ASE 479W Laboratory 2 Report
Quadcopter Feedback Control

Aaron Pandian

February 18, 2024

1 Introduction

Quadrotors become increasingly difficult to control if pushed to provide more complex dynamic functionality. Given a deep understanding of the employed physics, one can establish a high fidelity model to simulate this unmanned aerial vehicle (UAV). However, manually tuning parameters to achieve a desired simulated trajectory is less than ideal. Thus, the industry of controls engineering is introduced. A fundamental concept in the field is that of feedback control: given a reference, variables to control are compared to respective target values. The paper *Direct Acceleration Feedback Control of Quadrotor Aerial Vehicles* states this method "alleviates the need for accurate estimation of platform parameters."

For this laboratory assignment, a high fidelity quadrotor simulator was designed from the theory of Newtonian dynamics and applied in Matlab. Accounting for previously neglected effects, models for aerodynamic drag and voltage to angular rate delay were integrated. Furthermore, to ease control, a feedback control system was designed and experimented to implement a specific circular maneuver. The theory, application, and experiment results are discussed in the following report.

2 Theoretical Analysis

The simulation of a quadcopter relies on multiple underlying concepts that compound. This section poses as an introduction to such complex principles discussed later on.

2.1 Aerodynamic Drag

In the context of a UAV, aerodynamic drag represents the force opposing motion through the air. This principle plays a role accounted for in the quadrotor equation of motion (1) used for this experiment.

$$m\ddot{\boldsymbol{r}}_{\mathrm{I}} = -d_a \boldsymbol{v}_{\mathrm{I}}^{u} - mg\boldsymbol{Z}_{\mathrm{I}} + \sum_{i=1}^{4} \boldsymbol{F}_{i\mathrm{I}} + \boldsymbol{d}_{\mathrm{I}} \tag{1}$$

Above, m is the total mass of the quadcopter, $\mathbf{r_I}$ is the position of the center of mass in the inertial frame, g is the acceleration due to gravity, and $\boldsymbol{Z_I} = e3 \triangleq [0, 0, 1]^T$ is the inertial up-direction axis. $\boldsymbol{F_{iI}}$ is the force produced by the ith rotor, and $\boldsymbol{d_I}$ is an arbitrary disturbance force both on the center of mass expressed in I. As previously mentioned, drag force acts in opposition to motion through air, thus the $-d_a \boldsymbol{v}_{\mathrm{I}}^{u}$ term depicts the drag force d_a in the negative center of mass velocity unit vector direction $-\boldsymbol{v}_{\mathrm{I}}^{u}$, expressed in I. The complexity of aerodynamic drag can be seen by expansion of the drag force term seen below.

$$d_a = \tfrac{1}{2} C_d A_d f_d(\boldsymbol{z}_{\mathrm{I}}, \boldsymbol{v}_{\mathrm{I}}) \rho \tag{2}$$

In (2) the drag force term presents a nonlinear, non-constant system- dependent on the drag coefficient C_d, quadrotor area A_d in the ***x*-*y*** body plane, fluid density ρ, and a function $f_d(\mathbf{z_I}, \mathbf{v_I})$. For simplicity, the vehicle is assumed to be an infinitely thin circular disk, allowing the relationship in (2) to work.

It is important to note that multiple environmental conditions affect the induced aerodynamic drag force d_a causing this inconsistency. The quadcopter state- defined as the position, euler angle attitude, velocity, and angular rate- plays a large role in determining the aerodynamic drag. Specifically, the state velocity and euler angles, which determine directional speed and orientation, have the most prominent influence on drag force. The faster the quadcopter moves in opposition to airflow, the greater drag force is observed. Additionally, the more area in perpendicular contact with the airflow, the more resistance. This is intuitive. If the theoretically infinitely thin disk (defined earlier) flies where the airflow only impacts the thin sides- drag is minimized to *theoretically* zero. However, if the disk reorients and begins to move through the air where the airflow impacts the face side area, the flux increases because the impact area increases- in turn, increasing the drag force. Another state element that alters the induced aerodynamic drag is the angular rate of the quadrotor body with respect to the inertial world frame. Similar to velocity, when the angular rotation of the quadrotor opposes the direction of fluid flow, in this case air, the impact cross-sectional area will face drag resistance. Aside from state factors, the environment plays a large role as well. The density or compressibility, viscosity, and wind speed of air are all factors that impact how the quadcopter *moves through the air* during flight. An extremely dense, or incompressible fluid will greatly hinder movement of anything through itself; the same effect is seen in a highly viscous fluid, which can be thought of as fluid "thickness" or "stickiness." As such, to model aerodynamic drag force, multiple factors must be accounted for. The relationship can be seen in equation (3) with the analytical expansion of $f_d(\boldsymbol{z_I}, \boldsymbol{v_I})$.

$$f_d(\boldsymbol{z_I}, \boldsymbol{v_I}) = (\boldsymbol{z}_I \cdot \boldsymbol{v_I^u})(|v_I|^2) \tag{3}$$

Above, the term f_d can be described as the dot product of the body frame $\boldsymbol{z}$ axis expressed in I and the velocity unit vector expressed in I, multiplied by the magnitude of the quadcopter velocity. By integrating (3) into (2), it is evident how the state elements and environmental conditions discussed above lead to a non-linear aerodynamic drag system. Under ideal circumstances, a potential model for which the drag force d_a can be characterized is using a wind tunnel. In this experimental apparatus, the observer sets all conditions to solve for the aerodynamic drag coefficient C_d using a transformation of (2) and (3). To mimic real world utilization, a replica model for the quadrotor is placed inside the wind tunnel to calculate an accurate cross-sectional area A_d and set in a fixed orientation for $\boldsymbol{z}_I$. Positioning a pitot tube into the apparatus allows the observer to measure the airflow speed of known direction v_I and $\boldsymbol{v_I^u}$. Given air with density ρ, an electric strain gauge can be set on the quadcopter body to measure the force generated from the airflow, or drag force. Through multiple iterations, a robust value for the drag coefficient can then be used to calculate the drag force d_a across variations in fluid density, orientation, and velocity.

2.2 Error Direction-Cosine Matrix

Euler's formula expresses a rotation matrix in terms of an axis of rotation $\hat{\boldsymbol{a}}$ and a rotation angle ϕ:

$$R(\hat{\boldsymbol{a}}, \phi) = \cos(\phi) I_{3\times3} + (1 - \cos\phi)\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T - \sin(\phi)[\hat{\boldsymbol{a}}\times] \tag{4}$$

The $I_{3\times3}$ represents an identity matrix and $[\hat{\boldsymbol{a}}\times]$ denotes the skew-symmetric cross product equivalent, which is a varied notation of a cross product operation depicted in (5).

$$u \times v = [u\times]v \tag{5}$$

The profound result of (4) states that every rotation can be expressed as a rotation about a *single* axis, or the eigenvector of rotation $\hat{\boldsymbol{a}}$ denoted above. Expanding the general formula above produces the proper orthogonal matrix below:

$$R(\hat{a},\phi) = \begin{bmatrix} cos\phi + \hat{a}_x^2(1-cos\phi) & \hat{a}_x\hat{a}_y(1-cos\phi) - \hat{a}_z sin\phi & \hat{a}_x\hat{a}_z(1-cos\phi) + \hat{a}_y sin\phi \\ \hat{a}_y\hat{a}_x(1-cos\phi) + \hat{a}_z sin\phi & cos\phi + \hat{a}_y^2(1-cos\phi) & \hat{a}_y\hat{a}_z(1-cos\phi) - \hat{a}_x sin\phi \\ \hat{a}_z\hat{a}_x(1-cos\phi) - \hat{a}_y sin\phi & \hat{a}_z\hat{a}_y(1-cos\phi) + \hat{a}_x sin\phi & cos\phi + \hat{a}_z^2(1-cos\phi) \end{bmatrix} \tag{6}$$

The matrix present in (6) can assist in understanding control error in our developed system. Specifically, to control the quadcopter attitude, the following formula is derived.

$$\boldsymbol{N}_B = Ke_E + K_d\dot{e}_E + [\omega_B\times]J\omega_B \tag{7}$$

Where K and K_d are diagonal matrices for the derivative and proportional controls in the feedback system proportional-derivative controller. The moment of inertia J, angular rate of the quadcopter in the body frame ω_B, and the cross product equivalent, depicted in (5), $[\omega_B\times]$ term account for the current torque experienced by the quadcopter in the body frame. The error matrix e_E multiplied by the control matrices account for the variation from the reference; these values are combined to create a torque controlling the quadrotor to rotate in a desired motion. The error direction-cosine matrix R_E defined in (8) developed using (6) can help generate the error matrix e_E in (7).

$$R_E = R_{BI}^* R_{BI}^T = \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} \tag{8}$$

Where R_{BI} is the direction-cosine matrix to transform the inertial frame to body frame and R^*_{BI} is the direction-cosine matrix to transform the inertial frame to body frame based on the reference vectors for the body $\boldsymbol{x}$ and $\boldsymbol{z}$ axis expressed in the inertial frame. The correlation of the reference and current rotation matrix above generates the error rotation matrix R_E. Using (8) the derivation of the error matrix is as follows.

$$\boldsymbol{e}_E = \begin{bmatrix} e_{23} - e_{32} \\ e_{31} - e_{13} \\ e_{12} - e_{21} \end{bmatrix} \tag{9}$$

From (6), if the unit vector of rotation $\hat{\boldsymbol{a}} = [\hat{a}_x, \hat{a}_y, \hat{a}_z]^T = [a_1, a_2, a_3]^T$ about the angle ϕ, we can derive the elemental values of $\hat{\boldsymbol{a}}$.

$$R_{23} = a_2a_3(1-c\phi) - a_1s\phi \tag{10.1}$$
$$R_{32} = a_3a_2(1-c\phi) + a_1s\phi \tag{10.2}$$

$$R_{31} = a_3a_1(1-\mathrm{c}\phi) - a_2\mathrm{s}\phi \quad (10.3)$$
$$R_{13} = a_1a_3(1-\mathrm{c}\phi) + a_2\mathrm{s}\phi \quad (10.4)$$
$$R_{12} = a_1a_2(1-\mathrm{c}\phi) - a_3\mathrm{s}\phi \quad (10.5)$$
$$R_{21} = a_2a_1(1-\mathrm{c}\phi) + a_3\mathrm{s}\phi \quad (10.6)$$
$$a_1 = (R_{23} - R_{32})(2\mathrm{s}\phi)^{-1} \quad (10.7)$$
$$a_2 = (R_{31} - R_{13})(2\mathrm{s}\phi)^{-1} \quad (10.8)$$
$$a_3 = (R_{12} - R_{21})(2\mathrm{s}\phi)^{-1} \quad (10.9)$$

The terms, for example, $a_1a_2(1-\mathrm{c}\phi)$ and $a_2a_1(1-\mathrm{c}\phi)$, when subtracted equal zero, since the multiplication of matrices is commutative implying order does not matter. With the derivation equations (10.7 - 10.9) a relation can be formed between the error matrix e_E and the error direction-cosine matrix R_E. Utilizing the definition of the error matrix (9) and the aforementioned equations, the error matrix is written below.

$$e_\mathrm{E} = \begin{bmatrix} a_1(2s\phi) \\ a_2(2s\phi) \\ a_3(2s\phi) \end{bmatrix} \quad (11)$$

Thus, the error matrix can be written out using the elements of the error direction-cosine matrix (9) or by using the rotation angle for each element of the eigenvector of rotation (11) initialized in $\mathrm{R}(\hat{\boldsymbol{a}}, \phi)$. If the rotation angle varies across different axes, then the respective euler angles are to be used.

3 Implementation

To implement the high fidelity quadcopter simulation, a step-by-step approach was implemented. The code used is highlighted in brief demonstrations and discussed as to how functions relate to each other. This section works to explain the pieces of the simulation control algorithm for the quadrotor vehicle.

As mentioned prior, creating the base of the model is the rotation theorem from (1). The equation is entirely defined except for the skew-symmetric cross product equivalent, for which the following function is generated.

```
Unset
function [uCross] = crossProductEquivalent(u)

u1 = u(1,1); % extracted values from row 1, column 1
u2 = u(2,1);
u3 = u(3,1);
uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];

end
```

Figure 1: Cross product equivalent function

The application above is an expansion of (1) by outputting the cross-product-equivalent matrix *uCross* for an arbitrary 3-by-1 vector *u*. This function eases evaluating the cross product between two matrices, finding the vector that is perpendicular to both, which is critical when working with multiple directions and reference frames.

```
Unset
function [R] = rotationMatrix(aHat,phi)

I = [1 0 0; 0 1 0; 0 0 1];
aHatTranspose = aHat.';
R1 = cos(phi)*I;
R2 = (1-cos(phi))*aHat*aHatTranspose;
R3 = sin(phi)*crossProductEquivalent(aHat);
R = R1+R2-R3;

end
```

Figure 2: Rotation matrix function

To complicate the previous function, a rotation matrix function was developed. This function generates a rotation matrix through an angle about a specified axis. This function is a direct application of (4).

By utilizing the script in Fig. 2, a function defining a rotation sequence is defined. To explain, a quadrotor vehicle undergoes asymmetrical rotation to complete maneuvers, meaning it has control over its roll, pitch, and yaw. These allow the aircraft to rotate about the X, Y, and Z axis respectively. For this simulation, a 3-1-2 rotation sequence was enacted, for which the quadcopter rotates itself about three body frame axes in specific order to induce a change in attitude. Each rotation includes one axis and one angular shift about that axis, called Euler axes and Euler angles, and can be modeled using (1). Put together, an attitude matrix C can depict the 3-1-2 rotation, which is a rotation in the "ZXY" sequence.

$$C(\psi, \phi, \theta) = R_2(\boldsymbol{e}_2, \theta)R_1(\boldsymbol{e}_1, \phi)R_3(\boldsymbol{e}_3, \psi) \tag{12}$$

$$\boldsymbol{e}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \boldsymbol{e}_2 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \boldsymbol{e}_3 = \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{11.1}$$

The notation above depicts the euler axes **e** as X, Y, and Z in order from left to right in (13.1). As a result, it is obvious to state that the Euler angles ψ, ϕ, and θ are responsible for the yaw, roll, and pitch in order. Note that the 3-1-2 sequence is written in rotation matrices from right to left. This specific rotation sequence can be simplified using (12) and (6).

```
Unset
function [R_BW] = euler2dcm(e)

phi = e(1,1);
theta = e(2,1);
psi = e(3,1);

R1 = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R2 = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R3 = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];

R_BW = R2e*R1e*R3e;

end
```

Figure 3: Euler angles to rotation matrix

The formula in Fig. 3 converts matrix *e* containing Euler angles ψ, ϕ, θ (in radians) into a directed cosine matrix for a 3-1-2 rotation. Assuming the inertial or world and body frame are initially aligned, *R_BW* can then be used to cast a vector expressed in inertial frame coordinates as a vector in the body frame. The rotation matrices *R1*, *R2*, and *R3*, are expressions of (4) for the specified axis. An alternative function denoting the inverse transformation was also developed.

```
Unset
function [e] = dcm2euler(R_BW)

% 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.

phi = asin(R_BW(2,3));
assert(sin(phi)~=-pi/2 && sin(phi)~=pi/2,'Conversion is singular.');
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]';
end
```

Figure 4: Rotation matrix to euler angles

The function in Fig. 4 does the inverse operation from Fig. 3, whereby a rotation matrix is transformed to a matrix of euler angles, restricted by the limits above. The primary consideration is when the second rotation angle produces a singularity. In the case of a 3-1-2 sequence, when the roll angle $\phi = \pm \pi/2$, then the first and third rotations have equivalent effects, and thus, cannot be distinguished from one another. As a result, this function outputs an error instead of the euler angle matrix *in this situation*.

With these defined functions, the next step of implementation is to develop a model for the quadcopter dynamics. This function, given an initial state, will compute its derivative. This method will be iterated over using the Runge Kutta numerical ODE solver in Matlab. An oriented schematic of the quadcopter is shown in Fig. 5.

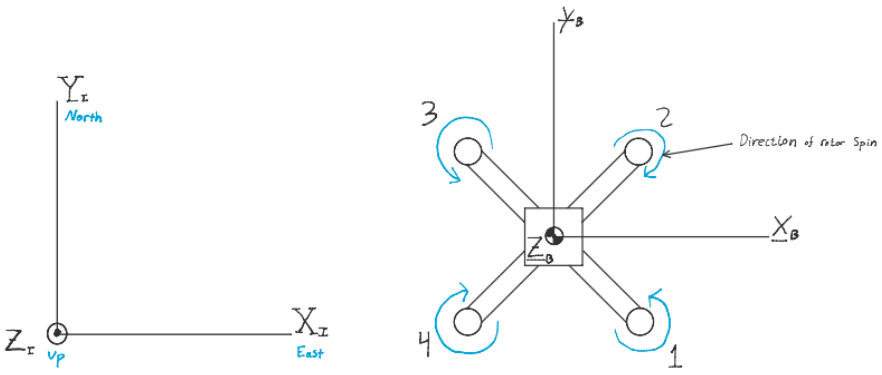


Figure 5: Quadcopter visualization

First it is important to understand the definition of state of the quadrotor, seen in (13).

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{r}_\mathrm{I} \\ \boldsymbol{e} \\ \boldsymbol{v}_\mathrm{I} \\ \boldsymbol{\omega}_\mathrm{B} \end{bmatrix} \tag{13}$$

In the above matrix, r_I represents the position vector of the quadcopter center of mass in the inertial frame and v_I is the velocity vector with respect to the inertial frame, in the inertial frame. The euler angle vector is as defined in Fig. 4 and ω_B is the angular velocity vector in the body frame. Considering the 3-dimensional size of each input matrix, the state matrix is a 12x1 matrix with all variables listed out- instead of demarketing four input matrices within the state matrix. To find the derivative of the state $\dot{X}$, the following equations will need to be computed for each state input.

$$\dot{\boldsymbol{r}}_\mathrm{I} = \boldsymbol{v}_\mathrm{I} \tag{14}$$

$$m\dot{\boldsymbol{v}}_\mathrm{I} = m\ddot{\boldsymbol{r}}_\mathrm{I} = -mg\boldsymbol{z}_\mathrm{I} + \sum_{i=1}^{4} \boldsymbol{F}_{i\mathrm{I}} + \boldsymbol{d}_\mathrm{I} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_\mathrm{BI}^\mathrm{T} \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ F_i \end{bmatrix} + \boldsymbol{d}_\mathrm{I} \tag{15}$$

$$\dot{\boldsymbol{\omega}}_{\mathrm{B}} = J^{-1}\left(\boldsymbol{N}_{\mathrm{B}} - [\boldsymbol{\omega}_{\mathrm{B}}\times]J\boldsymbol{\omega}_{\mathrm{B}}\right) \quad \text{where} \quad \boldsymbol{N}_{\mathrm{B}} = \sum_{i=1}^{4}\left(\boldsymbol{N}_{i\mathrm{B}} + \boldsymbol{r}_{i\mathrm{B}} \times \boldsymbol{F}_{i\mathrm{B}}\right) \tag{16}$$

$$\dot{R}_{\mathrm{BI}} = -[\boldsymbol{\omega}_{\mathrm{B}}\times]R_{\mathrm{BI}} \tag{17}$$

Functions (14) and (17) are explicit in that (14) simply defines the rate of change of position as velocity and (17) implements the theory that the derivative of the rotation matrix is derived by finding the negative cross product between it and the quadcopter angular rate vector ω_{B}. The latter equations are translational and rotational applications of Newton's second law of motion. However, due to the fact that this experiment generated a high fidelity model, two neglected effects are supplemented to update these latter equations.

$$m\ddot{\boldsymbol{r}}_{\mathrm{I}} = -d_a\boldsymbol{v}_{\mathrm{I}}^{u} - mg\boldsymbol{Z}_{\mathrm{I}} + \sum_{i=1}^{4}\boldsymbol{F}_{i\mathrm{I}} + \boldsymbol{d}_{\mathrm{I}} \tag{18}$$

The translation equation of motion (15) is updated to account for the drag force d_{a} in the direction opposite to velocity $-\boldsymbol{v}_{\mathrm{I}}^{\mathrm{u}}$. This drag force term can be written out by combining equations (2) and (3) previously discussed. Furthermore, to account for the fact that the rotor angular rates do not react instantaneously to a controlled voltage change, the transfer function detailing the relationship can be seen in (19).

$$\frac{\Omega(s)}{E_a(s)} = \frac{c_m}{\tau_m s + 1} \tag{19}$$

This phenomenon is further explained in the following section. Due to the fact that the angular rates are now varying with time, the rotor angular rate rate of change vector $\acute{\omega}_{\mathrm{i}}$ must be added to the state variable in our ODE function to be solved by Range-Kutta. To model this effect, the $\acute{\omega}_{\mathrm{i}}$ can be solved using (20) which utilizes a simplification of the transfer function (19).

$$\acute{\omega}_{\mathrm{i}} = (E_a C_{\mathrm{m}} - \omega_{\mathrm{i}})(\tau_{\mathrm{m}})^{-1} \tag{20}$$

The values C_{m} and τ_{m} are constants denoting the factor to convert motor voltage to motor angular rate in steady state in units rad/sec/volt and the unitless time constant governing the response time for input voltage both for each rotor motor in a 4x1 vector. The 4x1 vector for the applied voltage to each rotor motor is denoted as E_a and with C_{m} is used to estimate the target angular rate.

The formula in (16) is defined because the euler angles from the initial state matrix will be transformed, using the function in Fig. 3, to the equivalent rotation matrix. This is done to avoid the singularity error spurred by working with euler angles. Thus, the derivative state vector is formed, where $\dot{e}$ is replaced with the elements of $\dot{R}_{\mathrm{BI}}$ (unrolled column by column) to provide a now 22x1 state matrix. An overview of the process can be seen in Fig. 5.

```
Unset
function [Xdot] = quadOdeFunction(t,X,omegaVec,distVec,P)

% Find rotor_omega_dot for new state vector
Cm = P.quadParams.cm;
Tm = P.quadParams.taum;
omegaVecdot = (Cm.*eaVec - omegaVec) ./ Tm;

% Determine forces and torques for each rotor from rotor angular rates.  FMat =
[zeros(2,4);(P.quadParams.kF.*(omegaVec.^2))'];
NMat = [zeros(2,4);(P.quadParams.kN.*(omegaVec.^2).*(-P.quadParams.omegaRdir)')'];

% Assign some local variables for convenience
vIMag = norm(vI);
zI = RBI * [0 0 1]'; % 3x1
vIUnitVector = vI/vIMag;

% Find derivatives of state elements
rIdot = vI;
fd = dot(zI,vIUnitVector)*(vIMag^2);
da = 0.5*Cd*Ad*rho*fd;
fDragVec = vIUnitVector*da;
fDrag = [fDragVec(1) fDragVec(2) fDragVec(3)]';
vIdot = ([0;0;-mq*gE] + RBI'*sum(FMat,2) + fDrag)/mq;
RBIdot = -omegaBx*RBI;
NB = NB + cross(P.quadParams.rotor_loc(:,ii),FMat(:,ii));
omegaBdot = inv(Jq)*(NB - omegaBx*Jq*omegaB);

% Load the output vector
Xdot = [rIdot;vIdot;RBIdot(:);omegaBdot;omegaVecdot];
```

Figure 5: Quadrotor ordinary differential equation function

The output of this function is the derivative state vector $\dot{X}$, like (13) in construct, with the addition of the rotor angular speed rate of change. With the development of the quadrotor model, the plant in our control system is developed. A whole schematic of the desired feedback loop can be seen in the subsequent image.

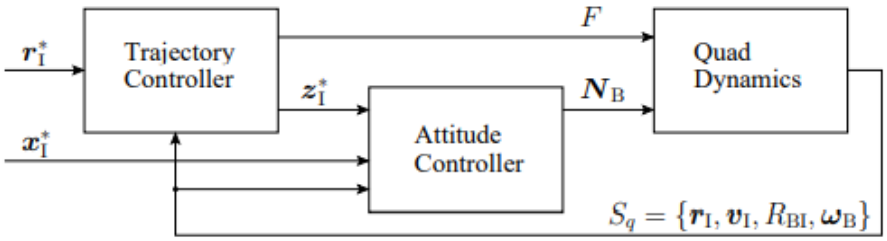


Figure 6: Top-level quadrotor control architecture

The next phase of implementation is to create the two control schemes seen in Fig. 6. The trajectory controller takes in the reference center of mass position vector in the inertial frame $\boldsymbol{r}_I^*$ and outputs the total thrust force the quadcopter needs to generate F and the desired direction of the body axis expressed in the inertial frame $\boldsymbol{z}_I^*$.

$$\boldsymbol{F}_I^* = k\boldsymbol{e}_r + k_d\dot{\boldsymbol{e}}_r + mg\boldsymbol{e}_3 + m\ddot{\boldsymbol{r}}_I^* \tag{21}$$

$$\boldsymbol{e}_r = \boldsymbol{r}^*_I - \boldsymbol{r}_I \tag{22}$$

$$\boldsymbol{z}_I^* = \frac{\boldsymbol{F}_I^*}{\|\boldsymbol{F}_I^*\|} \tag{23}$$

The position error vector $\boldsymbol{e}_r$ derived from the reference and calculated position vectors facilitate the calculation of the desired force using the proportional and derivative control vectors, the desired acceleration vector $\ddot{r}_I^*$, and gravity. With the desired force vector $\boldsymbol{F}_I^*$, the the calculation of $\boldsymbol{z}_I^*$ can be seen in (23). Equations (21) and (23) are brought together to calculate the output of the trajectory control in (24).

$$F = \boldsymbol{F}_I^* \cdot \boldsymbol{z}_I = (\boldsymbol{F}_I^*)^T R_{BI}^T \boldsymbol{e}_3 \tag{24}$$

This process is laid out in Fig. 7 with the code for the trajectory controller.

```
Unset
function [Fk,zIstark] = trajectoryController(R,S,P)

% Find error vectors
er = rIstark - rI;
```

```
er_dot = vIstark - vI;

% Control constants
K = [4 0 0; 0 4 0; 0 0 4];
Kd = [1.5 0 0; 0 1.5 0; 0 0 1.5]; % Both 3x3

% Finding total force
FIstark = K*er + Kd*er_dot + [0; 0; m*g] + m*aIstark;
zIstark = FIstark./norm(FIstark);
e3 = [0 0 1]';
zI = (RBI')*e3;
Fk = (FIstark')*zI;
```

Figure 7: Trajectory controller function

To create the attitude controller evident in Fig. 6, the guiding equation resembles (7), but we instead substitute $-\omega_B$ for $\dot{e}_E$ because of the complexity of deriving $\dot{e}_E$ from the attitude controller inputs z_I^* and x_I^*. This can be done for small error angles where the desired angular rate ω_B is zero evident from (25).

$$\dot{\boldsymbol{e}}_E \approx \boldsymbol{\omega}_B^* - \boldsymbol{\omega}_B \tag{25}$$

$$\boldsymbol{N}_B = K\boldsymbol{e}_E - K_d\boldsymbol{\omega}_B + [\boldsymbol{\omega}_B\times]\, J\boldsymbol{\omega}_B \tag{26}$$

Thus, we arrive at (26) by integrating equations (7) and (25).

```
Unset
function [NBk] = attitudeController(R,S,P)

% Small angle assumption
eE_dot = wB;

% Control parameters
K = [1 0 0; 0 1 0; 0 0 1];
Kd = [0.3 0 0; 0 0.3 0; 0 0 0.3];

% Deriving RE using equation (8)
b = cross(zIstark, xIstark)/(norm(cross(zIstark, xIstark))); % 3x1
a = cross(b,zIstark); % 3x1
RBIstark = [a, b, zIstark]'; % 3x3
RE = RBIstark*(RBI');
```

```
% Using equation (9)
eE = [RE(2,3)-RE(3,2), RE(3,1)-RE(1,3), RE(1,2)-RE(2,1)]'; % 3x1
NBk = K*eE - Kd*eE_dot + crossProductEquivalent(wB)*J*wB;
```

Figure 8: Attitude control function

The final piece in the control loop is a hidden function from Fig 6. where the total force F and control torque $\boldsymbol{N}_B$ vectors, output values from both control functions, are converted to voltages for each rotor motor to be *input* into the plant ODE function. The following equation is used to calculate the respective forces necessary for each rotor to generate to satisfy the control parameters.

$$\begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ y_1 & y_2 & y_3 & y_4 \\ -x_1 & -x_2 & -x_3 & -x_4 \\ -k_T & k_T & -k_T & k_T \end{bmatrix}^{-1} \begin{bmatrix} \min(F, 4\beta F_{\max}) \\ \alpha \boldsymbol{N}_B \end{bmatrix} \tag{27}$$

The last vector in the above equation consists of the minimum value between the control total thrust force and the max force output of each rotor F_{max} multiplied by four, and the elements of the control torque vector $\boldsymbol{N}_B$. Furthermore, k_T represents the ratio between the k_N and k_F torque and thrust constants. The alpha and beta values are used to ensure some minimum amount of rotor thrust is allocated to applying the torque N_B. Finally, x_i and y_i demarcate the ±1 coordinate positions of the rotors apparent in Fig. 5. The calculation of the rotor forces must satisfy the limitation condition of $0 \leq F_i \leq F_{max}$, where the maximum force output from a rotor motor F_{max} can be calculated using (28) and (29).

$$F_{max} = k_F(\omega^2_{max}) \tag{28}$$

$$\omega_{max} = C_m e_{a,max} \tag{29}$$

The known maximum voltage vector $e_{a,max}$ applied to each rotor allows the derivation of the maximum rotor force F_{max}. With the generation of the rotor force vector F_i from (27) equations (28) and (29) are used again to determine the desired voltage vector for each rotor motor to be input into the quadcopter ODE function.

Conclusively, the Runge Kutta ordinary differential equation solver in Matlab supports the simulation of the model above, as mentioned before. Given an arbitrary initial state structure, the simulator allows for visualization of the quadrotor dynamics across a preset sample time. The basic implementation of the control system in Fig. 6 is presented in the Fig. 9 simulator function.

Unset

```
function [Q] = simulateQuadrotorControl(R,S,P)

N = length(R.tVec);
dtIn = R.tVec(2) - R.tVec(1);
dtOut = dtIn/S.oversampFact;
RBIk = euler2dcm(S.state0.e);

% Initial state
S.state0.omegaVec = [0 0 0 0]';
Xk = [S.state0.r;S.state0.v;RBIk(:);S.state0.omegaB;S.state0.omegaVec];

XMat = []; tVec = [];

for kk=1:N-1
 tspan = [R.tVec(kk):dtOut:R.tVec(kk+1)]';
 distVeck = S.distMat(kk,:)';

 % Sk and Rk represent the state and reference structure for current time
 [Fk, zIstark] = trajectoryController(Rk,Sk,P);
 [NBk] = attitudeController(Rk,Sk,P);
 [eaVeck] = voltageConverter(Fk,NBk,P);
 [tVeck,XMatk] = ode45(@(t,X) quadOdeFunctionHF(t,X,eaVeck,distVeck,Pa),tspan,Xk);

 if(length(tspan) == 2)
   tVec = [tVec; tVeck(1)];
   XMat = [XMat; XMatk(1,:)];
 else
   tVec = [tVec; tVeck(1:end-1)];
   XMat = [XMat; XMatk(1:end-1,:)];
 end
 Xk = XMatk(end,:)';
 if(mod(kk,10) == 0)
   RBIk(:) = Xk(7:15);
   [UR,SR,VR]=svd(RBIk);
   RBIk = UR*VR'; Xk(7:15) = RBIk(:);
 end
end

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

% Create output
M = length(tVec);
Q.tVec = tVec;
Q.state.rMat = XMat(:,1:3);
Q.state.vMat = XMat(:,4:6);
Q.state.omegaBMat = XMat(:,16:18);
Q.state.eMat = zeros(M,3);
```

```
Q.state.eMat(mm,:) = dcm2euler(RBI)';
```

Figure 9: Runge-Kutta quadcopter ODE numerical simulator

This function outputs a structure containing matrix values to define all states through every iteration of time, till the defined sampling time. With this, a visualization is processed. Moreover, this simulator is employed to assess the control application in this experiment. To do so, a reference structure is further detailed in the subsequent section. A protocol generating the target matrices can be seen in Fig. 10.

Unset

```
% Total simulation time, in seconds
Tsim = 10;
% Update interval, in seconds.  Currently, 200 Hz.
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec = [0:N-1]'*delt;

% Nx3 matrix of desired CM positions in the I frame, in meters.
R.rIstar = [];

% Nx3 matrix of desired body x-axis direction, expressed as a unit vector in the I
frame.
R.xIstar = [];

% Nx3 matrix of desired CM velocities with respect to the I frame and expressed in
the I frame, in meters/sec.
R.vIstar = [];

% Nx3 matrix of desired CM accelerations with respect to the I frame and expressed
in the I frame, in meters/sec^2.
R.aIstar = [];

for kk=1:N-1
   t = tVec(kk);
   rIstar = [4*cos(pi+(t*(pi/5))), 4*sin(t*(pi/5)), 1]'; % Cartesian coordinates
   xIstar = -(rIstar)/norm(rIstar); % xIstar points opposite to position
   vIstar = (4*pi/5)*cross([0;0;1], xIstar); % velocity is tangent to xIstar
   aIstar = (((4*pi/5)^2)/4)*xIstar; % Centrifugal equation and xIstar direction

   % Update vector values
   R.rIstar = [R.rIstar, rIstar];
   R.xIstar = [R.xIstar, xIstar];
   R.vIstar = [R.vIstar, vIstar];
   R.aIstar = [R.aIstar, aIstar];
```

```
end
```

Figure 10: Quadrotor circular flight reference structure derivation

These functions set up the foundation for conducting dynamical analysis on a quadcopter for various initial conditions.

4 Results and Analysis

In this section, results of the lab experiments are discussed. Furthermore, the code theory is tested to attempt verification.

First, an analysis of the voltage to angular rate transfer function was performed. Below is a plot depicting a step input into (19).

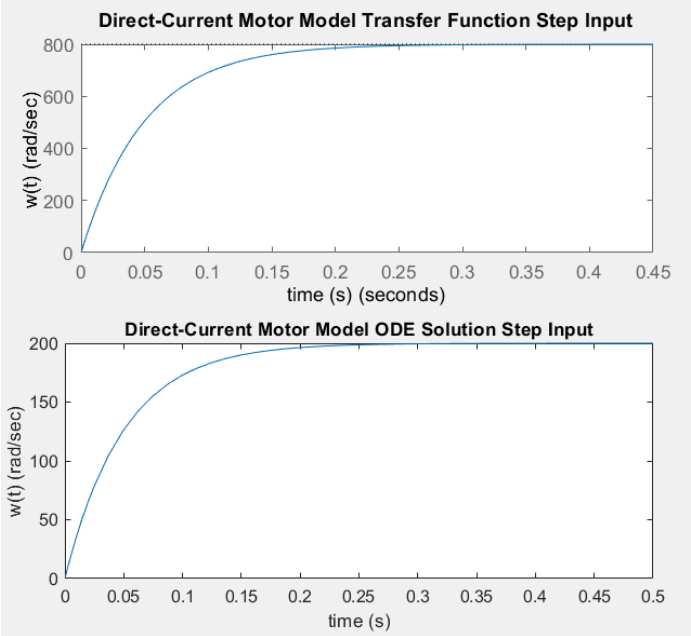


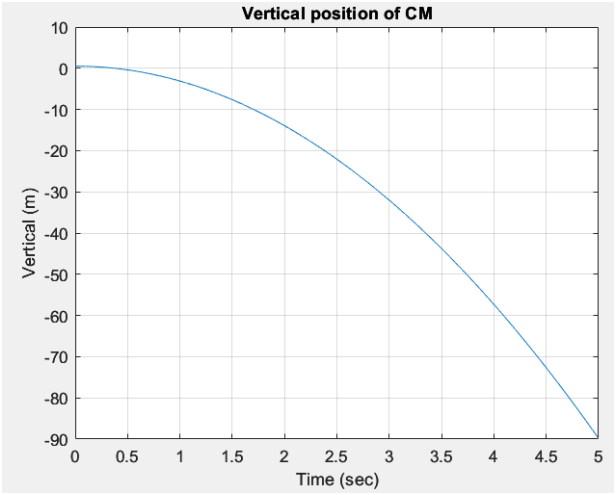
Figure 11: Voltage to angular rate conversion function using transfer function and ODE solver

As evident from Fig. 11, the 90% rise time is around 0.115 seconds. If this plot were to scale by multiplying the step input by any value, the rise time would stay the same, according to the transfer function. Nevertheless, in practice, one should expect the rise time to vary with the input voltage. For example, if the value of the input voltage e_a where to shift from 0 to an arbitrary value e_{as}, the value of the rise time would increase as well since the time for the motor to reach the desired angular rate if voltage is zero is zero. Conversely, when the voltage increases to an arbitrary value, the rise time would depend on the percentage

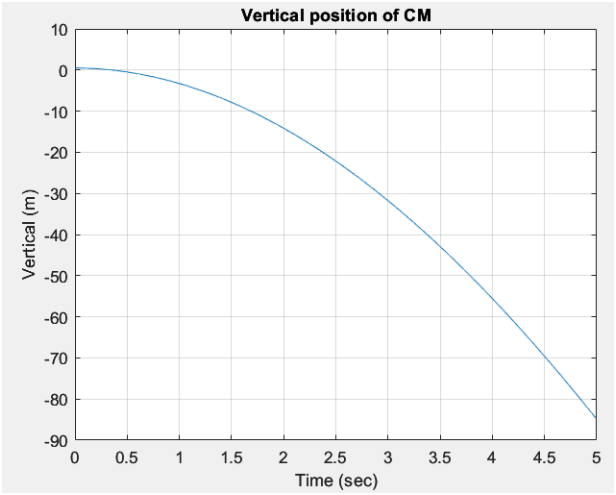
of that value which is still a higher magnitude. This would entail that the system would inherently take a longer time to reach the 90% steady state value than when the magnitude of the input was less. To verify the application of this model, the transfer function was converted to a first order differential equation and, using the Matlab Runge-Kutta ODE numerical solver, was also plotted in Fig. 11. The function can be seen below.

$$L^{-1}\{\frac{C_m}{\tau_m s+1}\} = \frac{C_m}{\tau_m} e^{-t/\tau_m} \tag{30}$$

The equation in (30) showcasing the inverse laplace transformation displays a relationship largely identical to the plot created by the transfer function. In integrating this previously neglected effect, alongside the aerodynamic drag, into the quadrotor dynamics model a more robust model is seemingly developed. To verify increased accuracy, a comparison between the previous dynamics function and the new high fidelity dynamics function was conducted. This comparison was done using a simulation along a five second interval and three different choices of ω_c where $\omega_i = \omega_c$ for i = 1, 2, 3, 4 and $e_a = \omega_c/C_m$ for the old and new dynamics function input, respectively, hold the rotor angular rates constant. Initiating the quadcopter at a near-zero-velocity hover state with $\boldsymbol{z}_I$ parallel to $\boldsymbol{Z}_I$, the results are generated in the following figures.



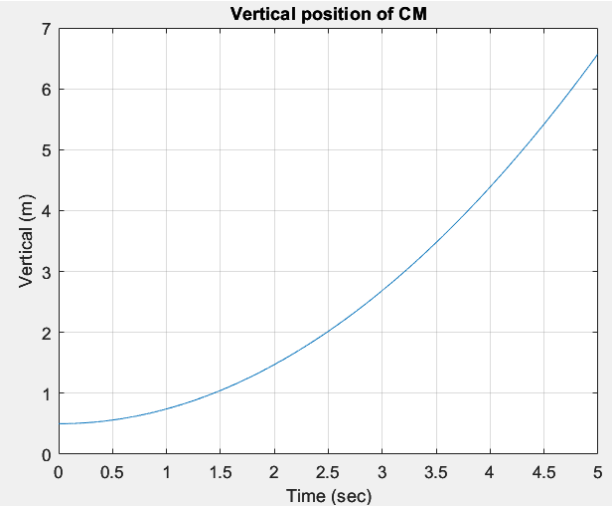
a. Basic quadrotor dynamics



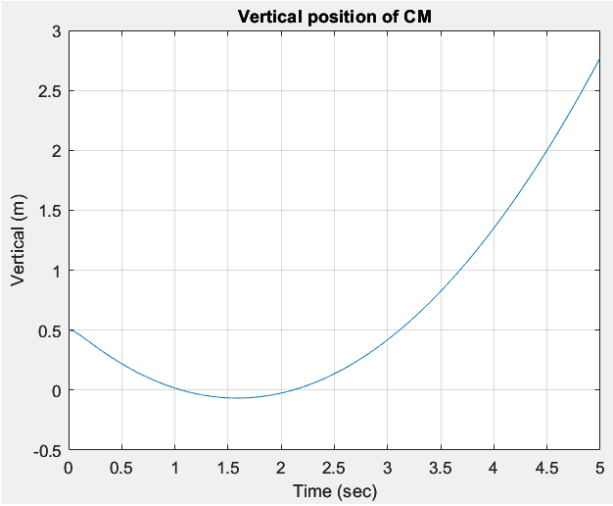
a. High fidelity quadrotor dynamics

Figure 12: Vertical distance plot, rest state, 0.5 meter altitude, ω_c = 300 rad/sec

Above the rotor rates are initialized to a value incapable of maintaining a hover state. As a result, both models slowly fall. The rate of the quadcopter falling increases as acceleration is maintained downwards; this is due to the fact that gravity is the primary force on the quadrotor, which acts downwards. The high fidelity model looks similar to the basic model here since the voltage input delay and drag force do not play much of a factor. However, it can be noted that the final distance of the basic model is further downward, likely due to the lack of air resistance in the model definition.



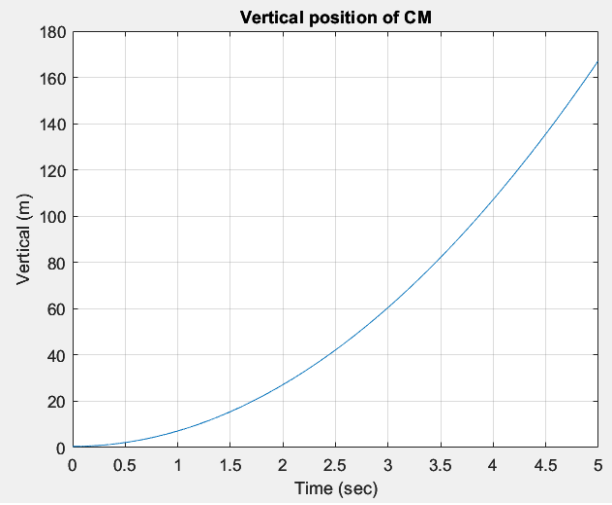
a. Basic quadrotor dynamics



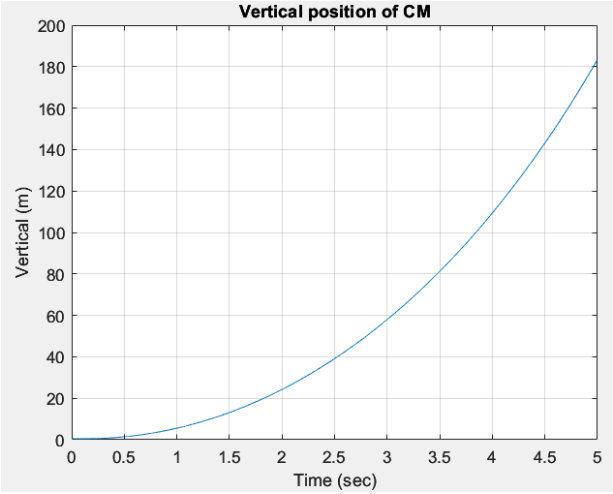
b. High fidelity quadrotor dynamics

Figure 13: Vertical distance plot, rest state, 0.5 meter altitude, ω_c = 600 rad/sec

In Fig. 13, the comparison was run with angular rates much closer to the hover speeds necessary for the thrust force to counteract gravity. In this case the angular speed causes a thrust that supersedes the gravitational force in the negative vertical direction. In this test, the implementation of the voltage input delay model into the high fidelity dynamics function is clearly evident. A large dip at the outset is observed due to rotor start-up delay, as opposed to the instantaneous generation of ω_c in the basic dynamics model. This is the primary cause for why the quadrotor in the basic model is predicted to travel much higher at the end of the sample time.



a. Basic quadrotor dynamics



b. High fidelity quadrotor dynamics

Figure 14: Vertical distance plot, rest state, 0.5 meter altitude, ω_c = 900 rad/sec

Lastly, a high rotor angular rate was initialized for Fig. 14. Due to the direct-motor voltage to rotor angular speed delay, the high fidelity model has a slightly slower increasing speed up to three seconds. After this point, the high fidelity model quadcopter moves at a faster rate and finishes the simulation at a further vertical distance than the basic quadrotor dynamic model. This is an interesting result, reasonably due to the initial small drop in height caused by voltage input reaction delay. The initial drop generates potential energy later converted to kinetic energy as the quadrotor moves back up, allowing the high fidelity model to move slightly faster than the basic quadrotor model afterwards.

With an increased accuracy present in the new dynamic model, understanding how to tune the control constants for the now developed feedback system control functions highlights the next step. To introduce the concept, tuning the proportional control and derivative control in the figure below to specific specifications marks a great segue.

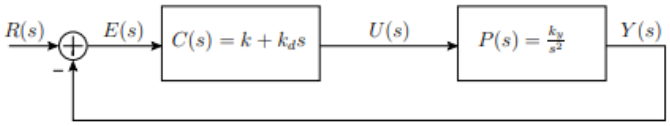


Figure 15: Proportional-derivative control of plant *P*(s)

The feedback control system employed in the final model for this experiment utilizes the same proportional-derivative control. For Fig. 15, assuming $k_y = 1$, to find control parameters that enable a rise time T_r under 0.25 sec, percent overshoot P_o under 30% and 2% settling time T_s under 2 sec- a Matlab Simulink model was generated. As such, the finalized parameters were found to be k = 32 and $k_d = 15$. The following unit step response to the system was used to validate if the control system passed the performance criterion.

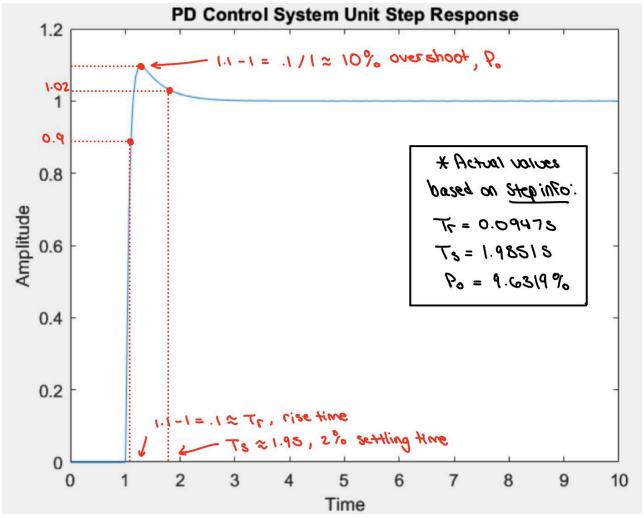


Figure 16: Double-integrator system controlled response with marked performance metrics

As evident from Fig. 16, the tuned control constants provide a system that passes all initial performance parameters. Approaching the process of tuning the control variables in each controller of the quadrotor feedback control loop, the final values can be found in Fig. 7 and 8.

With the control loop complete, the simulation from Fig. 9 can be run for experimentation. However, to achieve a specific trajectory utilizing this high fidelity simulation system, reference vectors for position, velocity, acceleration, and the body $\boldsymbol{x}_I$ axis must be developed.

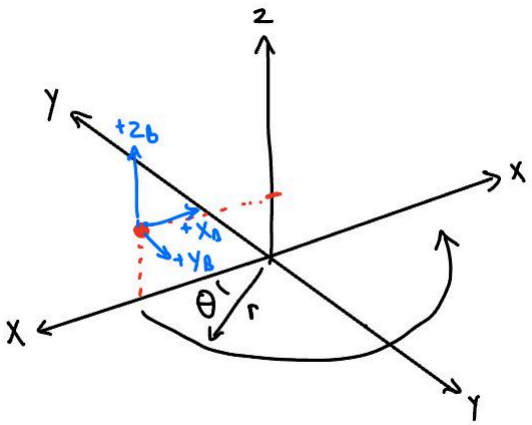


Figure 17: Initial state of quadcopter for circular trajectory

Using Fig. 17, a quadcopter starting at rest with an arbitrary starting position and orientation will travel a level, 4-diameter circular path with an orbital period of ten seconds. To first create the trajectory the, the euler angle attitude matrix was set to zero and the position vector $\boldsymbol{r}_I$ started at $[0\ 0\ -4]^T$ meters. As a result, the position matrix to follow a full circle with a diameter of 4 m at an altitude of 1 m was created using (31).

$$\boldsymbol{r}_I^* = \begin{bmatrix} 4 \cdot \cos(\pi + (t \cdot \frac{\pi}{5})) \\ 4 \cdot \sin(t \cdot \frac{\pi}{5}) \\ 1 \end{bmatrix} \text{ m} \tag{31}$$

Due to the fact that the $\boldsymbol{x}_I^*$ must be a unit vector and point into the center of the circle, or in this case the origin, the $\boldsymbol{x}_I^*$ is simply $-\boldsymbol{r}_I^*/|\boldsymbol{r}_I^*|$. This works for all time because the $\boldsymbol{x}_I$ vector is pointing at the circle center at the start of the simulation. To find the velocity magnitude, the orbital period T is now used like below.

$$v = (2\pi r)T^{-1} = 4\pi/5 \text{ m/s} \tag{32}$$

In terms of direction, the reference velocity vector must stay tangential to $\boldsymbol{x}_I^*$ and the $\boldsymbol{Z}_I$ axis. This the matrix for $\boldsymbol{v}_I^*$ can be set up as (33).

$$\boldsymbol{v}_I^* = (4\pi/5)([0\ 0\ 1]^T \times \boldsymbol{x}_I^*)\ \text{m/s} \tag{33}$$

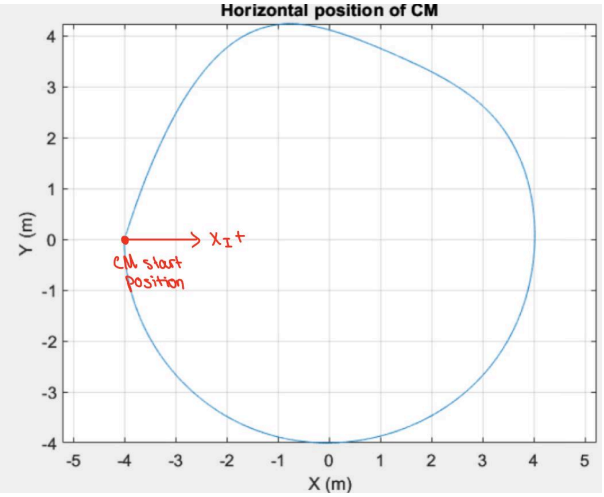
Conclusively, the centrifugal acceleration can be found using the formula below.

$$a_c = v^2 r^{-1} = (4\pi/5)^2/4\ \ \text{m}^2/\text{s}$$

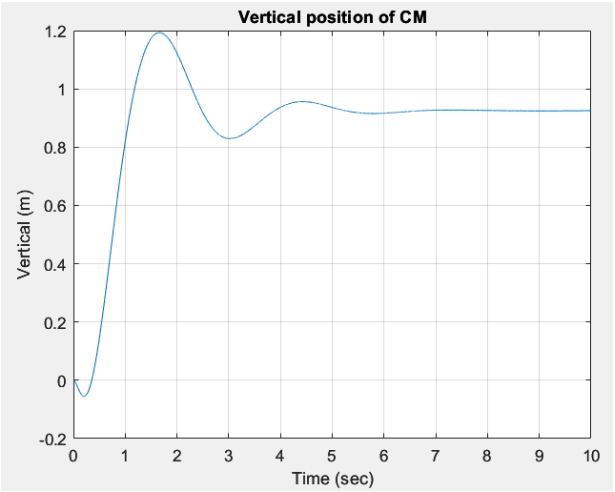
As is the definition of centrifugal force, the direction of the acceleration is also pointing into the circle, or the origin, thus, the acceleration aligns with $\boldsymbol{x}_I^*$. The final reference matrix is created as follows.

$$\boldsymbol{a}_I^* = ((4\pi/5)^2/4)(\boldsymbol{x}_I^*) \tag{34}$$

Inputting the derived matrices into the simulation explicit in Fig. 9 provides the resulting proof of a successful control application.



a. Horizontal position of quadcopter



b. Vertical position of quadcopter

Figure 18: Simulated quadcopter motion for controlled reference trajectory

From the information above, we in Fig. 18b that because the quadrotor starts at an initial position of no altitude, the control spikes the necessary input voltage for the quadrotor to catch up to the desired trajectory at an altitude of 1 m. Furthermore, at the start we see an initial dip in vertical position due to the fact that the rotor angular rates were at rest and the input voltage change does not occur instantaneously. We see a slight overshoot of the quadrotor vertical position but the system settles under seven seconds. Due to the initial error in the quadrotor vertical state, the horizontal position initially forms a malshapped circle. However,

once the quadrotor corrects its altitude at around the settling time, the horizontal position quickly adjusts and a complete circular trajectory is completed.

5 Conclusion

A closed loop control strategy was developed to generate rotor motor input voltages, necessary to increase rotor angular rate, causing the simulated quadcopter to fly in a complete circle on a horizontal plane. The strategy was based on a proportional-derivative (PD) controller as part of a feedback loop whereby, using a reference trajectory characteristics, trajectory and attitude controllers attempt to minimize the error between what the quadrotor dynamics predict and the target state. To increase the robustness of the system, the accuracy of the dynamics model was bolstered by accounting for neglected effects such as aerodynamic drag and the transfer from input voltage to rotor angular rate. As a result of development, the control parameters were tuned to allow successful implementation of the quadcopters circular flight. Based on the experiment conducted, it is much simpler and *more effective* to control a trajectory using a closed feedback loop in contrast to an open-loop controller where most parameters must be predefined. This can lead to complex lead times spent configuring an input structure to said simulation.

References

[1] M. Hamandi, M. Tognon and A. Franchi, "Direct Acceleration Feedback Control of Quadrotor Aerial Vehicles," *2020 IEEE International Conference on Robotics and Automation (ICRA)*, Paris, France, 2020, pp. 5335-5341, doi: 10.1109/ICRA40945.2020.9196557.

[2] Hamano, F. (n.d.). *Derivative of Rotation Matrix – Direct Matrix Derivation of Well-Known Formula*. https://arxiv.org/ftp/arxiv/papers/1311/1311.6010.pdf

[3] Reizenstein, Axel. Position and Trajectory Control of a Quadcopter Using PID and LQ Controllers, Linköping University, 2017, liu.diva-portal.org/smash/get/diva2:1129641/FULLTEXT01.pdf.

Appendix

The complete coding script for experimentation can be found in the Appendix.

```
Unset
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
%
%
% OUTPUTS
%
% uCross ----- 3-by-3 skew-symmetric cross-product equivalent matrix
%
%+-------------------------------------------------------------------------------+
% References: None
%
%
% Author: Aaron Pandian
%+===============================================================================+
u1 = u(1,1); % extracted values from row 1, column 1
u2 = u(2,1);
u3 = u(3,1);
uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];
end
```

Figure 1: Cross vector equivalent function

```
Unset
function [R] = rotationMatrix(aHat,phi)
% rotationMatrix : Generates the rotation matrix R corresponding to a rotation
% through an angle phi about the axis defined by the unit
% vector aHat. This is a straightforward implementation of
% Euler's formula for a rotation matrix.
%
% INPUTS
%
% aHat ------- 3-by-1 unit vector constituting the axis of rotation,
% synonmymous with K in the notes.
%
```

```
% phi -------- Angle of rotation, in radians.
%
%
% OUTPUTS
%
% R ---------- 3-by-3 rotation matrix
%
%+--------------------------------------------------------------------------+
% References: None
%
%
% Author: Aaron Pandian
%+==========================================================================+

function [uCross] = crossProductEquivalent(u)
u1 = u(1,1);
u2 = u(2,1);
u3 = u(3,1);
uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];
end

I = [1 0 0; 0 1 0; 0 0 1];
aHatTranspose = aHat.';
R1 = cos(phi)*I;
R2 = (1-cos(phi))*aHat*aHatTranspose;
R3 = sin(phi)*crossProductEquivalent(aHat);
R = R1+R2-R3;
end
```

Figure 2: Rotation matrix development function

```
Unset
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: None
%
%
% Author: Aaron Pandian
%+==========================================================================+

phi = asin(R_BW(2,3));
% assert() throws an error if condition is false
assert(sin(phi)~=-pi/2 && sin(phi)~=pi/2,'Conversion is singular.');
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]';
end
```

Figure 3: Direction cosine matrix to euler angles function

```
Unset
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: Attitude Transformations. VectorNav. (n.d.).
%https://www.vectornav.com/resources/inertial-navigation-primer/math-fundamentals/
%math-attitudetran
%
% Author: Aaron Pandian
%+===============================================================================+

phi = e(1,1);
theta = e(2,1);
psi = e(3,1);

% Method 1
R1e = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R2e = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R3e = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];

% Method 2
% function [R] = rotationMatrix(aHat,phi)
%
% function [uCross] = crossProductEquivalent(u)
% u1 = u(1,1); % extracted values from row 1, column 1
% u2 = u(2,1);
% u3 = u(3,1);
% uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];
% end
%
% I = [1 0 0; 0 1 0; 0 0 1];
% aHatTranspose = transpose(aHat);
% R1 = cos(phi)*I;
% R2 = (1-cos(phi))*aHat*aHatTranspose;
% % or I + crossProductEquivilant(aHat)^2 = a*a^T
% R3 = sin(phi)*crossProductEquivalent(aHat);
% R = R1+R2-R3;
% end
%
% R3e = rotationMatrix([0;0;1],psi);
```

```
% R1e = rotationMatrix([1;0;0],phi);
% R2e = rotationMatrix([0;1;0],theta);
% Using 3-1-2 Rotation
R_BW = R2e*R1e*R3e;
end
```

Figure 4: Euler angles to direction cosine matrix

Unset

```
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
%              vI = 3x1 velocity vector wrt I and in I, in meters/sec
%             RBI = 3x3 attitude matrix from I to B frame
%          omegaB = 3x1 angular rate vector of body wrt I, expressed in B
%                   in rad/sec
%        omegaVec = 4x1 vector of rotor angular rates, in rad/sec.
%                   omegaVec(i) is the angular rate of the ith rotor.
%
%    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
%
%     constants = Structure containing constants used in simulation and
```

```
%                 control, as defined in constantsScript.m
%
% OUTPUTS
%
% Xdot ------- Nx-by-1 time derivative of the input vector X
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+
% Extract quantities from state vector
rI = X(1:3);
vI = X(4:6);
RBI = zeros(3,3);
RBI(:) = X(7:15);
omegaB = X(16:18);
omegaVec = X(19:22);
% Convert voltage applied to rotor angular rates
% Find omega_dot for new state vector
Cm = P.quadParams.cm;
Tm = P.quadParams.taum;
omegaVecdot = (Cm.*eaVec - omegaVec) ./ Tm;
% Determine forces and torques for each rotor from rotor angular rates.  The
% ith column in FMat is the force vector for the ith rotor, in B.  The ith
% column in NMat is the torque vector for the ith rotor, in B.  Note that
% we negate P.quadParams.omegaRdir because the torque acting on the body is
% in the opposite direction of the angular rate vector for each rotor.
FMat = [zeros(2,4);(P.quadParams.kF.*(omegaVec.^2))'];
NMat = [zeros(2,4);(P.quadParams.kN.*(omegaVec.^2).*(-P.quadParams.omegaRdir)')'];
% Assign some local variables for convenience
mq = P.quadParams.m;
gE = P.constants.g;
Jq = P.quadParams.Jq;
omegaBx = crossProductEquivalent(omegaB);
Cd = P.quadParams.Cd;
Ad = P.quadParams.Ad;
rho = P.constants.rho;
vIMag = norm(vI);
zI = RBI * [0 0 1]'; % 3x1
% The problem on first iteration
if vIMag == 0
   vIUnitVector = [0 0 0];
else
   vIUnitVector = vI/vIMag;
end
```

```
% Find derivatives of state elements
rIdot = vI;
fd = dot(zI,vIUnitVector)*(vIMag^2);
da = 0.5*Cd*Ad*rho*fd;
fDragVec = vIUnitVector*da;
fDrag = [fDragVec(1) fDragVec(2) fDragVec(3)]';
vIdot = ([0;0;-mq*gE] + RBI'*sum(FMat,2) + fDrag)/mq;
RBIdot = -omegaBx*RBI;
NB = sum(NMat,2);
for ii=1:4
 NB = NB + cross(P.quadParams.rotor_loc(:,ii),FMat(:,ii));
end
omegaBdot = inv(Jq)*(NB - omegaBx*Jq*omegaB);
% Load the output vector
Xdot = [rIdot;vIdot;RBIdot(:);omegaBdot;omegaVecdot];
```

Figure 5: Quadcopter ordinary differential equation function

Unset

```
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.
%
%  oversampFact = Oversampling factor. Let dtIn = tVec(2) - tVec(1). Then the
%                 output sample interval will be dtOut =
%                 dtIn/oversampFact. Must satisfy oversampFact >= 1.
%
%         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).
%
%        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
%
%                   e = 3x1 vector of Euler angles, in radians, indicating the
```

```
%                       attitude
%
%                   v = 3x1 velocity with respect to the world frame and
%                       expressed in the world frame, in meters per second.
%
%               omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
%       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).
%
%    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).
%
%+-------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+===============================================================================+
N = length(S.tVec);
dtIn = S.tVec(2) - S.tVec(1);
dtOut = dtIn/S.oversampFact;
RBIk = euler2dcm(S.state0.e);
% Initial angular rates in rad/s, initialize here as per lab document
S.state0.omegaVec = [0 0 0 0]';
Xk = [S.state0.r;S.state0.v;RBIk(:);S.state0.omegaB;S.state0.omegaVec]; % Updated
state to include initial rotor rates
Pa.quadParams = S.quadParams;
Pa.constants = S.constants;
XMat = []; tVec = [];
for kk=1:N-1
 tspan = [S.tVec(kk):dtOut:S.tVec(kk+1)]';
 eaVeck = S.eaMat(kk,:)'; % Added voltage vector to be used in ODE
 distVeck = S.distMat(kk,:)';
 [tVeck,XMatk] = ...
     ode45(@(t,X) quadOdeFunctionHF(t,X,eaVeck,distVeck,Pa),tspan,Xk);
 if(length(tspan) == 2)
   tVec = [tVec; tVeck(1)];
   XMat = [XMat; XMatk(1,:)];
 else
   tVec = [tVec; tVeck(1:end-1)];
   XMat = [XMat; XMatk(1:end-1,:)];
 end
 Xk = XMatk(end,:)';
 % Ensure that RBI remains orthogonal
 if(mod(kk,10) == 0)
   RBIk(:) = Xk(7:15);
   [UR,SR,VR]=svd(RBIk);
   RBIk = UR*VR'; Xk(7:15) = RBIk(:);
 end
end
XMat = [XMat;XMatk(end,:)];
tVec = [tVec;tVeck(end,:)];
M = length(tVec);
P.tVec = tVec;
P.state.rMat = XMat(:,1:3);
P.state.vMat = XMat(:,4:6);
P.state.omegaBMat = XMat(:,16:18);
```

```
% P.state.omegaVec = XMat(:,19:22); % OmegaVec not included in state but
% recorded in QuadODE, so can optionally include in output.
P.state.eMat = zeros(M,3);
RBI = zeros(3,3);
for mm=1:M
 RBI(:) = XMat(mm,7:15);
 P.state.eMat(mm,:) = dcm2euler(RBI)';
end

   % oversampFact relative to the coarse timing of each segment.
   tspan = [tVec(k):dtOut:tVec(k+1)]';

   % Run ODE solver for time segment
   [tVeck,XMatk] = ode45(@(t,X)quadOdeFunction(t,X,omegaVec,distVec,S), tspan,
Xk);

   % Add the data from the kth segment to your storage vector
   tVecOut = [tVecOut; tVeck(1:end-1)];
   XMat = [XMat; XMatk(1:end-1,:)];

   % Prepare for the next iteration
   Xk = XMatk(end,:)';
end

% Store the final state of the final segment
XMat = [XMat; XMatk(end,:)];
tVecOut = [tVecOut; tVeck(end,:)];
% Updating N
N = length(tVecOut);
% Creating P Structure
P.tVec = tVecOut; % size N x 1?
rMatrix = zeros(3, N);
eMatrix = zeros(3, N);
vMatrix = zeros(3, N);
omegaBmatrix = zeros(3, N);

for i = 1:N % iterations on time so matrix should look like 18 x X
   Xi = XMat';
   rMatrix(1:end,i) = [Xi(1,i), Xi(2,i), Xi(3,i)]';
   vMatrix(1:end,i) = [Xi(4,i), Xi(5,i), Xi(6,i)]';
   RBIk = [Xi(7,i) -Xi(8,i) Xi(9,i);
           Xi(10,i) Xi(11,i) Xi(12,i);
           Xi(13,i) Xi(14,i) Xi(15,i)]';
   ek = dcm2euler(RBIk); % transform back to euler angles for output
   ek(3) = -ek(3); % accounting for negative error
   eMatrix(1:end,i) = ek;
   omegaBmatrix(1:end,i) = [Xi(16,i) Xi(17,i) Xi(18,i)]';
end
```

```
outputState.rMat = rMatrix';
outputState.eMat = eMatrix';
outputState.vMat = vMatrix';
outputState.omegaBMat = omegaBmatrix';
P.state = outputState;
```

Figure 6: Quadcopter high fidelity simulation function

```
Unset
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.
%
%        rIstar = Nx3 matrix of desired CM positions in the I frame, in
%                 meters.  rIstar(k,:)' is the 3x1 position at time tk =
%                 tVec(k).
%
%        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).
%
%        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).
%
%        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).
%
% 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.
%
```

```
%        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
%
%                   e = 3x1 vector of Euler angles, in radians, indicating the
%                       attitude
%
%                   v = 3x1 velocity with respect to the world frame and
%                       expressed in the world frame, in meters per second.
%
%               omegaB = 3x1 angular rate vector expressed in the body frame,
%                        in radians per second.
%
%       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).
%
% 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).
%
%+-------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+===============================================================================+
N = length(R.tVec);
dtIn = R.tVec(2) - R.tVec(1);
dtOut = dtIn/S.oversampFact;
RBIk = euler2dcm(S.state0.e);
% Initial angular rates in rad/s, initialize here as per lab document
S.state0.omegaVec = [0 0 0 0]';
Xk = [S.state0.r;S.state0.v;RBIk(:);S.state0.omegaB;S.state0.omegaVec];
Pa.quadParams = P.quadParams;
Pa.constants = P.constants;
Pa.sensorParams = P.sensorParams;
XMat = []; tVec = [];
% Initialize Sk values
Sk.statek.RBI = euler2dcm(S.state0.e); % Initial e to RBI for function input
Sk.statek.rI = S.state0.r;
Sk.statek.vI = S.state0.v;
Sk.statek.omegaB = S.state0.omegaB;
Sk.statek.omegaVec = S.state0.omegaVec;
for kk=1:N-1
 tspan = [R.tVec(kk):dtOut:R.tVec(kk+1)]';
 distVeck = S.distMat(kk,:)';
 % Get Rk values, P structure is constant
 Rk.tVec = R.tVec(kk);
 Rk.rIstark = R.rIstar(:,kk);
 Rk.vIstark = R.vIstar(:,kk);
 Rk.aIstark = R.aIstar(:,kk);
 Rk.xIstark = R.xIstar(:,kk);
 [Fk, zIstark] = trajectoryController(Rk,Sk,P); % Where S structure input needs to
be updated after initial state
 Rk.zIstark = zIstark;
```

```
 [NBk] = attitudeController(Rk,Sk,P);
 [eak] = voltageConverter(Fk,NBk,P);
 eaVeck = eak;
 [tVeck,XMatk] = ...
     ode45(@(t,X) quadOdeFunctionHF(t,X,eaVeck,distVeck,Pa),tspan,Xk); % Think
through omegaVec input between two simulators
 % Update Sk values after Quadrotor Function
 XstateNow = XMatk(end,:);
 Sk.statek.rI = XstateNow(1:3)';
 Sk.statek.vI = XstateNow(4:6)';
 Sk.statek.omegaB = XstateNow(16:18)';
 Sk.statek.RBI = [XstateNow(7) XstateNow(10) XstateNow(13); % 3 x 3
                  XstateNow(8) XstateNow(11) XstateNow(14);
                  XstateNow(9) XstateNow(12) XstateNow(15)];
 % Can also add omegaVec here but not needed since already stored in XMat
 % and not needed elsewhere, since its not part of state.
 if(length(tspan) == 2)
   tVec = [tVec; tVeck(1)];
   XMat = [XMat; XMatk(1,:)];
 else
   tVec = [tVec; tVeck(1:end-1)];
   XMat = [XMat; XMatk(1:end-1,:)];
 end
 Xk = XMatk(end,:)';
 if(mod(kk,10) == 0)
   RBIk(:) = Xk(7:15);
   [UR,SR,VR]=svd(RBIk);
   RBIk = UR*VR'; Xk(7:15) = RBIk(:);
 end
end
XMat = [XMat;XMatk(end,:)];
tVec = [tVec;tVeck(end,:)];
M = length(tVec);
Q.tVec = tVec;
Q.state.rMat = XMat(:,1:3);
Q.state.vMat = XMat(:,4:6);
Q.state.omegaBMat = XMat(:,16:18);
% Q.state.omegaVec = XMat(:,19:22); % OmegaVec not included in state but
% recorded in QuadODE, so can optionally include in output.
Q.state.eMat = zeros(M,3);
RBI = zeros(3,3);
for mm=1:M
 RBI(:) = XMat(mm,7:15);
 Q.state.eMat(mm,:) = dcm2euler(RBI)';
end
```

Figure 7: Quadcopter control simulation function

Unset

```
clear; clc;
% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
sensorParamsScript;
P.quadParams = quadParams;
P.constants = constants;
P.sensorParams = sensorParams;
%% Initializing R
% Total simulation time, in seconds
Tsim = 10;
% Update interval, in seconds.  Currently, 200 Hz.
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec = [0:N-1]'*delt;
R.tVec = tVec';
% Nx3 matrix of desired CM positions in the I frame, in meters.
% rIstar(k,:)' is the 3x1 position at time tk = tVec(k).
R.rIstar = [];
% 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).
R.xIstar = [];
% 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).
R.vIstar = [];
% 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).
R.aIstar = [];
for kk=1:N-1
   t = tVec(kk);
   rIstar = [4*cos(pi+(t*(pi/5))), 4*sin(t*(pi/5)), 1]';
   xIstar = -(rIstar)/norm(rIstar);
   vIstar = (4*pi/5)*cross([0;0;1], xIstar);
   aIstar = (((4*pi/5)^2)/4)*xIstar;
   R.rIstar = [R.rIstar, rIstar];
   R.xIstar = [R.xIstar, xIstar];
   R.vIstar = [R.vIstar, vIstar];
   R.aIstar = [R.aIstar, aIstar];
end
%% Initializing S
% 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 = [-4 0 0]';
% 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;
%% Run and Visualize
Q = simulateQuadrotorControl(R,S,P);
S2.tVec = Q.tVec;
S2.rMat = Q.state.rMat;
S2.eMat = Q.state.eMat;
S2.plotFrequency = 20;
S2.makeGifFlag = false;
S2.gifFileName = 'testGif.gif';
S2.bounds=1*[-5 5 -5 5 -5 5];
visualizeQuad(S2);
figure(1);clf;
plot(Q.tVec,Q.state.rMat(:,3)); 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));
axis equal; grid on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');
```

Figure 7: Top-level simulation script

```
Unset
function P = visualizeQuad(S)
% visualizeQuad : Takes in an input structure S and visualizes the resulting
%                 3D motion in approximately real-time.  Outputs the data
%                 used to form the plot.
%
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%           rMat = 3xM matrix of quad positions, in meters
```

```
%
%           eMat = 3xM matrix of quad attitudes, in radians
%
%           tVec = Mx1 vector of times corresponding to each measurement in
%                  xevwMat
%
%  plotFrequency = The scalar number of frames of the plot per each second of
%                  input data.  Expressed in Hz.
%
%         bounds = 6x1, the 3d axis size vector
%
%    makeGifFlag = Boolean (if true, export the current plot to a .gif)
%
%    gifFileName = A string with the file name of the .gif if one is to be
%                  created.  Make sure to include the .gif exentsion.
%
%
% OUTPUTS
%
% P ---------- Structure with the following elements:
%
%          tPlot = Nx1 vector of time points used in the plot, sampled based
%                  on the frequency of plotFrequency
%
%          rPlot = 3xN vector of positions used to generate the plot, in
%                  meters.
%
%          ePlot = 3xN vector of attitudes used to generate the plot, in
%                  radians.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:  Nick Montalbano
%+==============================================================================+
 % Important params
figureNumber = 42; figure(figureNumber); clf;
fcounter = 0; %frame counter for gif maker
m = length(S.tVec);
% UT colors
burntOrangeUT = [191, 87, 0]/255;
darkGrayUT = [51, 63, 72]/255;
% Parameters for the rotors
rotorLocations=[0.105 0.105 -0.105 -0.105
   0.105 -0.105 0.105 -0.105
   0 0 0 0];
r_rotor = .062;
```

```
% Determines the location of the corners of the body box in the body frame,
% in meters
bpts=[ 120  120 -120 -120  120  120 -120 -120
   28  -28   28  -28   28  -28   28  -28
   20   20   20   20   -30   -30   -30   -30 ]*1e-3;
% Rectangles representing each side of the body box
b1 = [bpts(:,1) bpts(:,5) bpts(:,6) bpts(:,2) ];
b2 = [bpts(:,1) bpts(:,5) bpts(:,7) bpts(:,3) ];
b3 = [bpts(:,3) bpts(:,7) bpts(:,8) bpts(:,4) ];
b4 = [bpts(:,1) bpts(:,3) bpts(:,4) bpts(:,2) ];
b5 = [bpts(:,5) bpts(:,7) bpts(:,8) bpts(:,6) ];
b6 = [bpts(:,2) bpts(:,6) bpts(:,8) bpts(:,4) ];
% Create a circle for each rotor
t_circ=linspace(0,2*pi,20);
circpts=zeros(3,20);
for i=1:20
   circpts(:,i)=r_rotor*[cos(t_circ(i));sin(t_circ(i));0];
end
% Plot single epoch if m==1
if m==1
   figure(figureNumber);

   % Extract params
   RIB = euler2dcm(S.eMat(1:3))';
   r = S.rMat(1:3);

   % Translate, rotate, and plot the rotors
   hold on
   view(3)
   rotor1_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,1)*ones(1,20));
   rotor1plot = plot3(rotor1_circle(1,:), rotor1_circle(2,:),
rotor1_circle(3,:),...
       'color',darkGrayUT);
   hold on
   rotor2_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,2)*ones(1,20));
   rotor2plot = plot3(rotor2_circle(1,:), rotor2_circle(2,:),
rotor2_circle(3,:),...
       'color',darkGrayUT);
   hold on
   rotor3_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,3)*ones(1,20));
   rotor3plot = plot3(rotor3_circle(1,:), rotor3_circle(2,:),
rotor3_circle(3,:),...
       'color',darkGrayUT);
   hold on
   rotor4_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,4)*ones(1,20));
   rotor4plot = plot3(rotor4_circle(1,:), rotor4_circle(2,:),
rotor4_circle(3,:),...
       'color',darkGrayUT);
```

```
    % Plot the body
    b1r=r*ones(1,4)+RIB*b1; b2r=r*ones(1,4)+RIB*b2; b3r=r*ones(1,4)+RIB*b3;
    b4r=r*ones(1,4)+RIB*b4; b5r=r*ones(1,4)+RIB*b5; b6r=r*ones(1,4)+RIB*b6;
    X = [b1r(1,:)' b2r(1,:)' b3r(1,:)' b4r(1,:)' b5r(1,:)' b6r(1,:)'];
    Y = [b1r(2,:)' b2r(2,:)' b3r(2,:)' b4r(2,:)' b5r(2,:)' b6r(2,:)'];
    Z = [b1r(3,:)' b2r(3,:)' b3r(3,:)' b4r(3,:)' b5r(3,:)' b6r(3,:)'];
    hold on
    bodyplot=patch(X,Y,Z,[.5 .5 .5]);

    % Plot the body axes
    bodyX=0.5*RIB*[1;0;0]; bodyY=0.5*RIB*[0;1;0]; bodyZ=0.5*RIB*[0;0;1];
    hold on
    axis1 = quiver3(r(1),r(2),r(3),bodyX(1),bodyX(2),bodyX(3),'red');
    hold on
    axis2 = quiver3(r(1),r(2),r(3),bodyY(1),bodyY(2),bodyY(3),'blue');
    hold on
    axis3 = quiver3(r(1),r(2),r(3),bodyZ(1),bodyZ(2),bodyZ(3),'green');
    axis(S.bounds)
    xlabel('X')
    ylabel('Y')
    zlabel('Z')
    grid on

    P.tPlot = S.tVec;
    P.rPlot = S.rMat;
    P.ePlot = S.eMat;

elseif m>1 % Interpolation must be used to smooth timing

    % Create time vectors
    tf = 1/S.plotFrequency;
    tmax = S.tVec(m); tmin = S.tVec(1);
    tPlot = tmin:tf:tmax;
    tPlotLen = length(tPlot);

    % Interpolate to regularize times
    [t2unique, indUnique] = unique(S.tVec);
    rPlot = (interp1(t2unique, S.rMat(indUnique,:), tPlot))';
    ePlot = (interp1(t2unique, S.eMat(indUnique,:), tPlot))';

    figure(figureNumber);

    % Iterate through points
    for i=1:tPlotLen

        % Start timer
        tic
```

```
        % Extract data
        RIB = euler2dcm(ePlot(1:3,i))';
        r = rPlot(1:3,i);

        % Translate, rotate, and plot the rotors
        hold on
        view(3)

rotor1_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,1)*ones(1,20));
        rotor1plot = plot3(rotor1_circle(1,:), rotor1_circle(2,:),...
            rotor1_circle(3,:), 'color', darkGrayUT);
        hold on

rotor2_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,2)*ones(1,20));
        rotor2plot = plot3(rotor2_circle(1,:), rotor2_circle(2,:),...
            rotor2_circle(3,:), 'color', darkGrayUT);
        hold on

rotor3_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,3)*ones(1,20));
        rotor3plot = plot3(rotor3_circle(1,:), rotor3_circle(2,:),...
            rotor3_circle(3,:), 'color', darkGrayUT);
        hold on

rotor4_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,4)*ones(1,20));
        rotor4plot = plot3(rotor4_circle(1,:), rotor4_circle(2,:),...
            rotor4_circle(3,:), 'color', darkGrayUT);

        % Translate, rotate, and plot the body
        b1r=r*ones(1,4)+RIB*b1; b2r=r*ones(1,4)+RIB*b2; b3r=r*ones(1,4)+RIB*b3;
        b4r=r*ones(1,4)+RIB*b4; b5r=r*ones(1,4)+RIB*b5; b6r=r*ones(1,4)+RIB*b6;
        X = [b1r(1,:)' b2r(1,:)' b3r(1,:)' b4r(1,:)' b5r(1,:)' b6r(1,:)'];
        Y = [b1r(2,:)' b2r(2,:)' b3r(2,:)' b4r(2,:)' b5r(2,:)' b6r(2,:)'];
        Z = [b1r(3,:)' b2r(3,:)' b3r(3,:)' b4r(3,:)' b5r(3,:)' b6r(3,:)'];
        hold on
        bodyplot=patch(X,Y,Z,[.5 .5 .5]);

        % Translate, rotate, and plot body axes
        bodyX=0.5*RIB*[1;0;0]; bodyY=0.5*RIB*[0;1;0]; bodyZ=0.5*RIB*[0;0;1];
        hold on
        axis1 = quiver3(r(1),r(2),r(3),bodyX(1),bodyX(2),bodyX(3),'red');
        hold on
        axis2 = quiver3(r(1),r(2),r(3),bodyY(1),bodyY(2),bodyY(3),'blue');
        hold on
        axis3 = quiver3(r(1),r(2),r(3),bodyZ(1),bodyZ(2),bodyZ(3),'green');
        % Fix up plot style
        axis(S.bounds)
        xlabel('X')
        ylabel('Y')
```

```
        zlabel('Z')
        grid on

        tP=toc;
        % Pause to stay near-real-time
        pause(max(0.001,tf-tP))

        % Gif stuff
        if S.makeGifFlag
            fcounter=fcounter+1;
            frame=getframe(figureNumber);
            im=frame2im(frame);
            [imind,cm]=rgb2ind(im,256);
            if fcounter==1
                imwrite(imind,cm,S.gifFileName,'gif','Loopcount',inf,...
                    'DelayTime',tf);
            else
                imwrite(imind,cm,S.gifFileName,'gif','WriteMode','append',...
                    'DelayTime',tf);
            end
        end

        % Clear plot before next iteration, unless at final time step
        if i<tPlotLen
            delete(rotor1plot)
            delete(rotor2plot)
            delete(rotor3plot)
            delete(rotor4plot)
            delete(bodyplot)
            delete(axis1)
            delete(axis2)
            delete(axis3)
        end
    end

    P.tPlot = tPlot;
    P.ePlot = ePlot;
    P.rPlot = rPlot;
end
end
```

Figure 8: Simulation visualization function

```
Unset
quadParamsScript;
```

```
constantsScript;
%% Part 3b
Cm = quadParams.cm(1);
Tm = quadParams.taum(1);
motor = tf([Cm],[Tm 1]);
subplot(2,1,1)
% Plot of transfer function for step input
step(motor*4)
title('Direct-Current Motor Model Transfer Function Step Input')
ylabel('w(t) (rad/sec)')
xlabel('time (s)')
% Plot of ODE  for step input, t=0 and w=0
w_dot = @(t, y) ((Cm/(Tm))*(exp(-t/Tm)));
[t, w] = ode45(w_dot,[0,0.5],0);
subplot(2,1,2)
plot(t,w)
title('Direct-Current Motor Model ODE Solution Step Input')
ylabel('w(t) (rad/sec)')
xlabel('time (s)')
%% Part 4
% Test old and new simulation models
plot(out.tout, out.yout)
title('PD Control System Unit Step Response')
ylabel('Amplitude')
xlabel('Time')
stepinfo(out.yout, out.tout)
```

Figure 9: Top-level simulator script for attempted circle control state

```
Unset
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.
%
%       xIstark = 3x1 desired body x-axis direction, expressed as a
%                 unit vector in the I frame.
%
% 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
%
%                  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.
%
% 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
%
%
% OUTPUTS
%
% NBk -------- Commanded 3x1 torque expressed in the body frame at time tk, in
%              N-m.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+
J = P.quadParams.Jq;
RBI = S.statek.RBI;
wB = S.statek.omegaB;
% Small angle assumption
eE_dot = wB;
zIstark = R.zIstark;
xIstark = R.xIstark;
K = [1 0 0; 0 1 0; 0 0 1]; % Parameters to tune
Kd = [0.3 0 0; 0 0.3 0; 0 0 0.3];
b = cross(zIstark, xIstark)/(norm(cross(zIstark, xIstark))); % 3x1
a = cross(b,zIstark); % 3x1
RBIstark = [a, b, zIstark]'; % 3x3
RE = RBIstark*(RBI'); % Double check if element by element multiplication is
```

```
needed here
eE = [RE(2,3)-RE(3,2), RE(3,1)-RE(1,3), RE(1,2)-RE(2,1)]'; % 3x1
NBk = K*eE - Kd*eE_dot + crossProductEquivalent(wB)*J*wB;
```

Figure 10: Attitude controller function

Unset

```
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.
%
%       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.
%
%       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.
%
% 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
%
%                  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.
%
% 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
%
%
% OUTPUTS
%
% Fk --------- Commanded total thrust at time tk, in Newtons.
%
% zIstark ---- Desired 3x1 body z axis expressed in I frame at time tk.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+
m = P.quadParams.m;
g = P.constants.g;
rIstark = R.rIstark;
vIstark = R.vIstark;
aIstark = R.aIstark;
rI = S.statek.rI;
RBI = S.statek.RBI;
vI = S.statek.vI;
er = rIstark - rI;
er_dot = vIstark - vI;
K = [4 0 0; 0 4 0; 0 0 4]; % Parameters to tune
Kd = [1.5 0 0; 0 1.5 0; 0 0 1.5]; % Both 3x3
FIstark = K*er + Kd*er_dot + [0; 0; m*g] + m*aIstark;
zIstark = FIstark./norm(FIstark);
e3 = [0 0 1]';
zI = (RBI')*e3;
Fk = (FIstark')*zI;
```

Figure 11: Trajectory controller function

Unset

```
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
%
%     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.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+
locations = P.quadParams.rotor_loc;
kF = P.quadParams.kF(1);
kN = P.quadParams.kN(1);
kT = kN/kF;
Cm = P.quadParams.cm(1);
eaMax = P.quadParams.eamax;
beta = 0.9; % constant
alpha = 1; % can reduce
% Finding max force value
wmax = Cm*eaMax;
Fmax = kF*(wmax^2);
FmaxTotal = Fmax*4*beta;
extraVecOne = [FmaxTotal, Fk];
G = [1, 1, 1, 1;
    locations(2,1), locations(2,2), locations(2,3), locations(2,4);
    -locations(1,1), -locations(1,2), -locations(1,3), -locations(1,4);
    -kT, kT, -kT, kT];
% Expressed by min force and torque vector
extraVec = [min(extraVecOne), alpha*NBk(1), alpha*NBk(2), alpha*NBk(3)]';
Fvec = (G^(-1))*extraVec;
% changing alpha to satisfy Fi <= Fmax
while max(all(Fvec > Fmax)) % if any element is over Fmax return a 1 to indicate
```

```
true, if all are false, max is a 0 or false.
   alpha = alpha - 0.05;
   extraVec = [min(extraVecOne), alpha*NBk(1), alpha*NBk(2), alpha*NBk(3)]';
   Fvec = (G^(-1))*extraVec; % recalculate Fvec to check again, if passes, this is
new value
end
% Check if motor force values are below zero and fix
for F = 1:4
   if Fvec(F) < 0
       Fvec(F) = 0;
   end
end
omegaVec = ((1/kF)*Fvec).^(0.5);
eak = (1/Cm)*omegaVec;
```

Figure 12: Controller to voltage converter function