Quadcopter: Hover Control

Andrew Kyong, Fayyad Bin Azhari, Mohammad Izwan Zainol Bahar, Luis Mendez Cossio

I. Introduction

IN this lab, we sought to test the hovering performance of our quadcopter. This experiment combines the components obtained from the previous labs - force maps (lab 2), state estimate (lab 3), and attitude control (lab 4). Before the lab, the quadcopter had control over its attitude, and produced a normalized thrust force. Now, we will implement the state estimation for the vertical motion and horizontal velocity. By the end of the lab, the quadcopter's maneuverability will be tested in a non-restricted flying area. The goal is to have the drone hover for over 10 seconds and make it return to its original position in a smooth manner.

Fig. 1. The quadcopter, ready to be tested.

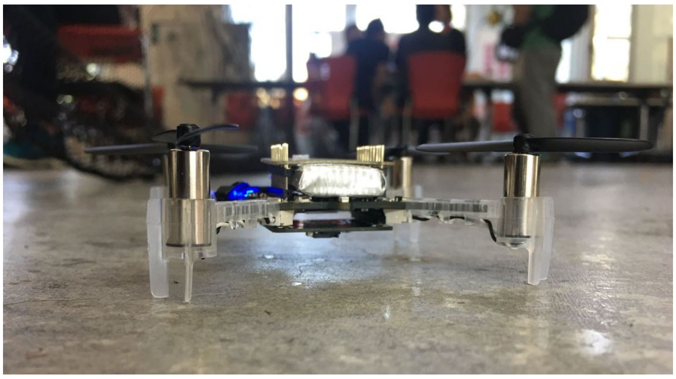


Fig. 2. Front view of the quadcopter.

II. Equipment, Methods, Procedure

A. Equipment List

- quadcopter kit
- goggles
- flight safety net

B. Preparations

Before beginning this lab, we first confirmed that the sensors are operating correctly by implementing the following code in $Figure$ 3.

```
//in PrintStatus():
printf("Last range = %6.3fm, ", \
  double(lastMainLoopInputs.heightSensor.value));
printf("Last flow: x=%6.3f, y=%6.3f\n", \
  double(lastMainLoopInputs.opticalFlowSensor.value_x), \
  double(lastMainLoopInputs.opticalFlowSensor.value_y));
```

Fig. 3. command that confirms sensors are working properly [1].

we check that the flow sensor is functioning properly by holding the vehicle still at roughly 20 cm above the table and confirming the output of the $quad\ status$ command. the command should say the sensor outputs roughly zero in all body-center directions. Moving the quadcopter in the $\underline{1}^B$ direction should yield a value in the x-direction but almost none in the y-direction. The same method is done in the $\underline{2}^B$ direction. $Figure$ 3 shows the location of the height and the optical flow sensors.

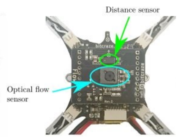


Fig. 4. The two sensors, located in the belly of the quadcopter [1].

The last preparation for the lab is stiffening the attitude controller by changing the time constant for the yaw rate, and the angles.

C. Procedure

1) Vertical state Estimator: We can calculate the height of the quadcopter $h_m(k)$ at any time step k by implementing the following vertical state estimator:

$$h_m(k) = \delta(k)cos\hat{\theta}(k)cos\hat{\phi}(k) \tag{1}$$

The vertical velocity $V_{3,m}(k)$ can be calculated by implementing the following equation:

$$V_{3m}(k) = \frac{h_m(k) - h_m(k-1)}{t_\delta(k) - t_\delta(k-1)} \tag{2}$$

Where δ is the range sensor output, $\hat{\phi}(k)$ and $\hat{\theta}(k)$ are estimates of the roll and pitch angle, respectively at time step k. For the second equation, $t_\delta(k)$ is the time for the distance measurement (k).

To test the height estimator, we placed the drone on the floor where it was free of obstacles. By starting the RadioAndJoystick, we get data when holding the vehicle at roughly 0.5m above the ground for 5 seconds. We then proceed to rotate the drone (pitch and roll angles) roughly 30 degrees to then return the drone to its original location at a constant height. The drone is then returned to the ground slowly while maintaining it at a horizontal position.

2) Horizontal state Estimator: We calculate the vehicle's velocity in the $\underline{1}^E$ and $\underline{2}^E$ directions by the expanded velocity pseudo-measurements, which depend on the optical flow (σ), rate gyroscope (γ), and range sensor (δ) measurements:

$$V_{1,m} = (-\sigma_1(k) + \gamma_2(k))\delta(k) \tag{3}$$

$$V_{2,m} = (-\sigma_2(k) - \gamma_1(k))\delta(k) \tag{4}$$

3) Horizontal Control: For lab 4, we implemented an attitude control scheme, letting us control roll, pitch and yaw. We now apply it to the velocity control. We first compute a desired horizontal acceleration, to counter the current velocity:

$$a_{1,d} = -\frac{1}{\tau_v} v_1 \tag{5}$$

$$a_{2,d} = -\frac{1}{\tau_v} v_2 \tag{6}$$

We can then compute the desired pitch and roll angles from the desired horizontal acceleration:

$$\phi_d = -\frac{1}{||\underline{g}||} a_{2,d} \tag{7}$$

$$\theta_d = \frac{1}{||\underline{g}||} a_{1,d} \tag{8}$$

To test the horizontal estimator, we placed the drone on the floor where it was free of obstacles. By starting the RadioAndJoystick, we get data when holding the vehicle at roughly 0.5m above the ground for 2 seconds and walking forward $(along\ \underline{1}^B)$ one step and hold for 2 seconds, we then repeat the process but now we walk left $(along\ \underline{2}^B)$ one step and hold for 2 seconds. We then proceed to rotate the drone (pitch and roll angles) roughly 30 degrees to then return the drone to its original location at a constant height. The drone is then returned to the ground slowly while maintaining it at a horizontal position.

4) Vertical Control: For the vertical control, the input a_3 is the change in total thrust normalized by mass

$$a_3 = \frac{\Delta c_\Sigma}{m^B} \tag{9}$$

Since $equation$ 9 is a double integrator, we can choose an input a_3 that makes the system behave like a damped system:

$$a_3 = -2\zeta\omega_n v_3 - \omega_n^2(s_3 - s_{3,d}) \tag{10}$$

with damping ratio ζ=0.7 and natural frequency ω_n=2 $\frac{rad}{s}$. Since we hope to obtain good measurements from the range sensor while ensuring the safety of the quadcopter, we initiate the value for the desired height $s_{3,d}$ to be 0.5m.

5) Set-up: Now that all the components have been implemented into the drone, we are ready to test it. The drone is placed inside the flying safety net and turned on. The drone was able to lift off the ground and hover for some time, but it kept flying slightly in one direction until it would crash against the walls of the net. We took some steps to debug the drone by analyzing the data obtained from the flight time of the drone. $Figure$ 5 shows a team member setting up the drone inside the safety zone.

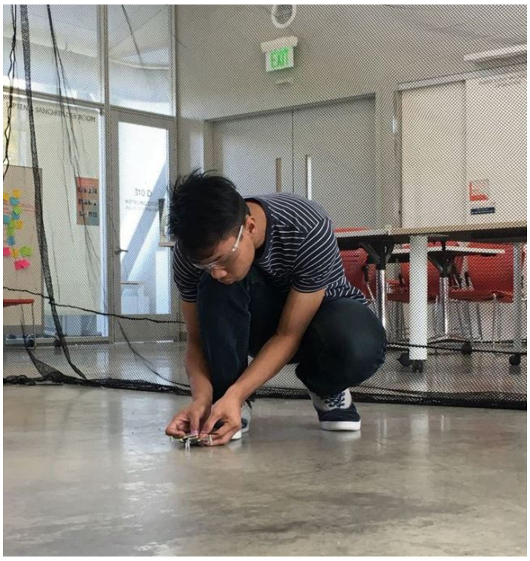


Fig. 5. Setting up the quadcopter for takeoff.

III. Results

A. *Height Estimator* **(Deliverable 2)**

The following data was obtained by performing the height estimator portion of the experiment.

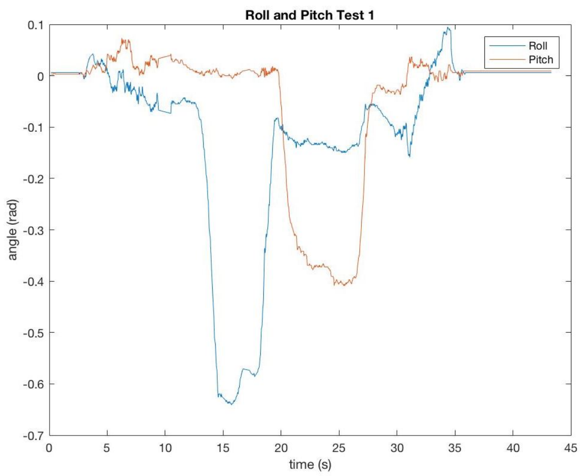


Fig. 6. Roll and pitch estimates for the height estimator- Test 1.

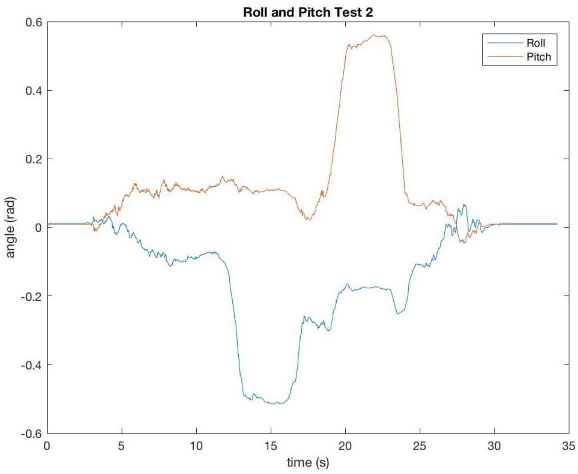


Fig. 7. Roll and pitch estimates for the height estimator- Test 2.

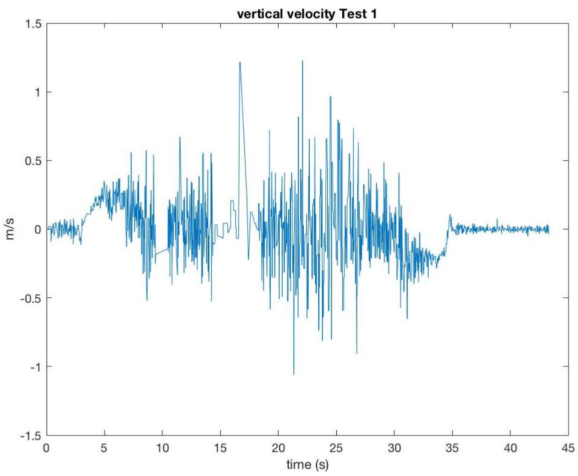


Fig. 8. Vertical velocity estimate for the height estimator- Test 1.

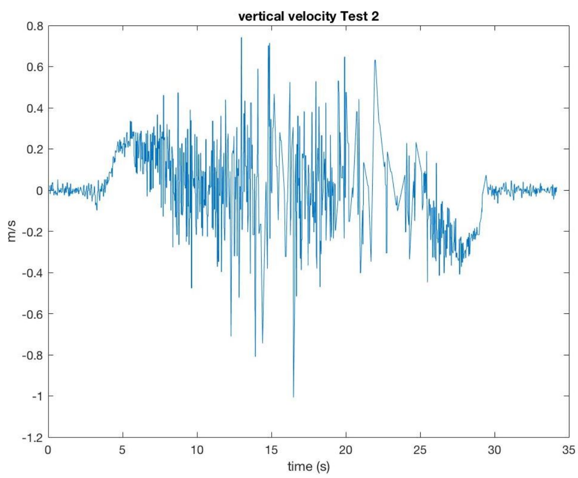


Fig. 9. Vertical velocity estimate for the height estimator- Test 2.

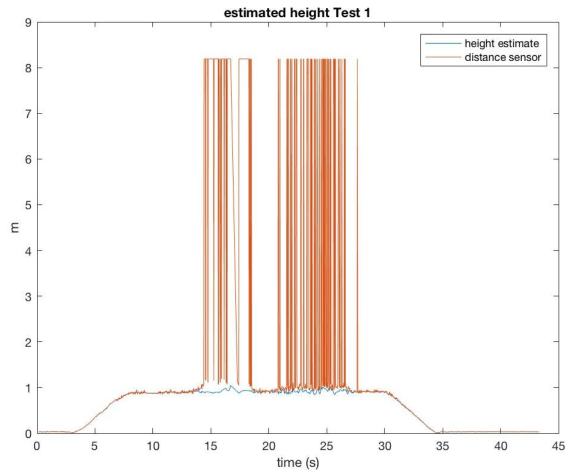


Fig. 10. Estimated height for height estimator- test 1.

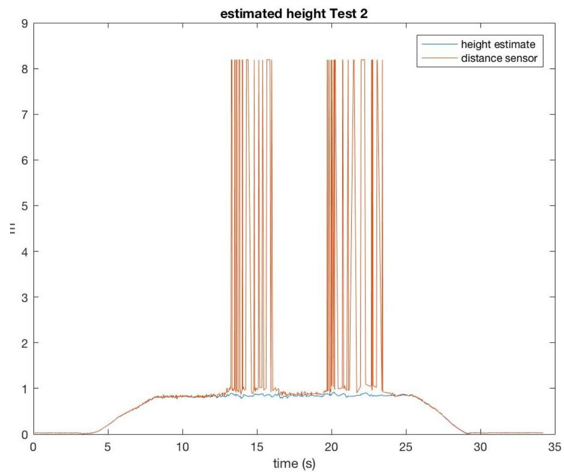


Fig. 11. Estimated height for height estimator- test 2.

From this data, we observe that as the roll and pitch angles deviate from zero degrees, the estimated height increases. The reason for such big values $(h_m(k) > 0.5m)$ is because the drone captures the distance from its body straight down, tilting the drone causes the distance from the sensor to the ground to increase. $Equation$ 1 also supports this argument because as θ and ϕ increase, the height estimated increases. Similarly, the estimated velocity is also affected by the changes in roll and pitch, this is most noticeable when the drone begins to drastically move in the directions of roll and pitch when it seemed to be stable at specific times.

B. Horizontal Estimator **(Deliverable 3)**

The following data was obtained by performing the horizontal estimator portion of the experiment.

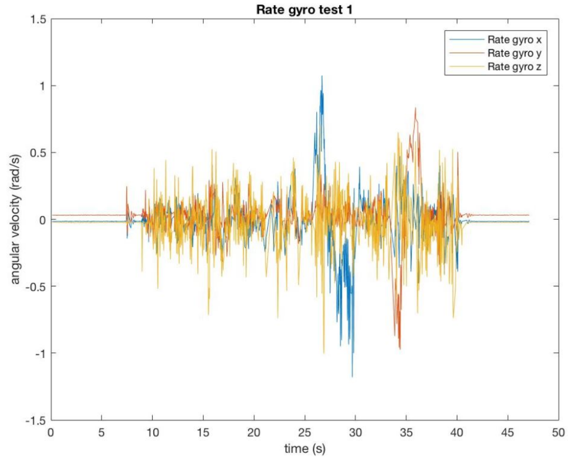


Fig. 12. Rate gyroscope measurements for horizontal estimator- test1.

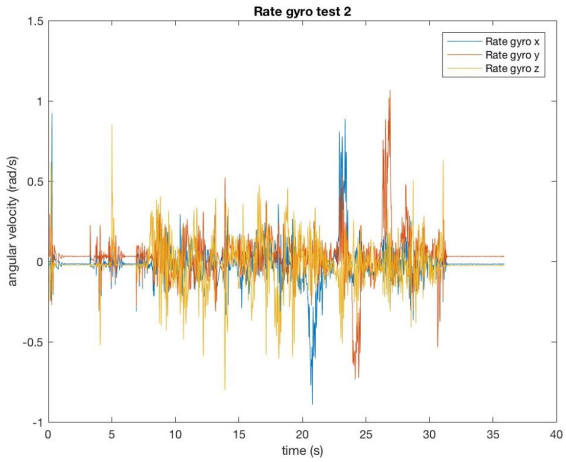


Fig. 13. Rate gyroscope measurements for horizontal estimator- test2.

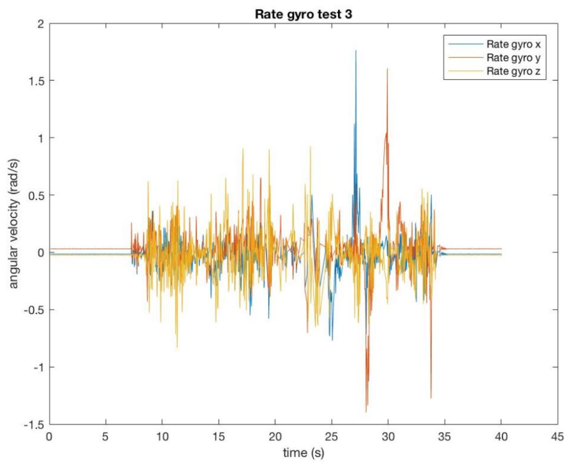


Fig. 14. Rate gyroscope measurements for horizontal estimator- test3.

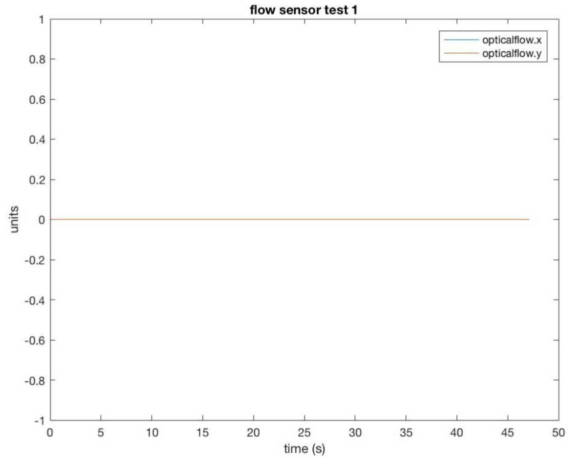


Fig. 15. flow sensor for horizontal estimator- test 1.

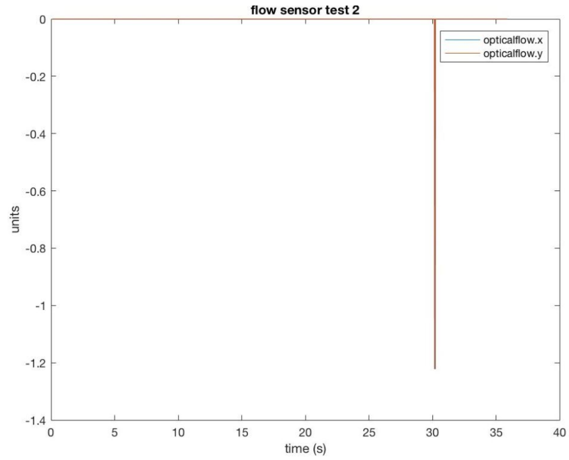


Fig. 16. flow sensor for horizontal estimator- test 2.

unit is wrong [-1]flow sensor scal

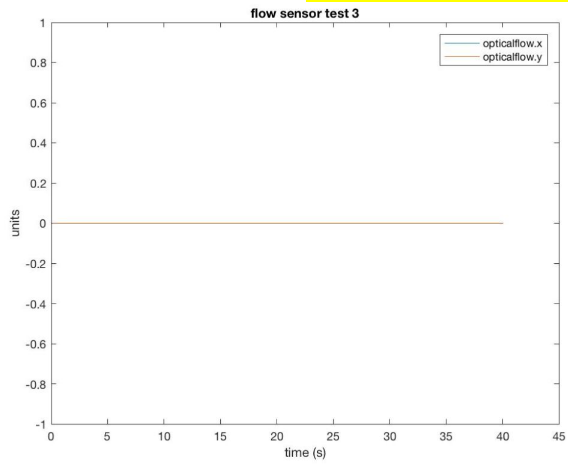


Fig. 17. flow sensor for horizontal estimator- test 3.

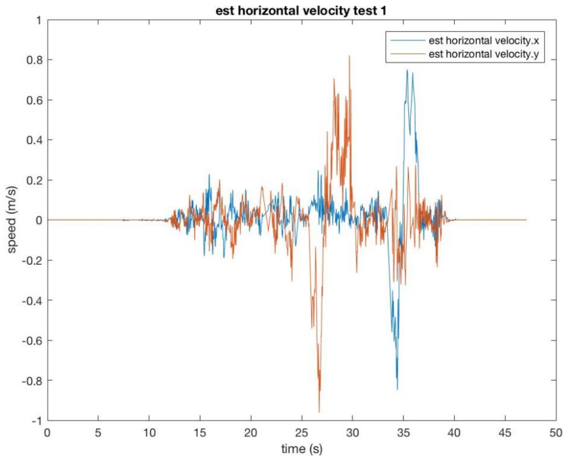


Fig. 18. estimated horizontal velocity for horizontal estimator- test 1.

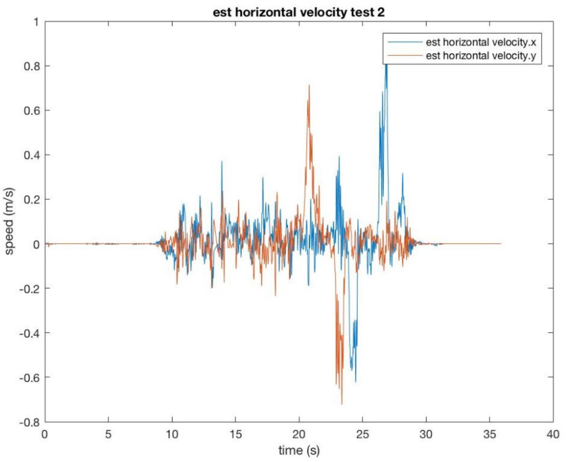


Fig. 19. estimated horizontal velocity for horizontal estimator- test 2.

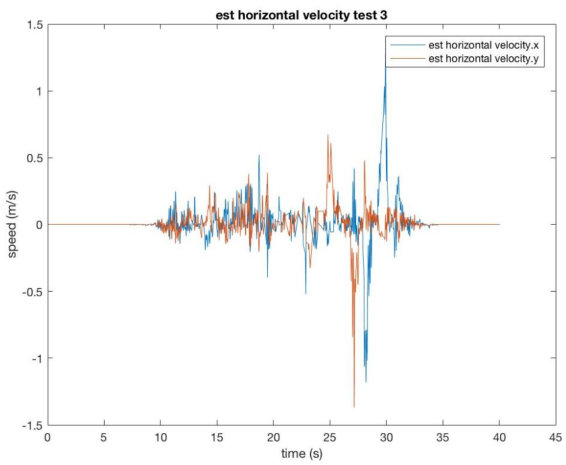


Fig. 20. estimated horizontal velocity for horizontal estimator- test 3.

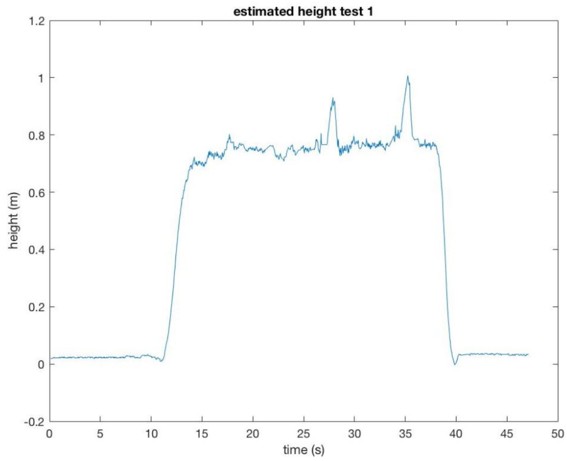


Fig. 21. height estimate for horizontal estimator- test 1.

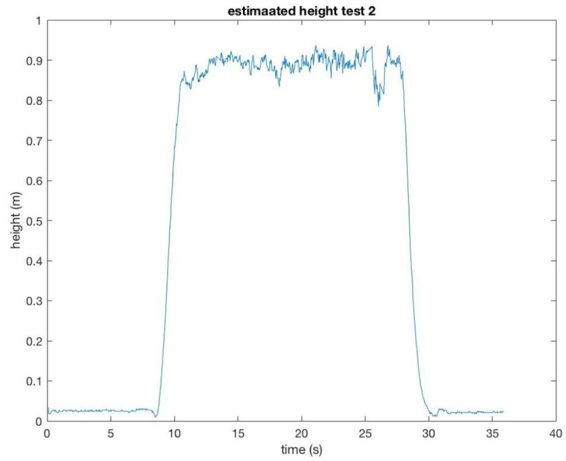


Fig. 22. height estimate for horizontal estimator- test 2.

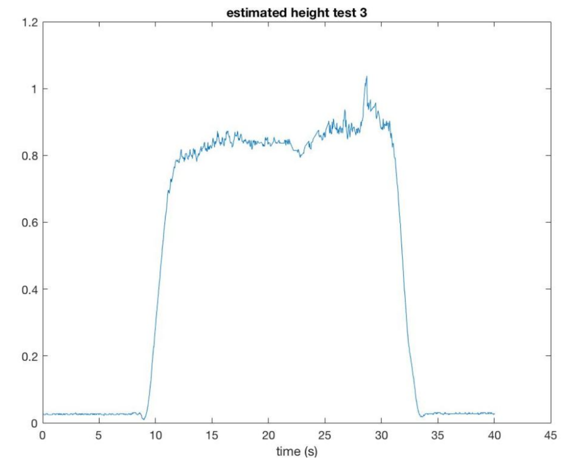


Fig. 23. height estimate for horizontal estimator- test 3.

The rate gyro estimator has a stable performance since it only oscillates averagely between 2 points. Sometimes, the reading will overshoot to a certain value but it will go to the opposite negative value and balance out the percent overshoot. The optical flow performance is good since it gives a constant value.The estimate horizontal velocity was quite stable at the beginning of the experiment. After a certain time (around 10 second), the reading of estimator start to overshoot a little, but it will come back to the stable oscillation. The reading for height estimator is within the expected values. Since the drone did not quite stay in stable height because of walking, the height changes a little bit throughout the experiment. But, the estimator did a good job in getting estimated reading in the expected range.

C. Closed-loop Hover Flight **(Deliverable 4)**

The following data was obtained by hovering the drone as long as possible.

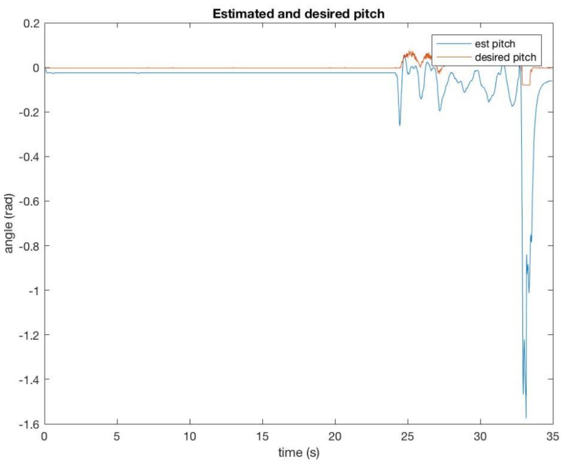


Fig. 24. estimate and desired pitch- test 1.

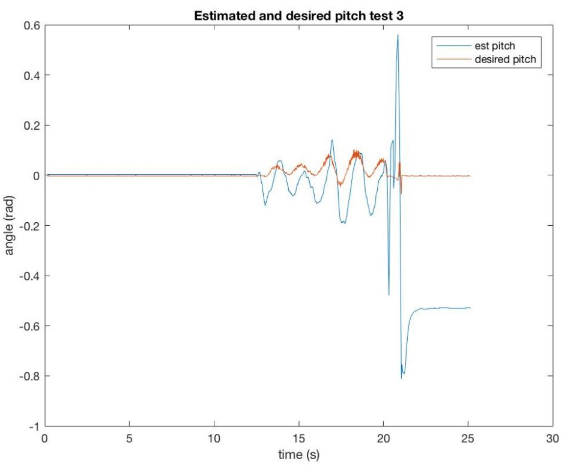


Fig. 25. estimate and desired pitch- test 3.

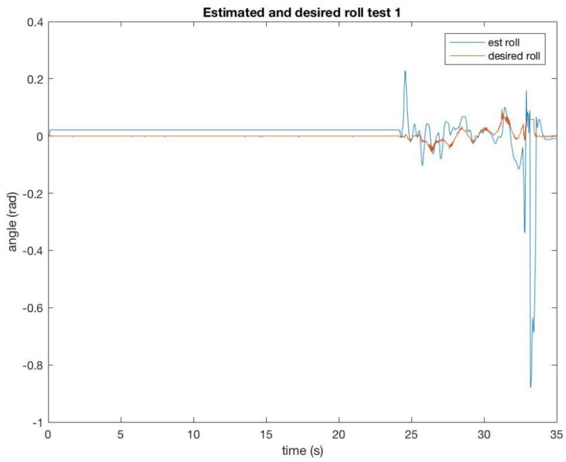


Fig. 26. estimate and desired roll- test 1.

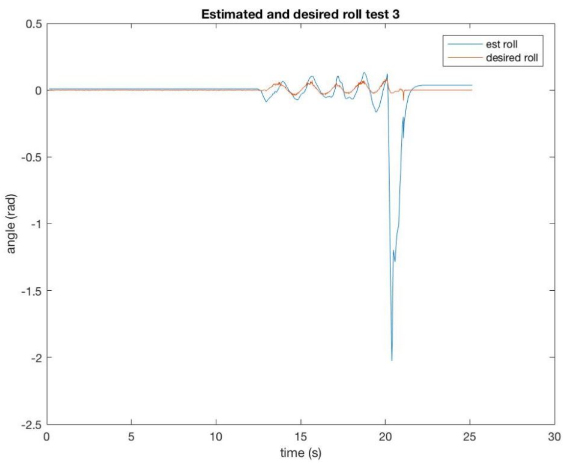


Fig. 27. estimate and desired roll- test 3.

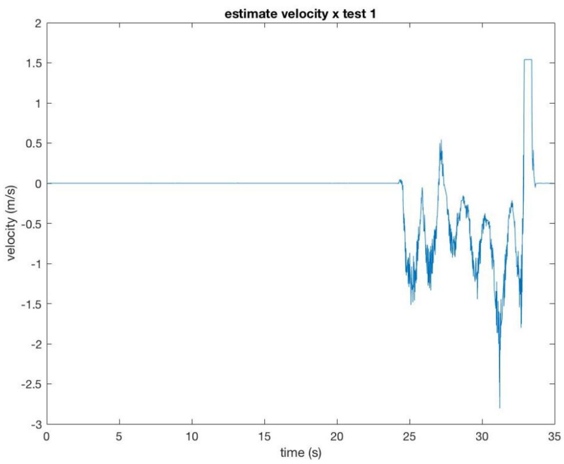


Fig. 28. estimate drone x-velocity- test 1.

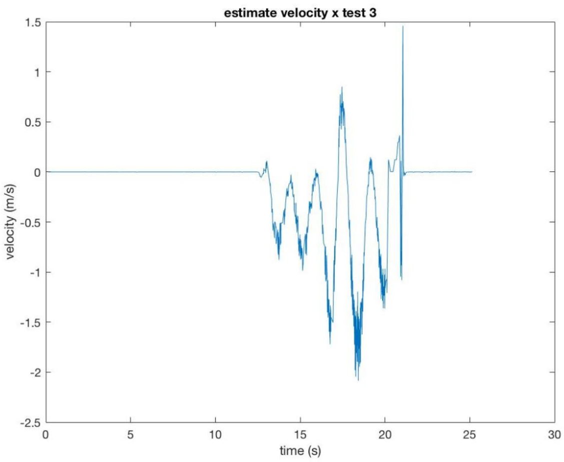


Fig. 29. estimate drone x-velocity- test 3.

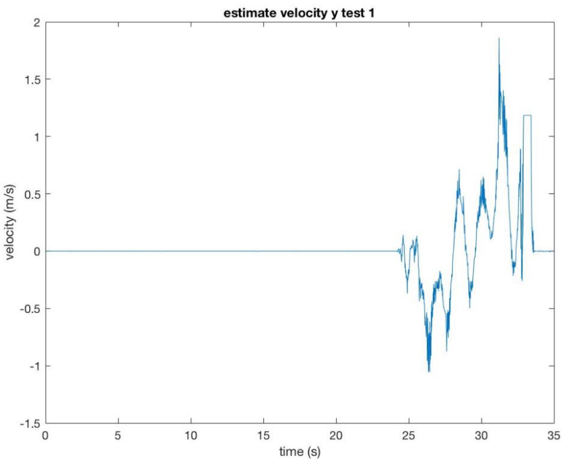


Fig. 30. estimate drone y-velocity- test 1.

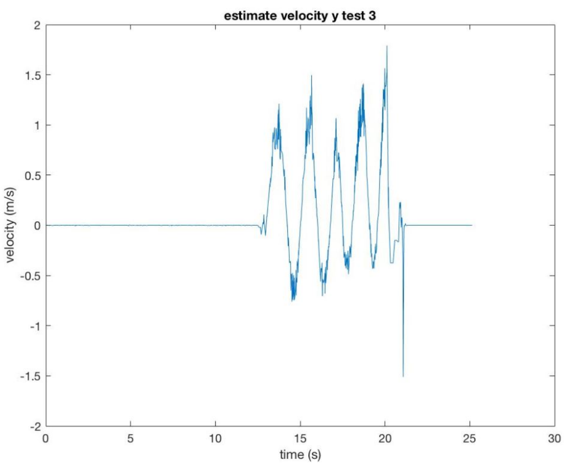


Fig. 31. estimate drone y-velocity- test 3.

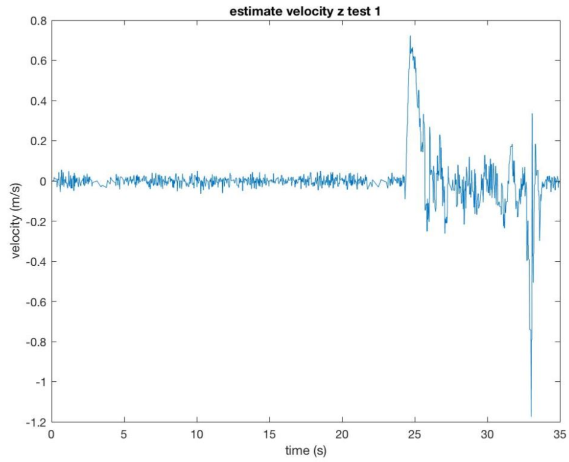


Fig. 32. estimate drone z-velocity- test 1.

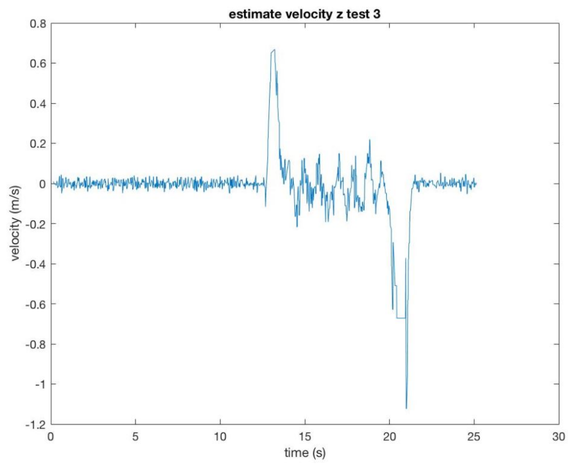


Fig. 33. estimate drone z-velocity- test 3.

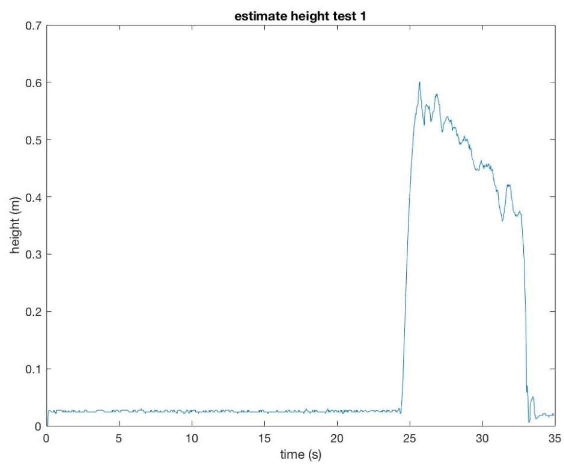


Fig. 34. estimate drone height- test 1.

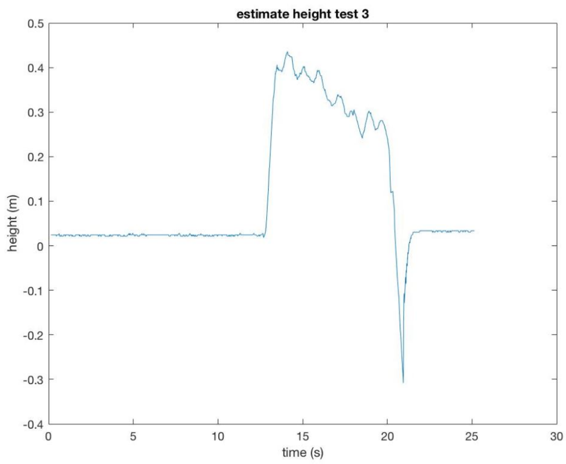


Fig. 35. estimate drone height- test 3.

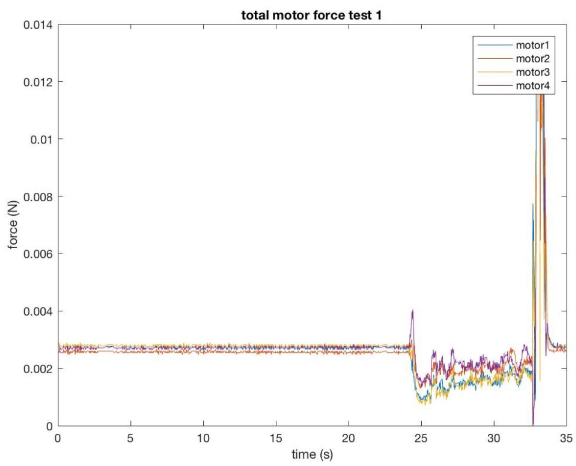


Fig. 36. Total motor force- test 1.

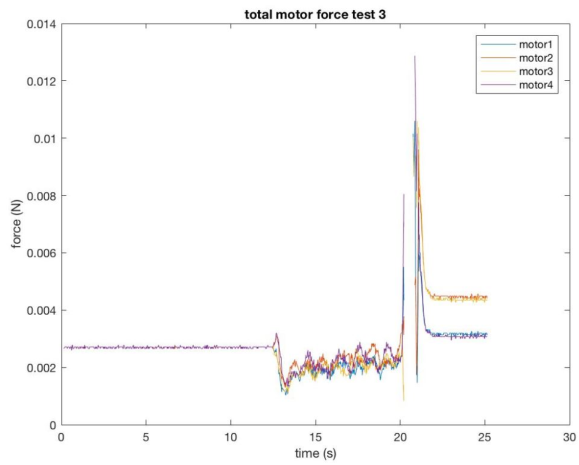


Fig. 37. Total motor force- test 3.

We ran 4 tests for this section of the experiment but chose to present on the two that showed progress had been made.

D. Analysis and Modification **(Deliverable 5)**

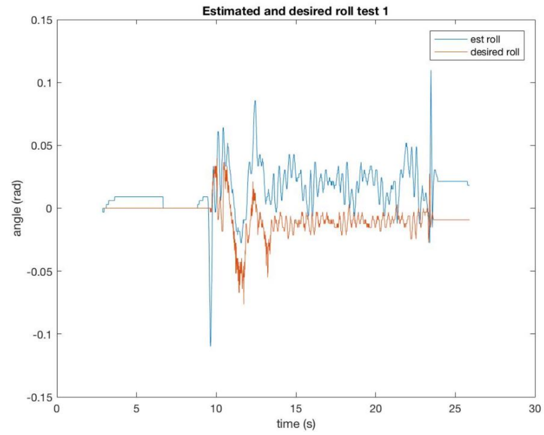


Fig. 38. Estimated and desired roll.

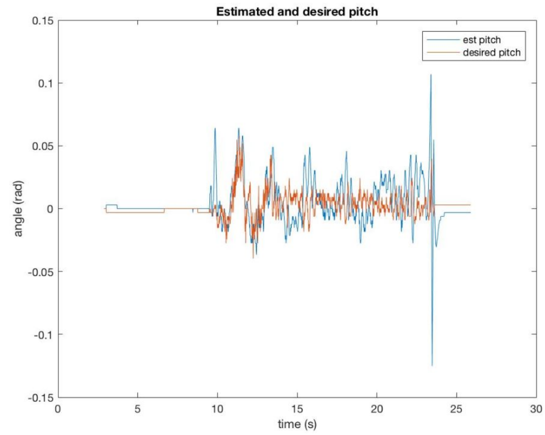


Fig. 39. Estimated and desired pitch.

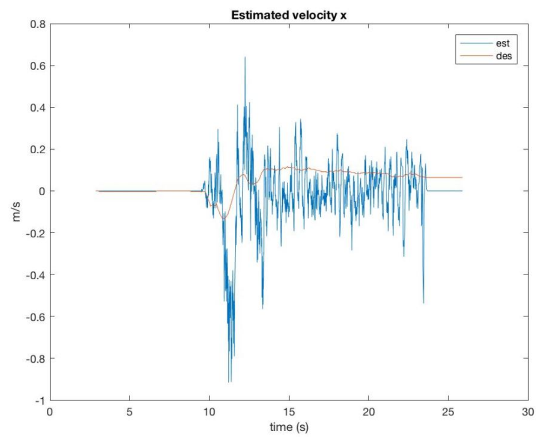


Fig. 40. Estimated velocity in the x- direction.

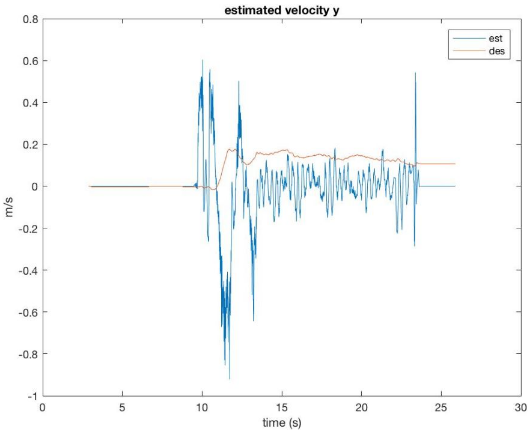


Fig. 41. Estimated velocity in the y- direction.

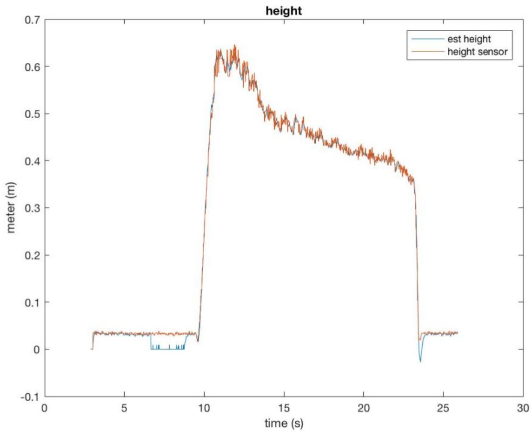


Fig. 42. Estimated height measurement.

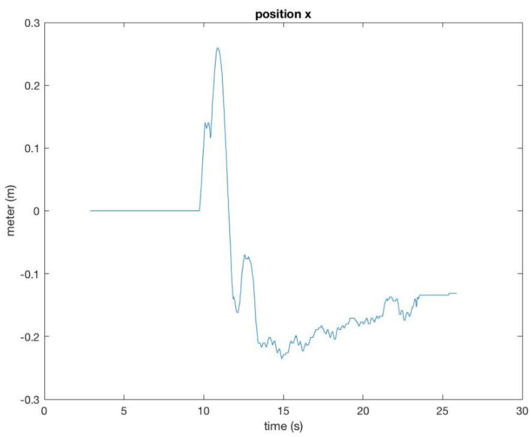


Fig. 43. Estimated position in the x-direction.

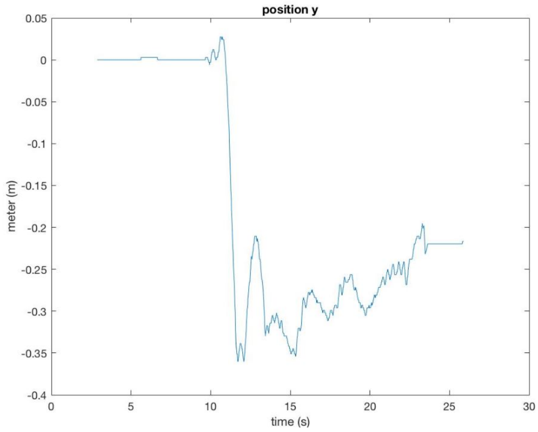


Fig. 44. Estimated position in the y-direction.

From the figures, we were able to see that the drone was not as stable in hovering flight as we would have like since the drone velocity in the y and z direction widely fluctuated. Furthermore, the pitch and roll angle also dropped drastically after 10 seconds. However, the total motor force graphs show that the motor outputs were all closely similar which meant that there was no problem with the motors distributing power unevenly. With all this in mind, we theorized that these behaviours could be because of the imbalance of the center of mass of the drone which led to the resultant tipping towards the end of the flight or because of drift error from the pitch and roll estimators after 10 seconds which could have led to unstable flight towards the end.

The first improvement we made to our original code was to change the time constant for the horizontal velocity, making it smaller so it is more sensitive to changes in horizontal velocity. We then proceeded to implemented the estimator reset button, $(yellow\ button)$ to reset all the calculations before flying again; this was to make sure that our drone would not behave in an unpredictable manner since its calculations were to reset. We also implemented the $green\ button$ to have a smooth landing. We then implemented the position estimator by integrating the estimated velocity.

We could have achieved better performance, what limited the achievement of our drone is that it sometimes keeps flying past the original place of launch and does not return.

IV. Discussion

The main problem we encountered was that the vehicle accelerated in one direction sometimes in a non-predictable manner. After analyzing the motion of the drone in this direction, it was concluded that it was the result of the center of mass being shifted and not located at the geometric center of the drone, thus causing a disturbance torque. The solution to this problem is to slightly shift the battery opposite in direction of where the drone was accelerating to.

Another problem we came across was that during operation, the drone tended to move back and forth and side-to-side. We fixed this problem by increasing the pith/roll and the pitch rate and roll rate to make the drone ”stiffer” and behave more controllable.

A small detail that made a difference during the lab was finding the correct location to hover the drone. We were initially testing the drone on a shiny hard wood floor that seemed to be slightly uneven. We changed the location to the surface of a pool table, this being carpet and the most leveled surface for this project. The carpet is a great material because it does not reflect; consequently, it does not offset the height sensor. The even surface of the pool table made it perfect for the height sensor.

V. Conclusion

With the extensive amount of effort put into the project to debug the program and understand the physical problems/ limitations of the drone and its components, the final version of the drone is ready to perform the task of hovering for over 10 seconds at an altitude of at least 0.5 meters and return to its original location in a smooth manner.

Reference

1. experimental set-up [total of 2 point]2. height estimate [total of 4 points] * 3 plot

[1] *ab 5. Hover Control*. 2018, https://bcourse
files/folder/Labs?preview=74011261

```cpp
#include "UserCode.hpp"
#include "UtilityFunctions.hpp"
#include "Vec3f.hpp"

#include <stdio.h> //for printf

//An example of a variable that persists beyond the function call.
float exampleVariable_float = 0.0f;  //Note the trailing 'f' in the number. This is to force single precision floating point.
Vec3f exampleVariable_Vec3f = Vec3f(0, 0, 0);
int exampleVariable_int = 0;

//We keep the last inputs and outputs around for debugging:
MainLoopInput lastMainLoopInputs;
MainLoopOutput lastMainLoopOutputs;

//Some constants that we may use:
const float mass = 32e-3f;  // mass of the quadcopter [kg]
const float gravity = 9.81f;  // acceleration of gravity [m/s^2]
const float inertia_xx = 16e-6f;  //MMOI about x axis [kg.m^2]
const float inertia_yy = inertia_xx;  //MMOI about y axis [kg.m^2]
const float inertia_zz = 29e-6f;  //MMOI about z axis [kg.m^2]

const float dt = 1.0f / 500.0f; //[s] period between successive calls to MainLoop

//initializes length and coefficient coupling propeller's force to torque
//about propeller's axis of rotation
const float l = 33e-3f;
const float k = 0.01f;

float desHeight = 0.0f;


//*Before* Main Loop

//* this initializes the estimated Gyroscope Bias vector and rate Gyroscope_corrected vector
Vec3f estGyroBias = Vec3f(0, 0, 0);
Vec3f rateGyro_corr = Vec3f(0, 0, 0);

//* initializes command angular accelerations and moment torque vector
Vec3f cmdAngAcc = Vec3f(0,0,0);
Vec3f n = Vec3f(0,0,0);
Vec3f cmdAngVel = Vec3f(0,0,0);

//initializes estimated roll, pitch, and yaw variables
//as well as measured pitch and roll variables and weighting constant ro
```

```

float estRoll = 0.0f;
float estPitch = 0.0f;
float estYaw = 0.0f;
float measPitch = 0.0f;
float measRoll = 0.0f;
float ro = 0.01f;

// initializes desired total force and forces on each propeller from mixer
//float des_tot_force = 8.0f*mass;
float cp1 = 0.0f;
float cp2 = 0.0f;
float cp3 = 0.0f;
float cp4 = 0.0f;

// initializes torque variables in x,y,z directions
float nx = 0.0f;
float ny = 0.0f;
float nz = 0.0f;

// initializes angular velocity and angular acceleration in x,y,z directions
float p_des = 0.0f;
float r_des = 0.0f;
float q_des = 0.0f;
float Roll_des = 0.0f;
float Pitch_des = 0.0f;
float Yaw_des = 0.0f;

// defining constant for vertical velocity
float estHeight = 0.0f;
float estVelocity_1 = 0.0f;
float estVelocity_2 = 0.0f;
float estVelocity_3 = 0.0f;
float lastHeightMeas_meas = 0.0f;
float lastHeightMeas_time = 0.0f;

// defining constant for position estimators
float estPosition_x = 0.0f;
float estPosition_y = 0.0f;
float const desPosition_x = 0.0f;
float const desPosition_y = 0.0f;
float desVelocity_1 = 0.0f;
float desVelocity_2 = 0.0f;



MainLoopOutput MainLoop(MainLoopInput const &in) {
  //Your code goes here
```

```
//*inside* Main Loop

//initializes time constants for roll, pitch, and yaw rates
float const timeConstant_rollRate = 0.04f;
float const timeConstant_pitchRate = timeConstant_rollRate;
float const timeConstant_yawRate = 0.1f;

//initializes time constants for roll, pitch, and yaw angle
float const timeConstant_rollAngle = 0.12f;
float const timeConstant_pitchAngle = timeConstant_rollAngle;
float const timeConstant_yawAngle = 0.2f;

//initializes time constants for horizontal velocity, height , position
float const timeConst_horizVel = 1.5f;
float const natFreq_height= 2.0f;
float const dampingRatio_height= 0.7f;

float const timeConstant_position= 2.0f;



//calculated rate gyroscope bias by measuring the rate gyroscope for a second's worth of measurement and assuming
//the drone wasn't moving
//the corrected rate gyroscope data was found by subtracting the measured rate gyroscope with the estimated
//gyroscope bias
if(in.currentTime < 1.0f) {
  estGyroBias = estGyroBias + (in.imuMeasurement.rateGyro / 500.0f);
}
rateGyro_corr = in.imuMeasurement.rateGyro - estGyroBias;

//implements a single axis attitude estimator for the pitch, roll, and yaw angles
//by integrating the corrected rate gyroscope data with a time step of 0.02
// and adding it to the current estimated pitch, roll, and yaw values
// estPitch  = estPitch + dt*rateGyro_corr.y;
// estRoll  = estRoll + dt*rateGyro_corr.x;



//calculates pitch and roll angles from accelerometer data in x and y
  measPitch = -in.imuMeasurement.accelerometer.x / gravity;
  measRoll = in.imuMeasurement.accelerometer.y / gravity;

//implements roll and pitch estimators that factor in measured pitch and
```

```
roll angles
  // to prevent drifting from occuring
  // ro and (1-ro) show confidence in whether we trust our estimated Euler
angles or our measured Euler angles
  // with higher ro meaning that we trust our measured roll and pitch angles
more while a smaller ro
  // means we trust our estimated roll and pitch angles more

  // reset all the estimator zero
    if ( in.joystickInput.buttonYellow == true) {
        estRoll= 0.0f;
        estPitch= 0.0f;
        estYaw= 0.0f;
        estPosition_x=0.0f;
        estPosition_y=0.0f;
        estHeight=0.0f;
        estVelocity_1=0.0f;
        estVelocity_2=0.0f;
        estVelocity_3=0.0f;

      }

  //  integrates corrected rate gyro to obtain estimated angle
  estRoll = (estRoll +dt*rateGyro_corr.x) * (1.0f-ro) + ro*(measRoll);
  estPitch = (estPitch +dt*rateGyro_corr.y) * (1.0f-ro) + ro*(measPitch);
  estYaw = estYaw + dt*rateGyro_corr.z;


  // height estimator
  // prediction step
  estHeight= estHeight+estVelocity_3*dt;
  estVelocity_3= estVelocity_3+0.0f*dt;

  // the correction step
  float const mixHeight =0.3f;
  if (in.heightSensor.updated) {
    if (in.heightSensor.value<5.0f) {
      float hMeas= in.heightSensor.value * cosf(estRoll) * cosf(estPitch);
      estHeight = (1-mixHeight)* estHeight +mixHeight*hMeas;

      float v3Meas = (hMeas - lastHeightMeas_meas)
          /(in.currentTime-lastHeightMeas_time);

      estVelocity_3= (1-mixHeight)*estVelocity_3+mixHeight*v3Meas;
      lastHeightMeas_meas= hMeas;
      lastHeightMeas_time=in.currentTime;
    }
  }
```

```

  //prediction
  //(just assume velocity is constant)
  estVelocity_1=estVelocity_1+0.0f*dt;
  estVelocity_2=estVelocity_2+0.0f*dt;

  //correction step
  float const mixHorizVel = 0.1f;
  if (in.opticalFlowSensor.updated) {
    float sigma_1 = in.opticalFlowSensor.value_x;
    float sigma_2 = in.opticalFlowSensor.value_y;
    float div = (cosf(estRoll)*cosf(estPitch));
    if (div>0.5f) {
      float deltaPredict = estHeight/div;
      float v1Meas= (-sigma_1 + in.imuMeasurement.rateGyro.y)*deltaPredict;
      float v2Meas= (-sigma_2 - in.imuMeasurement.rateGyro.x)*deltaPredict;
      estVelocity_1= (1-mixHorizVel)* estVelocity_1 + mixHorizVel* v1Meas;
      estVelocity_2= (1-mixHorizVel)* estVelocity_2 + mixHorizVel* v2Meas;

    }
  }

 // position estimator by integrating velocity

    estPosition_x=estVelocity_1*dt+estPosition_x;
    estPosition_y=estVelocity_2*dt+estPosition_y;

    desVelocity_1= -(1.0f/timeConstant_position)*( estPosition_x - desPosition_x);
    desVelocity_2= -(1.0f/timeConstant_position)*( estPosition_y - desPosition_y);

 // obtaining desired  acceleration from velocity
  float desAcc1= -(1.0f/ timeConst_horizVel)*(estVelocity_1-desVelocity_1);
  float desAcc2= -(1.0f/ timeConst_horizVel)*(estVelocity_2-desVelocity_2);

 // obtain desired roll, pitc and yaw
  Roll_des= -desAcc2/ gravity;
  Pitch_des= desAcc1/ gravity;
  Yaw_des= 0.0f;

 // desired height
  desHeight = 1.0f;

 // for smooth landing
  if ( in.joystickInput.buttonGreen == true) {
     desHeight = 0.5f;
  }
```

```cpp
// obtaining desired total acceleration from height choosen

  const float desAcc3 = -2.0f *dampingRatio_height*natFreq_height*estVelocity_3
      -natFreq_height*natFreq_height*(estHeight-desHeight);
  float desNormalizedAcceleration = (gravity+desAcc3)/ (cosf(estRoll)*cosf(estPitch));

  // float Pitch_des = 0.0f;
  // implements 30 angle degree tilt when pressing blue button
  //if ( in.joystickInput.buttonBlue == true) {
  //  Roll_des = -0.081f;
  //}

  // Commmand angular velocity
    cmdAngVel.x = (-1.0f/timeConstant_rollAngle)*( estRoll - Roll_des+0.0523f);
    cmdAngVel.y = (-1.0f/timeConstant_pitchAngle)*( estPitch - Pitch_des);
    cmdAngVel.z = (-1.0f/timeConstant_yawAngle)*( estYaw - Yaw_des);

  // Commmand angular acceleration
    cmdAngAcc.x = (-1.0f/timeConstant_rollRate)*(rateGyro_corr.x - cmdAngVel.x);
    cmdAngAcc.y = (-1.0f/timeConstant_pitchRate)*(rateGyro_corr.y - cmdAngVel.y);
    cmdAngAcc.z = (-1.0f/timeConstant_yawRate)*(rateGyro_corr.z - cmdAngVel.z);


  // Calculating torque in x,y,z direction
  nx = 16.0e-6f*cmdAngAcc.x+13.0e-6f*cmdAngVel.y*cmdAngVel.z;
  ny = 16.0e-6f*cmdAngAcc.y-13.0e-6f*cmdAngVel.x*cmdAngVel.z;
  nz = 29.0e-6f*cmdAngAcc.z;

  // Mixer to convert desired torque vector and total force to four individual
  // motor forces
  cp1 = (1.0f/4.0f)*(desNormalizedAcceleration*mass + (nx/l) -(ny/l) + ( nz/k));
  cp2 = (1.0f/4.0f)*(desNormalizedAcceleration*mass - (nx/l) -(ny/l) - ( nz/k));
  cp3 = (1.0f/4.0f)*(desNormalizedAcceleration*mass - (nx/l) +(ny/l) + ( nz/k));
  cp4 = (1.0f/4.0f)*(desNormalizedAcceleration*mass + (nx/l) +(ny/l) - ( nz/k));


  // The function input (named "in") is a struct of type
```

```
  // "MainLoopInput". You can understand what values it
  // contains by going to its definition (click on "MainLoopInput",
  // and then hit <F3> -- this should take you to the definition).
  // For example, "in.joystickInput.buttonBlue" is true if the
  // joystick's blue button is pushed, false otherwise.

  //Define the output numbers (in the struct outVals):
  MainLoopOutput outVals;
//  motorCommand1 -> located at body +x +y
//  motorCommand2 -> located at body +x -y
//  motorCommand3 -> located at body -x -y
//  motorCommand4 -> located at body -x +y
//  outVals.motorCommand1 = 0.0f;
//  outVals.motorCommand2 = 0.0f;
//  outVals.motorCommand3 = 0.0f;
//  outVals.motorCommand4 = 0.0f;

  //adds estimate angles to the telemetry channels
    outVals.telemetryOutputs_plusMinus100[0] = estRoll;
    outVals.telemetryOutputs_plusMinus100[1] = estPitch;
    outVals.telemetryOutputs_plusMinus100[2] = estHeight;


 // adds command angular accelerations to the telemetry channels
    outVals.telemetryOutputs_plusMinus100[3] = Roll_des;
    outVals.telemetryOutputs_plusMinus100[4] = Pitch_des;
 //   outVals.telemetryOutputs_plusMinus100[5] = estVelocity_3;
    outVals.telemetryOutputs_plusMinus100[5] = estVelocity_1;
 // adds command angular velocity to the telemetry channels
    outVals.telemetryOutputs_plusMinus100[6] = estVelocity_2;
    outVals.telemetryOutputs_plusMinus100[7] = desVelocity_1;
    outVals.telemetryOutputs_plusMinus100[8] = desVelocity_2;

 // adds command pitch desired to the telemetry channels
 //   outVals.telemetryOutputs_plusMinus100[9] = desNormalizedAcceleration;
    outVals.telemetryOutputs_plusMinus100[9] = estPosition_x;
    outVals.telemetryOutputs_plusMinus100[10] = estPosition_y;
    outVals.telemetryOutputs_plusMinus100[11] = estYaw;

  //if ( in.joystickInput.buttonBlue == true) {
      outVals.motorCommand1 = pwmCommandFromSpeed(speedFromForce(cp1));
      outVals.motorCommand2 = pwmCommandFromSpeed(speedFromForce(cp2));
      outVals.motorCommand3 = pwmCommandFromSpeed(speedFromForce(cp3));
      outVals.motorCommand4 = pwmCommandFromSpeed(speedFromForce(cp4));

  //if ( in.joystickInput.buttonBlue == true) {
  //  outVals.motorCommand1 = 130.0f;
  //  outVals.motorCommand2 = 130.0f;
```

```
  //  outVals.motorCommand3 = 130.0f;
  //  outVals.motorCommand4 = 130.0f;
  // }


  //copy the inputs and outputs:
  lastMainLoopInputs = in;
  lastMainLoopOutputs = outVals;
  return outVals;
}


void PrintStatus() {
  //For a quick reference on the printf function, see: http://www.cplusplus.com/reference/cstdio/printf/
  // Note that \n is a "new line" character.
  // Also, note that to print a `float` variable, you have to explicitly cast it to
  //  `double` in the printf function, and explicitly specify precision using something
  //  like %6.3f (six significant digits, three after the period). Example:
  //   printf("  exampleVariable_float = %6.3f\n", double(exampleVariable_float));


  //prints accelerometer measurement and rate gyroscope measurements in x,y,z directions
  printf("Acc: ");
  printf("x=%6.3f, ",
         double(lastMainLoopInputs.imuMeasurement.accelerometer.x));
  printf("\n");  //new line
  printf("y=%6.3f, ",
           double(lastMainLoopInputs.imuMeasurement.accelerometer.y));
  printf("\n");  //new line
  printf("z=%6.3f, ",
           double(lastMainLoopInputs.imuMeasurement.accelerometer.z));
  printf("\n");  //new line
  printf("Gyro: ");
  printf("x=%6.3f, ", double(lastMainLoopInputs.imuMeasurement.rateGyro.x));
  printf("\n");  //new line
  printf("y=%6.3f, ", double(lastMainLoopInputs.imuMeasurement.rateGyro.y));
  printf("\n");  //new line
  printf("z=%6.3f, ", double(lastMainLoopInputs.imuMeasurement.rateGyro.z));
  printf("\n");  //new line



//prints estimated rateGyro bias in x,y,z directions
```

```

 printf("Estimated Bias: ");
 printf("x=%6.3f, ",
            double(estGyroBias.x));
 printf("\n");  //new line
 printf("y=%6.3f, ",
            double(estGyroBias.y));
 printf("\n");  //new line
 printf("z=%6.3f, ",
            double(estGyroBias.z));
 printf("\n");  //new line

//prints corrected rate gyroscope measurements
 printf("Corrected Rate Gyro: ");
   printf("x=%6.3f, ",
            double(rateGyro_corr.x));
   printf("\n");  //new line
   printf("y=%6.3f, ",
            double(rateGyro_corr.y));
   printf("\n");  //new line
   printf("z=%6.3f, ",
            double(rateGyro_corr.z));
   printf("\n");  //new line

 // prints estimated Euler angles
   printf("Estimated Roll: ");
      printf("x=%6.3f, ",
               double(estRoll));
      printf("\n");  //new line
      printf("Estimated Pitch: ");
      printf("y=%6.3f, ",
                 double(estPitch));
      printf("\n");  //new line
      printf("Estimated Yaw: ");
      printf("z=%6.3f, ",
                 double(estYaw));
      printf("\n");  //new line

 // prints cp constants
         printf("cp1: ");
            printf("cp1=%6.3f, ",
                     double(cp1));
         printf("cp2: ");
            printf("cp2=%6.3f, ",
                     double(cp2));
         printf("cp3: ");
            printf("cp3=%6.3f, ",
                     double(cp3));
```

```
          printf("cp4: ");
             printf("cp4=%6.3f, ",
                     double(cp4));




// printf("Example variable values:\n");
// printf("  exampleVariable_int = %d\n", exampleVariable_int);
  //Note that it is somewhat annoying to print float variables.
  //  We need to cast the variable as double, and we need to specify
  //  the number of digits we want (if you used simply "%f", it would
  //  truncate to an integer.
  //  Here, we print 6 digits, with three digits after the period.
// printf("  exampleVariable_float = %6.3f\n", double(exampleVariable_float));

  //We print the Vec3f by printing it's three components independently:
  //printf("  exampleVariable_Vec3f = (%6.3f, %6.3f, %6.3f)\n",
  //       double(exampleVariable_Vec3f.x), double(exampleVariable_Vec3f.y),
  //       double(exampleVariable_Vec3f.z));

  //just an example of how we would inspect the last main loop inputs and outputs:
  printf("Last main loop inputs:\n");
  printf("  batt voltage = %6.3f\n",
         double(lastMainLoopInputs.batteryVoltage.value));
  printf("  JS buttons: ");
  if (lastMainLoopInputs.joystickInput.buttonRed)
    printf("buttonRed ");
  if (lastMainLoopInputs.joystickInput.buttonGreen)
    printf("buttonGreen ");
  if (lastMainLoopInputs.joystickInput.buttonBlue)
    printf("buttonBlue ");
  if (lastMainLoopInputs.joystickInput.buttonYellow)
    printf("buttonYellow ");
  if (lastMainLoopInputs.joystickInput.buttonStart)
    printf("buttonStart ");
  if (lastMainLoopInputs.joystickInput.buttonSelect)
    printf("buttonSelect ");
  printf("\n");
  printf("Last main loop outputs:\n");

  //prints out the motor commands for all 4 motors
  printf("  motor command 1 = %6.3f\n",
           double(lastMainLoopOutputs.motorCommand1));
  printf("  motor command 2 = %6.3f\n",
           double(lastMainLoopOutputs.motorCommand2));
```

```
printf("  motor command 3 = %6.3f\n",
          double(lastMainLoopOutputs.motorCommand3));
printf("  motor command 4 = %6.3f\n",
          double(lastMainLoopOutputs.motorCommand4));
printf("  Last range = %6.3fm, ",
          double(lastMainLoopInputs.heightSensor.value));
printf("  Last flow: x= %6.3f, y=%6.3f\n",
          double(lastMainLoopInputs.opticalFlowSensor.value_x),
          double(lastMainLoopInputs.opticalFlowSensor.value_y));

// prints position values
          printf("est position x: ");
             printf("est position x=%6.3f, ",
                      double(estPosition_x));
          printf("est position y: ");
             printf("est position y=%6.3f, ",
                      double(estPosition_y));
          printf("des position x: ");
             printf("des position x=%6.3f, ",
                      double(desPosition_x));
          printf("des position y: ");
             printf("des position y=%6.3f, ",
                      double(desPosition_y));

}

```

HW 5

(1) $\dot{x} = f(x,u) = \begin{bmatrix} x_2(x_1^2+1) \\ -x_1 + u \end{bmatrix}$

for equilibria find x^* where $\dot{x}=0$ $\qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

$$\begin{bmatrix} x_2(x_1^2+1) \\ -x_1 + u \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$x_2^*(x_1^{*2}+1) = 0 \qquad -x_1^* + u^* = 0$

$x_2^* = 0$ or $\cancel{x_1^* = i}$ $(x_1 \geq 0)$ $\qquad x_1^* = u^*$

$x^* = \begin{bmatrix} u^* \\ 0 \end{bmatrix}$ for any u^*

4/4

(2) $\dot{x} = f(x,u) = ux^3 + 2x + u$

$x^* = 0,\ u^* = 0$ is equilibria $\qquad u = -kx$

$A = \left[\frac{\partial f}{\partial x}\right] \qquad B = \left[\frac{\partial f}{\partial u}\right]$

$A = [3ux^2+2] \qquad B = [x^3+1]$

$$\frac{\partial F}{\partial x} = [3ux^2+2]\overset{x}{[\ ]} + [x^3+1]\overset{u}{[\ ]}$$

plug in x^*, u^*

$$A' = \frac{\partial F}{\partial x} = [2]\overset{x}{[\ ]} + [1][\ ]\ u$$

plug in $u = -kx$

$$\cancel{A' = [2][x] + [1][-kx]}$$

$$A' = [2]\,x + [1]\,-kx$$

$$= 2x - kx$$

$$A' = [2-k]\,x$$

SISOL means $2-k<0$

$\boxed{k>2}$ for system to be stable

4/4

(3) $q_{cmd} = -\frac{1}{\tau_\theta}(\hat{\theta} - \theta_{cmd})$

$\dot{q}_{cmd} = -\frac{1}{\tau_q}(\hat{q} - q_{cmd})$

$n_2 = J_{xx}\,\dot{q}_{cmd}$

$\theta = \theta_{cmd} = 0 \quad \hat{\theta} = \theta \quad \hat{q} = q$ $\quad J_{xx} = 16 \cdot 10^{-6}\ kg/m^2$

$|n_2| \leq \bar{n}_2 = 3.9 \cdot 10^{-3}\ N \cdot m$

$|\theta| \leq \bar{\theta} = \frac{\pi}{6}\ rad$

$|q| \leq \bar{q} = 20\ rad/s$

$\cancel{\tau_\theta = 2\tau_q}$ $\tau_\theta = 2\tau_q$

$q_{cmd} = -\frac{1}{\tau_\theta}(\theta - \theta_{cmd}) \quad \dot{q}_{cmd} = -\frac{1}{\tau_q}(q - q_{cmd})$

$\dot{q}_{cmd} = \frac{n_2}{J_{xx}} = \frac{3.9 \cdot 10^{-3}\ N \cdot m}{16 \cdot 10^{-6}\ kg/m^2} = 243.75\ rad/s^2$

$q_{cmd} = -\frac{1}{\tau_\theta}(\theta - 0) = -\frac{1}{\tau_\theta}\theta = -\frac{1}{\tau_\theta} \cdot \frac{\pi}{6}\ rad$

$\dot{q}_{cmd} = -\frac{1}{\tau_q}\left(\frac{20\ rad}{s} - \frac{1}{\tau_\theta} \cdot \frac{\pi}{6}\ rad\right)$

$\dot{q}_{cmd} = -\frac{1}{\tau_q} \cdot \left(\frac{20\ rad}{s} - \frac{1}{2} \cdot \frac{1}{\tau_q} \frac{\pi}{6}\ rad\right)$

$$243.75 \, \text{rad}/s^2 = \ddot{q}_{cmd} = +\frac{\pi \, \text{rad}}{12}\left(\frac{1}{\tau_q}\right)^2 - 20 \frac{\text{rad}}{s}\left(\frac{1}{\tau_q}\right)$$

$$x = \frac{1}{\tau_q}$$

$$0 = \frac{\pi}{12} \text{rad} \left(\frac{1}{\tau_q}\right)^2 - 20 \frac{\text{rad}}{s}\left(\frac{1}{\tau_q}\right) - 243.75 \, \text{rad}/s^2$$

$$0 = \frac{\pi}{12} x^2 - 20 x - 243.75$$

$$x = \frac{+20 \pm \sqrt{400 - 4 \cdot \frac{\pi}{12} \cdot (-243.75)}}{\left(\frac{2\pi}{12}\right)}$$

$$x = 87.086 \text{ or } \cancel{-10.691}$$

$$\tau_q = \frac{1}{x} = \frac{1}{87.086} = 0.0115$$

$$\tau_\theta = 0.023$$

~0.094s, 0.188s

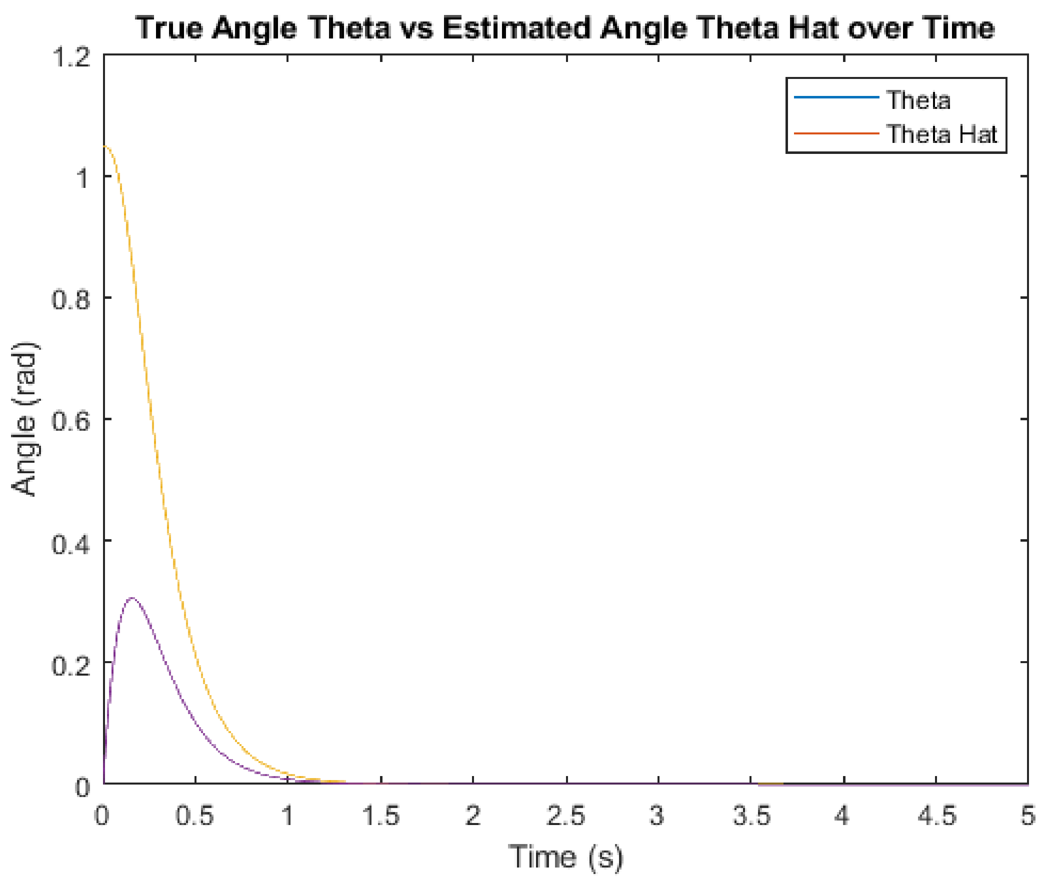
4/6

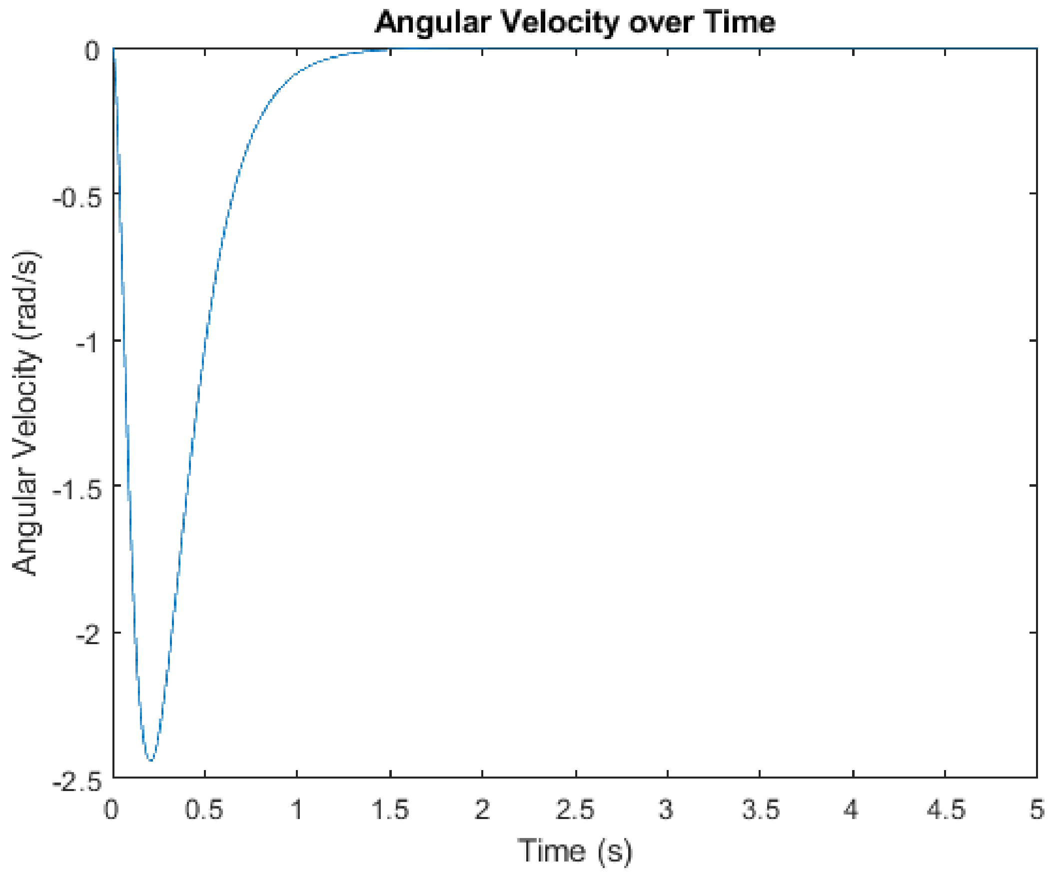
HW 5 Problem 4

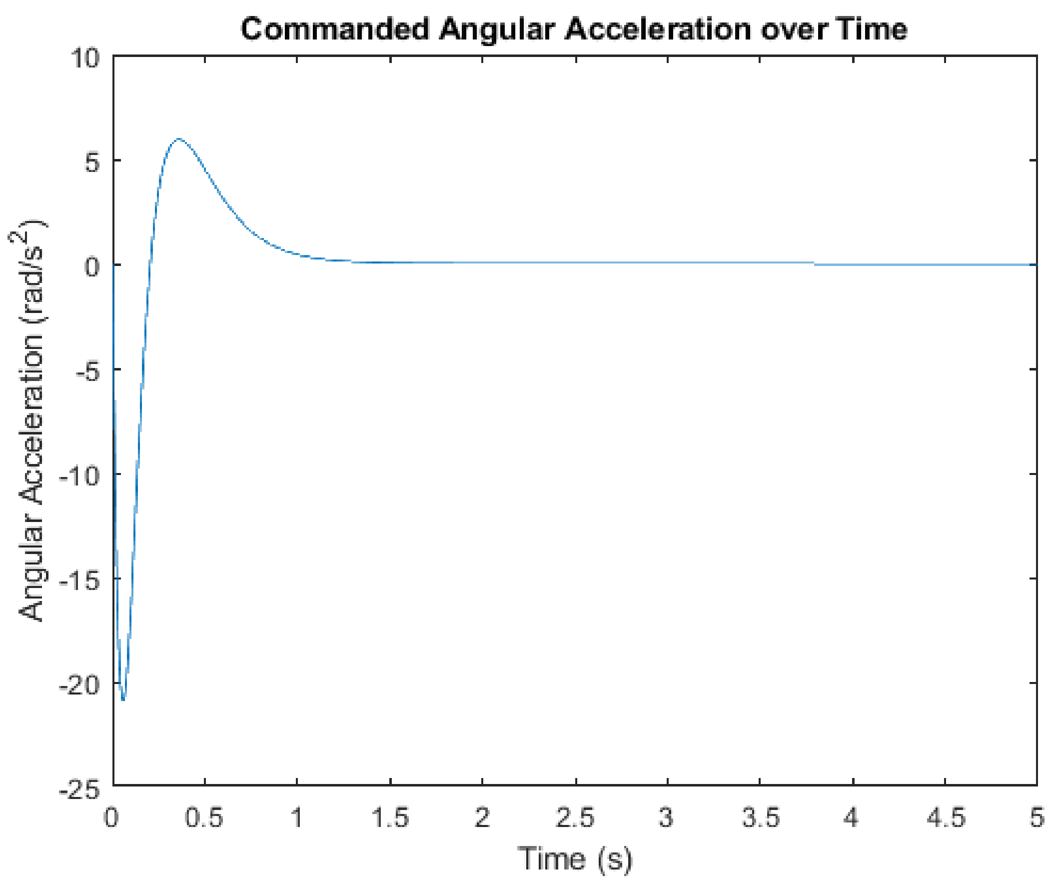
initial conditions

```
theta_hat(1) = 0;
q_hat(1) = 0;
theta(1) = pi/3;
q(1) = 0;
theta_command = 0;

time = [0:0.002:5];

%constants
delta_t = 0.002;
ro = 0.01;
g = 9.81;
tau_q = 0.04;
tau_theta = 0.12;


for k = 1:(5/0.002)
    %simulating rate gyroscope
    y2(k) = q(k);

    %estimator of rate gyro
    q_hat(k) = y2(k);

    %commanded inputs
    q_cmd(k) =  (-1/tau_theta) * (theta_hat(k) - theta_command);
    q_dot_cmd(k) = (-1/tau_q) * (q_hat(k) - q_cmd(k));

    %Euler integration of system dynamic equations
    theta(k + 1) = theta(k) + q(k) * delta_t + 0.5*q_dot_cmd(k)*delta_t^2;
    q(k + 1) = q(k) + q_dot_cmd(k) * delta_t;

    %simulating accelerometer
    alpha1(k) = -g* sin(theta(k));

    %estimator equations
    theta_hat_predicted(k + 1) = theta_hat(k) + q_hat(k) * delta_t;
    theta_measured(k + 1) = -alpha1(k) / g;
    theta_hat(k + 1) = (1-ro)*theta_hat_predicted(k + 1) + ro*theta_measured(k + 1);

end


figure(1)
plot(time, theta)
hold on
plot(time, theta_hat)
title('True Angle Theta vs Estimated Angle Theta Hat over Time')
xlabel('Time (s)')
ylabel('Angle (rad)')
legend('Theta', 'Theta Hat')
```

```
figure(2)
plot(time, q)
title('Angular Velocity over Time')
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')

figure(3)
plot(time(2:end), q_dot_cmd)
title('Commanded Angular Acceleration over Time')
xlabel('Time (s)')
ylabel('Angular Acceleration (rad/s^2)')
```




Angular Velocity over Time
Angular Velocity (rad/s)
Time (s)
0
-0.5
-1
-1.5
-2
-2.5
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5


Commanded Angular Acceleration over Time
Angular Acceleration (rad/s^2)
Time (s)
10
5
0
-5
-10
-15
-20
-25
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5

6/6