16-665: Robot Mobility
Fall, 2020

Air Mobility
Project Description
Due: November 5, 2020

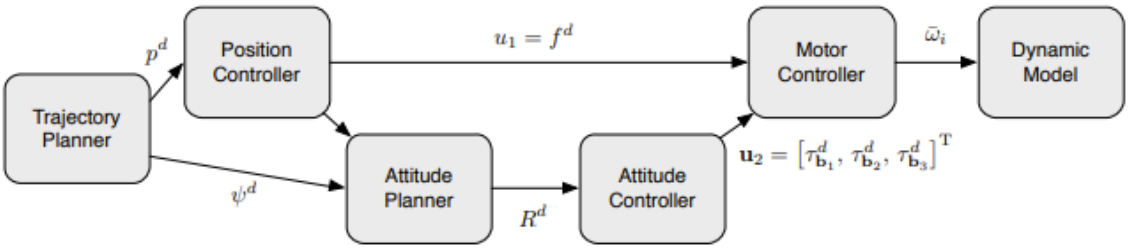
Feng Xiang

Quadcopter Simulation Project

16-665

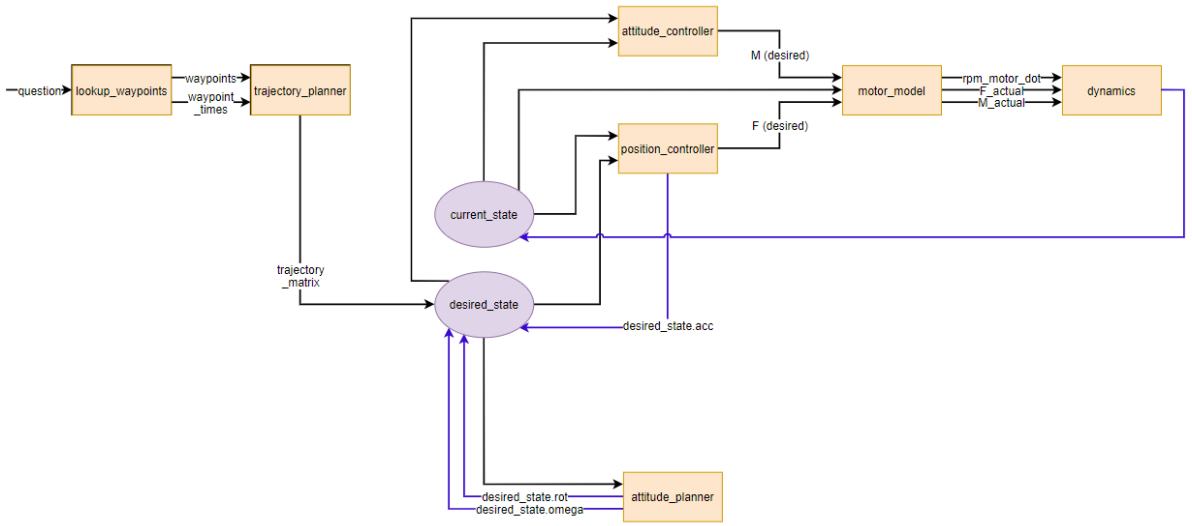
Question 1

Initially, one would describe a quadcopter simulation model to be as the following shown in class:



The flow of the model begins at the trajectory planner, which maps the desired path and states that the quadcopter model is following. From the trajectory planner, the position controller maps the desired pose values (x, y, and z positions and velocities) and outputs both a desired thrust to the motor controller and differences between the desired and actual phi and theta, respectively, to the attitude planner. Keep in mind that the position controller utilizes a feedback loop to output an appropriate desired thrust and delta values based on the gains of the feedback loop. From there the attitude planner takes in the desired heading, and error values and outputs desired rotation matrices that the model induces to follow the desired path. The attitude controller takes in those rotation matrices and outputs desired moment values across each of the three body axes and passes that information to the motor controller. Keep in mind that the attitude controller utilizes a feedback loop to output desired moment values based on the errors multiplied by the gains of the feedback loop. The motor controller would take in both the desired thrust and the desired moment values to input into the mixing matrix equation. The mixing matrix equation in the motor controller outputs actual rotor RPM values and passes them into the dynamic model. From the given inputs, the dynamic model updates the current state of the model. From there the model continuously updates as new waypoints on the trajectory are inputted in to the model and new values are being updated.

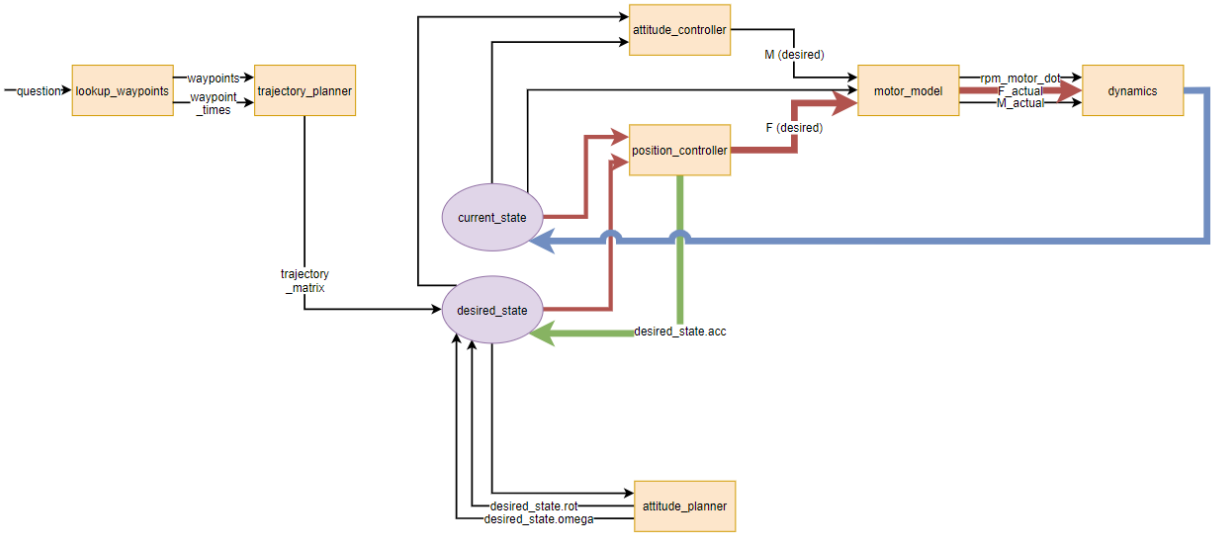
The figure shown below displays the software pipeline of the simulation model being developed:



Each rectangular block denotes the respective major *MATLAB* functions created in the model. Variables *current_state* and *desired_state* are included in the diagram as well and are visualized as circular blocks. These state blocks were included because they are continuously being updated in the model feed into various functions. The purple arrows that feed back into the state variables denote a feedback of information between the two types of states and the respective function passing the data into the state variables.

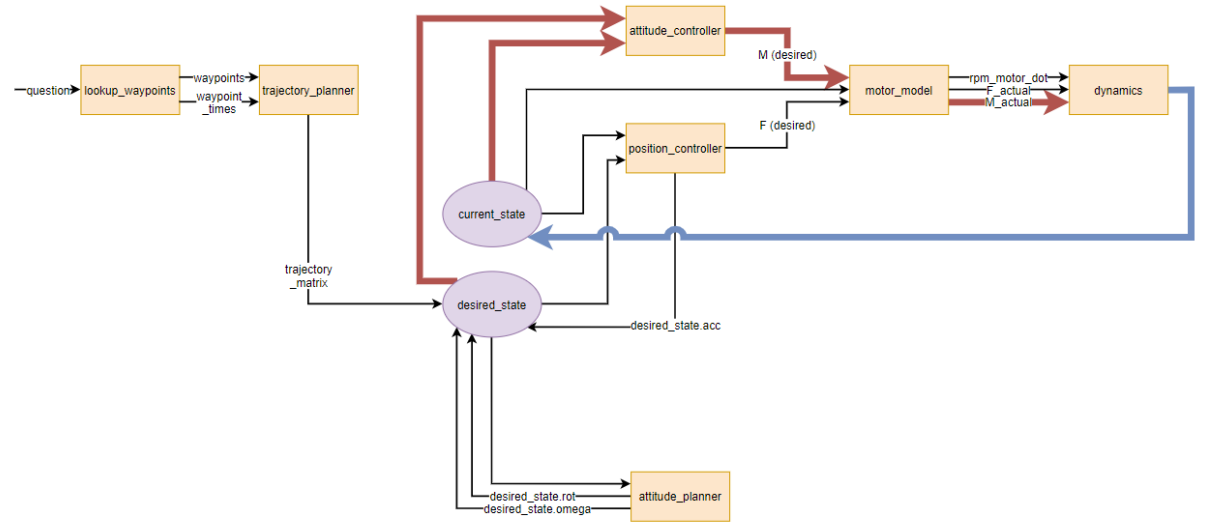
One can observe a near perfect similarity between the system diagram shown above and the theoretical system diagram discussed earlier in this document. Though it may seem like the *attitude_planner* function is operating outside of the system loop, keep in mind that the output values of the function feed into the *desired_state* variable. These variable are used in the functions further along the system flow, namely the *attitude_controller* function which uses those desired values to compute the error between the current state and the desired state at that point in time.

The following iteration of the system diagram shown below highlights the closed loop feedback of the position controller:



The red arrows denote the forward flow of the closed loop whereas the blue and green arrows denote respective feedback loops to either the current state or desired state variables. Internal to the position controller, the function outputs an appropriate *F* desired value based on the errors between the current and desired states at the current point in time, multiplied by the closed loop gain specified in the function. The *positional_controller* function also outputs an updated desired acceleration value into the desired state variable. Actual state outputs from the dynamics model feed into the current state vector, and on the next iteration of the model, the desired state is updated as well.

The following iteration of the system diagram shown below highlights the closed loop feedback of the attitude controller:

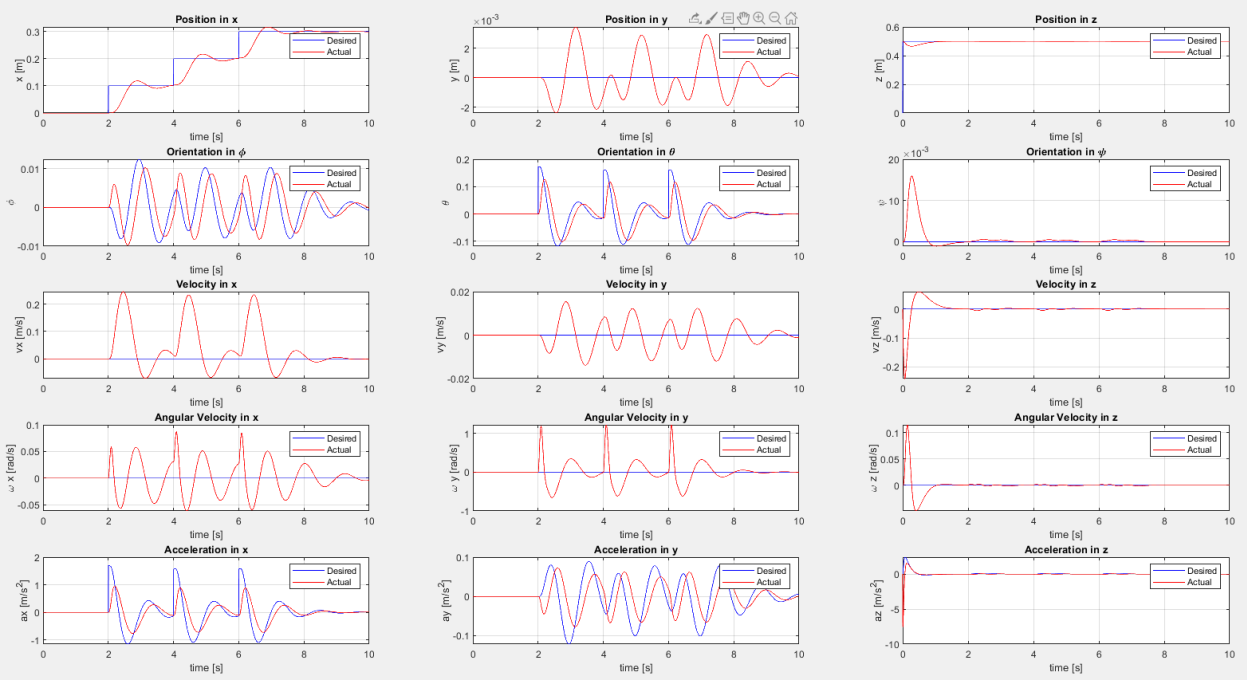


Similar to the previous closed loop diagram for the position controller, the red arrows denote forward flow of the closed loop whereas the blue arrow denote the feedback loop to the current state variable. Similar to the function of the position controller in the model, the attitude controller outputs a desired

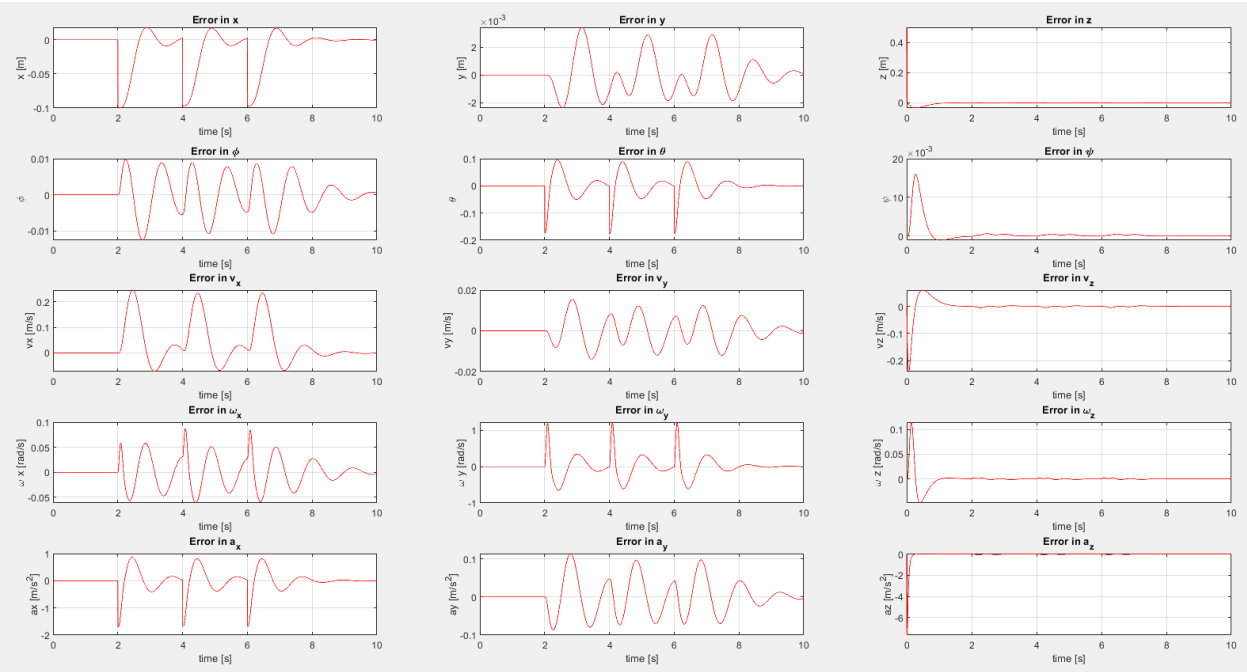
M value based on the phi, theta, and psi errors between the desired and current states. The output state from the dynamics model serves to update the *current_state* variable, wherein the *desired_state* variable also updates on the next iteration of the model.

Question 2

The following plot shown below denotes the model simulation plots for the desired and actual values from using the default gain values for both the position controller and the attitude controller:



The following plot shown below denotes the error plots over time from using the default gain values for both the position controller and attitude controller:



Observing the actual vs desired x-position of the model shown above, there exists a degree of oscillation about each updated waypoint. As such, one can observe that the closed loop system behavior along the x-axis of the model is an underdamped system as can be observed by the temporal overshoot of the actual diagram compared to the desired.

As an initial reconfiguration, the proportional gains of the position controller were changed from the default provided values and set to the follow:

Kp1 = 12

Kd1 = 6.6

Kp2 = 12

Kd2 = 6.6

Kp3 = 14

Kd3 = 9

Comparing the following values with the default values, one can see that the proportional gains denoted to “Kd” decreased from the default values. The motivation behind decreasing the proportional control of the system is to provide a closed loop subsystem that will be less reactive and energetic to desired trajectories in order to prevent overshoot. As such, reducing the proportional gains of the positional control drove the x-control of the system from an underdamped system to a system that is more critically or slightly overdamped, which is seen in the actual x-axis distance traveled over time.

The following attitude controller gains were used:

Kpphi = 210

Kdphi = 28

Kptheta = 218

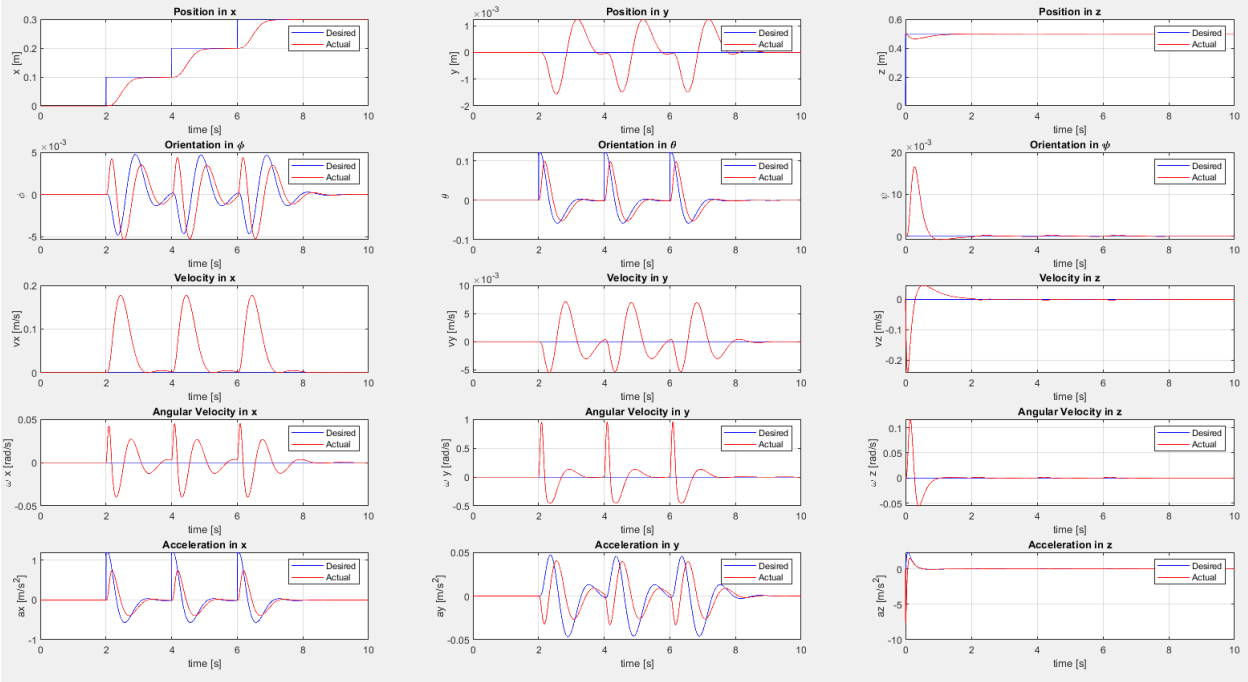
Kdtheta = 28

Kppsi = 80

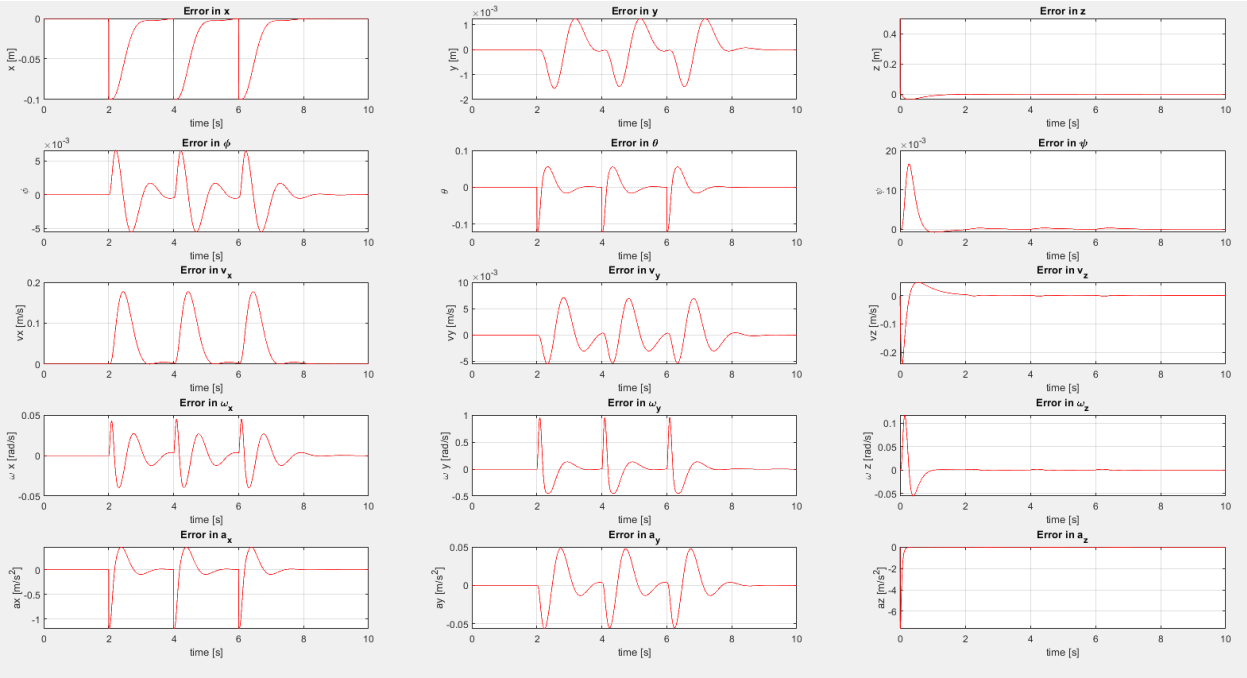
Kdpsi = 15.88

The motivation behind increasing the proportional gains of the attitude controller while slightly decreasing the derivative gains was slightly different compared to the motivations behind adjusting the positional controller gains. Since the orientations and derivatives of phi and theta are more oscillatory in nature, it is more desirable to have an attitude which is more reactive and can track the oscillatory trajectory of phi and theta more closely and aggressively. This was the main motivation behind increasing the proportional gains for *Kpphi* and *Kptheta*, which would effectively decrease the rise time, assuming the output behavior is similar to that of a 2nd order system. The motivation behind slightly reducing the derivative gains of the system was in part to giving more control to the proportional gain and to have the attitude controller react slightly less to sudden changes to the derivative of phi and theta in the system. Even though decreasing the derivative gains would effectively increase the settling time and percent overshoot values, the oscillatory nature of phi and theta gave less of an emphasis of these increased values.

The following plot shown below denotes the model simulation plots for the desired and actual values from using custom gain values for both the position controller and the attitude controller:



The following plot shown below denotes the error plots over time from using custom gain values for both the position controller and attitude controller:



As seen in the figures above, the custom gain values to the position controller and attitude controller gave more desirable results in several fields. For one, the x-direction positional response has no percent

overshoot while following the desired trajectory. Actual tracking of the y-direction position over time has also contracted in overshoot to lie more closely between the zero axis. Several other plots have reduced percent overshoots, such as the phi orientation and theta orientation over time.

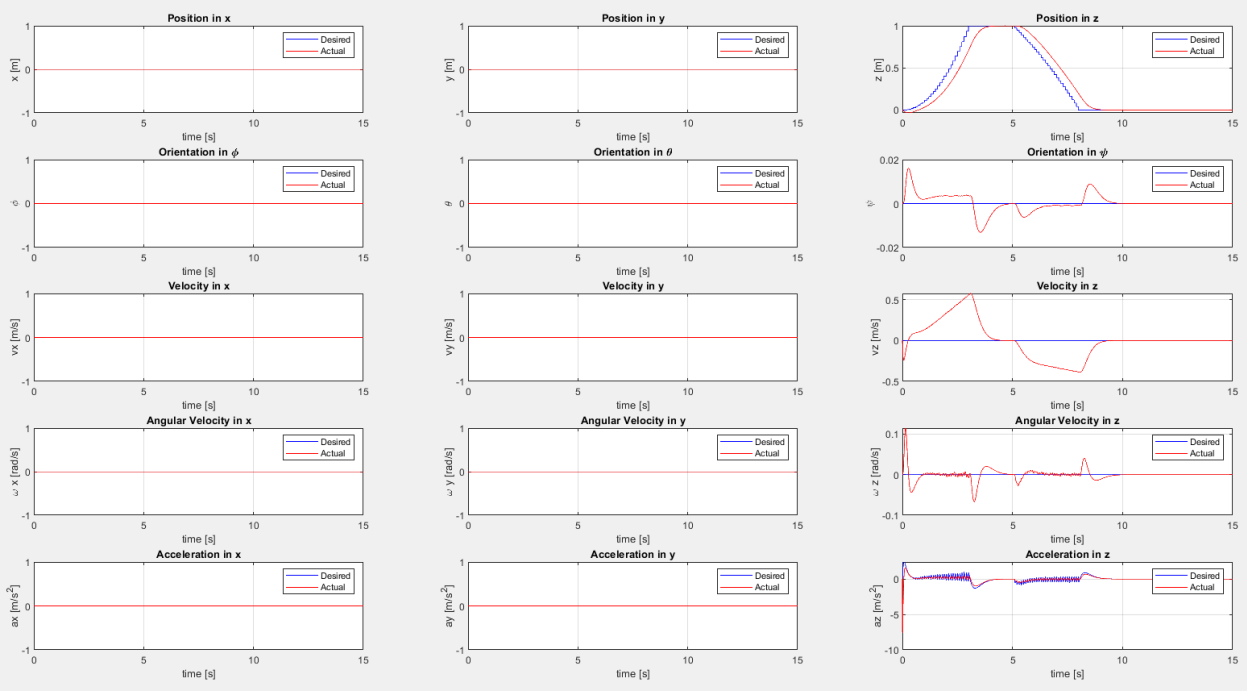
Question 3

The take-off, hover, and landing desired profile of the model was determined by the following parametric equations:

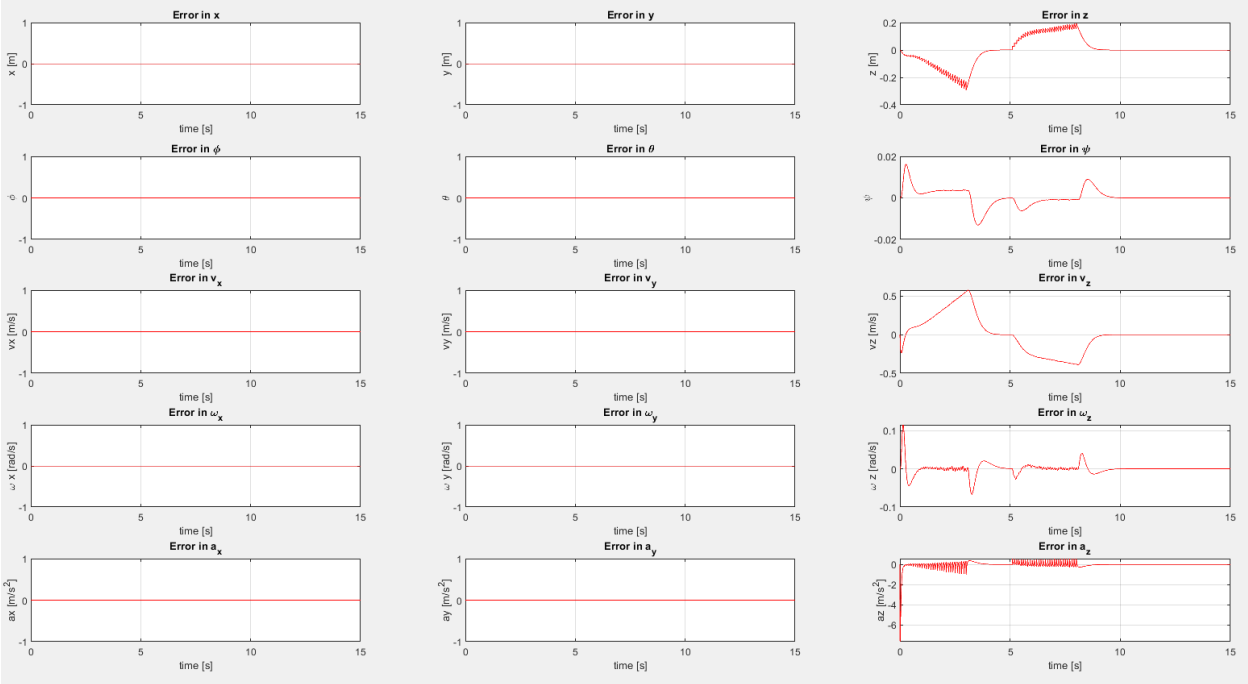
$$[0,3]: \; h = \frac{1}{9}t^2$$
$$(3,5]: \; h = 1$$
$$(5,8]: \; h = -\frac{1}{39}t^2 + \frac{64}{39}$$

Where h is the z-axis height of the quadcopter in meters and t is the time parameter of the equation in seconds. Converting the waypoints into a trajectory, it was determined to use the same method employed by the *trajectory_planner* function for question 2. The method is this: between waypoint locations, create a flatline function and perform a step function to the next waypoint.

The following plot shown below denotes the model simulation plots for the desired and actual values from the default gain values for both the position controller and the attitude controller:



The following plot shown below denotes the error plots over time from the default gain values for both the position controller and attitude controller:

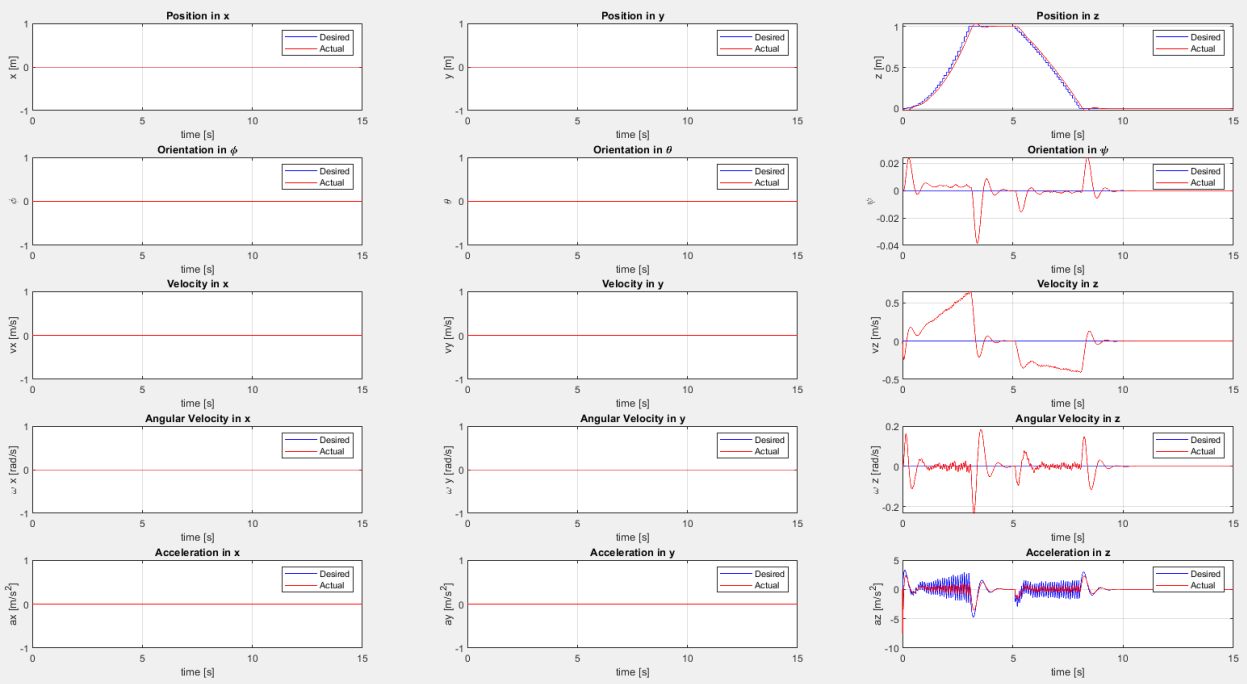


Analyzing the figures shown above, one can observe an actual z-axis distance travel plot that is following the desired trajectory path but never converging until the trajectory flat-lines. In essence, one can deduce that the system does not oscillate about the waypoints, and instead performs like an overdamped system towards each updated waypoint. In addition, because this trajectory was computed using constant values between waypoints, there was no desired velocity profile to pass into the model. After some time and before the model plots ended, however, one can clearly see a stable and converging model as the error plots denote a model that is converging the various errors towards zero.

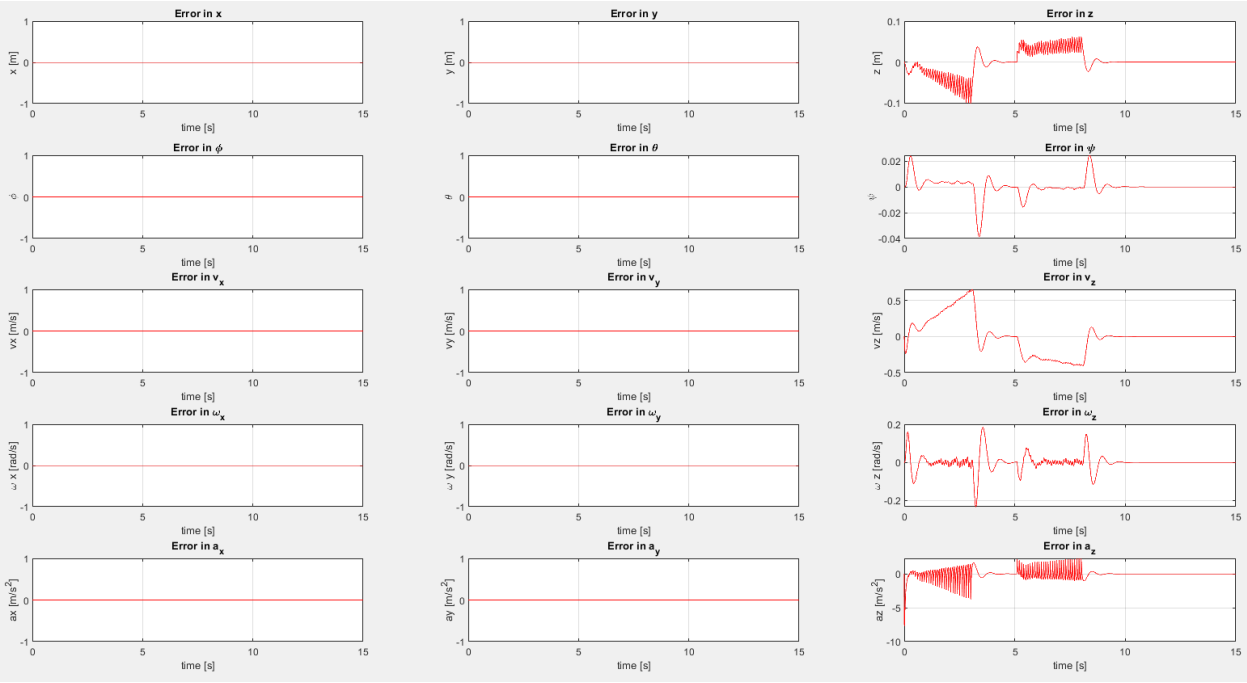
The proportional gain associated with the z-direction controller in the *position_controller* function was reconfigured to be the following:

Kp3 = 80

The following plot shown below denotes the model simulation plots for the desired and actual values using custom gain values for both the position controller and the attitude controller:

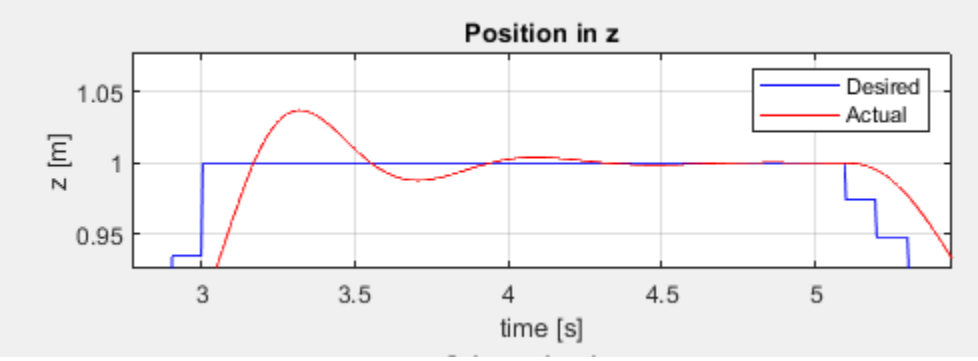


The following plot shown below denotes the error plots over time using custom gain values for both the position controller and attitude controller:

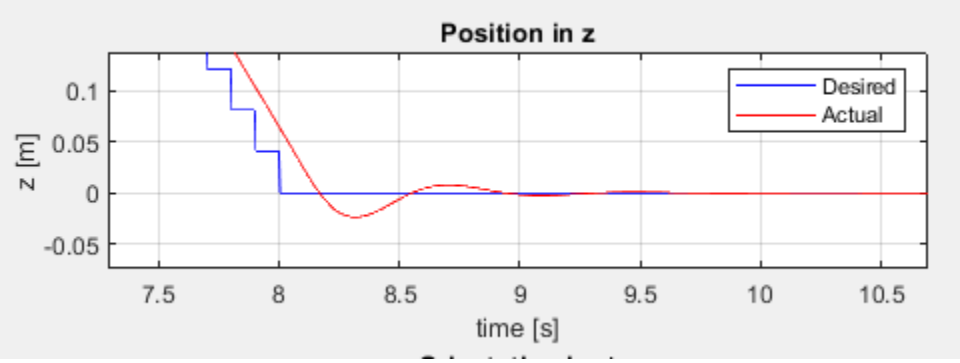


Observing the figures shown above, one can determine that the z-direction tracking between the actual and desired trajectories are more closely tracking compared to using the default gain values given to perform the simulation. One can also observe that the actual model plots overshoot and oscillate about some desired waypoints when the z-direction desired trajectory holds a constant value for too long, such as at the peak of the desired trajectories and after the quadcopter lands.

The close-up figure shown below displays the actual vs desired z-direction trajectories about the plateau of the trajectory:



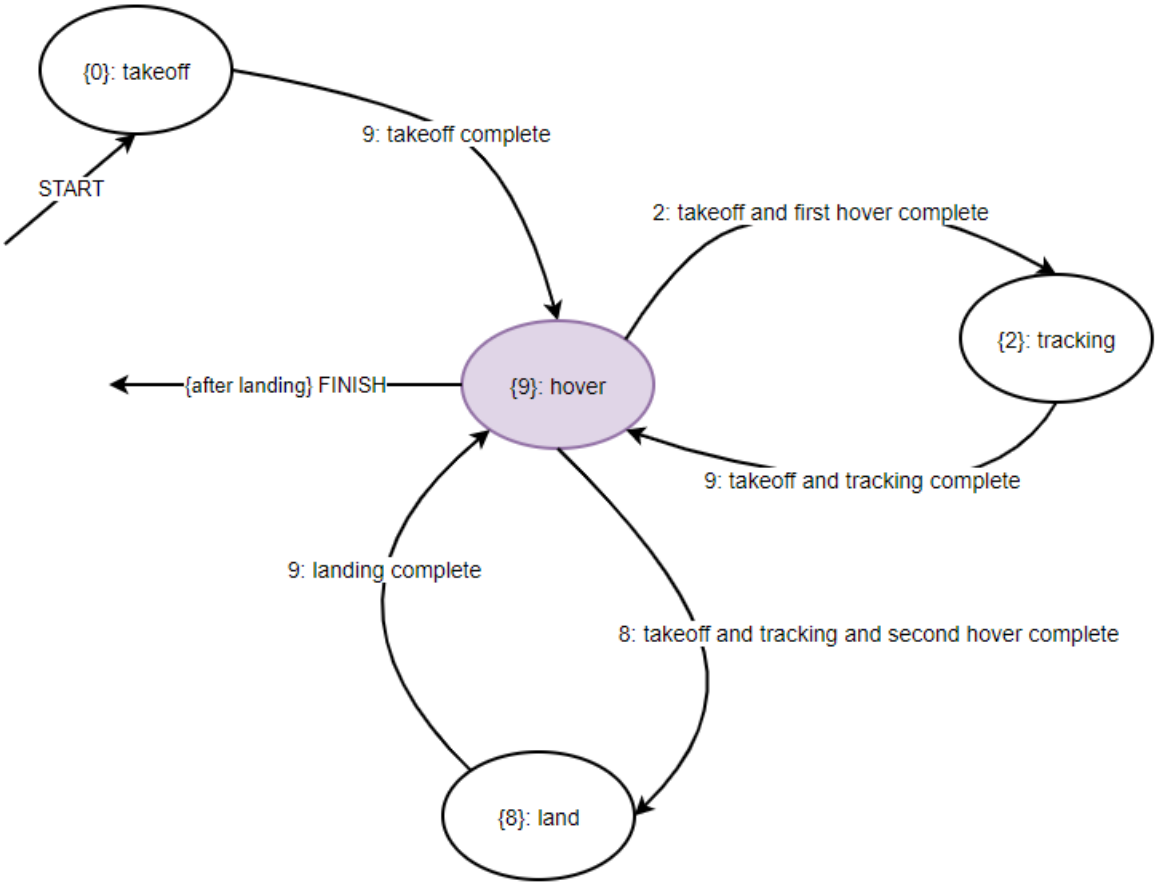
The close-up figure shown below displays the actual vs desired z-direction trajectories about the landing conclusion of the model:



The actual trajectory of the model as shown above may seem unrealistic because if the zero axis of the plot is the ground, then the actual trajectory is indicating that the quadcopter is travelling below the ground before rising back up. Regardless, the close-up of the landing of the quadcopter does give insight into the oscillatory nature of the control policy that was put in place in the quadcopter and the details of difference between the simulated model and what a real-world robot would encounter.

Question 4

The state machine design for this model was designed as shown in the visual diagram below:



In summary, the state machine consists of four distinct states where one state is called upon more than once and flows out of the "internal" node requires additional conditionals. At the start of the program, the state machine begins at the 0^{th} state. In this state, the quadcopter begins at rest, or ground level, and takes off to a specified height in the air based on a desired rise trajectory.

After a specified time passes after takeoff, the quadcopter reaches to a hover state as denoted by the purple-highlight node in the diagram above. In this hover state, the quadcopter will stabilize to the final desired x, y, and z positions of the previous state for a set amount of time. In this scenario, the quadcopter will be in the hover state for about 1 seconds and stabilize about the final specified height from the takeoff desired trajectory.

After the specified time set for the hover state passes, the state machine transitions into the tracking state. In this state, the model tracks and navigating a specified desired trajectory. The desired trajectory and waypoints were prespecified in the program and is specified for a certain amount of time. After the specified time for the desired trajectory passes, the state transitions back to the hover state. Identical to

the previous interaction with the hover state, the model will stabilize about the final desired position of the previous state for a set amount of time. Finally, with takeoff and tracking complete, the hover state will transition to the landing state.

In the landing state, the quadcopter tracking a desired z-direction trajectory from its current height back to a height of zero. Based on the current algorithm, the landing trajectory takes into account the current height of the quadcopter after hover and sets ramp trajectory from that current height to the zero axis. After this point the state will once again convert to a state of 9 and enter the “hover” state. Though the quadcopter is on the ground at the zero axis and is no longer hovering, the state is still encounter so as to hold the last desired trajectory value (zero) and drive the system to zero. Afterwards, the system will change its *fsm_state* to 3, thus closing the while loop and ending the model. Afterwards the program moves on to output the trajectory and error plots.

In order to concatenate the outputs of the model at different states, the default *actual_state_matrix* and *actual_desired_state_matrix* variables were modified. Instead of initializing the variables as have a set length based on the time vector, the variables were instead initialized as a column vector. Through each iteration of the quadcopter model for each serial state, the two variables are appended by updated values, as opposed to change the zero values at each consecutive index. Through this method, a prespecified length of time does not have to specified and instead the program can focus on specifying time for each state trajectory. The updated initialization of the two variables as done through MATLAB are shown in the figure below:

```
%% Create a matrix to hold the actual state at each time step

actual_state_matrix = zeros(15, 1);
% [x; y; z; xdot; ydot; zdot; phi; theta; psi; phidot; thetadot; psidot; xacc; yacc; zacc];
actual_state_matrix(:,1) = vertcat(state(1:12), 0, 0, 0);
```

```
%% Create a matrix to hold the actual desired state at each time step

% Need to store the actual desired state for acc, omega dot and omega as
% it will be updated by the controller
actual_desired_state_matrix = zeros(15, 1);
```

To the same philosophy as modifying the *actual_state_matrix* and *actual_desired_state_matrix* variables, the total time vector of the model was modified to fit the flow and execution of the state machine. Instead of specifying an allotted time vector for the entirety of the model, a blank time variable was initialized. After the execution of each state and before transitioning the model to the next state, the *tot_time_vect* variable is appended by the time vector of the specific state. Initialization of the *tot_time_vect* variable is shown below:

```
%declare blank total time vector
%will append to the time as model runs through states
tot_time_vect = [];
```

After the execution of the state machine and before the plots and outputs of the model are created, the *tot_time_vect* is reconfigured in order to have a linear time progression. The motivation behind reinitializing the variable is because each state time vector starts at zero and progresses to their respective end time values. Concatenating those vectors to the same vector, the time vector is not progressing linearly. Therefore, the time vector variable is reinitialized back on the time step of the

model and the total length of the *tot_time_vect* itself. The reinitialization command is shown in the figure below:

```
tot_time_vect = linspace(0,length(tot_time_vect)*time_step,length(tot_time_vect));
```

The state machine was implemented in MATLAB using a switch-case implementation nested within a while loop. A snippet of the loop is shown below:

```
if question == 4
    while fsm_state ~= 3
        switch fsm_state
            case 0 %takeoff case
                %initialize waypoints for takeoff
                [temp_waypoints,temp_waypoint_time
                % Populate the state vector with t
```

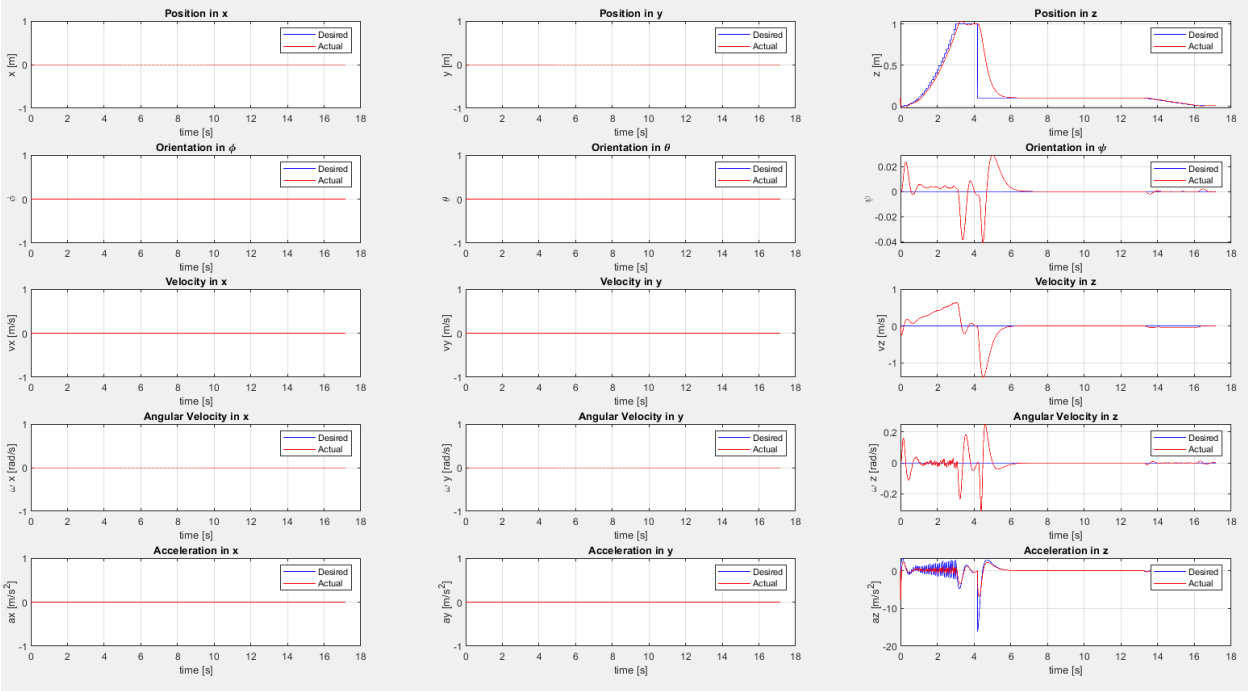
As shown in the figure above, the while loop is run through only when the input to the *main* function is *question*=4. Prior to the if-conditional, the *fsm_state* varible is set to zero, indicating that the model has not flown yet. From there the body of the *case=0* code executes, where the quadcopter ran through a takeoff trajectory. The *lookup_waypoints* and *trajectory_planner* functions is then appended with appropriate code to create a desired takeoff trajectory.

At the end of the body of the case code, the *fsm_state* variable is updated to the next state as per the state machine design described earlier. In addition, the *tot_time_vect* variable is appended with the time vector used in the model. This procedure is repeated for the other states as well. A snippet of the *fsm_state* update and *tot_time_vect* append after the takeoff state code block is executed is shown below:

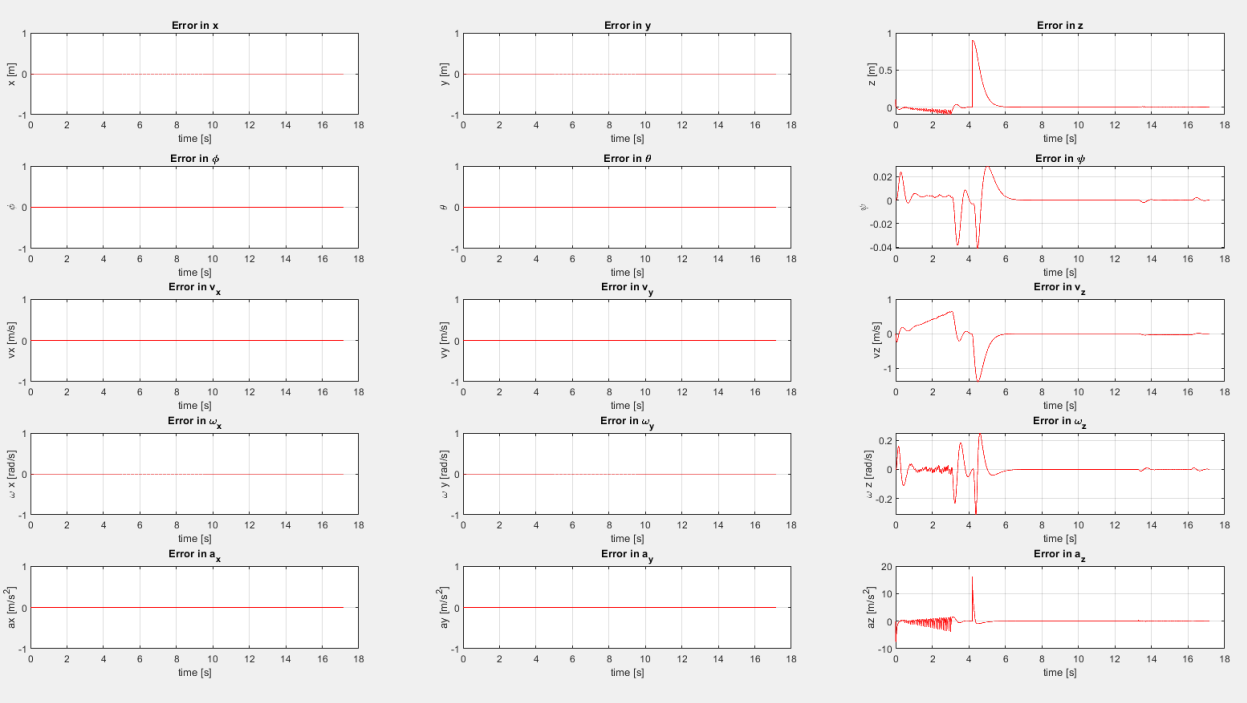
```
    end
    fsm_state = 9;
    tot_time_vect = [tot_time_vect temp_time_vect];
```

Question 5

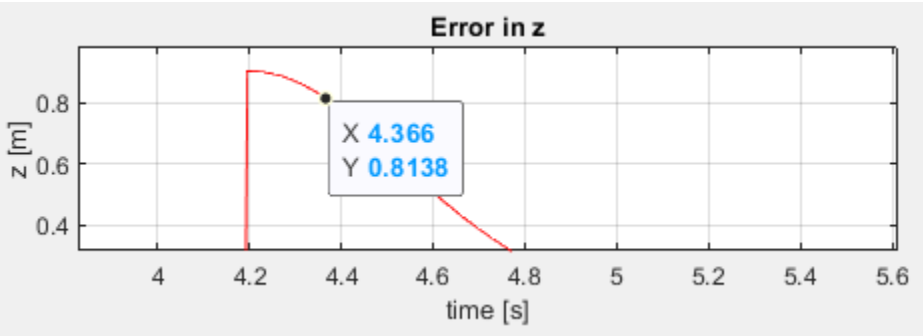
The following actual and desired trajectory plots of the model following a z=0.1 tracking trajectory is shown below. The plots shown below used gains from Gain Set 1 for the *position_controller* and *attitude_controller* functions:

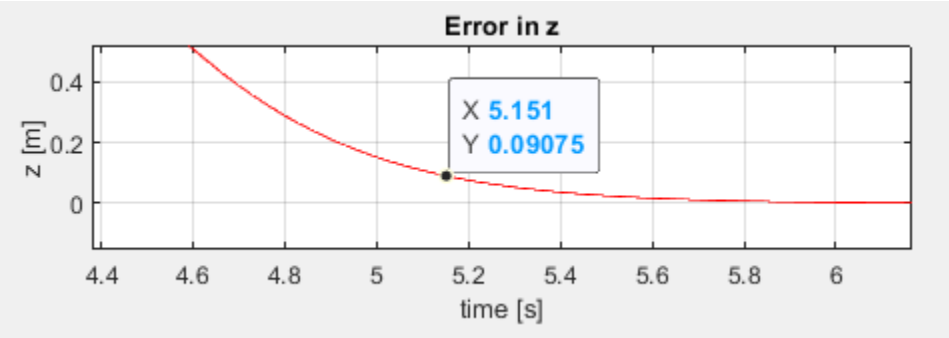


Using the same Gain Set 1, the error plots between the desired and actual trajectories is shown below:



The method to compute the 10-90% rise time and 10% settling time values was done through manually analyzing the approximate point to the respective y-value and compute the x-value (time) value on the error plots. For the figures shown above using Gain Set 1, the 10% and 90% times are shown below for the positional plot, respectively:





The start time for the trajectory tracking state was at t=4.2 seconds. From the figures shown above, the position rise time of the system from 10% to 90% is about 0.78 seconds.

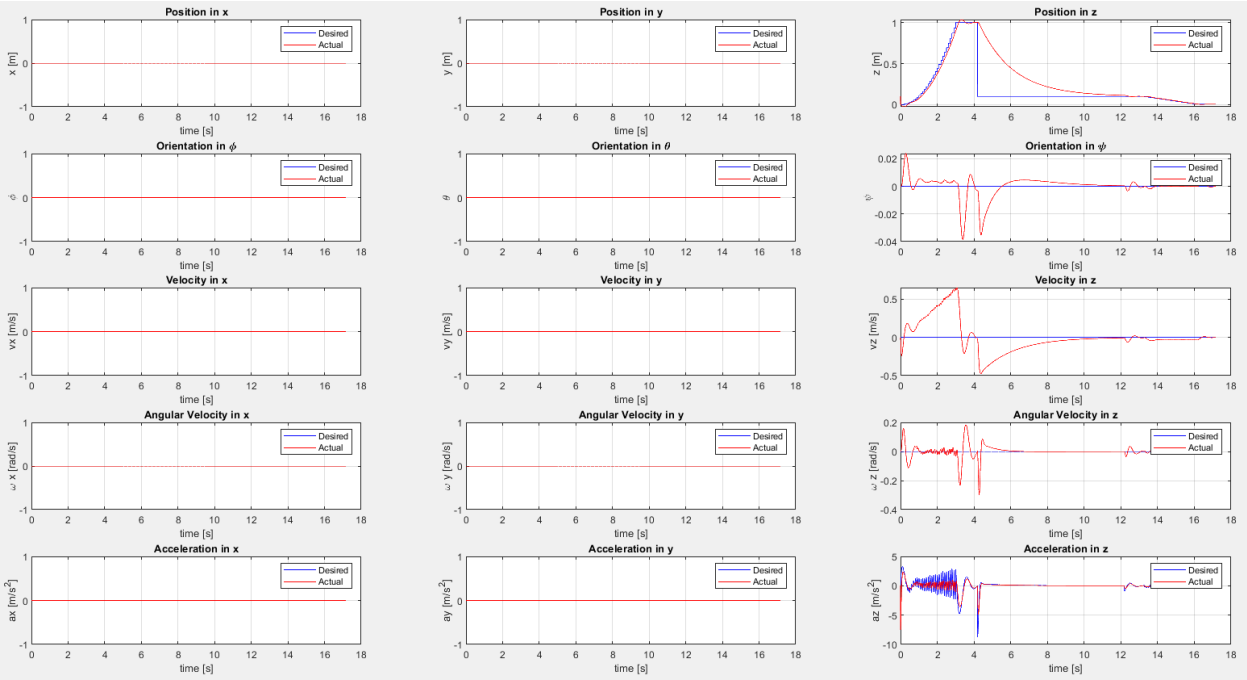
From the time determined from the figure above and computing the difference from the start time of the trajectory tracking state, the position settling time was determined to be about 0.951 seconds.

The linear velocity rise and settling times were computed to be 0.78 seconds and 0.951 seconds, respectively.

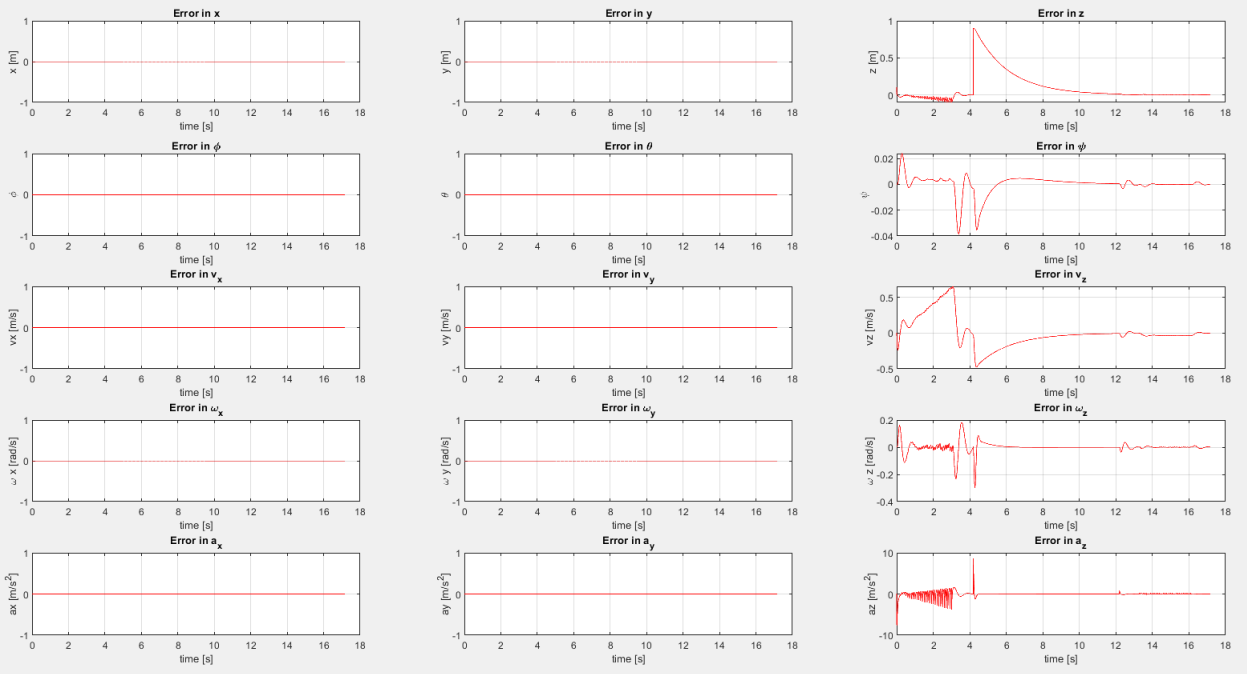
The steady state error value during the tracking trajectory state for both position and linear velocity plots were at zero.

Because the model exhibits critically or overdamped characteristics, the system does not have any percent overshoot.

The following actual and desired trajectory plots of the model following a z=0.1 tracking trajectory is shown below. The plots shown below used gains from Gain Set 2 for the *position_controller* and *attitude_controller* functions:



Using the same Gain Set 2, the error plots between the desired and actual trajectories is shown below:

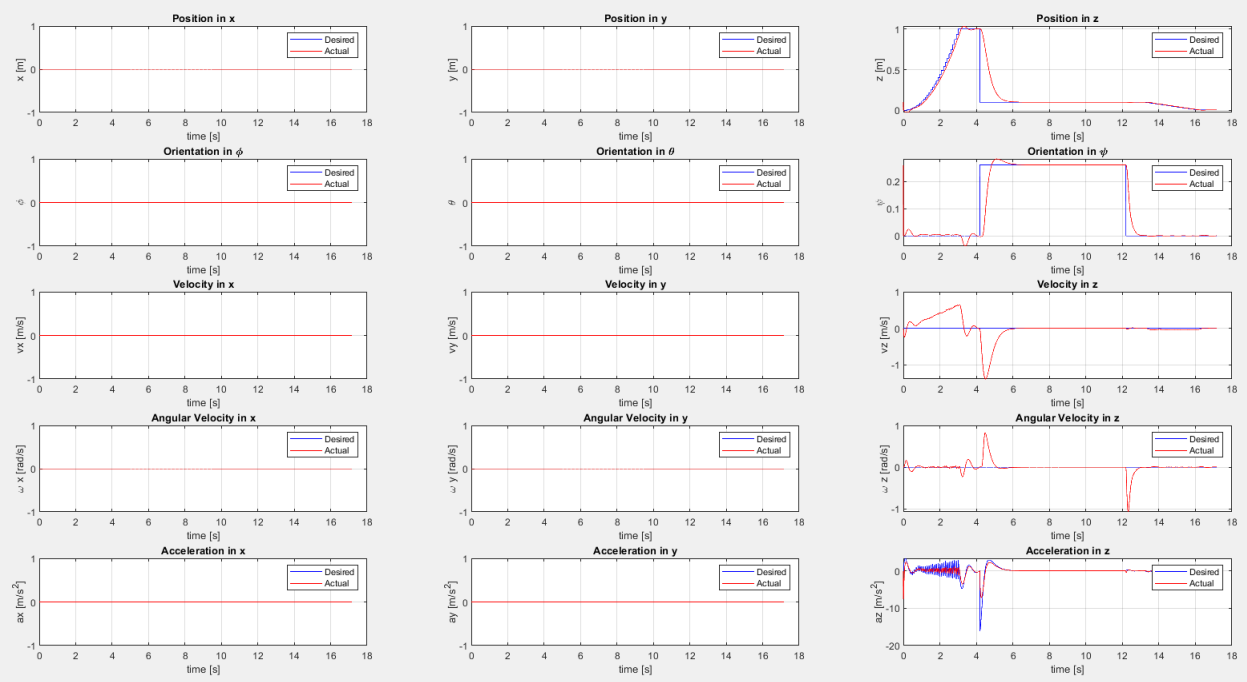


Using the same methods as described earlier, the position rise time was computed to be about 4.025 seconds. The linear velocity rise time was computed to be about 4.016 seconds. The position settling time was computed to be about 4.025 seconds. The linear velocity settling time was computed to be about 4.016 seconds.

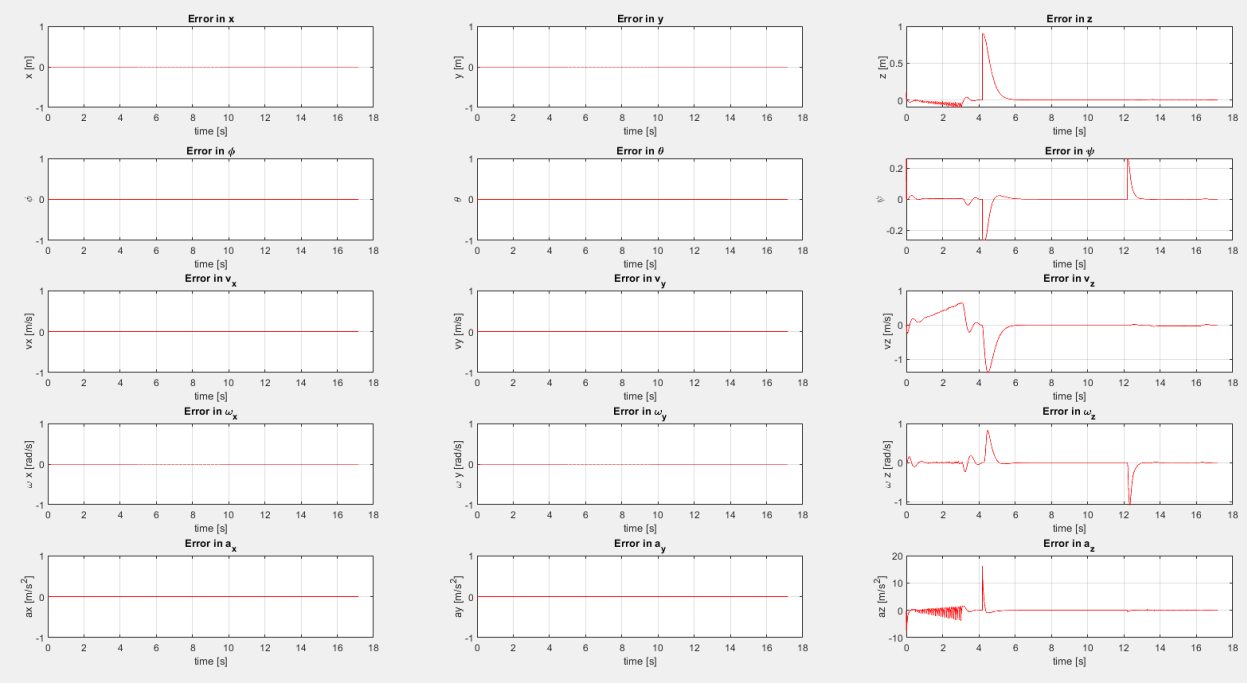
The steady state value was determined to be around the desired height at 0.1m.

The percent overshoot is nonexistent because the model performed like an overdamped system.

The following actual and desired trajectory plots of the model following a z=0.1 and psi=15 deg tracking trajectory is shown below. The plots shown below used gains from Gain Set 1 for the *position_controller* and *attitude_controller* functions:



Using the same Gain Set 1, the error plots between the desired and actual trajectories is shown below:

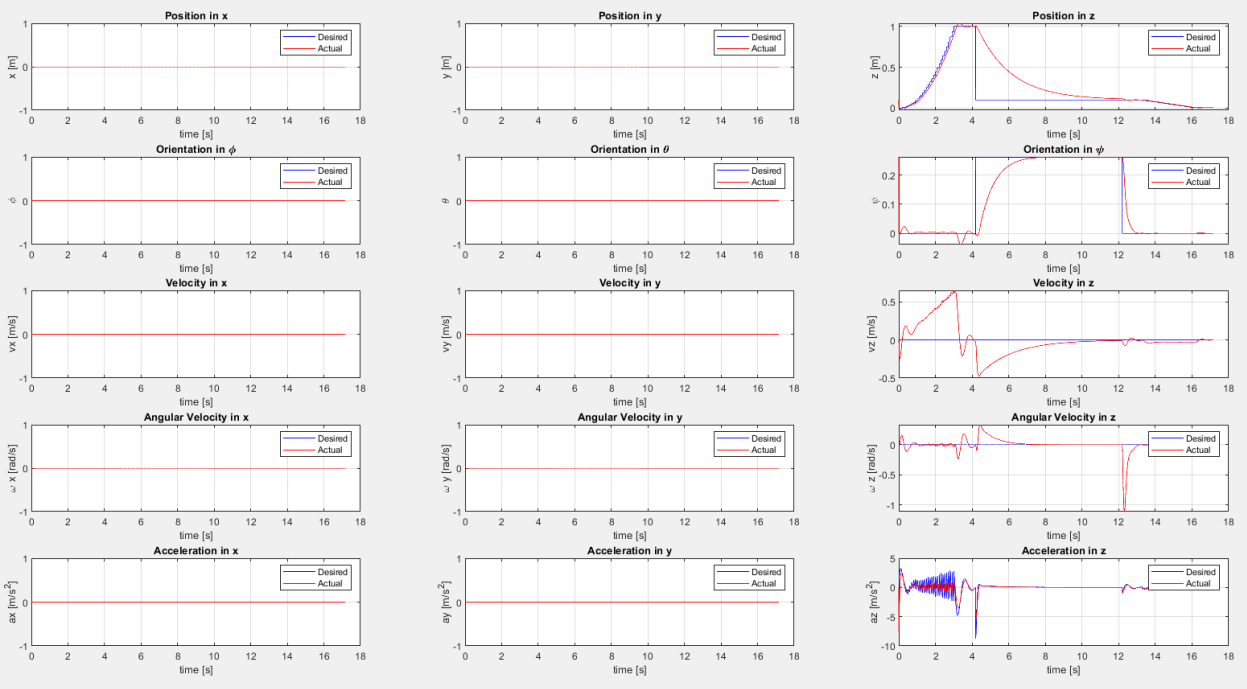


Based on the psi error plots, the rise time was computed to be about 0.335 seconds. The settling time was computed to be about 0.541 seconds.

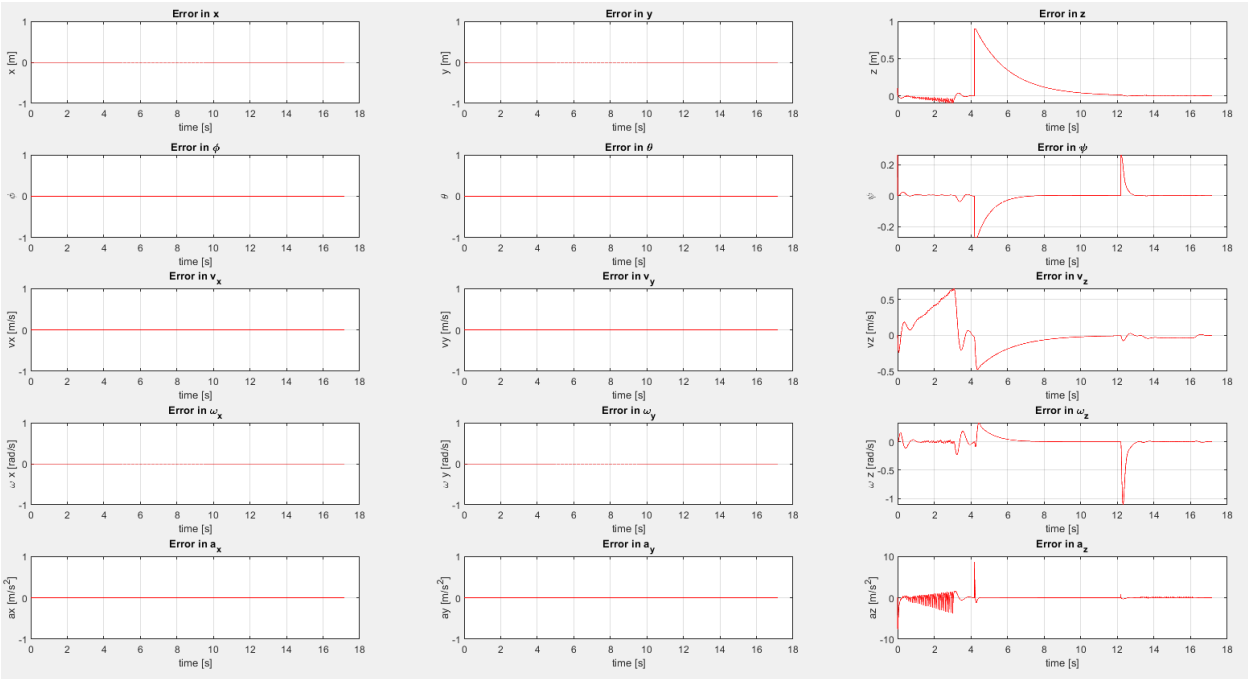
The steady state error value was observed to converge at zero.

The percent overshoot was computed to be about 8.4% overshoot.

The following actual and desired trajectory plots of the model following a z=0.1 and psi=15 deg tracking trajectory is shown below. The plots shown below used gains from Gain Set 2 for the *position_controller* and *attitude_controller* functions:



Using the same Gain Set 2, the error plots between the desired and actual trajectories is shown below:



Based on the psi error plots, the rise time was computed to be about 1.621 seconds. The settling time was computed to be about 1.852 seconds.

The steady state error value was determined to converge at zero.

Because the system behaves similar to an overdamped system, the percent overshoot does not exist.

For the positional and heading controllers, there exists two types of gains, proportional and derivative. Assuming all other variables are constant, as the proportional gain increases for both the positional and derivative gains, the rise time decreases. To an effect, if the proportional is at a high enough level, the model will become underdamped, as can be seen with Gain Set 1 on the psi trajectory at 15 degrees. As a mirror, decreases in the proportional gain will increase the rise time characteristic. Decreasing the proportional gain enough will yield an overdamped tracking system. For the derivative gain, it is observed that increasing the derivative gain decreased settling time and decreased percent overshoot. Conversely, decreasing the derivative gain increased the settling time and increased percent overshoot.