simplified model is a continuous-time model, whereas the original model is a discrete-time model. This means that we need to determine the discrete-time equivalent of the precompensator we designed in the previous subsubsection. After studying a continuous-to-discrete-transformation tutorial, I came up with the following code for performing the transformation using the Tustin Approximation: (Note that the function `syms` requires the installation of the Symbolic Math Toolbox beforehand.)

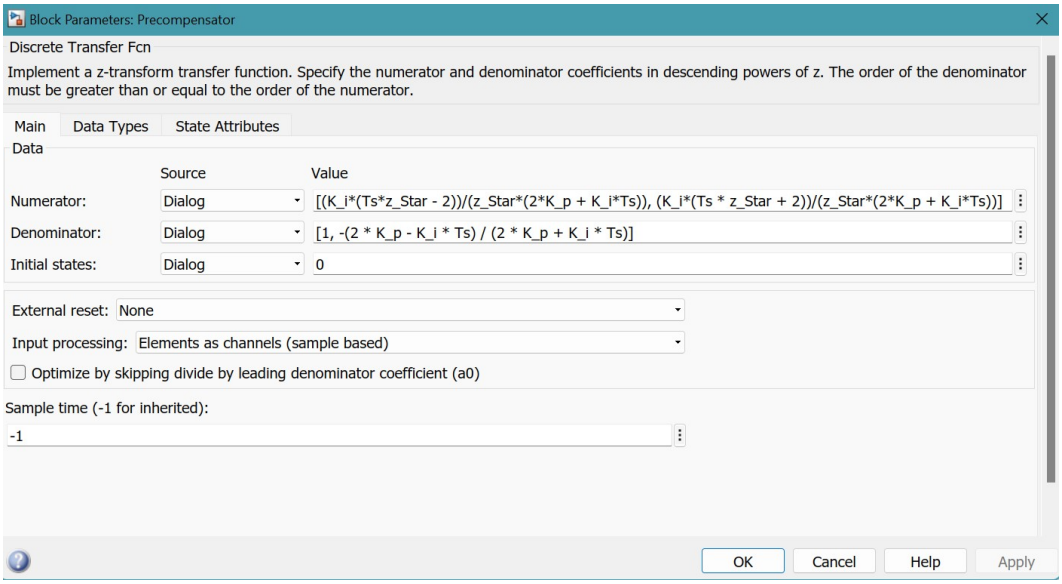
```
syms s z Kp Ki T zStar;
% continuous-time transfer function of interest:
tf_cont = (-1 / zStar * Ki * s + Ki) / (Kp * s + Ki);

[num, den] = numden(subs(tf_cont, s, 2 / T * (1 - z ^ -1) / (1 + z ^ -1)));

num = flip(eval(coeffs(num, z)));
den = flip(eval(coeffs(den, z)));

num = num ./ den(1) % numerator of c2d(tf([-1 / zStar * Ki,  Ki], [Kp, Ki]), T, 'tustin')
den = den ./ den(1) % denominator of c2d(tf([-1 / zStar * Ki,  Ki], [Kp, Ki]), T, 'tustin')
```

In the code above, `Kp` and `Ki` represent the proportional and integral gains, respectively. `T` represents the sample time, and `zStar` represents the desired location of the new zero. I deliberately named these symbolic variables in a way such that they would not overwrite the corresponding numerical variables (such as `Ts` for the sample time) in the original model.

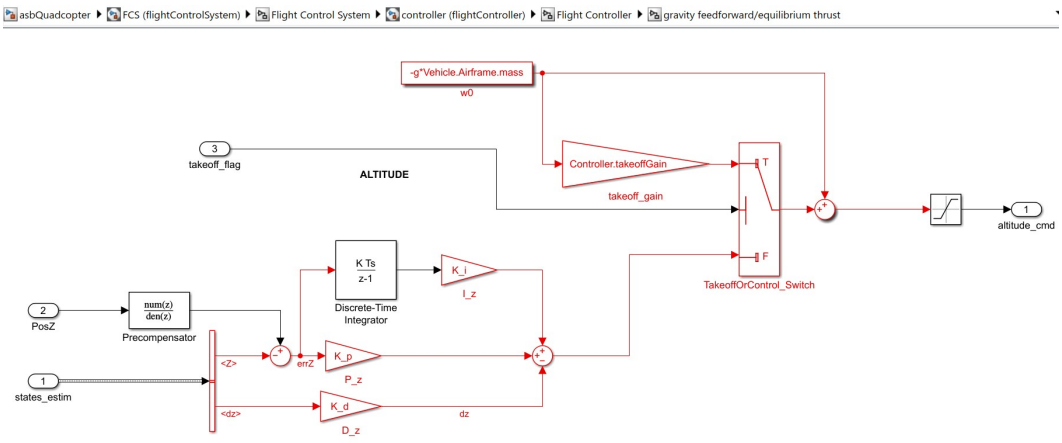
The outputs for lines 10-11 give the numerator and denominator of the discrete-time precompensator:

```
Command Window
>> num

num =

[(Ki*(T*zStar - 2))/(zStar*(2*Kp + Ki*T)), (Ki*(T*zStar + 2))/(zStar*(2*Kp + Ki*T))]

>> den

den =

[1, -(2*Kp - Ki*T)/(2*Kp + Ki*T)]
```

Using the variable `z_Star` for the desired location of the new zero, we can set up the precompensator for the original model by first adding a Discrete Transfer Fcn block, then copying the outputs into the numerator and denominator configurations, and finally replacing the symbolic variables with the numerical variables everywhere:

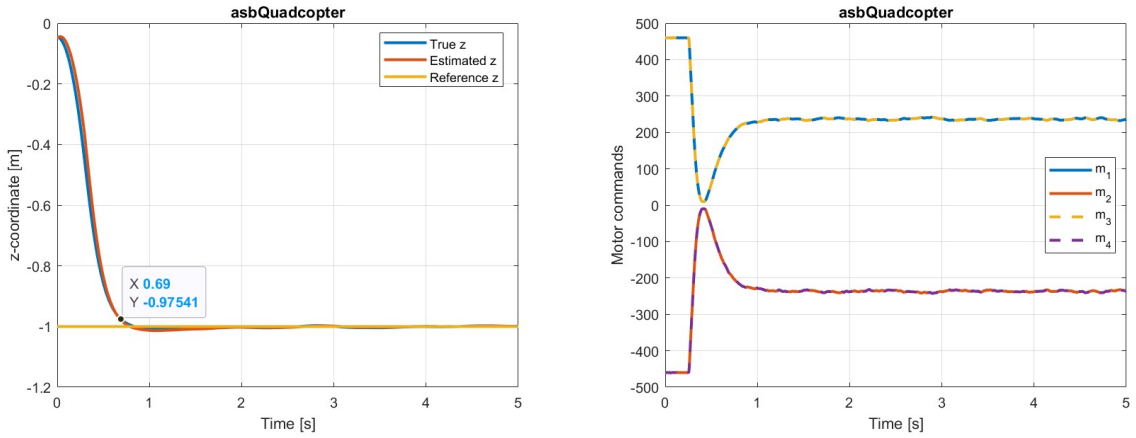


The final diagram for the upgraded altitude controller would thus look like the following:



(The Discrete-Time Integrator block does not need to be configured; leave the integration method as "Integration: Forward Euler", the gain value as 1.0, the initial condition as 0, and the sample time as -1.)

Finally, through trial and error, I found that a value of `z_Star = -3.8` would yield a very satisfactory altitude performance:



Compared to the performance in section 5.4.3, this new altitude performances has a smaller overshoot, faster speed of response, but also more motor saturation. With the upgraded altitude controller, the 97.5% settling time is about 0.69 s, and the amount of overshoot is about 1%. The price we pay is that the motor commands are saturated for much of the first 0.5 seconds of the flight. The plots above also show that the continuous-to-discrete transformation technique presented in this section is successful.

5.5 Controlling the Yaw

Out of the six states of interest (x, y, z, roll, pitch, and yaw), I consider yaw to be the second simplest state to control. Unlike altitude control, however, I did not work with any simplified model for the yaw dynamics.

5.5.1 Creating the *Constant-Altitude-and-Then-Constant-Yaw Scenario*

To help us evaluate the yaw performance, we will consider the scenario where throughout the simulation, the z reference is set equal to a nonzero constant, the yaw reference is first set equal to zero for a period of time but then set equal to a nonzero constant for the remainder of the simulation, and everything else is set equal to zero. We refer to such a scenario as a *constant-altitude-and-then-constant-yaw scenario*. For this subsubsection, we will create the scenario where the total simulation time is `10`, the constant z reference value is `-1`, the constant nonzero yaw reference value is `1` (which corresponds to about 57.3 degrees), and the initial period of time is `5`.

To create the scenario, I recommend duplicating the "ConstantAltitude" scenario, renaming the

duplicate scenario "ConstantAltitudeAndThenConstantYaw", and then editing the data points of the "ConstantAltitudeAndThenConstantYaw" scenario. The final data points should look like the following (these data points are consistent with the reference yaw signals the plots in sections 5.5.2 and 5.5.3):

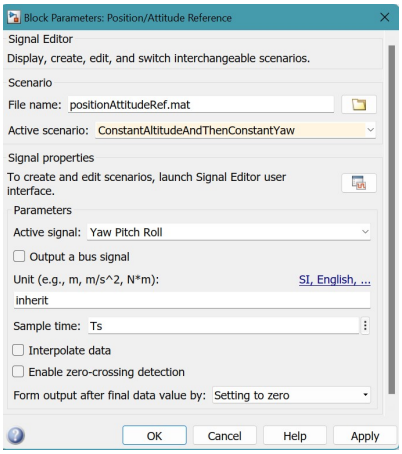
ConstantAltitudeAndThenConstantYaw.XYZ

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	-1
10	0	0	-1

ConstantAltitudeAndThenConstantYaw.Yaw Pitch Roll

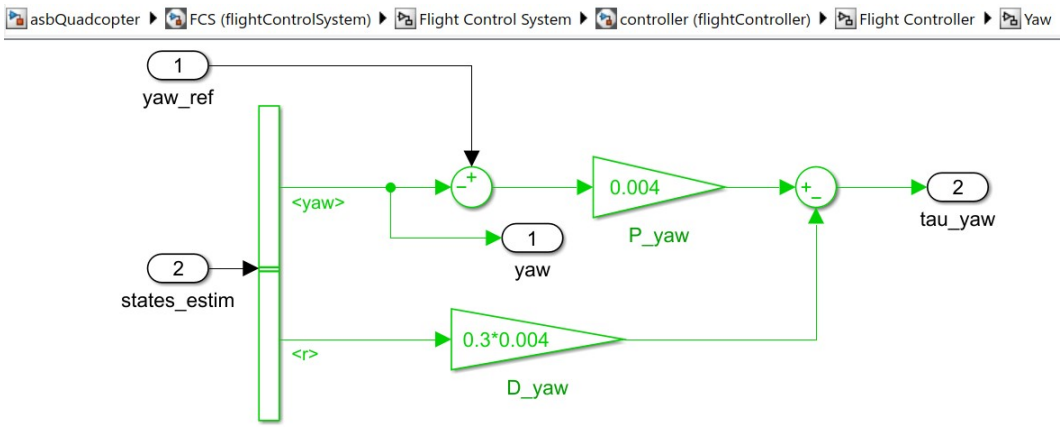
TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	0
5	1	0	0
10	1	0	0

Also, make sure that the box for "Interpolate data" is unchecked when "Yaw Pitch Roll" is selected as the active signal for this scenario:



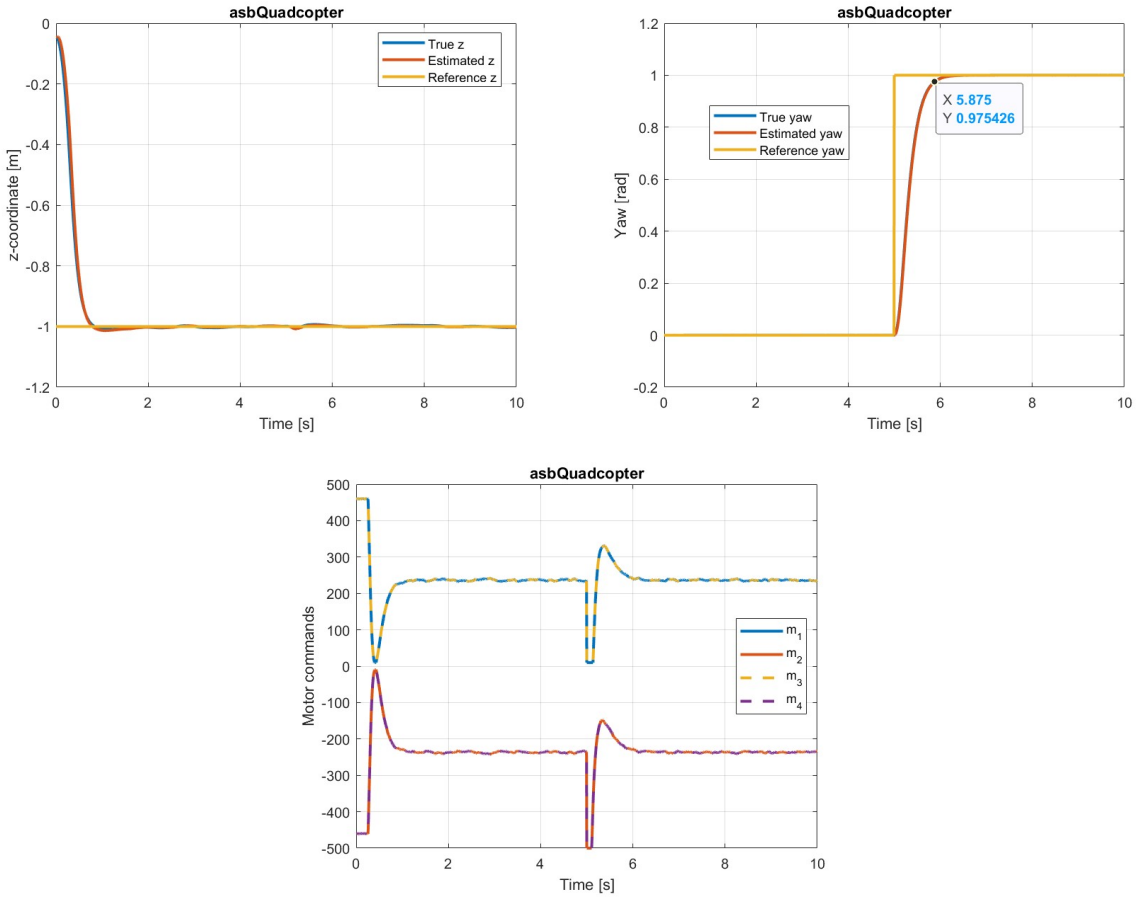
5.5.2 Simulating with the Default Controller

The default controller (the one in the prebuilt model) for yaw is a PD controller with a proportional gain of `0.004` and a derivative gain of `0.0012`:



Using the technique presented in subsection 5.1, we can simulate the performance of the quadcopter

with the newly created "ConstantAltitudeAndThenConstantYaw" scenario and obtain the following plots:

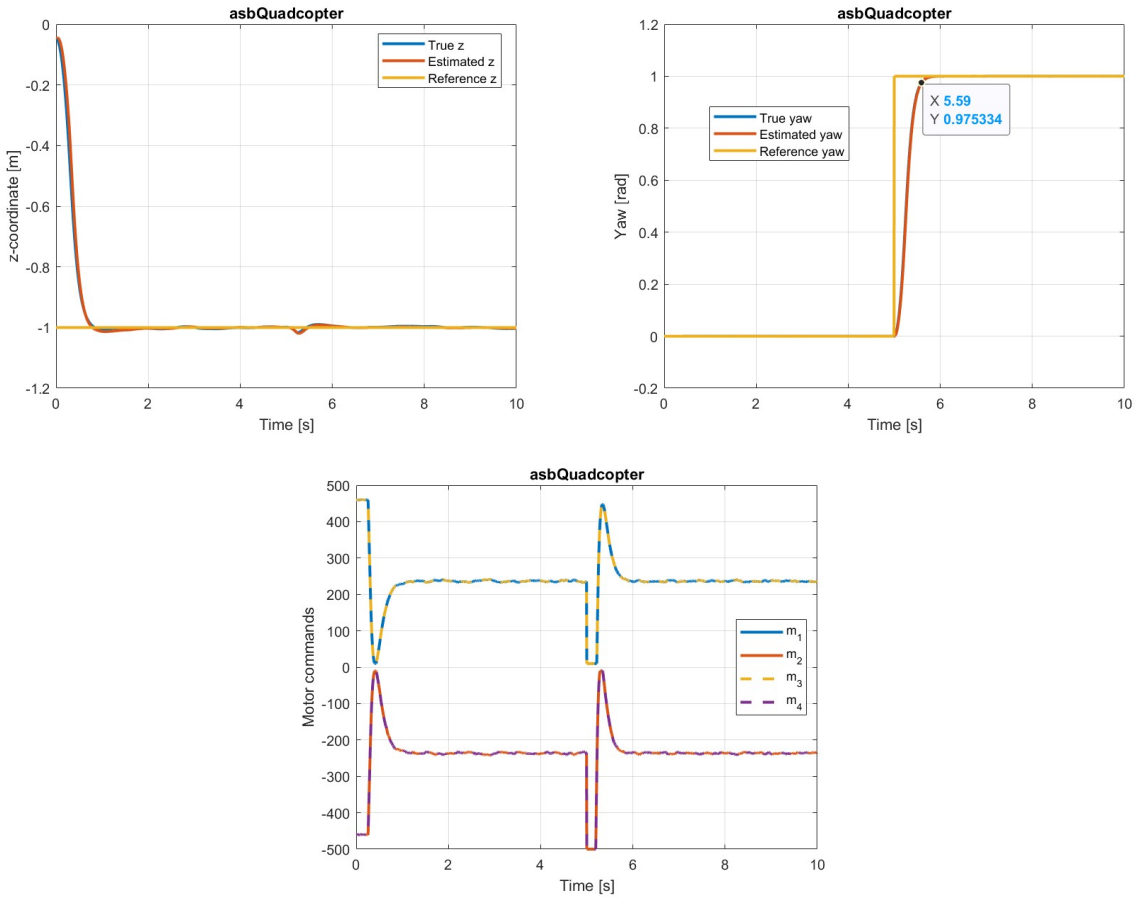


The yaw performance with the default controller is characterized by a slow speed of response and brief period of motor-command saturation. The first plot shows the success of the upgraded altitude controller in section 5.4.5: because of the presence of integral control in the upgraded altitude controller, we see that the true altitude is able to track the reference altitude with zero steady-state error. The second plot shows that the 97.5% settling time for yaw is about 0.875 s. The third plot shows the presence of saturation of motor commands for a brief period after 5 s (when we compare this performance with the one in the next subsubsection (section 5.5.3), we see that the upgraded yaw controller achieves a slightly faster speed of response).

5.5.3 Upgrading the Yaw Controller

Since the yaw response with the default controller appears to be largely satisfactory already (in terms of the speed of response and the amount of overshoot), I decided to keep things simple and simply hand-tune the existing proportional and derivative controllers (instead of adding an integral controller and tuning the three of them).

With a proportional gain of `0.01` and a derivative gain of `0.0018`, we can achieve a slightly faster speed of response (a 97.5% settling time of about 0.59 s) for the yaw performance:



5.6 Controlling the Pitch and Roll

The complexity of dealing with pitch and roll is greater than that with altitude and yaw; whereas we would intuitively expect changes in altitude and yaw to have relatively trivial impacts on the other four states of interest, changes in roll and pitch are always physically associated with changes in x- and y-coordinates.

5.6.1 Creating Two Scenarios

To help us evaluate the pitch performance, we will consider the scenario where throughout the simulation, the z reference is set equal to a nonzero constant, the pitch reference is first set equal to zero for a period of time but then set equal to a nonzero constant for the remainder of the simulation, and everything else is set equal to zero. We refer to such a scenario as a *constant-altitude-and-then-constant-pitch scenario*.

Similarly, the *constant-altitude-and-then-constant-roll scenario* is one where throughout the simulation, the z reference is set equal to a nonzero constant, the roll reference is first set equal to zero for a period

of time but then set equal to a nonzero constant for the remainder of the simulation, and everything else is set equal to zero.

For this subsubsection, we will create the two scenarios where the total simulation time is `30`, the constant z reference value is `-1`, the constant nonzero pitch and yaw reference values are both `0.1` (which corresponds to about 5.73 degrees), and the initial period of time is `5`.

To create the scenarios, I recommend duplicating the "ConstantAltitudeAndThenConstantYaw" scenario twice, renaming the duplicate scenarios "ConstantAltitudeAndThenConstantPitch" and "ConstantAltitudeAndThenConstantRoll", and then editing the data points of the "ConstantAltitudeAndThenConstantPitch" and "ConstantAltitudeAndThenConstantRoll" scenarios. The final data points should look like the following (these data points are consistent with the reference pitch and roll signals in the plots in sections 5.6.3 and 5.6.4):

ConstantAltitudeAndThenConstantPitch.XYZ

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	-1
30	0	0	-1

ConstantAltitudeAndThenConstantPitch.Yaw Pitch Roll

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	0
5	0	0.1	0
30	0	0.1	0

ConstantAltitudeAndThenConstantRoll.XYZ

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	-1
30	0	0	-1

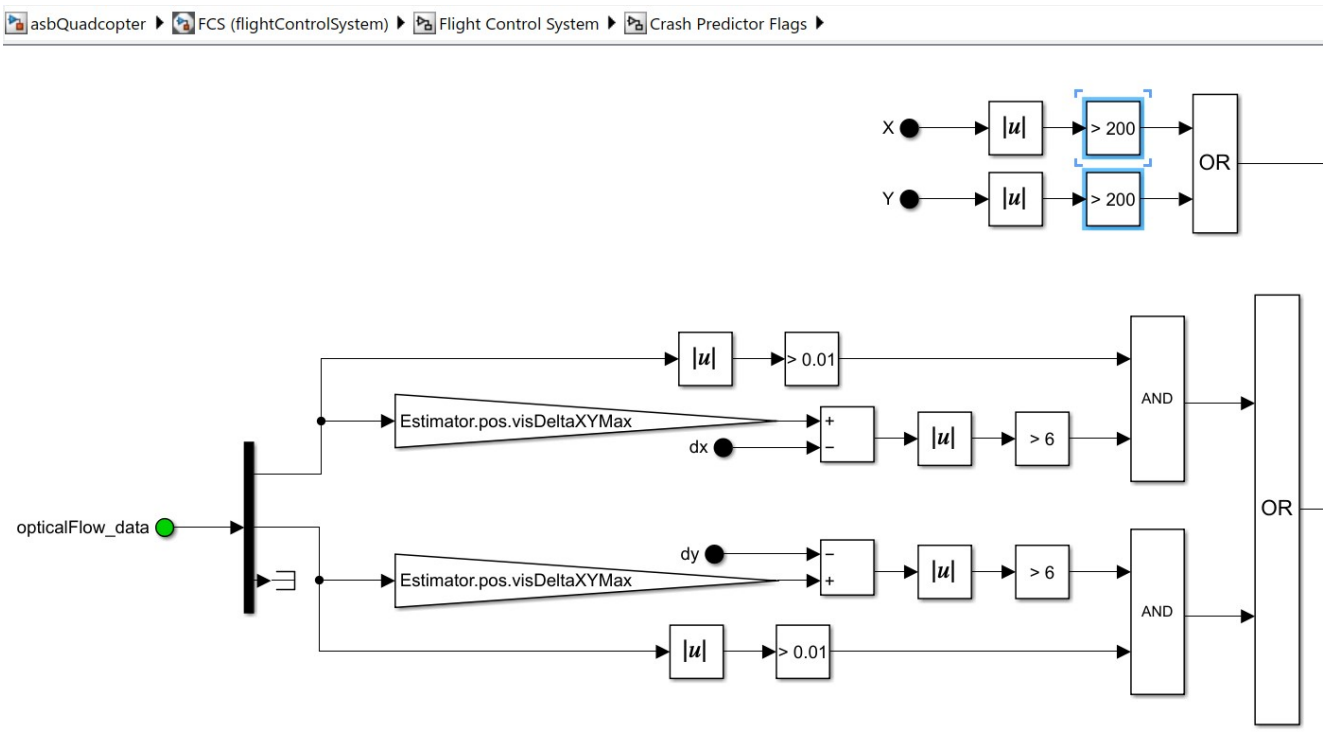
ConstantAltitudeAndThenConstantRoll.Yaw Pitch Roll

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	0
5	0	0	0.1
30	0	0	0.1

As with the constant-altitude-and-then-constant-yaw scenario, make sure that the box for "Interpolate data" is unchecked when "Yaw Pitch Roll" is selected as the active signal for both of the newly created scenarios.

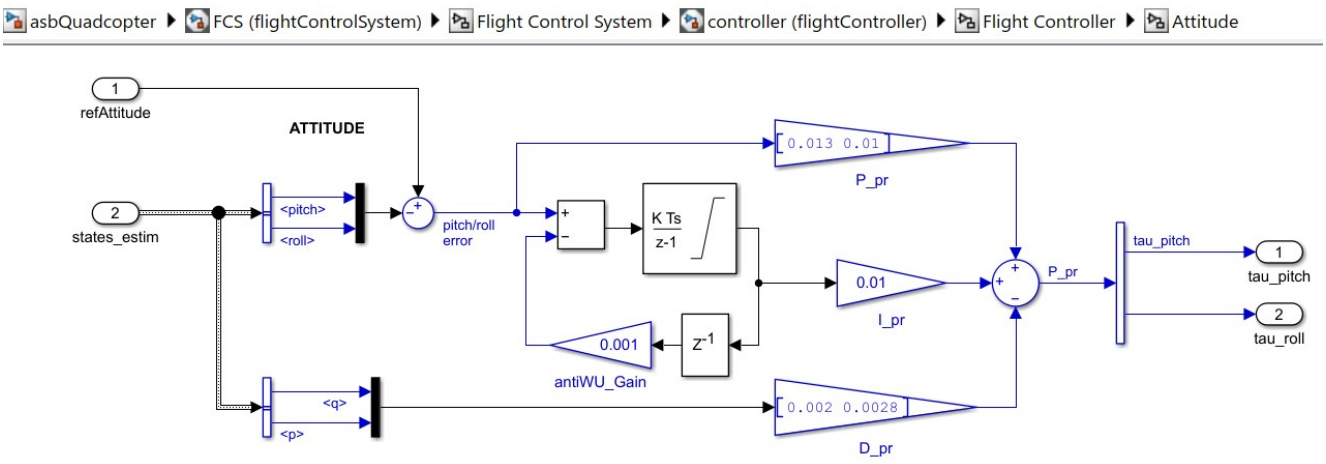
5.6.2 Extending the x and y Limits

As discussed in section 1, pitch and roll motions are accompanied by x and y movements, respectively. Since our prebuilt model has a mechanism to stop the simulation whenever the estimated x- or y-coordinate exceeds a certain limit, we should extend the limits before we simulate with the two scenarios. To do so, go to the "Crash Predictor Flags" subsystem and edit the constant values of the two Compare To Constant blocks (a value of `200` would be sufficient for the purpose of simulating with the two scenarios we just created):

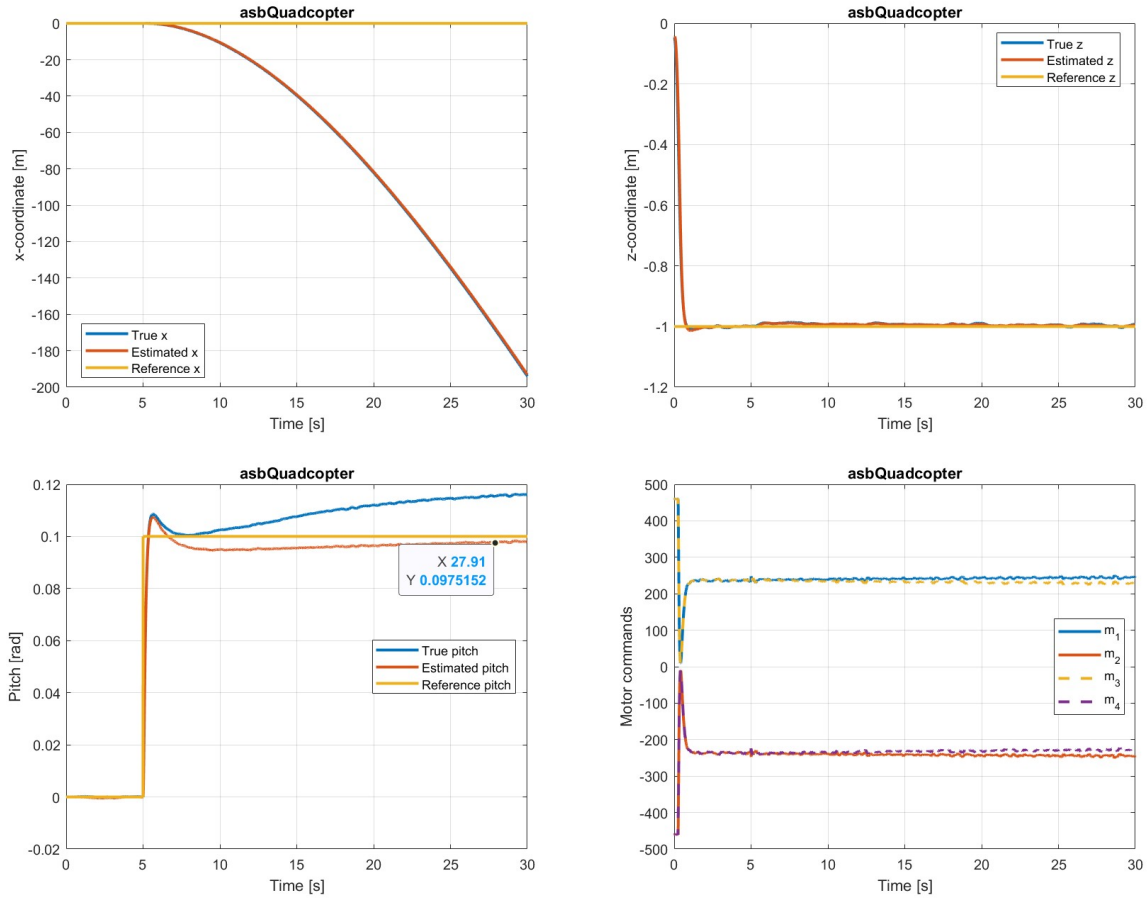


5.6.3 Simulating with the Default Controllers

The default controller (the one in the prebuilt model) for pitch is a PID controller with a proportional gain of `0.013`, an integral gain of `0.01`, and a derivative gain of `0.002`, and the default controller for roll is a PID controller with a proportional gain of `0.01`, an integral gain of `0.01`, and a derivative gain of `0.0028`:



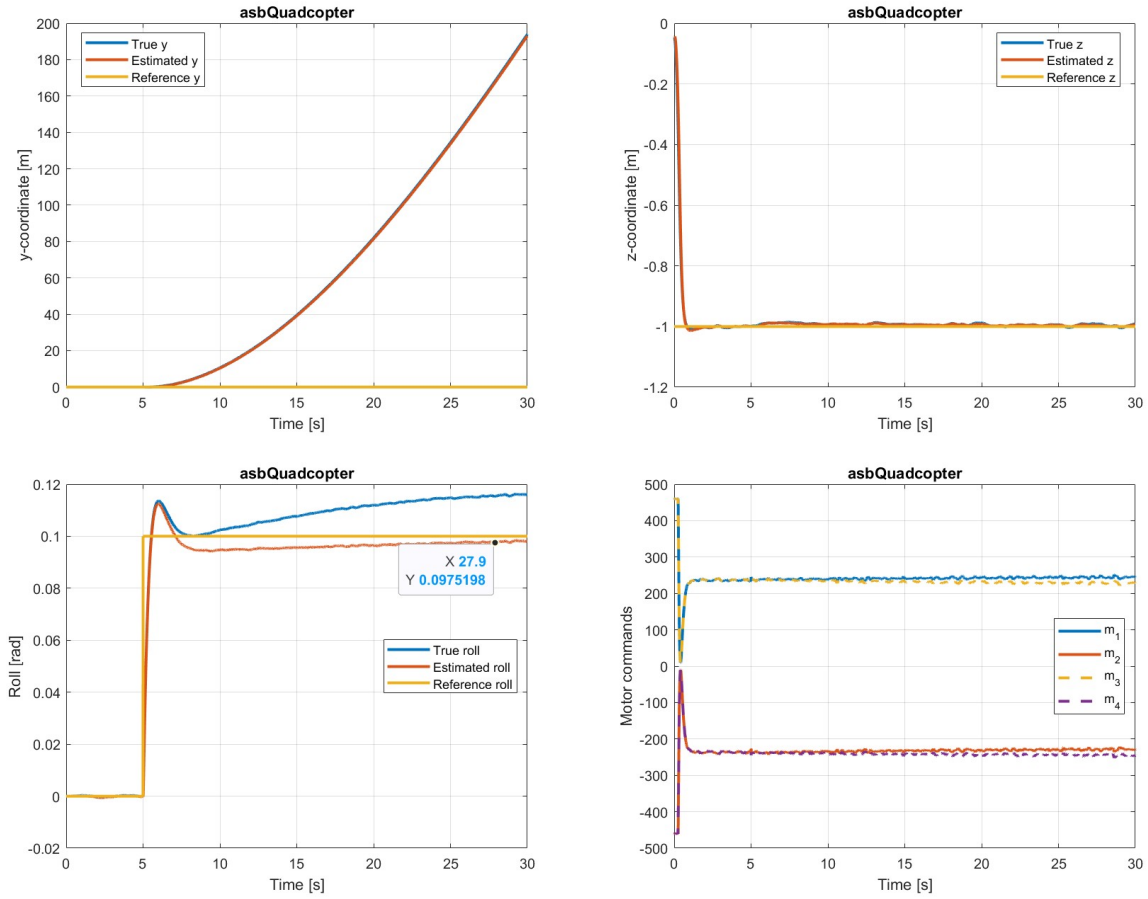
Using the technique presented in subsection 5.1, we can simulate the performance of the quadcopter with the "ConstantAltitudeAndThenConstantPitch" scenario and obtain the following plots: (Recall from figure 2 and figure 4 that when the reference pitch signal is nonzero, the commanded pitch is identical to the reference pitch, so, for simplicity, we may omit the commanded pitch when plotting.)



The above four plots illustrate the relation between pitch and x, demonstrate the success of the previously tuned altitude controller, and show the relatively insignificant impact a nonzero pitch command has on motor commands. Recall from section 1 that a positive pitch would decrease the x-coordinate, which is evident in the top left plot above. The top right plot shows that the true z is still able to track the reference z with zero steady-state error. Note, from the bottom left plot, that there appears to be a bias in the estimated pitch and the true pitch does not appear to even converge to the reference pitch. The 97.5% settling time for the estimated pitch is 22.91 s, and the amount of overshoot is almost 10%. Despite the pitch poor performance (in terms of the speed of response and amount of overshoot), the motor commands remain well within saturation limits starting from 5 s, as shown in the bottom right plot. Also, note that the bottom right plot makes sense qualitatively: according to section 1 and subsection 5.3, we would expect motor commands #3 and #4 to be smaller in magnitude than motor commands #1 and #2 starting from 5 s.

Similarly, the four plots below illustrate the relation between roll and y, demonstrate the success of the previously tuned altitude controller, and show the relatively insignificant impact a nonzero roll command has on motor commands.When we simulate the "ConstantAltitudeAndThenConstantRoll" scenario, we would expect the y-coordinate to increase, the z response to have zero steady-state error,

the estimated roll to differ from the true roll in a nontrivial manner, the settling time for the estimated roll to be very slow, and the motor commands to be well within saturation limits starting from 5 s. Moreover, according to section 1 and subsection 5.3, we would expect motor commands #1 and #4 to be greater in magnitude than motor commands #2 and #3 starting from 5 s. This is all confirmed in the following plots:

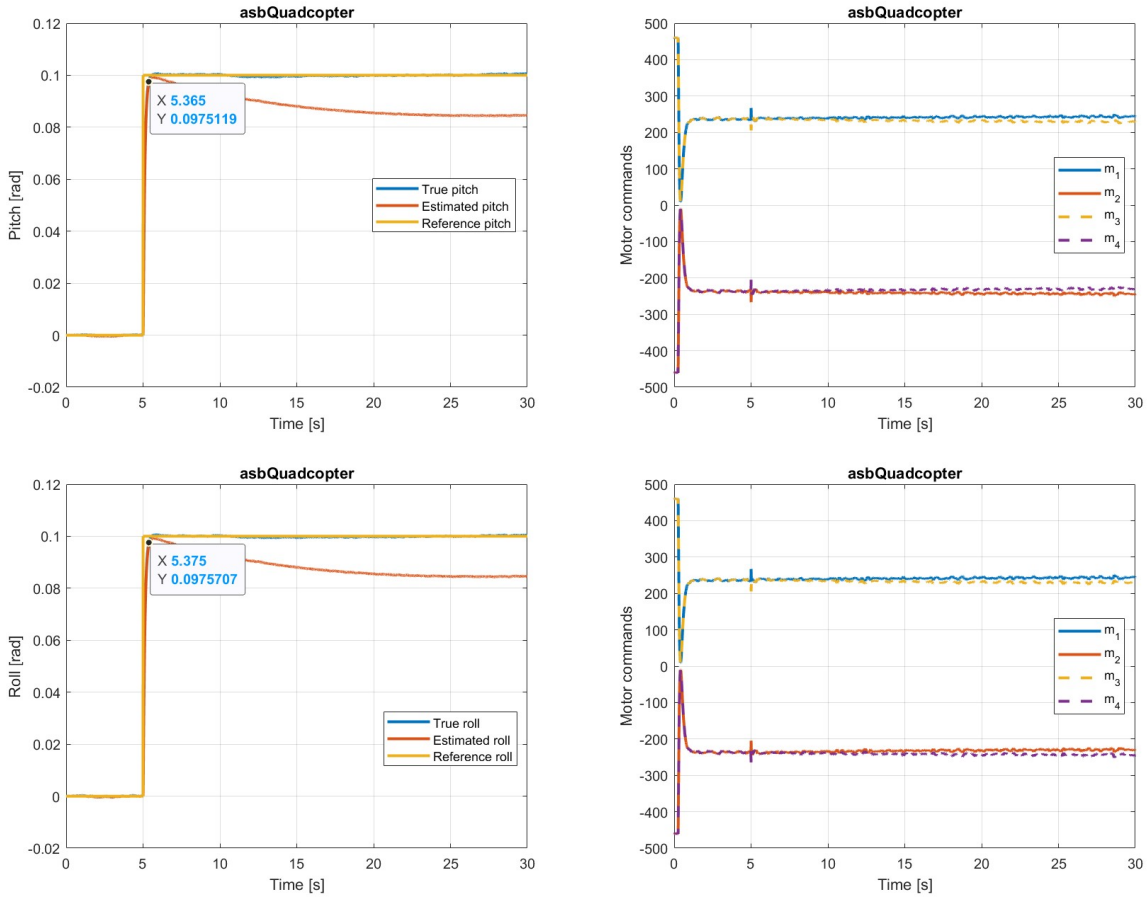


5.6.4 Upgrading the Pitch and Roll Controllers

The performances of the default pitch and roll controllers are far from ideal, but upgrading the two controllers is not at all an easy task. *Drone Simulation and Control, Part 5: Tuning the PID controller* demonstrates how to perform PID autotuning in Simulink. I spent a lot of time trying this technique for the pitch and roll controllers, but none of my attempts resulted in a success. In the end, I settled for hand-tuning the two controllers.

The four plots below show that it is possible to build upgraded pitch and roll controllers that are clearly superior to the default ones; the speed of response is much faster and the amount of overshoot is much smaller, while the motor command profile is similar. With a proportional gain of `0.035`, an integral gain of `0.001`, and a derivative gain of `0.004` for both controllers, I was able to improve the pitch and roll performances as much as I could for the two aforementioned scenarios; I managed

to get the true pitch (or roll) to converge to the reference pitch (or roll) with a 97.5% settling time of about 0.37 s and an overshoot of about 0.5%. The motor commands also remain well within saturation limits starting from 5 s. Below, the top two plots are for the "ConstantAltitudeAndThenConstantPitch" scenario and the bottom two plots are for the "ConstantAltitudeAndThenConstantRoll" scenario: (The plots for x, y, and z are all similar to the plots for the default controllers and are thus not shown.)



Two final remarks regarding the pitch and roll performances are in order. First, a case can be made for having the estimated pitch (or roll) instead of true pitch (or roll) to converge to the reference pitch (or roll); after all, the controller only has access to the estimated value, so it is fair to say that the controller's job is to have the estimated value converge to the true value. However, in the spirit of this entire simulation phase (step 2 of the general strategy discussed in section 3), I prioritized the true value over the estimated value for pitch and roll controller tuning; I aimed to achieve the best possible theoretical performance in the simulation phase.

Second, with the upgraded controllers, the performance of the pitch (or roll) controller is actually dependent on the reference value. A larger reference value (than 0.1) would result in a steady-state true value that is below the reference value, while a smaller reference value (than 0.1) would result in a steady-state true value that is above the reference value. This is because 1) I deliberately set the integral

gain to be small enough such that a nonzero steady-state difference always exists between the estimated value and the reference value and 2) as it turns out, the steady-state difference (in units of rad) between the estimated value and the true value appears to be independent of the reference value. Therefore, the upgraded controllers work best when the reference value is around 0.1 rad.

5.7 Controlling the Horizontal Movements

The x- and y-coordinates are the remaining states that our flight controller is designed to control (see figure 4).

5.7.1 Creating Two Scenarios

When evaluating the performance of any of the other four states (altitude, yaw, pitch, and roll), we invariably considered the step response. However, it turns out that the step reference is not necessarily the best reference for evaluating the performances of x- and y-coordinates; recall from section 1 that the change in x- or y-coordinates is initiated by the change in pitch or roll, so if a step reference is used for x (or y), the commanded pitch (or roll) will be very large initially, potentially leading to large overshoot and oscillations for the x (or y) response.

Through experience, I found that the following scenario would be appropriate for the purpose of evaluating the performance of x (or y): throughout the simulation, the z reference is set equal to a nonzero constant, the x (or y) reference is first set equal to zero for a period of time but then increases linearly from zero to a nonzero constant for a period of time and then remains at that nonzero constant for the remainder of the simulation, and everything else is set equal to zero. We refer to such a scenario as a *constant-altitude-and-then-linear-x scenario* (or *constant-altitude-and-then-linear-y scenario*).

For this subsubsection, we will create the two scenarios where the total simulation time is `20`, the constant z reference value is `-1`, the final nonzero reference value (for x or y) is `1`, the initial period of time is `5`, and the second period of time is `2`. This corresponds to the following data points (these data points are consistent with the reference x and y signals in the plots in sections 5.7.2 and 5.7.3):

ConstantAltitudeAndThenLinearX.XYZ

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	-1
5	0	0	-1
7	1	0	-1
20	1	0	-1

ConstantAltitudeAndThenLinearX.Yaw Pitch Roll

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	0
20	0	0	0

ConstantAltitudeAndThenLinearY.XYZ

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	-1
5	0	0	-1
7	0	1	-1
20	0	1	-1

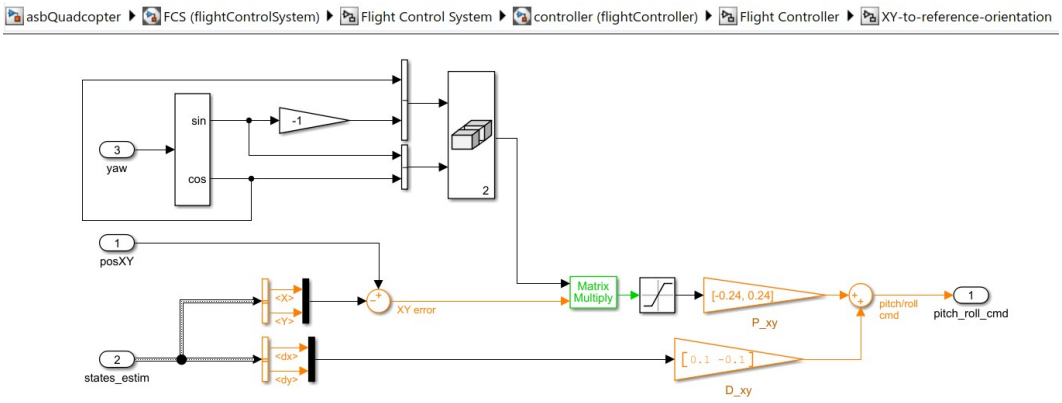
ConstantAltitudeAndThenLinearY.Yaw Pitch Roll

TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	0
20	0	0	0

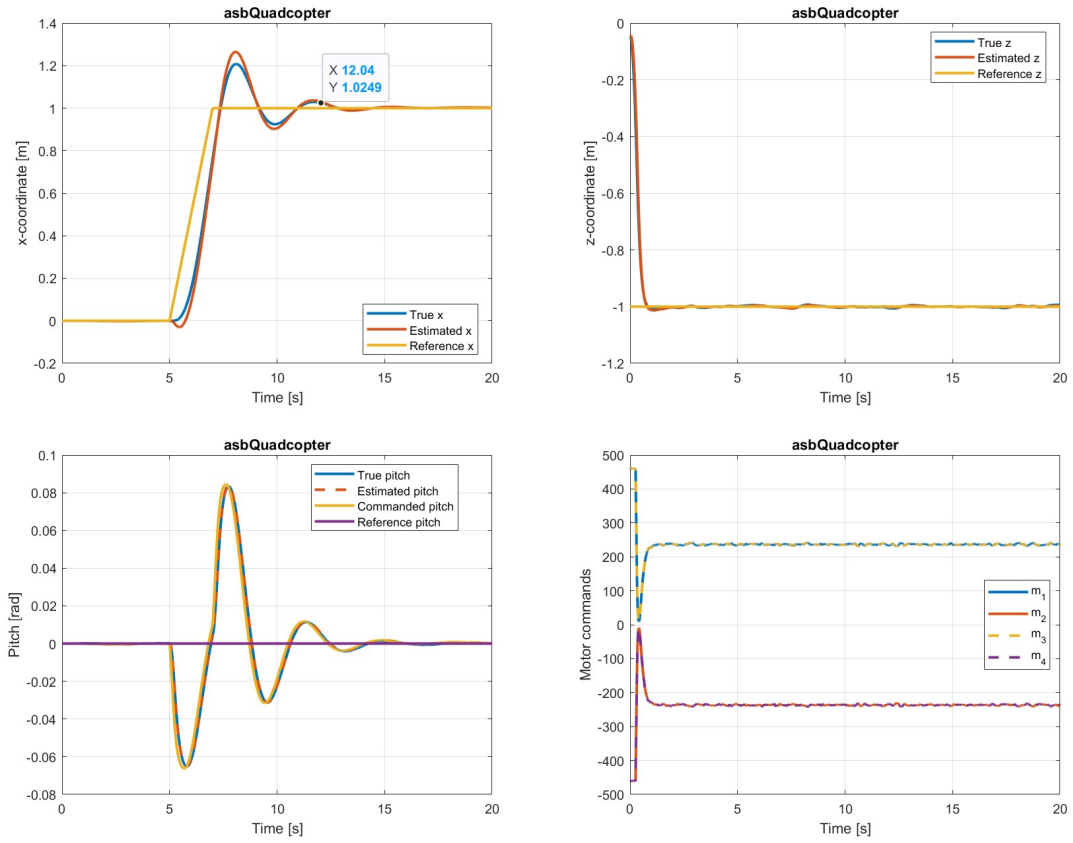
Since we want the x (or y) reference to increase linearly during the second period, for both of the scenarios we just created, we need to check the box for "Interpolate data" when "XYZ" is selected as the active signal.

5.7.2 Simulating with the Default Controllers

The default x and y controllers are PD controllers with the same magnitudes of the controller gains: a proportional gain magnitude of `0.24` and a derivative gain magnitude of `0.1`. The signs of the gains are determined by the physical dynamics; increasing x requires decreasing pitch but increasing y requires increasing roll, as discussed in section 1.

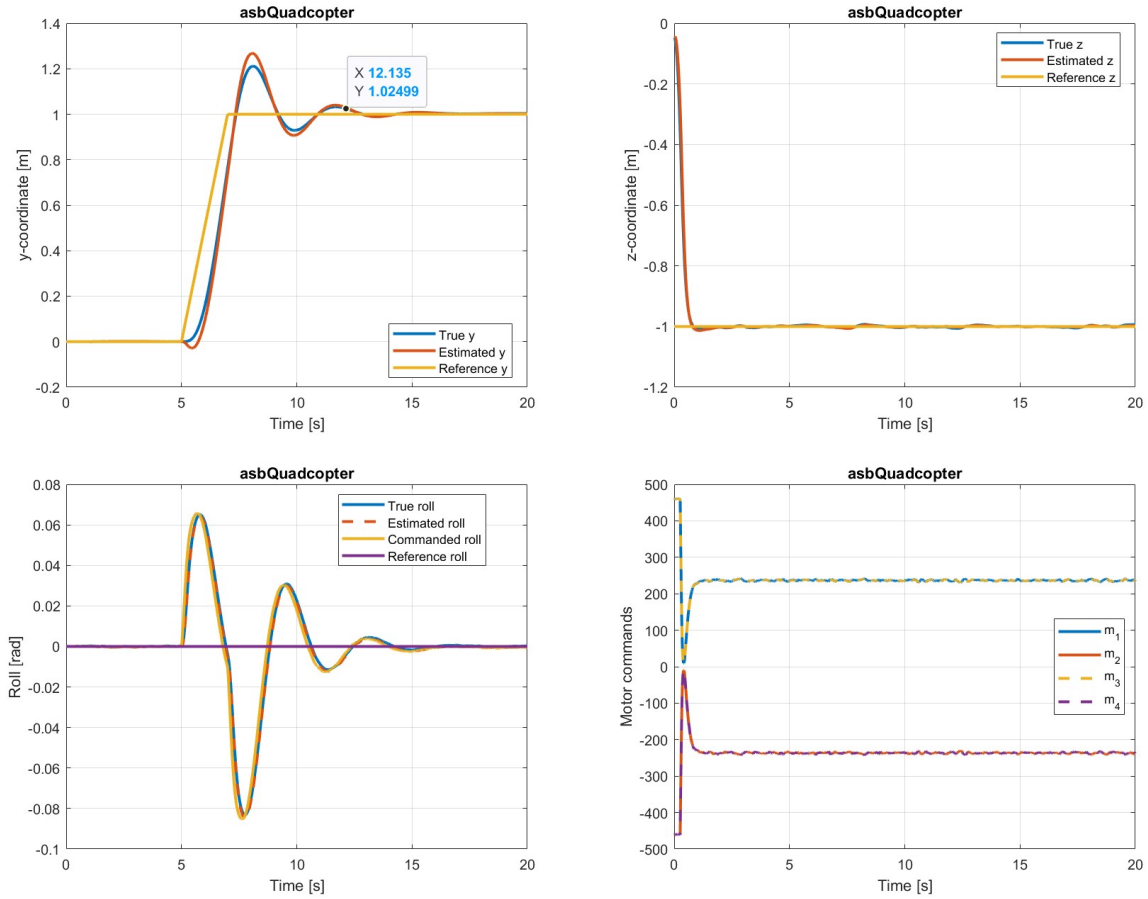


Using the technique presented in subsection 5.1, we can simulate the performance of the quadcopter with the "ConstantAltitudeAndThenLinearX" scenario and obtain the following plots:



In summary, the above four plots show the success of the previously tuned z and pitch controllers but also the need for the default x controller to be tuned (because of the large overshoot and slow speed of response). Starting from 5 s, the performances of z, pitch, and motor commands all appear to be perfectly desirable. The x response, however, is not necessarily ideal; the amount of overshoot is about 20% and it is not until 12.04 s that the x response reaches the steady state (by the standard of staying within 2.5% of the reference value). Since our goal in this scenario is to get the quadcopter to increase its x-coordinate from 0 to 1 (starting from 5 s), we can informally consider the reference x part of the control strategy and say that the settling time of the x response is about 7.04 s.

Similar statements can be made when we look at the plots for the "ConstantAltitudeAndThenLinearY" scenario. Specifically, the following four plots show the success of the previously tuned z and roll controllers but also the room for improvement for the default y controller (because of the large overshoot and slow speed of response):

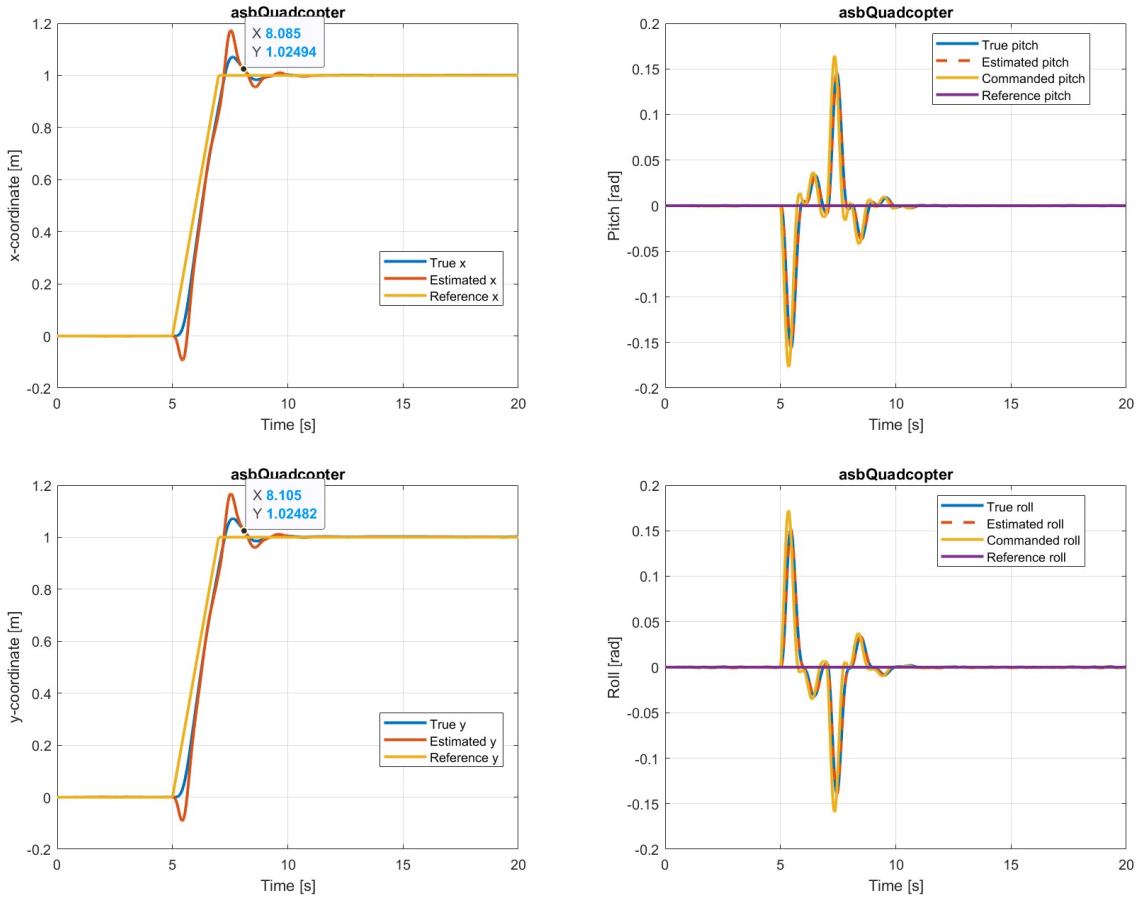


5.7.3 Upgrading the x and y Controllers

I tackled the tasks of improving the x and y responses from two perspectives. First, the duration of the linear increase of the reference signal can be tuned. Smaller durations lead to larger overshoots require longer times for the response to eventually reach steady state once it reaches the reference value, whereas longer durations by definition lead to longer times for the response to reach the reference value in the first place. When I started dealing with the subject of controlling the x- and y-coordinates, I did not know what the optimum duration should be. In fact, I started with the *constant-altitude-and-then-constant-x scenario* where the duration is zero. Through trial and error, I found that a duration of 2 seconds would lead to reasonable behavior of the response, which is why I suggested creating the scenarios where the reference signals reach the final reference values at 7 s in subsubsection 5.7.1.

Second, the controller gains for the x and y controllers can be tuned. According to the classical control theories in EECS 460, increasing the proportional gain generally leads to faster speed of response. I observed this exact phenomenon when tuning the x and y controllers. However, increasing the proportional gain also comes at the cost of increasing the overshoot. To alleviate the issue, we can increase the derivative gain. Increasing the derivative gain turns out to be extremely effective in reducing the overshoot, but at the same time we need to make sure the derivative gain is not large enough to cause

oscillations or unstable behavior. After many rounds of tuning the proportional and derivative gains, I decided that increasing the magnitude of the proportional gain from `0.24` to `0.52` and that of the derivative gain from `0.1` to `0.15` would lead to the best performance that I was able to achieve. Below, the top two plots are for the "ConstantAltitudeAndThenLinearX" scenario and the bottom two plots are for the "ConstantAltitudeAndThenLinearY" scenario: (The plots for z and motor commands are all similar to the ones for the default controllers and are thus not shown.)



The above four plots show that it is possible to achieve much better performances with the upgraded controllers (compared to the ones with the default controllers). With the upgraded controllers, the peak values of pitch and roll are a little larger (than those with the default controllers), but the responses of x and y have been significantly improved: for either response, the overshoot has been reduced from about 20% to about 7% and the settling time has been reduced from about 7.1 s to about 3.1 s. While this is still not quite comparable to the performances of the tuned altitude, yaw, pitch, and roll controllers, I believe that tuning x and y controllers is on a different complexity level than tuning the other controllers and thus the performances I achieved could be considered largely satisfactory (because of the drastically reduced amount of overshoot and much faster speed of response).

5.8 Testing with the *Constant-Altitude-and-Then-Alternate-x-y-Yaw Scenario*

To wrap up section 5, I will demonstrate the performance of the updated flight controller with the *constant-altitude-and-then-alternate-x-y-yaw scenario*. This scenario is defined to be one where throughout the simulation, the z reference is set equal to z nonzero constant, but the x, y, and yaw references take turns (for a total of two turns) to increase their reference values. Specifically, we will consider the scenario defined by the following data points:

ConstantAltitudeAndThenAlternateXYYaw.XYZ

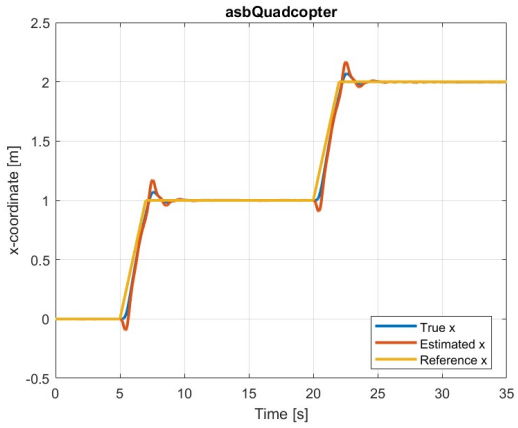
TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	-1
5	0	0	-1
7	1	0	-1
10	1	0	-1
12	1	1	-1
20	1	1	-1
22	2	1	-1
25	2	1	-1
27	2	2	-1
30	2	2	-1
35	2	2	-1

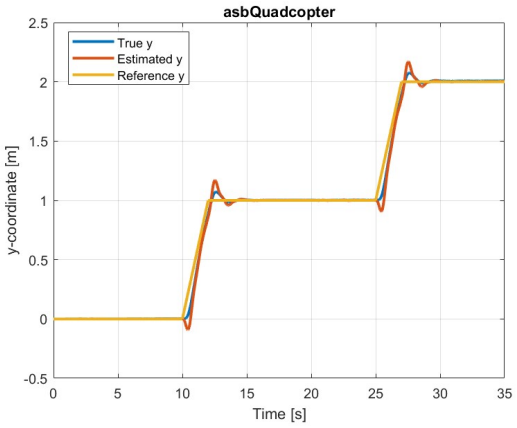
ConstantAltitudeAndThenAlternateXYYaw.Yaw Pitch Roll

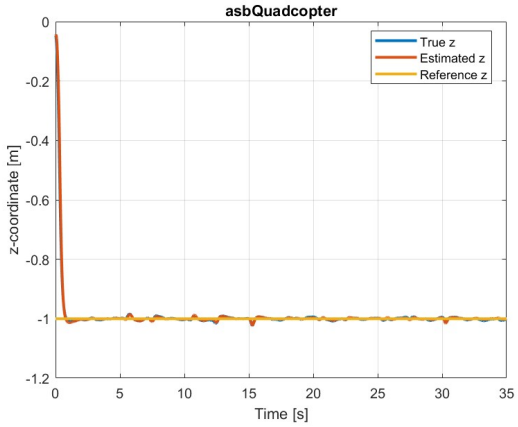
TIME	DATA(1,1,:)	DATA(2,1,:)	DATA(3,1,:)
0	0	0	0
15	1	0	0
30	2	0	0
35	2	0	0

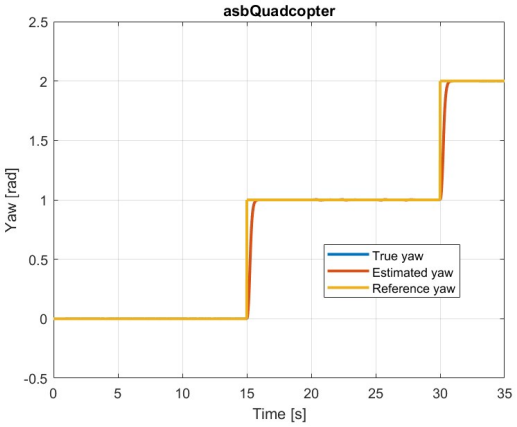
Essentially, if we compare a reference signal that we focus on as the "protagonist," then during the first 5 seconds, the z reference is the protagonist, and then from 5 s to 10 s, the x reference is the protagonist, and then from 10 s to 15 s, the y reference is the protagonist, and then from 15 s to 20 s, the yaw reference is the protagonist, and then from 20 s to 25 s, the x reference is the protagonist, and then from 25 s to 30 s, the y reference is the protagonist, and finally from 30 s to 35 s, the yaw reference is the protagonist.

Using the technique presented in subsection 5.1, we can simulate the performance of the quadcopter with this scenario and obtain the following plots:

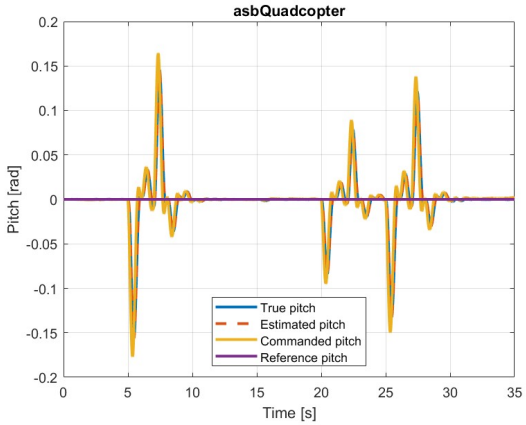


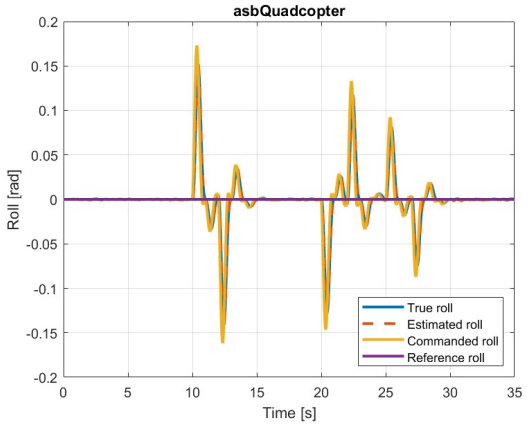


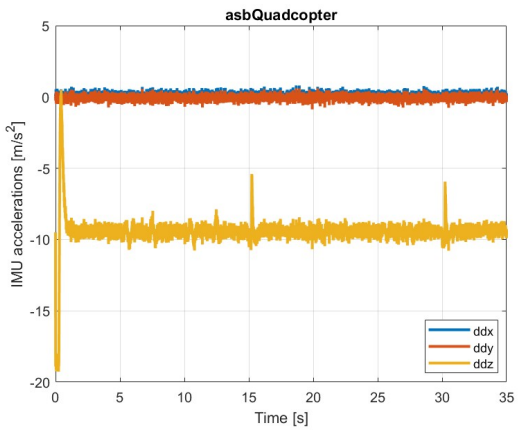


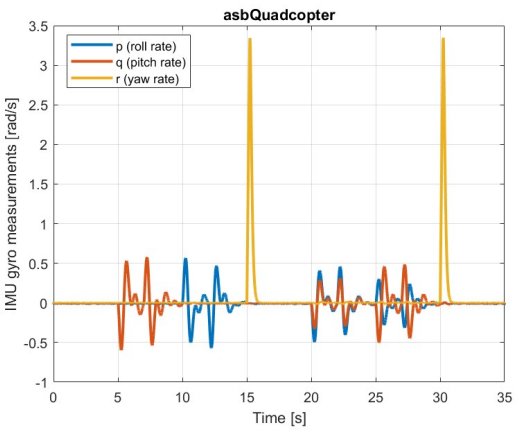


The above four plots show that the four states of interest are all successfully controlled as expected.







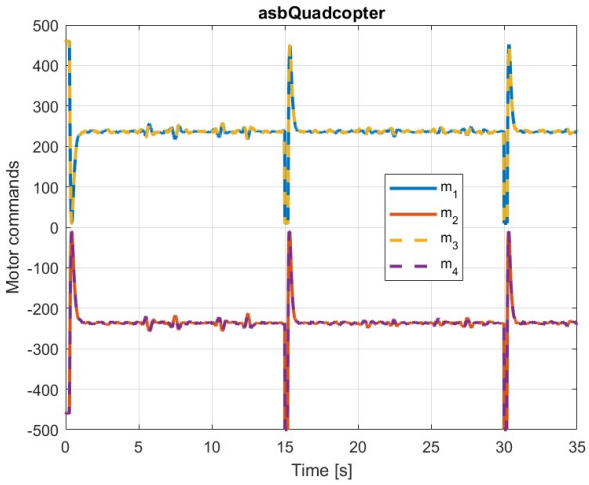


Overall, the above four plots show the success of the previously tuned pitch, roll, and yaw controllers and that the trends of the sensor measurements are very much expected.

Note from the top two plots that from 5 s to 10 s and from 10 s to 15 s, the commanded roll and the commanded pitch are zero, respectively, whereas from 20 s to 30 s, both are nonzero signals. This is because from 15 s to 20 s, the yaw of the quadcopter yaw has changed, meaning that the x- and y-directions in the body reference frame are no longer the same as those in the world reference frame starting from 20 s. Also, even though the difference between the true and estimated pitches or rolls is apparent in subsection 5.6, we see that when we use pitches and rolls as a means of changing x- and y-coordinates, the difference between the two is no longer significant and they both track the commanded value very well.

In the bottom left plot, it is interesting to see that despite the changes in x- and y-coordinates during the simulation, the IMU measurements of the accelerations in x- and y-directions appear to be roughly constant throughout the simulation, whereas the measured z-direction acceleration experiences a spike whenever the yaw of the quadcopter changes quickly.

In the bottom right plot, the measurements are very much expected. Due to the drastic changes in yaw at 15 s and 30 s, the measured yaw rate experience jumps at those times. The pitch and roll rates have nonzero values whenever the pitch and roll change, respectively.



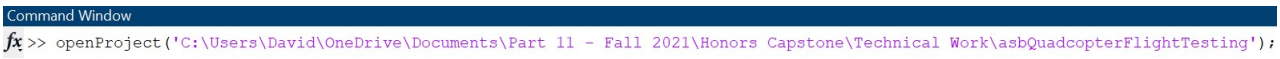
Finally, the profile of the motor commands are consistent with our expectations as well. We tuned the altitude and yaw controllers ambitiously enough to have the motor commands briefly saturate shortly after 0 s, 15 s, and 30 s. When the reference x or y changes, however, the motor commands still stay well within saturation limits.

As a whole, the simulation phase (step 2 of our general strategy in section 3) has been a complete success. All six controllers are tuned one at a time such that they achieve much better performances than the default controllers in the prebuilt model do. With the upgraded controllers, the motor commands

do occasionally saturate, but the simulation phase is very much theoretically oriented such that control theories can be applied without too much practical concern.

6 Flight Testing

Unfortunately, in my experience, even though the tuned controllers work extremely well in simulations, they could lead to catastrophic flight tests in practice. This is why I suggest starting with the default controllers for flight testing (this reiterates section 4.3):



6.1 Understanding How to Make Bluetooth Connections

The default firmware of Parrot Mambo (which can be downloaded from this page) is not designed for Bluetooth connections with computers, so we should first update the firmware (see appendix C for more information on the firmware). This firmware update only needs to be completed once. Once the quadcopter has the correct firmware, we can connect the quadcopter to the computer via Bluetooth. Bluetooth connection is a required step for each flight test. This official documentation covers how to perform firmware updates and make Bluetooth connections.

When I went through this procedure for the first time, I struggled to get my computer to detect the quadcopter in the first place (which is a prerequisite for updating the firmware). It turned out that the cables I tried were all intended for charging only; after obtaining a cable that was capable of both charging and transferring data, the issue was solved.

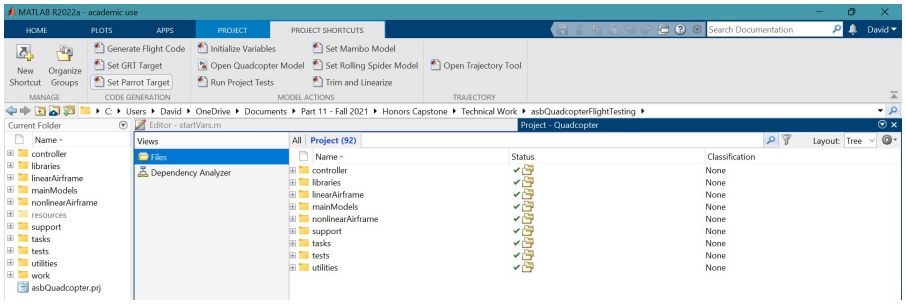
6.2 Procedure for Flight Testing and Postflight Analysis

6.2.1 Setting the Scenario of Interest

Before beginning a flight test, we should first manually set the scenario of the Signal Editor block in the "landing logic" subsystem to be the desired scenario. Note that we have to manually perform this step (i.e. double click the block, make the selection, and apply the change) in order to tell the flight code compiler the scenario of interest. For flight testing for the first time, I recommend creating and using the constant-altitude scenario with a conference z reference of `-0.7`.

6.2.2 Configuring the Models for Flight Code Generation

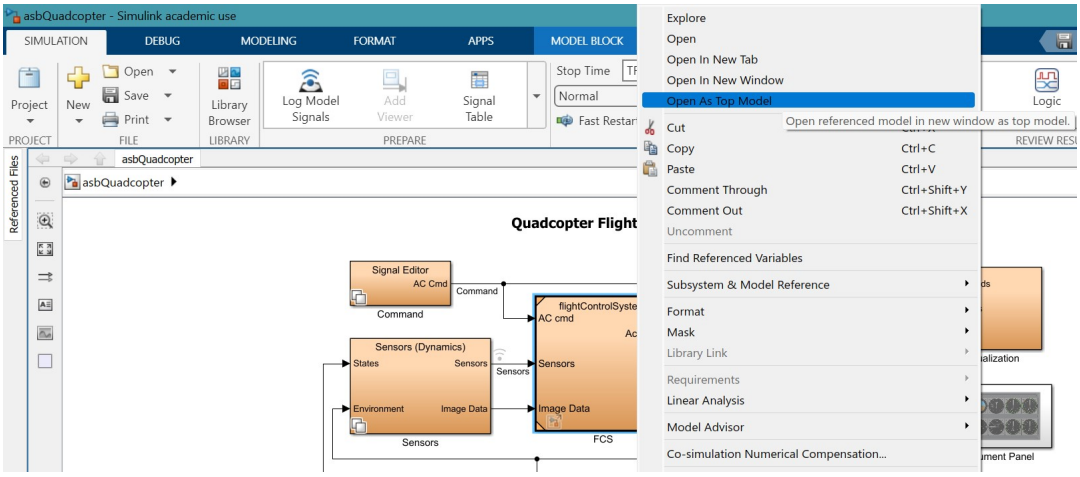
After deciding which scenario to use, we should configure the models for flight code generation by clicking on the "Set Parrot Target" button under the "PROJECT SHORTCUTS" tab:



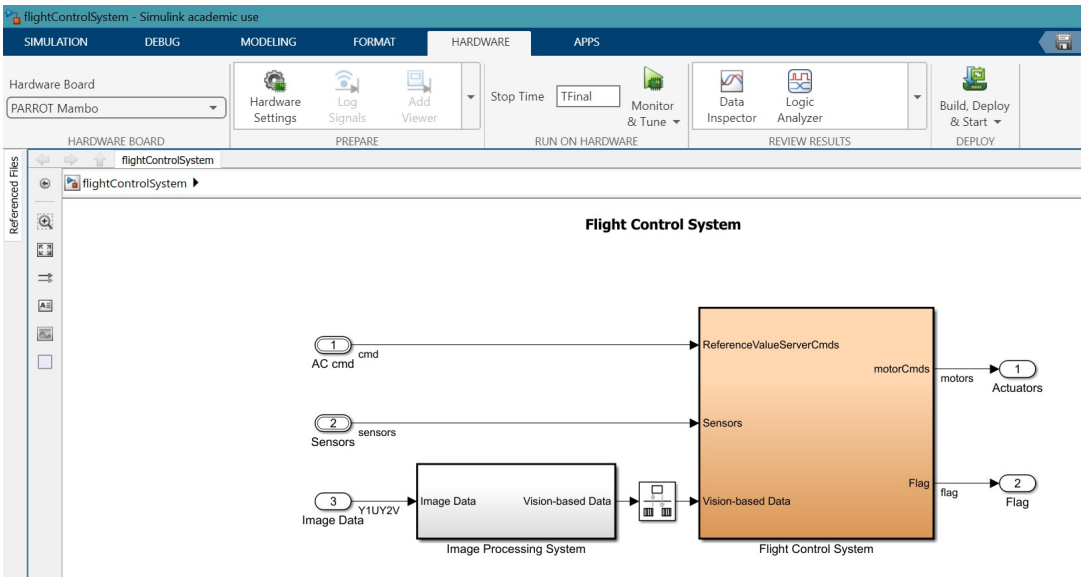
Note that this step is a one-time step; once the model is saved, closed, and reopened, we do not need to click on the "Set Parrot Target" button again if we want to perform another flight test.

6.2.3 Generating Flight Code

Next, we should right click and open "FCS" as the top model (this is a necessary step for every flight test):



Next, under the "HARDWARE" tab (which should show up if we installed the add-ons I mentioned before running `asbQuadcopterStart`, as discussed in subsection 4.1), click on the "Built, Deploy & Start" button to start the process of flight-code generation:



6.2.4 Completing the Flight Test

The rest is straightforward. While the flight code is being generated, we should connect the quadcopter to the computer via Bluetooth (in fact, if it was previously connected, I recommend disconnecting and reconnecting). Then, the "Parrot Flight Control Interface" should pop up, and we set the power gain to 100%, set the simulation time, start the flight, and download the flight data afterwards (all of these steps are intuitive to complete with the interface).

6.2.5 Making Plots for Postflight Analysis

Finally, once we have the flight data (an "RSdata.mat" file), the following code can be used for postflight analysis:

```
load('RSdata.mat');

t_real = rt_posref.time(1 : end, :);
posref_real = rt_posref.signals.values(1 : end, :);
estim_real = rt_estim.signals.values(1 : end, :);
motor_real = rt_motor.signals.values(1 : end, :);
sensor_real = rt_sensor.signals.values(1 : end, :);

plot(t_real, estim_real(:, 1), t_real, posref_real(:, 1), 'LineWidth', 2);
legend('Estimated x', 'Reference x', 'Location', 'best');
xlabel('Time [s]');
ylabel('x-coordinate [m]');
title('realQuadcopter');
grid on;
```

```
15
16 plot(t_real, estim_real(:, 2), t_real, posref_real(:, 2), 'LineWidth', 2);
17 legend('Estimated y', 'Reference y', 'Location', 'best');
18 xlabel('Time [s]');
19 ylabel('y-coordinate [m]');
20 title('realQuadcopter');
21 grid on;
22
23 plot(t_real, estim_real(:, 3), t_real, posref_real(:, 3), 'LineWidth', 2);
24 legend('Estimated z', 'Reference z', 'Location', 'best');
25 xlabel('Time [s]');
26 ylabel('z-coordinate [m]');
27 title('realQuadcopter');
28 grid on;
29
30 plot(t_real, estim_real(:, 6), t_real, posref_real(:, 8), 'LineWidth', 2);
31 legend('Estimated roll', 'Commanded roll', 'Location', 'best');
32 xlabel('Time [s]');
33 ylabel('Roll [rad]');
34 title('realQuadcopter');
35 grid on;
36
37 plot(t_real, estim_real(:, 5), t_real, posref_real(:, 7), 'LineWidth', 2);
38 legend('Estimated pitch', 'Commanded pitch', 'Location', 'best');
39 xlabel('Time [s]');
40 ylabel('Pitch [rad]');
41 title('realQuadcopter');
42 grid on;
43
44 plot(t_real, estim_real(:, 4), t_real, posref_real(:, 4), 'LineWidth', 2);
45 legend('Estimated yaw', 'Reference yaw', 'Location', 'best');
46 xlabel('Time [s]');
47 ylabel('Yaw [rad]');
48 title('realQuadcopter');
49 grid on;
50
51 plot(t_real, motor_real(:, 1), t_real, motor_real(:, 2), t_real, motor_real(:, 3), ...
52     t_real, motor_real(:, 4), 'LineWidth', 2);
53 legend('m_1', 'm_2', 'm_3', 'm_4', 'Location', 'best');
54 xlabel('Time [s]');
55 ylabel('Motor commands');
```

```
title('realQuadcopter');
grid on;

plot(t_real, sensor_real(:, 1), t_real, sensor_real(:, 2), t_real, sensor_real(:, 3), ...
    'LineWidth', 2);
legend('ddx', 'ddy', 'ddz', 'Location', 'best');
xlabel('Time [s]');
ylabel('IMU accelerations [m/s^2]');
title('realQuadcopter');
grid on;

plot(t_real, sensor_real(:, 4), t_real, sensor_real(:, 5), t_real, sensor_real(:, 6), ...
    'LineWidth', 2);
legend('p (roll rate)', 'q (pitch rate)', 'r (yaw rate)', 'Location', 'best');
xlabel('Time [s]');
ylabel('IMU gyro measurements [rad/s]');
title('realQuadcopter');
grid on;

plot(t_real, sensor_real(:, 8), 'LineWidth', 2);
xlabel('Time [s]');
ylabel('Pressure [Pa]');
title('realQuadcopter');
grid on;

plot(t_real, sensor_real(:, 7), 'LineWidth', 2);
xlabel('Time [s]');
ylabel('Ultrasonic altitude [m]');
title('realQuadcopter');
grid on;
```

6.3 Potential Sources for Performance Improvement

At the time of the completion of this report, while I have had some level of success of having the quadcopter take off and maintain at an altitude for a period of time, I am still working on improving its flight performance in general. I have identified many sources that could be potentially the keys to better performance:

1) updating the mass of the quadcopter

   - This refers to updating the value of the variable `Vehicle.Airframe.mass`

- The steady-state error of the altitude response can be used to calculate the updated value for the mass

2) changing the take-off duration

   - This refers to changing the value of the variable `takeOffDuration`
   - This is related to the transient behavior of the quadcopter

3) changing the take-off gain

   - This refers to changing the value of the variable `Controller.takeoffGain`

4) tuning the controllers

   - The altitude controller (PD controller) could be tuned or even upgraded to a PID controller
   - The pitch and roll controllers could be tuned to reduce the oscillations of the quadcopter in the air

5) updating the state estimation logic

   - Simulink Gain blocks could be introduced if the estimates of certain quantities are consistently overestimates
   - Some kind of special behavior may be introduced for certain state estimates during the transient period (for example, by modifying or even hardcoding state estimates during the first one second)

6) introducing a landing mechanism

   - This refers to setting `enableLanding` equal to `1` and modifying the landing mechanism in figure 2
   - This should not be a priority and may only be attempted once the quadcopter has achieved satisfactory performances in all the other aspects (including taking off and not drifting away)

7 Summary and Conclusion

In this document, a qualitative introduction to the quadcopter control problem is given. Extremely detailed instructions on how to use MATLAB and Simulink to perform simulations of the quadcopter model are then given. Concrete examples of upgrading the controllers are also illustrated. Finally, the

procedure for sending flight code to the physical quadcopter is presented and the possible ways to improve the flight performance in practice are discussed.

Throughout the document, I strive to keep things concise and simple. My actual project experience was much more difficult than this document suggests. I did not find any shortcuts or comprehensive documentation (which I consider this one to be) for this Parrot Mambo project along the way, and I learned a lot of the technical procedures the hard way by curiosity and perseverance (especially the procedure for sending flight code). I made a lot more changes to the model than the ones I discussed in this document, but while drafting this document, I discovered a much cleaner way to perform a lot of the steps and decided it was better to keep things as simple as possible.

I look forward to continuing the work with the quadcopter (for at least another year or so). My immediate next steps include estimating the mass of the quadcopter and enhancing the smoothness of the take-off performance. After the altitude performance has been improved to my satisfaction, I will examine and improve the yaw, x, and y performances. If all things go as planned, I will follow through my initial capstone proposal and have the quadcopter complete a complex trajectory autonomously. Eventually, I am also interested in incorporating real-time image-processing capabilities into the quadcopter (see this website, which also contains video demos of quadcopter flight tests in practice, for the image-processing algorithms I have already developed).

Appendix

A Clarification of the Meaning of *Active Signal*

In subsubsection 4.2.1, I discussed the need to select "Yaw Pitch Roll" as the *active signal* before configuring the sample time. The exact meaning of this term was a source of confusion for me when I started working on the project. For example, I thought that if we selected "Yaw Pitch Roll" as the active signal, then during the simulation, the other signal ("XYZ") would be disabled and effectively set equal to zero. However, this does not turn out to be the case at all. According to the MathWorks Support Team, the active-signal setting of a Signal Editor block is simply a mechanism that allows us to output different signals with different properties (such as different sample times and units). In other words, both the "XYZ" and "Yaw Pitch Roll" signals are "active" (in the sense that if they would certainly not be automatically set equal to zero) during the simulation even if only one of them is selected as the active signal for the Signal Editor block.

B Preventing Unintentional Live Script Simulation Output

I wrote most of my code of this project in the MATLAB live script environment. When a line of code in live script opens a specific window, however, an additional figure of the screenshot of that window may show up in the live script output. I personally find it distracting, so I will present the solution for preventing it in this section.

When opening the project, the unintentional output may look like the one in figure 5. The simple way to avoid this is to simply run the code in the Command Window instead, as described in subsection 4.1.

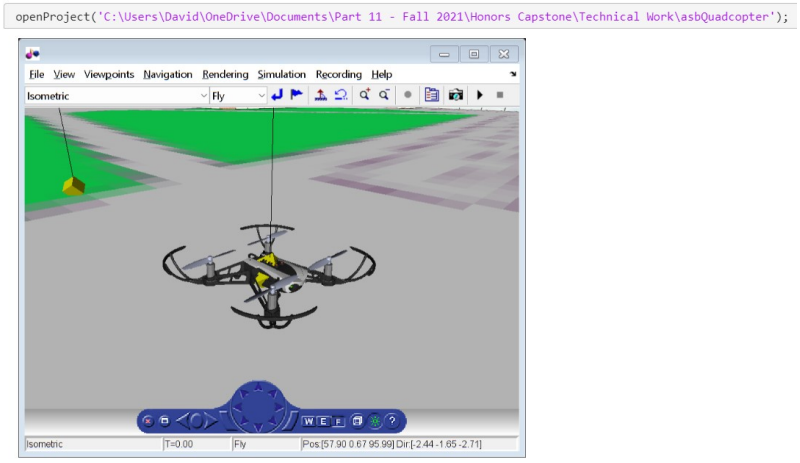


Figure 5: Unintentional live script simulation output when opening the project.

When programmatically running a simulation from live script, the unintentional figure output may look like figure 6. One effective and simple way to prevent this output is to comment out the Video Viewer block of the quadcopter camera, as shown in figure 7.

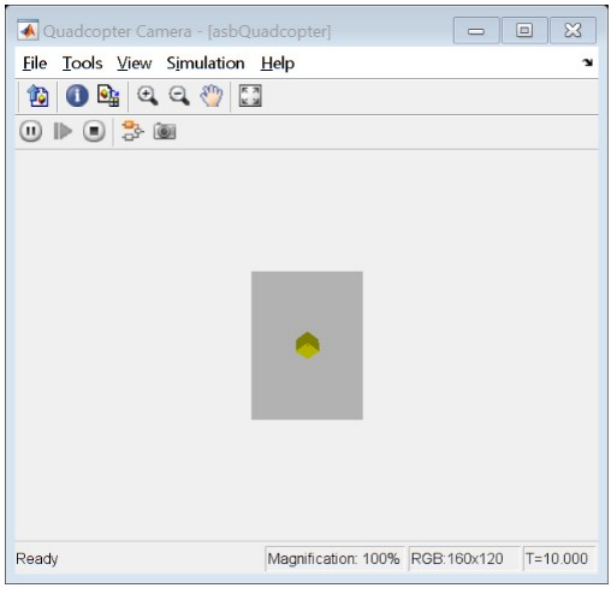


Figure 6: Unintentional live script simulation output when running a simulation.

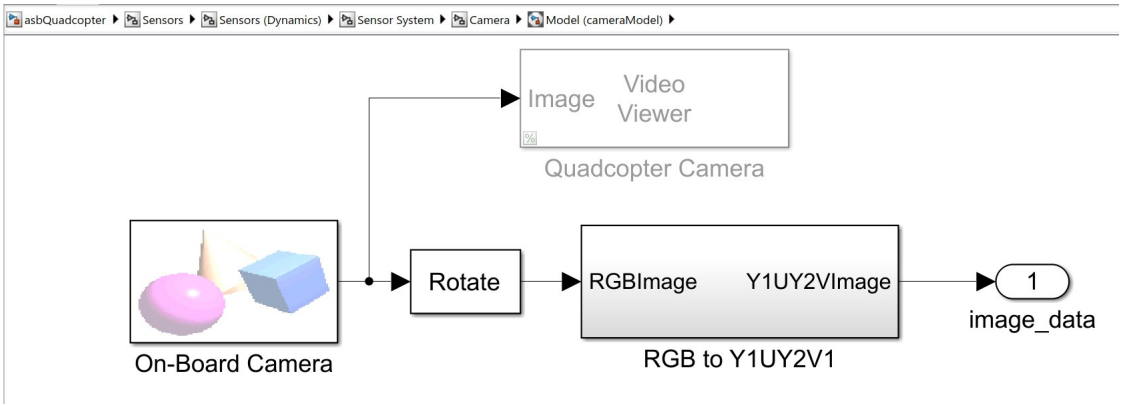


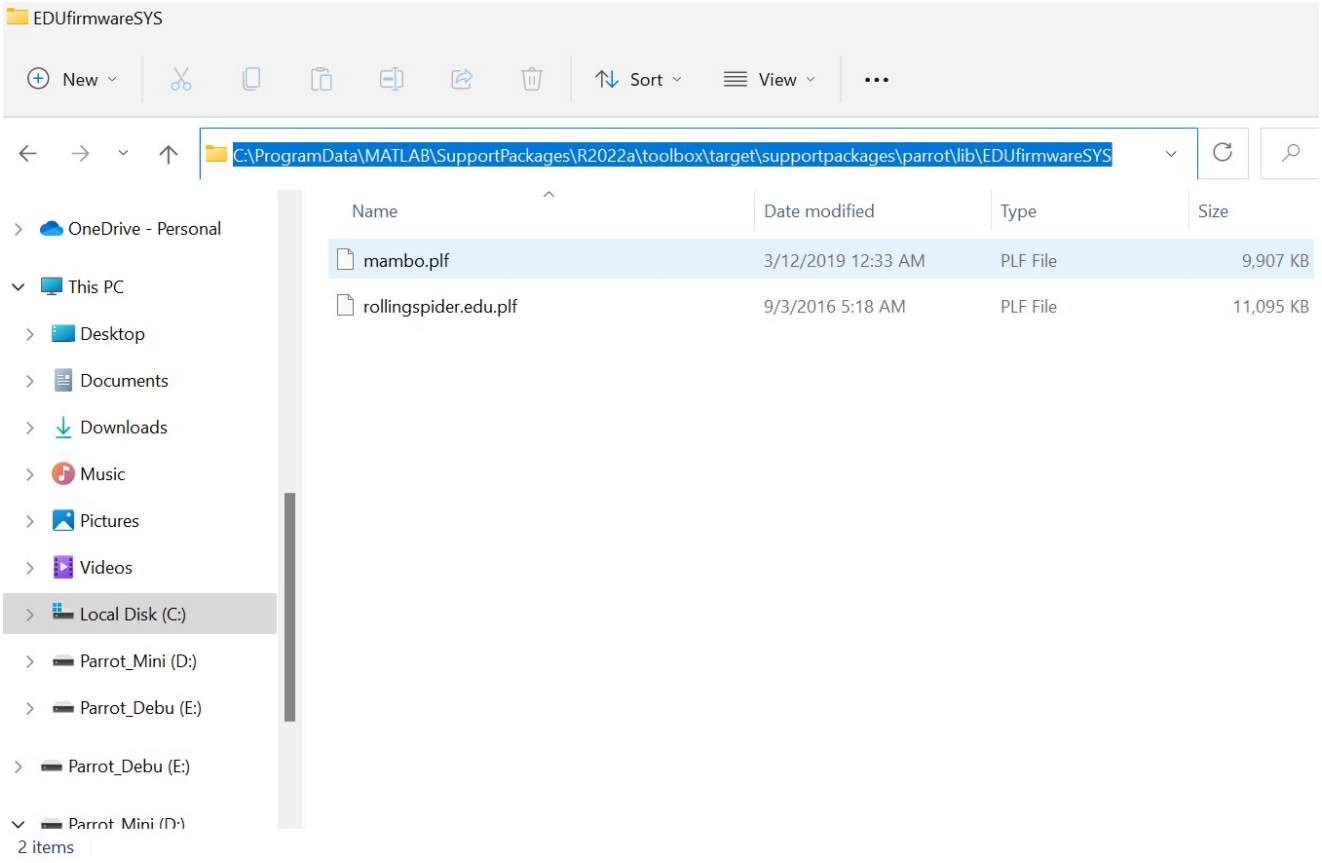
Figure 7: Commenting out the Video Viewer block in order to prevent unintentional live script simulation output.

C More Information on the Firmware of Parrot Mambo

The default (official) firmware of Parrot Mambo is designed for Bluetooth connections with phones (as opposed to computers). Specifically, for an Android phone, the Freeflight Mini app can be used to remotely control the quadcopter via Bluetooth connection. The flight performance achieved with this app is extraordinarily smooth and stable (although I could not find any technical documentation that sheds light on its control logic).

When we update the firmware of Parrot Mambo following this guide, however, we lose the ability to connect the quadcopter to the app. Thankfully, it is very straightforward to switch back to the original firmware (which can be downloaded from this page). As demonstrated in this YouTube video, every time we want to update the firmware of Parrot Mambo, we can simply connect the quadcopter to the computer using a data-transfer cable, copy and paste the desired firmware (a PLF-file) into the home directory of Parrot Mambo's drive, and disconnect the quadcopter from the computer.

In addition, to change from the default firmware to the MATLAB firmware, we do not have to follow this guide every time. I discovered that the MATLAB firmware is located in a specific directory, and we could simply copy and paste this PLF-file into the home directory of Parrot Mambo's drive just as we would for the default firmware. For PC, the directory looks like the one in the following figure:



D List of Add-Ons Used for the Project

Throughout this document, the MATLAB add-ons are introduced on a case-by-case basis; whenever a specific add-on is needed to accomplish a specific purpose, the purpose is explained and the add-on is mentioned. Readers who are interested in following my footsteps should have no problem installing the add-ons on this on-demand basis, but some may prefer to install all of them at once using the following list (MATLAB and Simulink should be installed first):

- Aerospace Toolbox
- Control System Toolbox
- Image Processing Toolbox
- Aerospace Blockset
- Optimization Toolbox
- Simulink Control Design
- Signal Processing Toolbox
- Computer Vision Toolbox
- Simulink 3D Animation
- MATLAB Coder
- Simulink Coder
- MATLAB Support for MinGW-w64 C/C++ Compiler
- Embedded Coder
- Simulink Support Package for Parrot Minidrones
- DSP System Toolbox
- Symbolic Math Toolbox