ASE 389 Laboratory 2 Report
Simulator Refinements and Quadcopter Control

Justin Hart

February 14, 2023

1 Introduction

In the simulator developed in Lab 1, certain effects were neglected for simplicity. Two significant effects, aerodynamic forces on the quad's body, and motor dynamics (described in [1]), can easily be accounted for and increase the accuracy of the simulator greatly. Once the simulator is refined, a simple controller as developed in [3] can be used to follow input trajectories, increasing the autonomy of the quad.

In this laboratory assignment, a high fidelity quadrotor dynamics simulator was developed. This was done by using first principles equations for drag forces and motor dynamics. Then, a controller was developed and integrated into the simulator. The controller tested with the refined simulator to force the quad to follow a desired trajectory of a full circle at constant vertical position.

2 Theoretical Analysis

2.1 Aerodynamic Forces on the Quad

In Lab 1, aerodynamic forces on the quad's body were neglected in the simulator. To approximate these forces, a drag force term is added to (3.6) in [2]. The drag force term is approximated as a drag coefficient, d_a, multiplied by the negative unit vector of velocity of the center of mass of the quad relative to I, expressed in I (this force is only in the direction opposite the center of mass' velocity).

The elements of the quad's state that affect d_a include the attitude of the quad as this would increase the area of the quad perpendicular to the velocity vector. Also, the velocity of the quad would affect the drag coefficient as it is proportional to the square of the velocity. Environmental conditions affecting d_a would be the density of the air as an increased air density would impart more drag on the quad's surface. Air disturbance velocity would also affect the drag coefficient as this would change the relative velocity vector.

In order to determine the drag coefficient experimentally, the quad could be placed in a wind tunnel and the air speed imparted on the quad as well as the quad's attitude could be adjusted within the bounds of required operating conditions. The force imparted on the quad due to drag could be calculated for this range of conditions allowing for a model of d_a to be developed.

If d_a is described by (1):

$$d_a = \frac{1}{2} C_d A_d \rho f_d(z_I, v_I) \tag{1}$$

And is assumed to be proportional to the cross-sectional area of the quad perpendicular to the direction of travel as well as the squared magnitude of the velocity of the center of mass relative to I, expressed in I, v_I, then an equation for $f_d(z_I, v_I)$ can be found as follows. To get the area of the quad perpendicular to the direction of travel, we multiply the body z axis of the quad expressed in I by the velocity unit vector. This gives us (2):

$$\underline{z_I} \cdot \hat{v}_I = R_{BI}^T \underline{z_B} \cdot \hat{v}_I = \left(R_{BI}^T \underline{z_B}\right)^T \frac{\underline{v_I}}{\|v_I\|} \tag{2}$$

Since f_d is also proportional to the square of the magnitude of v_I, we multiply (2) by this magnitude to get (3):

$$f_d = \left(R_{BI}^T \underline{z_B}\right)^T \frac{\underline{v_I}}{\|v_I\|} \|v_I\|^2 = \left(R_{BI}^T \underline{z_B}\right)^T \underline{v_I} \|v_I\| \tag{3}$$

2.2 Motor Dynamics

Another neglected effect from the Lab 1 model was motor dynamics. In section 3.4 of [2] a model for motor dynamics is derived and a transfer function from voltage to angular rate is presented at (3.25). The step response plot is shown below in Fig. 1.

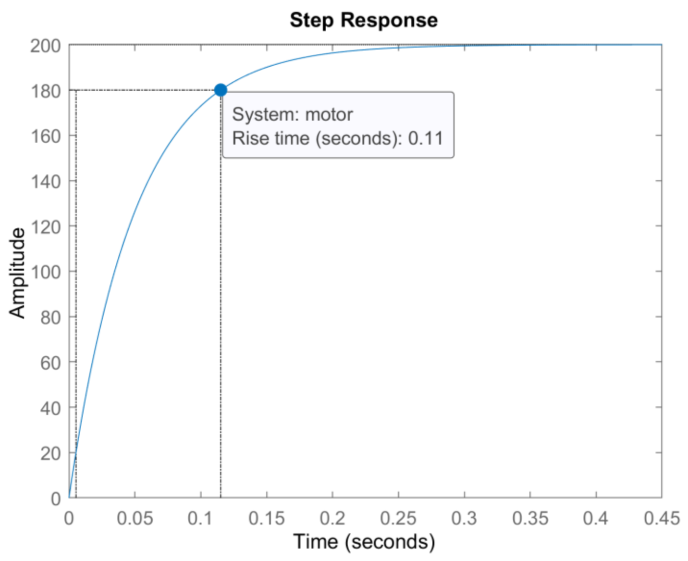


Figure 1: Step Response of quad voltage to angular rate as derived in section 3.4 of [2]

As shown in the plot, it takes 0.11 seconds for the rotors to achieve 90% of their commanded velocity after a step input. According to the transfer function given in (3.25) of [2], the rise time does not depend on the magnitude of the step input. This is because rise time in a first system is approximated as $2.2 * t_m$ which does not change based on the magnitude of the step input. In practice I would expect the rise time for a step input from $e_a = 0\ to\ e_a = e_{as}$ to depend on the value of e_{as}. This is because I would think as e_{as} increases, so do losses from friction as well as resistance within the circuitry of the quad. This would negatively affect the rise time for larger values of e_as.
The transfer function given by (3.25) in [2] can be converted to a first order differential equation in the angular rate ω to give:

$$\dot{\omega} = \frac{1}{t_m}(e_a c_m - \omega) \tag{4}$$

This can be plugged into a Matlab function and solved with ode45 for a unit step input (see TF2odecomp.m in Appendix A). The plot of this response is shown in Fig. 2.

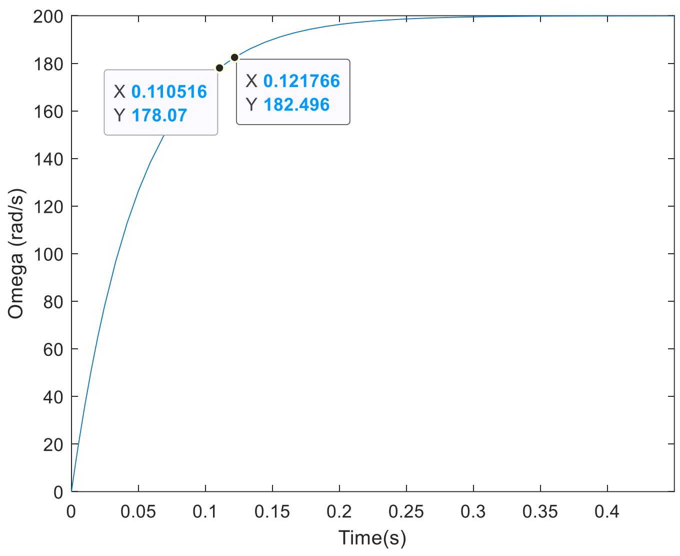


Figure 2: Unit step response of converted transfer function to first order differential equation (4)

The plot in Fig. 2 does look identical to the step response of the transfer funciton in Fig. 1.

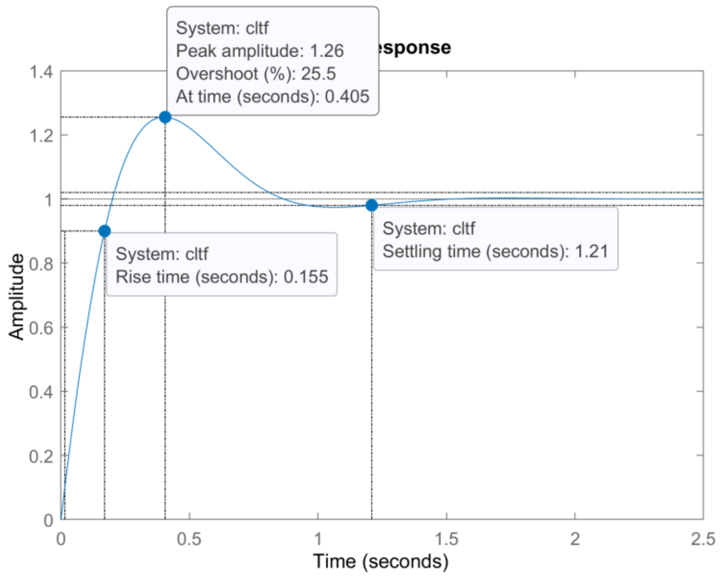
2.3 Basic PD Control

For the quad, if its inertia matrix is diagonal and the attitude errors are small enough, then all six degrees of freedom are mutually decoupled. Each degree of freedom acts as a plant with double integrator dynamics and can be controlled with a basic proportional-derivative (PD) controller. To make the controller responsive and accurate, values for k and k_d can be found to meet three basic performance metrics: 90% rise time (T_r) less than 0.25 seconds, less than 30% overshoot (P_o), and a settling time (T_s) to within 2% of the reference value of less than 2 seconds. Referencing Fig. 1 in Lab 2, the closed loop transfer function (CLTF) of the system can be derived as $\frac{k+k_d s}{s^2+k_d s+k}$ if $k_y = 1$. To find k and k_d, treat the

CLTF as a standard second order system of the form $\frac{\omega_n^2}{s^2+2\varsigma\omega_n s+\omega_n^2}$ which makes $k = \omega_n^2$ and $k_d = 2\varsigma\omega_n$. Then, the standard equations for performance metrics of a step response to a second order system can be used. These are:

$$\varsigma = \left[1 + \left(\frac{\pi}{\ln(P_o)}\right)^2\right]^{-\frac{1}{2}}, T_s = \frac{4}{\varsigma\omega_n}, T_r \approx \frac{1.8}{\omega_n} \tag{5}$$

By plugging in the required performance metrics and slightly adjusting values as necessary (since the system isn't exactly of second order form), values of $k = 26.45$ and $k_d = 5.32$ were found to satisfy the requirements. A plot of the step response and measured performance values is shown in Fig. 3:



```
        RiseTime: 0.1548
    SettlingTime: 1.2089
     SettlingMin: 0.9214
     SettlingMax: 1.2552
       Overshoot: 25.5248
      Undershoot: 0
            Peak: 1.2552
        PeakTime: 0.4053
```

Figure 3: Step response characteristics of a closed loop system with PD controller.

2.4 Attitude Controller

In order to develop the full quadrotor control architecture, an attitude controller is derived in Lab 2. This controller relies on a Euler angle error vector, e_E, which uses elements from the error direction-cosine matrix, R_E, as derived in lectures. To explore the relationship between the Euler angle error vector and error direction-cosine matrix, let a unit eigenvector of rotation $\hat{a} = [a_1, a_2, a_3]^T$. The rotation matrix $R(\hat{a}, \varphi)$ full expression is worked out as follows (where $c\varphi = \cos(\varphi)$ *and* $s\varphi = \sin(\varphi)$):

$$R(\hat{a}, \varphi) = c\varphi \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + (1 - c\varphi) \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} - s\varphi \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{6}$$

$$R(\hat{a},\varphi) = \begin{bmatrix} c\varphi + a_1^2 - a_1^2 c\varphi & a_1a_2 - a_1a_2c\varphi + a_3s\varphi & a_1a_3 - a_1a_3c\varphi - a_2s\varphi \\ a_1a_2 - a_1a_2c\varphi - a_3s\varphi & c\varphi + a_2^2 - a_2^2c\varphi & a_2a_3 - a_2a_3c\varphi + a_1s\varphi \\ a_1a_3 - a_1a_3c\varphi + a_2s\varphi & a_2a_3 - a_2a_3c\varphi - a_1s\varphi & c\varphi + a_3^2 - a_3^2c\varphi \end{bmatrix} \tag{7}$$

Then it can be shown that $a_1 = \frac{R_{23}-R_{32}}{2sin\varphi}, a_2 = \frac{R_{31}-R_{13}}{2sin\varphi}, a_3 = \frac{R_{12}-R_{21}}{2sin\varphi}$ by the process in (7):

$$a_1 = \frac{R_{23} - R_{32}}{2sin\varphi} = \frac{(a_2a_3 - a_2a_3c\varphi + a_1s\varphi) - (a_2a_3 - a_2a_3c\varphi - a_1s\varphi)}{2sin\varphi} = a_1 \tag{8}$$

Which can be repeated in similar fashion for $a_2 and\ a_3$. Given this derivation, it can be concluded that $\frac{e_E}{2sin\varphi}$ is the unit eigenvector of rotation for R_E where φ is the rotation angle. By adding the diagonal elements in (6) the trace of $R(\hat{a},\varphi)$ is found to be $1 + 2cos\varphi$ by setting the terms $(a_1^2 + a_2^2 + a_3^2) = 1$ since this is equal to the magnitude of a unit vector which is 1.

3 Implementation

The quadrotor dynamics HF simulator is designed similarly to the original dynamics simulator in the sense that there are still three levels of code. These are at the lowest level, an ordinary differential equation (ODE) function (quadOdeFunctionHF.m) that is fed into a Matlab ODE solver. The second level is the new HF simulation function (simulateQuadrotorDynamicsHF.m) that adds neglected effects from the previous lab. The top level of code is the topSimulateHF.m script that provides inputs and runs lower level functions in order to output plots of the simulation. All applicable code is listed in Appendix A.

To add the drag force in the HF simulator, the vIdot equation in the original quadOdeFunction is updated to include the calculated drag coefficient from section 2.1 above. The implementation in code is shown below:

```
% vIdot
Ft = 0;
for i = 1:4
    F = kF(i)*omegaVec(i)^2;
    Ft = Ft + F;
end
da = 1/2*Cd*Ad*rho*(RBI'*[0;0;1])'*vI;

vIdot = 1/m*(-da*vI+[0;0;-m*g]+RBI'*[0;0;Ft]+distVec);
```

To implement motor dynamics into the HF simulator, the rotor angular rate is added as a state to the original state matrix. The input of rotor angular rate is removed from the quadOdeFunction and a motor voltage vector is input instead. The differential equation derived in section 2.2 above is added to the original quadOdeFunction using this input motor voltage vector and the constants taum and cm which are the rotor time constants and motor voltage to motor angular rate conversion factors respectively. This differential equation code is shown below:

```
% omegaVecdot
```

```
omegaVecdot = 1/taum(1)*(eaVec*cm(1) - omegaVec);
```

The overall control design architecture is shown in Fig. 2 of Lab 2. First, the trajectory controller is designed according to the relatively straightforward equations given in the Lab 2 guide. The desired trajectory and current state of the quad is fed into the trajectory controller, and a desired z-axis direction and commanded total thrust is output.

Then, the attitude controller is created with diagonal K and K_d controller gains. The desired z-axis direction from the trajectory controller as well as the desired x-axis direction and current state of the quad is input into the controller. A desired direction cosine matrix is calculated and used to find the error direction cosine matrix. This is used to find the Euler angle error vector which is plugged into the attitude control law for the commanded torque which is the output of the attitude controller. The code for the commanded torque is shown below:

```
OUTPUTS
%
% NBk -------- Commanded 3x1 torque expressed in the body frame at
% time tk, in N-m.

    K11 = .35;
    K22 = .35;
    K33 = .35;
    K = diag([K11 K22 K33]);

    Kd11 = .05;
    Kd22 = .05;
    Kd33 = .05;
    Kd = diag([Kd11 Kd22 Kd33]);


    yIstark = cross(zIstark,xIstark)/norm(cross(zIstark,xIstark));
    xIstark_proj = cross(yIstark,zIstark);
    RBIstark = [xIstark_proj,yIstark,zIstark]';
    Re = RBIstark*RBI'; %error direction cosine matrix

    eE = [Re(2,3)-Re(3,2);Re(3,1)-Re(1,3);Re(1,2)-Re(2,1)]; %error
euler angles

    NBk = K*eE - Kd*omegaB + crossProductEquivalent(omegaB)*Jq*omegaB;
```

Next, a conversion function is required to convert the commanded thrust and torque into voltages to be applied to each motor. First, a conversion matrix is derived (from lecture) that converts commanded thrust and torque into individual rotor required thrust. Then, in order to keep the input voltages below the maximum motor voltage, each calculated thrust is iterated upon using a scalar value, alpha, that is reduced until the commanded thrust to each motor is smaller than the maximum allowable thrust (a function of the maximum input motor voltage). Finally, any negative rotor thrust values are set to zero. This voltageConverter function is shown below:

```
% OUTPUTS
%
```

```
% eak -------- Commanded 4x1 voltage vector to be applied at time tk,
% in volts. eak(i) is the voltage for the ith motor.
    omegamax = eamax*cm;
    Fmax = kF.*omegamax.^2;

    kT = kN./kF;
    convmat = [1 1 1 1;rotor_loc(2,1) rotor_loc(2,2) rotor_loc(2,3) 
rotor_loc(2,4);...
        -rotor_loc(1,1) -rotor_loc(1,2) -rotor_loc(1,3) -
rotor_loc(1,4);...
        -kT(1) kT(1) -kT(1) kT(1)];

    beta = 0.9;
    alpha = 1;
    minmat = [min([Fk,4*beta*Fmax(1)]);alpha*NBk];
    Fmat = inv(convmat)*minmat;

    for q = 1:4
        while Fmat(q) > Fmax(q)
            alpha = alpha-0.01;
            minmat = [min([Fk,4*beta*Fmax(q)]);alpha*NBk];
            Fmat = inv(convmat)*minmat;
        end
    end

    for h = 1:4
        if Fmat(h) < 0
            Fmat(h) = 0;
        end
    end

    eak = sqrt(Fmat/kF(1))*1/cm(1);
```

Lastly, the controller and converter functions are embedded into a simulator function that draws in desired trajectories and generates input voltages within the simulator. Within the ODE solver loop, input structures are created for desired values and current state at that time step. The trajectory controller function outputs the commanded total thrust and desired body z-axis direction at that step and the attitude controller takes that and outputs the commanded torque. All of this is fed into the voltage converter to get the commanded input motor voltages which are used as the motor voltage vector input to the quadOdeFunctionHF. Then the ODE solver solves for the current state at that time step and iterates to the next step. The simulateQuadrotorControl function is shown below:

```
M = (N-1)*oversampFact + 1;
    tVecOut = [];
    XMat = [];

    Xk = X0;

    for k = 1:N-1
```

```
        tspan = [tVecIn(k):dtOut:tVecIn(k+1)]';

        %R structure input to functions
        Rk.rIstark = rIstar(k,:)';
        Rk.vIstark = vIstar(k,:)';
        Rk.aIstark = aIstar(k,:)';
        Rk.xIstark = xIstar(k,:)';

        %S structure input to functions
        Sk.statek.rI = Xk(1:3);
        Sk.statek.RBI = [Xk(7:9),Xk(10:12),Xk(13:15)];
        Sk.statek.vI = Xk(4:6);
        Sk.statek.omegaB = Xk(16:18);

        %trajectory controller
        [Fk,zIstark] = trajectoryController(Rk,Sk,P);
        Rk.zIstark = zIstark;

        %attitude controller
        NBk = attitudeController(Rk,Sk,P);

        %voltage conversion
        eak = voltageConverter(Fk,NBk,P);

        %ODE solver
        [tVeck,XMatk] =
ode45(@(t,X)quadOdeFunctionHF(t,X,eak,distMat(k,:)',P),tspan,Xk);

        if(length(tspan) == 2)
            % Deal with S.oversampFact = 1 case
            tVecOut = [tVecOut; tVeck(1)];
            XMat = [XMat; XMatk(1,:)];
        else
            tVecOut = [tVecOut; tVeck(1:end-1)];
            XMat = [XMat; XMatk(1:end-1,:)];
        end

        Xk = XMatk(end,:)';
        RBIk = [Xk(7:9),Xk(10:12),Xk(13:15)];
        %Ensure that RBI remains orthogonal
        if(mod(k,10) == 0)
            RBIk = [Xk(7:9),Xk(10:12),Xk(13:15)];
            [UR,SR,VR]=svd(RBIk);
            RBIk = UR*VR'; Xk(7:15) = RBIk(:);
        end
%         etest = dcm2euler(RBIk);
    end

    XMat = [XMat; XMatk(end,:)];
    tVecOut = [tVecOut; tVeck(end,:)];

    Q.tVec = tVecOut;
```

```
Q.state.rMat = XMat(:,1:3);

eMat = [];
for k = 1:M
    RBIk = [XMat(k,7:9)',XMat(k,10:12)',XMat(k,13:15)'];
    ek = dcm2euler(RBIk);
    eMat = [eMat;ek'];
end

Q.state.eMat = eMat;
Q.state.vMat = XMat(:,4:6);
Q.state.omegaBMat = XMat(:,16:18);
Q.state.omegaVec = XMat(:,19:22);
```

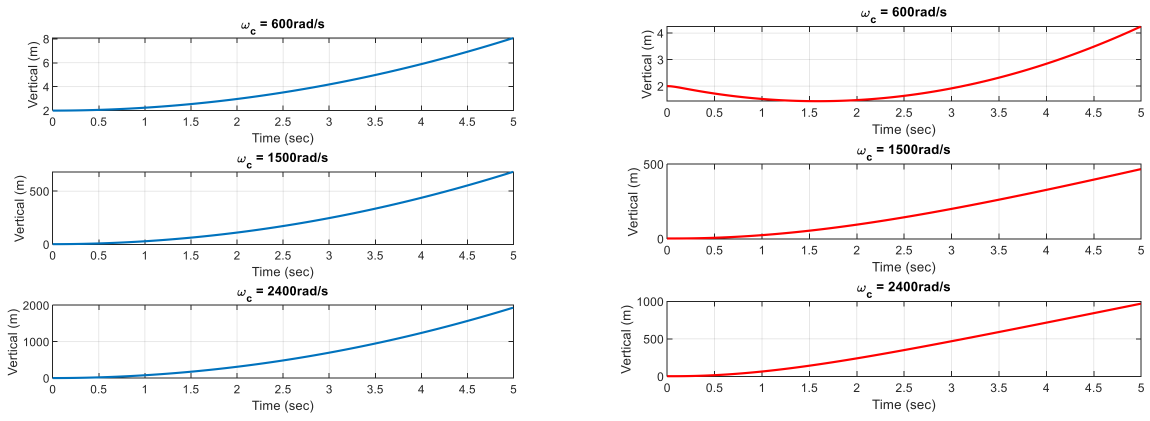
The last step of the controller design is to tune the controller. This is done with an iterative guess and check method by first setting k and k_d equal to zero and then slowly increasing k until oscillations appear. Then k_d is increased until the oscillations disappear. These steps are repeated until acceptable performance is achieved. The attitude controller gains are tuned similarly, with each element along the diagonal of the gains tuned independently. The final computed gains for the trajectory and attitude controllers respectively are:

$$k = 5, k_d = 2.7, K = \begin{bmatrix} 1.1 & & \\ & 1.3 & \\ & & 1.4 \end{bmatrix}, K_d = \begin{bmatrix} .075 & & \\ & .11 & \\ & & .13 \end{bmatrix}$$

4 Results and Analysis

4.1 Simulator Refinements

In order to visualize the difference between the Lab 1 simulator and the high fidelity (HF) simulator developed in Lab 2, three different steady state rotor rates were input into each. The rotor rates chosen were 600, 1500, and 2400 rad/s as low, moderate, and extreme values. These values were input into the HF simulator by applying a constant voltage to each motor. The quad started from an initial zero-velocity hover state with z_I parallel to Z_I so that the quad flew directly upward. The plots of vertical distance from hover as a function of time are shown in Fig. 4:



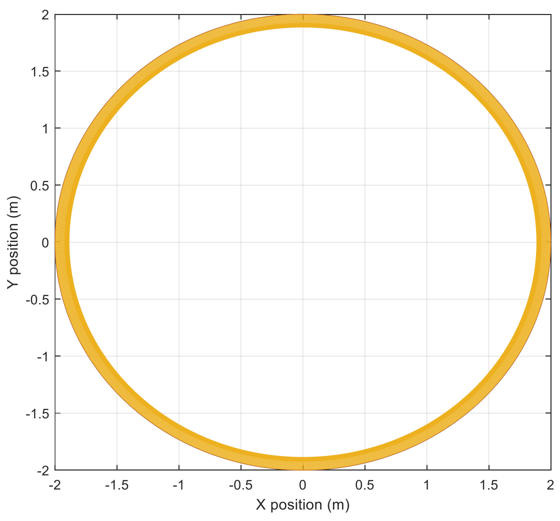
(a) (b)

Figure 4: Vertical distance of CM from hover for (a) Lab 1 simulator and (b) HF simulator, for steady state rotor rates.

The Lab 1 model without refinements clearly reached greater heights than the high fidelity simulation over the time period tested. The first reason is due to the addition of the motor model in the high fidelity simulation. This adds a spin-up time for the motors (time to reach desired speed) vs the original model which assumes instantaneous motor speed. This results in a slight dip at the start of the HF simulation until the motor reaches its desired speed, coupled with a delay in vertical motion of the quad. The second reason is because with the addition of the drag force in the model, there is additional resistance to motion (besides gravity and disturbance forces). This resistance to motion is proportional to the square of the velocity so as the quad moves faster, the drag force resisting motion increases exponentially.

4.2 Quadrotor Controller Experimentation

To test the controller design, a desired CM trajectory was developed that articulated a circular path with a 4-meter diameter, and orbital period of 10 seconds. Then, this was used to find desired CM velocity and acceleration over the course of this trajectory. This was done by taking the difference in trajectory and velocity respectively of each next iteration value and the current iteration value and dividing by the time delta. A desired direction of the body x axis was also created to keep the quad always pointing towards the center of the circle. The desired trajectory, velocity, acceleration, and body x axis direction is shown below in Figs. 5, 6, and 7.



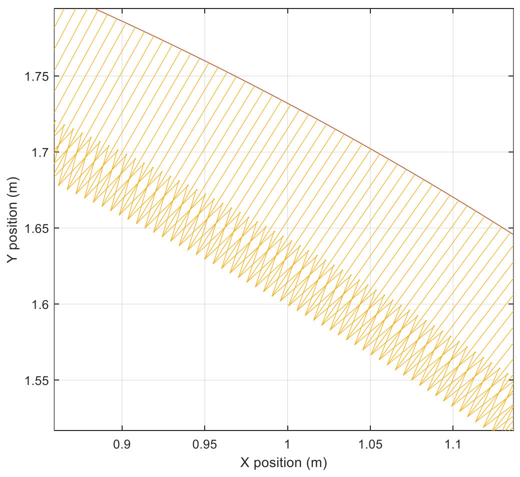


Figure 5: Desired trajectory in the $X_I - Y_I$ plane and a zoomed in section of the trajectory showing the desired body x axis direction is pointed towards the trajectory center.

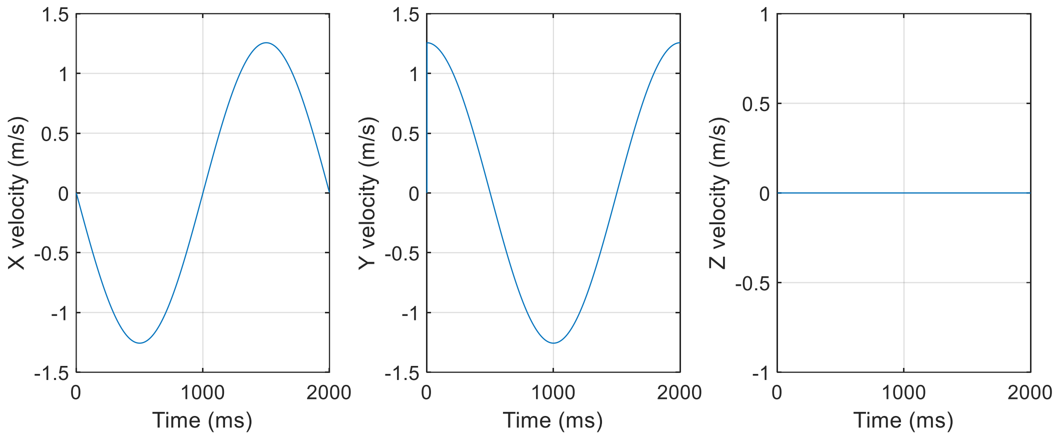


Figure 6: Desired X,Y, and Z velocity of the quad’s CM for the circular trajectory.

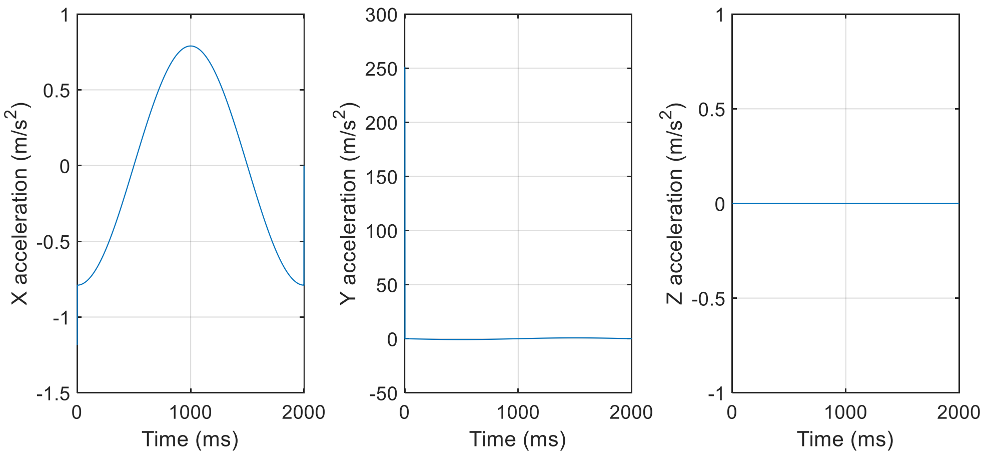


Figure 7: Desired X,Y, and Z acceleration of the quad’s CM for the circular trajectory.

This desired trajectory, velocity, acceleration, and body x axis direction was fed into the integrated HF simulator with controller. The quad was started at rest – with initial velocity, body angular rate, and rotor angular rates all equal to zero. The results of the experiment are shown below in Figs. 8 and 9.

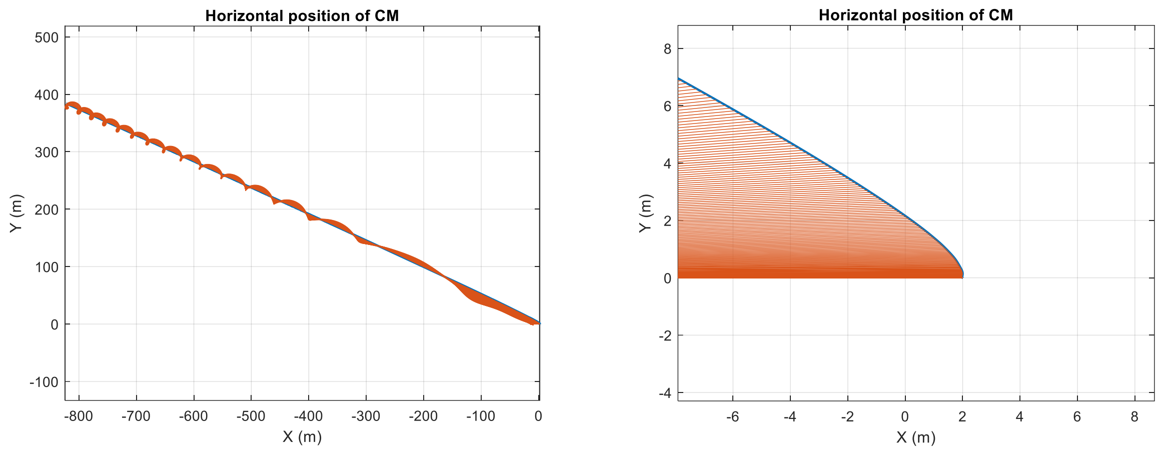


Figure 8: Quad CM position in the $X_I - Y_I$ plane with vectors pointing in the body x axis direction projected onto the $X_I - Y_I$ plane and a magnified plot at the origin.

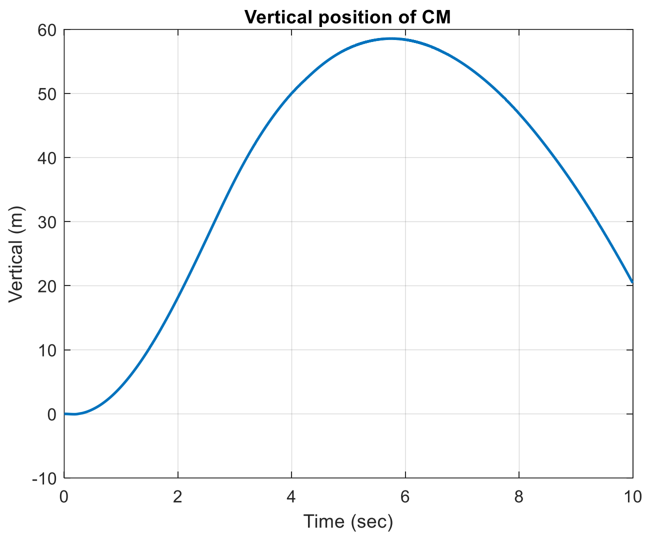


Figure 9: Quad's CM position in the Z_I direction.

Ultimately, after attempting to tune both by an iterative guess and check method as well as by tuning each decoupled second order ODE for each degree of freedom, a circular trajectory could not be followed by the integrated controller/simulator. The simulation starts out relatively circular as shown in Fig. 8, but quickly becomes unstable. It could not be determined whether this was due to a poor desired trajectory/velocity/acceleration input, a fault in the code, or a lack of adequate controller tuning.

5 Conclusion

A refined simulator for the quadrotor was created, accounted for previously neglected effects (aerodynamic forces and motor dynamics). A basic controller combined trajectory and attitude control was derived using linear, time-invariant system control techniques. The controller was integrated into the simulator and tested using a circular input trajectory with constant altitude.

References

[1] D. Mellinger, N. Michael, and V. Kumar, “Trajectory generation and control for precise aggressive maneuvers with quadrotors,” *The International Journal of Robotics Research*, vol. 31, no. 5, pp. 664–674, 2012.

[2] Humphreys, T 2023, *Aerial Robotics Course Notes for ASE 479W/ASE 389*, lecture notes, Aerial Robotics ASE 389, University of Texas at Austin, delivered 10 January 2023.

[3] T. Lee, M. Leoky, and N. H. McClamroch, “Geometric tracking control of a quadrotor UAV on SE(3),” in *Decision and Control (CDC), 2010 49th IEEE Conference on*, pp. 5420–5425, IEEE, 2010.

Appendix A: MATLAB Code

A.1 attitudeController.m

```
function [NBk] = attitudeController(R,S,P)
% attitudeController : Controls quadcopter toward a reference attitude
%
%
% INPUTS
%
% R ---------- Structure with the following elements:
%
%       zIstark = 3x1 desired body z-axis direction at time tk,
expressed as a
%                 unit vector in the I frame.
    zIstark = R.zIstark;
%       xIstark = 3x1 desired body x-axis direction, expressed as a
%                 unit vector in the I frame.
    xIstark = R.xIstark;
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure
with the
%                 following elements:
%
%                  rI = 3x1 position in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude
        RBI = S.statek.RBI;
%                  vI = 3x1 velocity with respect to the I frame and
%                       expressed in the I frame, in meters per
second.
%
%              omegaB = 3x1 angular rate vector expressed in the body
frame,
%                       in radians per second.
```

```
        omegaB = S.statek.omegaB;
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
    Jq = P.quadParams.Jq;
%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m
%
%
% OUTPUTS
%
% NBk -------- Commanded 3x1 torque expressed in the body frame at
time tk, in
%              N-m.

    K11 = 1.1;
    K22 = 1.3;
    K33 = 1.4;
    K = diag([K11 K22 K33]);

    Kd11 = .075;
    Kd22 = .11;
    Kd33 = .13;
    Kd = diag([Kd11 Kd22 Kd33]);


    yIstark = cross(zIstark,xIstark)/norm(cross(zIstark,xIstark));
    xIstark_proj = cross(yIstark,zIstark);
    RBIstark = [xIstark_proj,yIstark,zIstark]';
    Re = RBIstark*RBI'; %error direction cosine matrix

    eE = [Re(2,3)-Re(3,2);Re(3,1)-Re(1,3);Re(1,2)-Re(2,1)]; %error
euler angles

    NBk = K*eE - Kd*omegaB + crossProductEquivalent(omegaB)*Jq*omegaB;
```

A.2 basicPDcontrol.m

```
%basic PD control
clear
clc

Mp = 0.15;
tr = 0.35;

omega_n = 1.8/tr;
zeta = (1+(pi()/log(Mp))^2)^(-.5);
```

```
ky = 1/0.78;
k = omega_n^2;
kd = 2*zeta*omega_n;

cs = tf([kd k],1);
ps = tf(ky,[1 0 0]);

oltf = series(cs,ps);
cltf = feedback(oltf,1)

step(cltf)
stepinfo(cltf)
```

A.3 ControlTuning.m

```
%Trajectory Controller tuning
clear

quadParamsScript;

Mp = 0.10;
tr = 0.40;
ts = 2;

omega_n = 1.8/tr;
zeta = (1+(pi()/log(Mp))^2)^(-.5);
ts_calc = 4/(zeta*omega_n);

ky = 1/quadParams.m;
% k = (omega_n^2)/ky
% kd = (2*zeta*omega_n)/ky

k = 5;
kd = 2.7;

cs = tf([kd k],1);
ps = tf(ky,[1 0 0]);

oltf = series(cs,ps);
cltf = feedback(oltf,1)

step(cltf)
stepinfo(cltf)

%%
%Attitude Controller tuning Jx
clear

quadParamsScript;

Mp = 0.15;
```

```
tr = 0.35;
ts = 2;

omega_n = 1.8/tr;
zeta = (1+(pi()/log(Mp))^2)^(-.5);
ts_calc = 4/(zeta*omega_n);

ky = 1/quadParams.Jq(1,1);
% k = (omega_n^2)/ky
% kd = (2*zeta*omega_n)/ky

k = 1.1;
kd = 0.075;

cs = tf([kd k],1);
ps = tf(ky,[1 0 0]);

oltf = series(cs,ps);
cltf = feedback(oltf,1)

step(cltf)
stepinfo(cltf)

%%
%Attitude Controller tuning Jy
clear

quadParamsScript;

Mp = 0.15;
tr = 0.35;
ts = 2;

omega_n = 1.8/tr;
zeta = (1+(pi()/log(Mp))^2)^(-.5);
ts_calc = 4/(zeta*omega_n);

ky = 1/quadParams.Jq(2,2);
% k = (omega_n^2)/ky
% kd = (2*zeta*omega_n)/ky

k = 1.3;
kd = .11;

cs = tf([kd k],1);
ps = tf(ky,[1 0 0]);

oltf = series(cs,ps);
cltf = feedback(oltf,1)

step(cltf)
stepinfo(cltf)
```

```
%%
%Attitude Controller tuning Jz
clear

quadParamsScript;

Mp = 0.15;
tr = 0.35;
ts = 2;

omega_n = 1.8/tr;
zeta = (1+(pi()/log(Mp))^2)^(-.5);
ts_calc = 4/(zeta*omega_n);

ky = 1/quadParams.Jq(3,3);
% k = (omega_n^2)/ky
% kd = (2*zeta*omega_n)/ky

k = 1.4;
kd = 0.13;

cs = tf([kd k],1);
ps = tf(ky,[1 0 0]);

oltf = series(cs,ps);
cltf = feedback(oltf,1)

step(cltf)
stepinfo(cltf)
```

A.4 crossProductEquivalent.m

```
function [uCross] = crossProductEquivalent(u)
% crossProductEquivalent : Outputs the cross-product-equivalent matrix
uCross
% such that for arbitrary 3-by-1 vectors u and v,
% cross(u,v) = uCross*v.
%
% INPUTS
%
% u ---------- 3-by-1 vector

u1 = u(1);
u2 = u(2);
u3 = u(3);

%
% OUTPUTS
%
% uCross ----- 3-by-3 skew-symmetric cross-product equivalent matrix
```

```
uCross = [0 -u3 u2;u3 0 -u1;-u2 u1 0];
```

A.5 dcm2euler.m

```
function [e] = dcm2euler(R_BW)
% dcm2euler : Converts a direction cosine matrix R_BW to Euler angles
phi =
%              e(1), theta = e(2), and psi = e(3) (in radians) for a 3-
1-2
%              rotation. If the conversion to Euler angles is singular
(not
%              unique), then this function issues an error instead of
returning
%              e.
%
% Let the world (W) and body (B) reference frames be initially
aligned.  In a
% 3-1-2 order, rotate B away from W by angles psi (yaw, about the body
Z
% axis), phi (roll, about the body X axis), and theta (pitch, about
the body Y
% axis).  R_BW can then be used to cast a vector expressed in W
coordinates as
% a vector in B coordinates: vB = R_BW * vW
%
% INPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
%
% OUTPUTS
%
% e ---------- 3-by-1 vector containing the Euler angles in radians:
phi =
%               e(1), theta = e(2), and psi = e(3).  By convention,
these
%               should be constrained to the following ranges: -pi/2 <=
phi <=
%               pi/2, -pi <= theta < pi, -pi <= psi < pi.
%
%+----------------------------------------------------------------------
----------+
% References:
%
%
% Author:
%+======================================================================
==========+

epsilon = 1e-15;
```

```
phi = asin(R_BW(2,3));
% Note that if both arguments of the atan2 below are zero, as occurs
when
% cos(phi) = 0, then the result is undefined. Thus, for phi = pi/2 +
n*pi, for
% n an integer, the 3-1-2 Euler representation is singular.  The
intuition here
% is that when the roll angle is pi/2, the first and third rotations
(about the
% z and y axes, respectively), have exactly the same effect, and so
can't be
% distinguished, leading to a non-unique representation for psi and
theta; only
% psi + theta is constrained, but not psi and theta individually.
This
% function detects this singularity and issues an error.
if(abs(cos(phi)) < epsilon)
  error('The 312 attitude representation is singular: cos(phi) is near
0');
end
theta = atan2(-R_BW(1,3),R_BW(3,3));
if(theta == pi)
  theta = -pi;
end
psi = atan2(-R_BW(2,1),R_BW(2,2));
if(psi == pi)
  psi = -pi;
end
e = [phi; theta; psi];
```

A.6 euler2dcm.m

```
function [R_BW] = euler2dcm(e)
% euler2dcm : Converts Euler angles phi = e(1), theta = e(2), and psi
= e(3)
%             (in radians) into a direction cosine matrix for a 3-1-2
rotation.
%
% Let the world (W) and body (B) reference frames be initially
aligned.  In a
% 3-1-2 order, rotate B away from W by angles psi (yaw, about the body
Z
% axis), phi (roll, about the body X axis), and theta (pitch, about
the body Y
% axis).  R_BW can then be used to cast a vector expressed in W
coordinates as
% a vector in B coordinates: vB = R_BW * vW
%
% INPUTS
%
```

```
% e ---------- 3-by-1 vector containing the Euler angles in radians:
phi =
%              e(1), theta = e(2), and psi = e(3)
%
%
% OUTPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
%+----------------------------------------------------------------------
----------+
% References:
%
%
% Author:
%+======================================================================
==========+

cPhi = cos(e(1)); sPhi = sin(e(1));
cThe = cos(e(2)); sThe = sin(e(2));
cPsi = cos(e(3)); sPsi = sin(e(3));

R_BW = [cPsi*cThe - sPhi*sPsi*sThe, cThe*sPsi + cPsi*sPhi*sThe, -
cPhi*sThe;
        -cPhi*sPsi,                                  cPhi*cPsi,
sPhi;
        cPsi*sThe + cThe*sPhi*sPsi, sPsi*sThe - cPsi*cThe*sPhi,
cPhi*cThe];
```

A.7 motorE2omegaTF.m

```
%Motor TF from voltage to angular rate

quadParamsScript;
constantsScript;
S.quadParams = quadParams;
S.constants = constants;
S.E = 1; %sets input voltage in order to compare step responses

motor = tf([S.E*S.quadParams.cm(1)],[S.quadParams.taum(1) 1]);

step(motor)
```

A.8 quadOdeFunctionHF.m

```
function [Xdot] = quadOdeFunctionHF(t,X,eaVec,distVec,P)
% quadOdeFunctionHF : Ordinary differential equation function that
models
%                      quadrotor dynamics -- high-fidelity version.
For use
%                      with one of Matlab's ODE solvers (e.g., ode45).
%
```

```
%
% INPUTS
%
% t ---------- Scalar time input, as required by Matlab's ODE function
%              format.
%
% X ---------- Nx-by-1 quad state, arranged as
%
%              X =
[rI',vI',RBI(1,1),RBI(2,1),...,RBI(2,3),RBI(3,3),...
%                   omegaB',omegaVec']'
%
%              rI = 3x1 position vector in I in meters
    rI = [X(1:3)];
%              vI = 3x1 velocity vector wrt I and in I, in meters/sec
    vI = [X(4:6)];
%             RBI = 3x3 attitude matrix from I to B frame
    RBI = [X(7:9),X(10:12),X(13:15)];
%          omegaB = 3x1 angular rate vector of body wrt I, expressed
in B
%                   in rad/sec
    omegaB = [X(16:18)];
%        omegaVec = 4x1 vector of rotor angular rates, in rad/sec.
%                   omegaVec(i) is the angular rate of the ith rotor.
    omegaVec = [X(19:22)];
%    eaVec --- 4x1 vector of voltages applied to motors, in volts.
eaVec(i)
%              is the constant voltage setpoint for the ith rotor.

%  distVec --- 3x1 vector of constant disturbance forces acting on the
quad's
%              center of mass, expressed in Newtons in I.
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
    kF = P.quadParams.kF;
    kN = P.quadParams.kN;
    omegaRdir = P.quadParams.omegaRdir;
    rotor_loc = P.quadParams.rotor_loc;
    m = P.quadParams.m;
    Jq = P.quadParams.Jq;
    Ad = P.quadParams.Ad;
    Cd = P.quadParams.Cd;
    taum = P.quadParams.taum;
    cm = P.quadParams.cm;
    eamax = P.quadParams.eamax;
%    r_rotor = P.quadParams.r_rotor;
%
%     constants = Structure containing constants used in simulation
and
```

```
%                  control, as defined in constantsScript.m
    rho = P.constants.rho;
    g = P.constants.g;
%
% OUTPUTS
%
% Xdot ------- Nx-by-1 time derivative of the input vector X

% rIdot
rIdot = vI;

% vIdot
Ft = 0;
for i = 1:4
    F = kF(i)*omegaVec(i)^2;
    Ft = Ft + F;
end
da = 1/2*Cd*Ad*rho*(RBI'*[0;0;1])'*vI;

vIdot = 1/m*(-da*vI+[0;0;-m*g]+RBI'*[0;0;Ft]+distVec);

% RBIdot
omegaBcross = crossProductEquivalent(omegaB);
RBIdot = -omegaBcross*RBI;

% omegaBdot
NB = [0;0;0];
for i = 1:4
    ri = rotor_loc(:,i);
    Fi = [0;0;kF(i)*omegaVec(i)^2];
    Ni = [0;0;omegaRdir(i)*kN(i)*omegaVec(i)^2];
    Nnext = Ni+cross(ri,Fi);
    NB = NB + Nnext;
end

omegaBdot = Jq\(NB-omegaBcross*Jq*omegaB);

% omegaVecdot
test = eaVec(eaVec > eamax);
if isempty(test)

else
    error(['motor voltage must not be greater than ' num2str(eamax) '
volts.'])
end

omegaVecdot = 1/taum(1)*(eaVec*cm(1) - omegaVec);

% Xdot
Xdot =
[rIdot',vIdot',RBIdot(1,1),RBIdot(2,1),RBIdot(3,1),RBIdot(1,2),RBIdot(
2,2),...
RBIdot(3,2),RBIdot(1,3),RBIdot(2,3),RBIdot(3,3),omegaBdot',omegaVecdot
']';
```

A.9 simulateQuadrotorControl.m

```
function [Q] = simulateQuadrotorControl(R,S,P)
% simulateQuadrotorControl : Simulates closed-loop control of a
quadrotor
%                            aircraft.
%
%
% INPUTS
%
% R ---------- Structure with the following elements:
%
%          tVec = Nx1 vector of uniformly-sampled time offsets from
the
%                 initial time, in seconds, with tVec(1) = 0.
    tVecIn = R.tVec;
    N = length(tVecIn);
%        rIstar = Nx3 matrix of desired CM positions in the I frame,
in
%                 meters.  rIstar(k,:)' is the 3x1 position at time tk
=
%                 tVec(k).
    rIstar = R.rIstar;
%        vIstar = Nx3 matrix of desired CM velocities with respect to
the I
%                 frame and expressed in the I frame, in meters/sec.
%                 vIstar(k,:)' is the 3x1 velocity at time tk =
tVec(k).
    vIstar = R.vIstar;
%        aIstar = Nx3 matrix of desired CM accelerations with respect
to the I
%                 frame and expressed in the I frame, in meters/sec^2.
%                 aIstar(k,:)' is the 3x1 acceleration at time tk =
%                 tVec(k).
    aIstar = R.aIstar;
%        xIstar = Nx3 matrix of desired body x-axis direction,
expressed as a
%                 unit vector in the I frame. xIstar(k,:)' is the 3x1
%                 direction at time tk = tVec(k).
    xIstar = R.xIstar;
% S ---------- Structure with the following elements:
%
%  oversampFact = Oversampling factor. Let dtIn = R.tVec(2) -
R.tVec(1). Then
%                 the output sample interval will be dtOut =
%                 dtIn/oversampFact. Must satisfy oversampFact >= 1.
```

```
    oversampFact = S.oversampFact;
    if oversampFact >= 1
        dtIn = tVecIn(2) - tVecIn(1);
        dtOut = dtIn/oversampFact;
    else
        error('oversampFact must be >= 1');
    end

%        state0 = State of the quad at R.tVec(1) = 0, expressed as a
structure
%                 with the following elements:
%
%                   r = 3x1 position in the world frame, in meters
        r0 = S.state0.r;
%                   e = 3x1 vector of Euler angles, in radians,
indicating the
%                       attitude
        e0 = S.state0.e;
        RBI0 = euler2dcm(e0);
%                   v = 3x1 velocity with respect to the world frame
and
%                       expressed in the world frame, in meters per
second.
        v0 = S.state0.v;
%              omegaB = 3x1 angular rate vector expressed in the body
frame,
%                       in radians per second.
        omegaB0 = S.state0.omegaB;

%              omegaVec - added this to template. 4x1 initial value of
%              rotor velocity in rad/s
            omegaVec0 = [0;0;0;0];

        X0 =
[r0',v0',RBI0(1,1),RBI0(2,1),RBI0(3,1),RBI0(1,2),RBI0(2,2),...
    RBI0(3,2),RBI0(1,3),RBI0(2,3),RBI0(3,3),omegaB0',omegaVec0']';

%       distMat = (N-1)x3 matrix of disturbance forces acting on the
quad's
%                 center of mass, expressed in Newtons in the world
frame.
%                 distMat(k,:)' is the constant (zero-order-hold) 3x1
%                 disturbance vector acting on the quad from R.tVec(k)
to
%                 R.tVec(k+1).
    distMat = S.distMat;
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m

%  sensorParams = Structure containing sensor parameters, as defined
in
%                 sensorParamsScript.m
%
% OUTPUTS
%
% Q ---------- Structure with the following elements:
%
%          tVec = Mx1 vector of output sample time points, in seconds,
where
%                 Q.tVec(1) = R.tVec(1), Q.tVec(M) = R.tVec(N), and M
=
%                 (N-1)*oversampFact + 1.
%
%         state = State of the quad at times in tVec, expressed as a
%                 structure with the following elements:
%
%                rMat = Mx3 matrix composed such that rMat(k,:)' is
the 3x1
%                       position at tVec(k) in the I frame, in meters.
%
%                eMat = Mx3 matrix composed such that eMat(k,:)' is
the 3x1
%                       vector of Euler angles at tVec(k), in radians,
%                       indicating the attitude.
%
%                vMat = Mx3 matrix composed such that vMat(k,:)' is
the 3x1
%                       velocity at tVec(k) with respect to the I
frame
%                       and expressed in the I frame, in meters per
%                       second.
%
%           omegaBMat = Mx3 matrix composed such that omegaBMat(k,:)'
is the
%                       3x1 angular rate vector expressed in the body
frame in
%                       radians, that applies at tVec(k).

    M = (N-1)*oversampFact + 1;
    tVecOut = [];
    XMat = [];

    Xk = X0;

    for k = 1:N-1

        tspan = [tVecIn(k):dtOut:tVecIn(k+1)]';
        %R structure input to functions
        Rk.rIstark = rIstar(k,:)';
        Rk.vIstark = vIstar(k,:)';
        Rk.aIstark = aIstar(k,:)';
        Rk.xIstark = xIstar(k,:)';

        %S structure input to functions
        Sk.statek.rI = Xk(1:3);
        Sk.statek.RBI = [Xk(7:9),Xk(10:12),Xk(13:15)];
        Sk.statek.vI = Xk(4:6);
        Sk.statek.omegaB = Xk(16:18);

        %trajectory controller
        [Fk,zIstark] = trajectoryController(Rk,Sk,P);
        Rk.zIstark = zIstark;

        %attitude controller
        NBk = attitudeController(Rk,Sk,P);

        %voltage conversion
        eak = voltageConverter(Fk,NBk,P);

        %ODE solver
        [tVeck,XMatk] =
ode45(@(t,X)quadOdeFunctionHF(t,X,eak,distMat(k,:)',P),tspan,Xk);

        if(length(tspan) == 2)
            % Deal with S.oversampFact = 1 case
            tVecOut = [tVecOut; tVeck(1)];
            XMat = [XMat; XMatk(1,:)];
        else
            tVecOut = [tVecOut; tVeck(1:end-1)];
            XMat = [XMat; XMatk(1:end-1,:)];
        end

        Xk = XMatk(end,:)';
        RBIk = [Xk(7:9),Xk(10:12),Xk(13:15)];
        %Ensure that RBI remains orthogonal
        if(mod(k,10) == 0)
            RBIk = [Xk(7:9),Xk(10:12),Xk(13:15)];
            [UR,SR,VR]=svd(RBIk);
            RBIk = UR*VR'; Xk(7:15) = RBIk(:);
        end
%         etest = dcm2euler(RBIk);
    end

    XMat = [XMat; XMatk(end,:)];
    tVecOut = [tVecOut; tVeck(end,:)];

    Q.tVec = tVecOut;
    Q.state.rMat = XMat(:,1:3);
    eMat = [];
    for k = 1:M
        RBIk = [XMat(k,7:9)',XMat(k,10:12)',XMat(k,13:15)'];
        ek = dcm2euler(RBIk);
        eMat = [eMat;ek'];
    end

    Q.state.eMat = eMat;
    Q.state.vMat = XMat(:,4:6);
    Q.state.omegaBMat = XMat(:,16:18);
    Q.state.omegaVec = XMat(:,19:22);
```

A.10 simulateQuadrotorDynamicsHF.m

```
function [P] = simulateQuadrotorDynamicsHF(S)
% simulateQuadrotorDynamicsHF : Simulates the dynamics of a quadrotor
%                               aircraft (high-fidelity version).
%
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%           tVec = Nx1 vector of uniformly-sampled time offsets from
the
%                  initial time, in seconds, with tVec(1) = 0.
    tVecIn = S.tVec;
    N = length(tVecIn);
%
%  oversampFact = Oversampling factor. Let dtIn = tVec(2) - tVec(1).
Then the
%                  output sample interval will be dtOut =
%                  dtIn/oversampFact. Must satisfy oversampFact >= 1.
    oversampFact = S.oversampFact;
    if oversampFact >= 1
        dtIn = tVecIn(2) - tVecIn(1);
        dtOut = dtIn/oversampFact;
    else
        error('oversampFact must be >= 1');
    end
%
%          eaMat = (N-1)x4 matrix of motor voltage inputs.  eaMat(k,j)
is the
%                  constant (zero-order-hold) voltage for the jth motor
over
%                  the interval from tVec(k) to tVec(k+1).
    eaMat = S.eaMat;

%         state0 = State of the quad at tVec(1) = 0, expressed as a
structure
```

```
%                  with the following elements:
%
%                    r = 3x1 position in the world frame, in meters
            r0 = S.state0.r;

%                    e = 3x1 vector of Euler angles, in radians,
indicating the
%                        attitude
            e0 = S.state0.e;
            RBI0 = euler2dcm(e0);

%                    v = 3x1 velocity with respect to the world frame
and
%                        expressed in the world frame, in meters per
second.
            v0 = S.state0.v;

%               omegaB = 3x1 angular rate vector expressed in the body
frame,
%                        in radians per second.
            omegaB0 = S.state0.omegaB;
%
%               omegaVec - added this to template. 4x1 initial value of
%               rotor velocity in rad/s
            omegaVec0 = [0;0;0;0];

         X0 =
[r0',v0',RBI0(1,1),RBI0(2,1),RBI0(3,1),RBI0(1,2),RBI0(2,2),...
    RBI0(3,2),RBI0(1,3),RBI0(2,3),RBI0(3,3),omegaB0',omegaVec0']';

%       distMat = (N-1)x3 matrix of disturbance forces acting on the
quad's
%                 center of mass, expressed in Newtons in the world
frame.
                  distMat(k,:)' is the constant (zero-order-hold) 3x1
%                 disturbance vector acting on the quad from tVec(k)
to
%                 tVec(k+1).
            distMat = S.distMat;

%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m
%
%
% OUTPUTS
%
% P ---------- Structure with the following elements:
%
```

```
%           tVec = Mx1 vector of output sample time points, in seconds,
where
%                  P.tVec(1) = S.tVec(1), P.tVec(M) = S.tVec(N), and M
=
%                  (N-1)*oversampFact + 1.
%
%
%          state = State of the quad at times in tVec, expressed as a
structure
%                  with the following elements:
%
%                 rMat = Mx3 matrix composed such that rMat(k,:)' is
the 3x1
%                        position at tVec(k) in the world frame, in
meters.
%
%                 eMat = Mx3 matrix composed such that eMat(k,:)' is
the 3x1
%                        vector of Euler angles at tVec(k), in radians,
%                        indicating the attitude.
%
%                 vMat = Mx3 matrix composed such that vMat(k,:)' is
the 3x1
%                        velocity at tVec(k) with respect to the world
frame
%                        and expressed in the world frame, in meters
per
%                        second.
%
%            omegaBMat = Mx3 matrix composed such that omegaBMat(k,:)'
is the
%                        3x1 angular rate vector expressed in the body
frame in
%                        radians, that applies at tVec(k).

    M = (N-1)*oversampFact + 1;
    tVecOut = [];
    XMat = [];

    Xk = X0;

    for k = 1:N-1

        tspan = [tVecIn(k):dtOut:tVecIn(k+1)]';

        [tVeck,XMatk] =
ode45(@(t,X)quadOdeFunctionHF(t,X,eaMat(k,:)',distMat(k,:)',S),tspan,X
k);

        if(length(tspan) == 2)
        % Deal with S.oversampFact = 1 case
            tVecOut = [tVecOut; tVeck(1)];
            XMat = [XMat; XMatk(1,:)];
        else
            tVecOut = [tVecOut; tVeck(1:end-1)];
            XMat = [XMat; XMatk(1:end-1,:)];
        end

        Xk = XMatk(end,:)';

        % Ensure that RBI remains orthogonal
        if(mod(k,10) == 0)
            RBIk = [Xk(7:9),Xk(10:12),Xk(13:15)];
            [UR,SR,VR]=svd(RBIk);
            RBIk = UR*VR';
            Xk(7:15) = RBIk(:);
        end

    end

    XMat = [XMat; XMatk(end,:)];
    tVecOut = [tVecOut; tVeck(end,:)];

    P.tVec = tVecOut;
    P.state.rMat = XMat(:,1:3);

    eMat = [];
    for k = 1:M
        RBIk = [XMat(k,7:9)',XMat(k,10:12)',XMat(k,13:15)'];
        ek = dcm2euler(RBIk);
        eMat = [eMat;ek'];
    end

    P.state.eMat = eMat;
    P.state.vMat = XMat(:,4:6);
    P.state.omegaBMat = XMat(:,16:18);
    P.state.omegaVec = XMat(:,19:22);
```

A.11 TF2odecomp.m

```
%ODE vs TF comparison
function [L] = TF2odecomp

quadParamsScript;
constantsScript;
S.quadParams = quadParams;
S.constants = constants;
S.ea = 1;

omega0 = 0;
tspan = [0 0.45];

[t,omega] = ode45(@(t,omega)odefun(t,omega,S),tspan,omega0);
```

```
L.time = t;
L.speed = omega;

plot(t,omega)
xlabel('Time(s)')
ylabel('Omega (rad/s)')



end
    function [omegadot] = odefun(t,omega,S)
        omegadot = (1/S.quadParams.taum(1))*(S.ea*S.quadParams.cm(1)-
omega);
        return
    end
```

A.12 topSimulate_lab1.m

```
% Top-level script for calling simulateQuadrotorDynamics
clear; clc;

%600rad/s rotor speed

% Total simulation time, in seconds
Tsim = 5;
% Update interval, in seconds.  This value should be small relative to
the
% shortest time constant of your system.
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
S.tVec = [0:N-1]'*delt;
% Matrix of disturbance forces acting on the body, in Newtons,
expressed in I
S.distMat = zeros(N-1,3);
% Rotor speeds at each time, in rad/s
S.omegaMat = 600*ones(N-1,4);
% Initial position in m
S.state0.r = [0 0 2]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 0]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in
rad/s
S.state0.omegaB = [0 0 0]';
% Oversampling factor
S.oversampFact = 10;
% Quadrotor parameters and constants
quadParamsScript;
```

```
constantsScript;
S.quadParams = quadParams;
S.constants = constants;
P1 = simulateQuadrotorDynamics(S);

%1500rad/s rotor speed
S.omegaMat = 1500*ones(N-1,4);
P2 = simulateQuadrotorDynamics(S);

%2400rad/s rotor speed
S.omegaMat = 2400*ones(N-1,4);
P3 = simulateQuadrotorDynamics(S);


figure(1);clf;
subplot(3,1,1);
plot(P1.tVec,P1.state.rMat(:,3),'LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('\omega_c = 600rad/s');

subplot(3,1,2);
plot(P2.tVec,P2.state.rMat(:,3),'LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('\omega_c = 1500rad/s');

subplot(3,1,3);
plot(P3.tVec,P3.state.rMat(:,3),'LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('\omega_c = 2400rad/s');
```

A.13 topSimulateContol.m

```
% Top-level script for calling simulateQuadrotorControl
clear; clc;

%--------P Structure----------------
% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
sensorParamsScript;
P.quadParams = quadParams;
P.constants = constants;
P.sensorParams = sensorParams;

%---------R Structure--------------
% Total simulation time, in seconds
Tsim = 10;
```

```
% Update interval, in seconds.  This value should be small relative to
the
% shortest time constant of your system.
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
R.tVec = [0:N-1]'*delt;
%size of circle to be flown
d = 4;
r = d/2;
%vector of angles at each point around the circle
dalpha = 2*pi()/(N-1);
davec = [0:N-1]'*dalpha;
%desired position
rIstar = [2 0 0];
for j = 2:N
    rIstar = [rIstar;r*cos(davec(j)),r*sin(davec(j)),0];
end
rIstar = smoothdata(rIstar);
R.rIstar = rIstar;
%desired velocity
vIstar = [];
for n = 1:N-1
    xdot = (rIstar(n+1,1)-rIstar(n,1))/delt;
    ydot = (rIstar(n+1,2)-rIstar(n,2))/delt;
    zdot = 0;
    vIstar = [vIstar;xdot,ydot,zdot];
end
vIstar = smoothdata([vIstar;vIstar(end,:)]);
R.vIstar = vIstar;
%desired acceleration
aIstar = [];
for m = 1:N-1
    xdotdot = (vIstar(m+1,1)-vIstar(m,1))/delt;
    ydotdot = (vIstar(m+1,2)-vIstar(m,2))/delt;
    zdotdot = 0;
    aIstar = [aIstar;xdotdot,ydotdot,zdotdot];
end
aIstar = smoothdata([aIstar;aIstar(end,:)]);
R.aIstar = aIstar;
%desired body x-axis direction
xIstar = [-1 0 0];
for p = 2:N
    xIstar = [xIstar;-cos(davec(p)),-sin(davec(p)),0];
end
R.xIstar = xIstar;

%----------S Structure----------------
% Matrix of disturbance forces acting on the body, in Newtons,
expressed in I
S.distMat = zeros(N-1,3);
% Initial position in m
```

```
S.state0.r = [2 0 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0.4 pi]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in
rad/s
S.state0.omegaB = [0 0 0]';
% Oversampling factor
S.oversampFact = 10;

% for kk = 0.05:0.15:0.95
%     P.Kd = [kk 0 0;0 kk 0;0 0 kk];
%     for ll = 0.5:0.2:1.5
%         P.K = [ll 0 0;0 ll 0;0 0 ll];
%         for ii = 1:0.5:4
%             P.kd = ii;
%             for jj = 2:0.5:5
%                P.k = jj;
%
%                k = P.k
%                kd = P.kd
%                K = P.K
%                Kd = P.Kd
[Q] = simulateQuadrotorControl(R,S,P);

xIvec = [];
for mm = 1:N-1
    RBIk = euler2dcm(Q.state.eMat(mm,:)');
    xk = RBIk*[1;0;0];
    xIvec = [xIvec;xk'];
end

figure(1);clf;
plot(Q.tVec,Q.state.rMat(:,3),'LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('Vertical position of CM');

figure(2);clf;
plot(Q.state.rMat(:,1), Q.state.rMat(:,2),'LineWidth',1.5);
axis equal; grid on;
hold on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');
quiver(Q.state.rMat(1:10:end-1,1), Q.state.rMat(1:10:end-
1,2),xIvec(:,1),xIvec(:,2));

%             end
%         end
%     end
% end
```

A.14 topSimulateHF.m

```
% Top-level script for calling simulateQuadrotorDynamics
clear; clc;

%600rad/s rotor speed

% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
S.quadParams = quadParams;
S.constants = constants;
% Total simulation time, in seconds
Tsim = 5;
% Update interval, in seconds.  This value should be small relative to
the
% shortest time constant of your system.
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
S.tVec = [0:N-1]'*delt;
% Matrix of disturbance forces acting on the body, in Newtons,
expressed in I
S.distMat = zeros(N-1,3);
% Voltage input to motor at each time, in volts
S.eaMat = (600/S.quadParams.cm(1))*ones(N-1,4);
% Initial position in m
S.state0.r = [0 0 2]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 0]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in
rad/s
S.state0.omegaB = [0 0 0]';
% Oversampling factor
S.oversampFact = 10;

P1 = simulateQuadrotorDynamicsHF(S);

%1500rad/s rotor speed
S.eaMat = (1500/S.quadParams.cm(1))*ones(N-1,4);
P2 = simulateQuadrotorDynamicsHF(S);

%2400rad/s rotor speed
S.eaMat = (2400/S.quadParams.cm(1))*ones(N-1,4);
P3 = simulateQuadrotorDynamicsHF(S);
```

```
figure(1);clf;
subplot(3,1,1);
plot(P1.tVec,P1.state.rMat(:,3),'r','LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('\omega_c = 600rad/s');

subplot(3,1,2);
plot(P2.tVec,P2.state.rMat(:,3),'r','LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('\omega_c = 1500rad/s');

subplot(3,1,3);
plot(P3.tVec,P3.state.rMat(:,3),'r','LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('\omega_c = 2400rad/s');
```

A.15 trajectoryController.m

```
function [Fk,zIstark] = trajectoryController(R,S,P)
% trajectoryController : Controls quadcopter toward a reference 
trajectory.
%
%
% INPUTS
%
% R ---------- Structure with the following elements:
%
%       rIstark = 3x1 vector of desired CM position at time tk in the 
I frame,
%                 in meters.
    rIstark = R.rIstark;
%       vIstark = 3x1 vector of desired CM velocity at time tk with 
respect to
%                 the I frame and expressed in the I frame, in 
meters/sec.
    vIstark = R.vIstark;
%       aIstark = 3x1 vector of desired CM acceleration at time tk 
with
%                 respect to the I frame and expressed in the I frame, 
in
%                 meters/sec^2.
    aIstark = R.aIstark;
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure 
with the
%                 following elements:
```

```
%
%                   rI = 3x1 position in the I frame, in meters
        rI = S.statek.rI;
%                  RBI = 3x3 direction cosine matrix indicating the
%                        attitude
        RBI = S.statek.RBI;
%                   vI = 3x1 velocity with respect to the I frame and
%                        expressed in the I frame, in meters per
second.
        vI = S.statek.vI;
%               omegaB = 3x1 angular rate vector expressed in the body
frame,
%                        in radians per second.

% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                  quad, as defined in quadParamsScript.m

    m = P.quadParams.m;

%     constants = Structure containing constants used in simulation
and
%                  control, as defined in constantsScript.m

    g = P.constants.g;
%
% OUTPUTS
%
% Fk --------- Commanded total thrust at time tk, in Newtons.
%
    e3 = [0;0;1];
    k = 5;
    kd = 2.7;
    er = rIstark - rI; %error vector for quad position in m
    erdot = vIstark - vI; %error vector for quad velocity in m/s

    %trajectory controller desired force vector
    FIstark = k*er+kd*erdot+m*g*e3+m*aIstark;

    Fk = FIstark'*RBI'*e3;

% zIstark ---- Desired 3x1 body z axis expressed in I frame at time
tk.
%
    zIstark = FIstark/norm(FIstark);
```

A.16 voltageConverter.m

```
function [eak] = voltageConverter(Fk,NBk,P)
% voltageConverter : Generates output voltages appropriate for desired
%                     torque and thrust.
%
%
% INPUTS
%
% Fk --------- Commanded total thrust at time tk, in Newtons.
%
% NBk -------- Commanded 3x1 torque expressed in the body frame at
time tk, in
%              N-m.
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
    kF = P.quadParams.kF;
    kN = P.quadParams.kN;
    omegaRdir = P.quadParams.omegaRdir;
    rotor_loc = P.quadParams.rotor_loc;
    m = P.quadParams.m;
    Jq = P.quadParams.Jq;
    Ad = P.quadParams.Ad;
    Cd = P.quadParams.Cd;
    taum = P.quadParams.taum;
    cm = P.quadParams.cm;
    eamax = P.quadParams.eamax;
%
%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m
%
%
% OUTPUTS
%
% eak -------- Commanded 4x1 voltage vector to be applied at time tk,
in
%              volts. eak(i) is the voltage for the ith motor.
    omegamax = eamax*cm;
    Fmax = kF.*omegamax.^2;

    kT = kN./kF;
    convmat = [1 1 1 1;rotor_loc(2,1) rotor_loc(2,2) rotor_loc(2,3)
rotor_loc(2,4);...
        -rotor_loc(1,1) -rotor_loc(1,2) -rotor_loc(1,3) -
rotor_loc(1,4);...
        -kT(1) kT(1) -kT(1) kT(1)];

    beta = 0.9;
    alpha = 1;
    minmat = [min([Fk,4*beta*Fmax(1)]);alpha*NBk];
    Fmat = inv(convmat)*minmat;
```

```
for q = 1:4
    while Fmat(q) > Fmax(q)
        alpha = alpha-0.01;
        minmat = [min([Fk,4*beta*Fmax(q)]);alpha*NBk];
        Fmat = inv(convmat)*minmat;
    end
end

for h = 1:4
    if Fmat(h) < 0
        Fmat(h) = 0;
    end
end

eak = sqrt(Fmat/kF(1))*1/cm(1);
```