ASE 479W Laboratory 3 Report
Measurement Simulation and State Estimation

Aaron Pandian

March 4, 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). Implementing this model with feedback control allows for a scalable approach to achieving target motion. The paper *Direct Acceleration Feedback Control of Quadrotor Aerial Vehicles* states this method “alleviates the need for accurate estimation of platform parameters.” One key aspect of this system is the measurement of the UAV state using sensors. These measurements are used to calculate the reference position- which is to be minimized in error from the provided target trajectory.

For this laboratory assignment, a high fidelity quadrotor simulator was designed from the theory of Newtonian dynamics and applied in Matlab. To ease model control, a feedback system was designed and experimented to employ a specific circular maneuver. Furthermore, three sensors were modeled: an inertial measurement unit (IMU), a global navigation satellite system (GNSS) receiver, and a camera. These sensors were modeled with noisy outputs for realism, and the resulting state was estimated using these measurements. This state was then replaced by the true state determined by the high fidelity model itself. 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 Rank of Baseline Covariance Matrix

In modeling the GNSS measurements, to measure the secondary antenna position relative to the primary antenna, the baseline vector $\boldsymbol{r}_{bG}$ is denoted.

$$\boldsymbol{r}_{b\mathrm{G}} = \boldsymbol{r}_{s\mathrm{G}} - \boldsymbol{r}_{p\mathrm{G}} \tag{1}$$

$$\tilde{\boldsymbol{r}}_{b\mathrm{G}}(t_k) = \boldsymbol{r}_{b\mathrm{G}}(t_k) + \boldsymbol{w}_{b\mathrm{G}}(t_k) \tag{2}$$

The baseline vector derived from (1) is altered along time t_k with some noise $\boldsymbol{\omega}_{\mathrm{bG}}$ in (2). For this experiment $\boldsymbol{\omega}_{\mathrm{bG}}$ is a white discrete-time Gaussian noise process that we model statistically as follows.

$$\mathbb{E}\left[\boldsymbol{w}_{b\mathrm{G}}(t_k)\right] = 0, \quad \mathbb{E}\left[\boldsymbol{w}_{b\mathrm{G}}(t_k)\boldsymbol{w}_{b\mathrm{G}}^{\mathsf{T}}(t_j)\right] = R_{b\mathrm{G}}\delta_{kj} \tag{3}$$

This noise model implies that the noise on the GNSS measurement has zero mean, a covariance matrix of R_{bG}, and is uncorrelated in time.

$$\|\tilde{\boldsymbol{r}}_{b\mathrm{G}}\| = \|\boldsymbol{r}_{b\mathrm{G}}\| \tag{4}$$

Due to the fact that the length of $\tilde{\boldsymbol{r}}_{bG}$ is known, the constraint in (4) is presented. As a result, the calculation of the covariance matrix R_{bG} used to derive the noise $\boldsymbol{\omega}_{\mathrm{bG}}$ is uniquely conducted in (5).

$$R_{b\mathrm{G}} = \|\boldsymbol{r}_{b\mathrm{G}}\|^2\sigma_b^2[I_{3\times 3} - \boldsymbol{r}_{b\mathrm{G}}^u(\boldsymbol{r}_{b\mathrm{G}}^u)^{\mathsf{T}}] \tag{5}$$

The σ_{b} represents the known standard deviation of the angular error (in radians) of the baseline vector measurement. Due to the constraint presented in (4) the matrix R_{bG} is rank 2. This can be shown by symbolizing the values above and making some assumptions.

$$R_{bG} = A^2B^2\left(\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} - \begin{bmatrix}x\\ y\\ z\end{bmatrix}[xyz]\right) \tag{6}$$

Setting up (5) as (6) the values of the unit vector $\boldsymbol{r}_{bG}$ denoted as $\boldsymbol{r}^u{}_{bG}$ are important. Since the length of the $\tilde{\boldsymbol{r}}_{bG}$ is fixed, that rules out one degree of freedom the value can vary. To depict this, any term from the $\boldsymbol{r}^u{}_{bG}$ vectors in (5) can be equated to zero, for this example it was the x term. This resulted in (6.1) where y, z are dependant non-zero values in the unit vector.

$$R_{bG} = \begin{bmatrix}1 & 0 & 0\\ 0 & 1-Y^2 & -YZ\\ 0 & -YZ & 1-Z^2\end{bmatrix}A^2B^2 \tag{6.1}$$

Now, using the minor method, it is possible to determine the rank of the system. If det(R_{bG}) ≠ 0, then the rank of R_{bG} is the order of the matrix, which is 3, if not- further calculation will continue. Calculating the determinant leaves $A^2B^2(1-Y^2-Z^2)$. However, remembering that $\boldsymbol{r}^u{}_{bG}$ is a unit vector, the Y^2 and Z^2 must sum to one. This leaves $\det(R_{bG}) = 0$, meaning that the rank of $R_{bG} \neq 3$. To find the rank, calculations were conducted to find any non-zero matrix minor of order two. In taking the top right section of the matrix in (6.1), neglecting the A and B terms, the following assessment was conducted.

$$\det \begin{bmatrix} 1 & 0 \\ 0 & 1-Y^2 \end{bmatrix} = 1 - Y^2 \neq 0 \tag{6.2}$$

Where it is found that the determinant of the minor is a non-zero value, following the assumption that z was non-zero, so y cannot equal one. As a result, it is proven that the rank of matrix RbG is 2.

2.2 Homogeneous Coordinates Cross Product of Lines

In simulating a camera and projecting detected features from a 3D plane to a 2D plane, we use the concept of homogeneous coordinates. Any 2-D point $[x, y]^T$ can be represented in homogeneous coordinates as $\boldsymbol{x} = [\gamma x, \gamma y, \gamma]^T$ for any $\gamma \neq 0$. Dividing out the third coordinate of a homogeneous point $[x, y, z]^T$ converts it back to its 2D equivalent $[x/z, y/z]^T$.

Similarly, a line represented as $\boldsymbol{l} = [a, b, c]^T$, can be expanded as the following.

$$ax + by + c = 0 \tag{7}$$

It is known that the point defined by the intersection of two lines $\boldsymbol{l}_1$ and $\boldsymbol{l}_2$ can be found as $\boldsymbol{x} = \boldsymbol{l}_1 \times \boldsymbol{l}_2$. Starting with the homogeneous coordinate $\boldsymbol{x} = [\gamma x, \gamma y, \gamma]^T$ where $\gamma = 1$, we arrive at $\boldsymbol{x} = [x, y, 1]^T$. Concurrently, taking the cross product of $\boldsymbol{l}_1$ and $\boldsymbol{l}_2$ is shown in the following calculation.

$$\begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} \times \begin{bmatrix} a_2 \\ b_2 \\ c_2 \end{bmatrix} = \begin{bmatrix} b_1c_2 - c_1b_2 \\ c_1a_2 - a_1c_2 \\ a_1b_2 - a_2b_1 \end{bmatrix} \tag{7.1}$$

This results in a 3D vector, or a set of homogeneous coordinates. This can be converted to its 2D counterpart as mentioned previously by dividing out the third coordinate.

$$\boldsymbol{x} = [\frac{b_1c_2 - b_2c_1}{a_1b_2 - a_2b_1}, \frac{c_1a_2 - c_2a_1}{a_1b_2 - a_2b_1}, 1]^T \tag{7.2}$$

To represent the homogenous point in the 2D plane we can remove the third coordinate entirely from (7.2). This can be shown to be the intersection point of two lines using an example. Consider the two lines.

$$\boldsymbol{l}_1\textbf{:}\ x + y - 5 = 0 \qquad (7.3)$$

$$\boldsymbol{l}_2\text{: } 4x - 5y + 7 = 0 \qquad (7.4)$$

A plot generated from the two lines on the same plane is shown below.

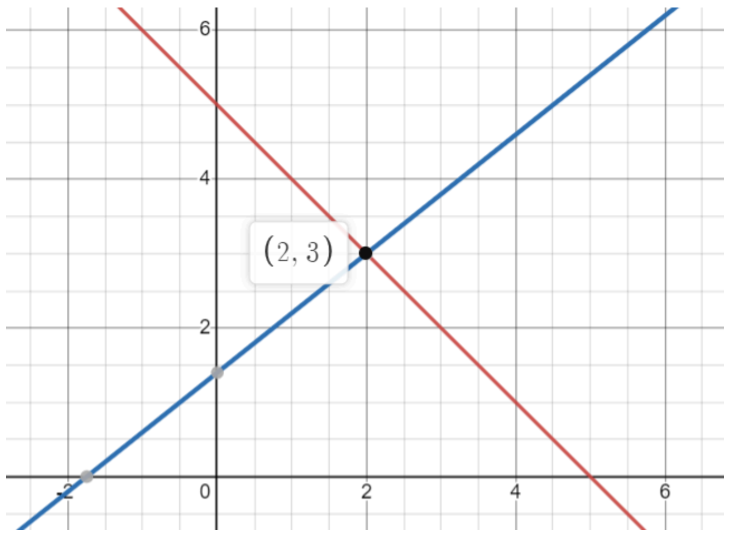


Figure 1: A plot of lines $\boldsymbol{l}_1$ (red) and $\boldsymbol{l}_2$ (blue)

Through graphical representation the intersection point in Fig. 1 can be denoted as $\boldsymbol{x} = [2,\ 3,\ 1]^T$. However, to show that the cross product of the two lines can produce the same answer, the calculation in (7.5) is conducted.

$$\begin{bmatrix} 1 \\ 1 \\ -5 \end{bmatrix} \times \begin{bmatrix} 4 \\ -5 \\ 7 \end{bmatrix} = \begin{bmatrix} -18 \\ -27 \\ -9 \end{bmatrix} \qquad (7.5)$$

By dividing out the third coordinate, the resultant is equivalent to (7.2) in process. The homogeneous coordinate point produced is $\boldsymbol{x} = [2,\ 3,\ 1]^T$, exactly the point found prior.

2.3 Inertial Acceleration Derivation

The IMU accelerometer is capable of measuring the experienced acceleration of the UAV.

$$\tilde{\boldsymbol{f}}_{\mathrm{B}} = R_{\mathrm{BI}}\left(\boldsymbol{a}_{\mathrm{I}} + g\boldsymbol{e}_3\right) + \boldsymbol{b}_a + \boldsymbol{v}_a \qquad (8)$$

The accelerometer measurement model can be seen in (8) to derive the specific force. The bias $\boldsymbol{b}_a$ and noise $\boldsymbol{v}_a$ terms are calculated alongside the quadrotor acceleration in the I frame $\boldsymbol{a}_I$. However, this measurement model assumes that the accelerometers axes are in line with the body frame axes, and that the local east north up (ENU) frame is inertial. For the accelerometers axes to be in line with the body frame axes, the vector pointing from the CM to the accelerometer proof mass $\boldsymbol{l}_B$ must be zero.

$$\boldsymbol{a}_\mathrm{I} = \ddot{\boldsymbol{r}}_\mathrm{I} = \ddot{\boldsymbol{R}}_\mathrm{I} \tag{9}$$

Assuming $\boldsymbol{l}_B = 0$, the acceleration of a point fixed in the B frame with respect to the I (ENU) frame is related to R_I, or the vector pointing from the I to the B frame, through (8). In the succeeding models this is assumed for simplicity. However, in reality this is not the case. To derive $\boldsymbol{a}_I$ located at some non-zero, constant $\boldsymbol{l}_B$, equation (10) can be exercised.

$$C\ddot{\boldsymbol{r}}_\mathrm{I} = \underbrace{C\ddot{\boldsymbol{R}}_\mathrm{I}}_{\text{accel. of B origin}} + \underbrace{\ddot{\boldsymbol{l}}_\mathrm{B}}_{\text{rel. accel.}} + \underbrace{(\dot{\boldsymbol{\omega}}_\mathrm{B} \times \boldsymbol{l}_\mathrm{B})}_{\text{Euler}} + \underbrace{2(\boldsymbol{\omega}_\mathrm{B} \times \dot{\boldsymbol{l}}_\mathrm{B})}_{\text{Coriolis}} + \underbrace{\boldsymbol{\omega}_\mathrm{B} \times (\boldsymbol{\omega}_\mathrm{B} \times \boldsymbol{l}_\mathrm{B})}_{\text{centripetal}} \tag{10}$$

Taken from the derivation of point P expressed in the I frame, with a distance of $\boldsymbol{l}$ from the B frame origin, as expressed by the figure below.

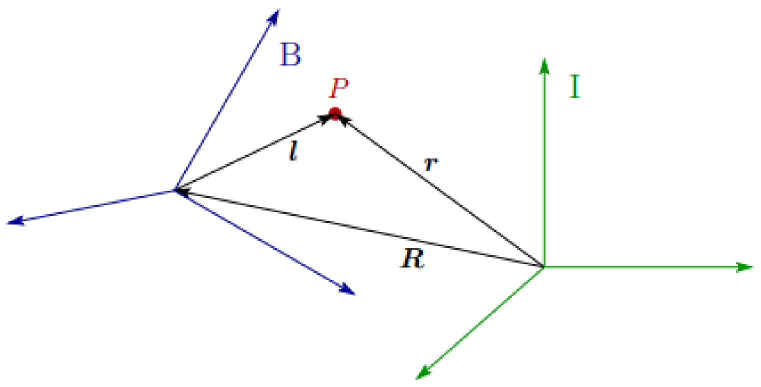


Figure 2: Point P with respect to both I and B frame

With (10), where C is rotation matrix from the I to B frame otherwise noted as R_{BI}, a derivation of the desired $\boldsymbol{a}_I$ can be obtained by making a few adjustments. Using an understanding of the system, first the relative acceleration term can be negated since, in this case, the value of $\boldsymbol{l}_B$ is fixed once the IMU is mounted on the quadrotor. Additionally, the Coriollos term from (10) can also be removed. The Coriollos effect is defined by an imposed force inherently existing in a frame that rotates. Due to the fact that $\boldsymbol{a}_I$ is in the inertial (non-rotating) frame, this effect does not need to be accounted for. As a result, the acceleration of a point fixed at $\boldsymbol{l}_\mathrm{B}$ in the B frame, with respect to I is shown in (11).

$$\boldsymbol{a}_\mathrm{I} = \ddot{\boldsymbol{R}}_\mathrm{I} + R_\mathrm{BI}^\mathsf{T}\left[\boldsymbol{\omega}_\mathrm{B} \times (\boldsymbol{\omega}_\mathrm{B} \times \boldsymbol{l}_\mathrm{B}) + (\dot{\boldsymbol{\omega}}_\mathrm{B} \times \boldsymbol{l}_\mathrm{B})\right] \tag{11}$$

In the equation above, only the euler and centripetal terms from (10) are represented.

2.4 Rate Gyro Measurement Model

For the rate gyro measurement model, the equation is presented below.

$$\tilde{\boldsymbol{\omega}}_\mathrm{B} = \boldsymbol{\omega}_\mathrm{B} + \boldsymbol{b}_g + \boldsymbol{v}_g \tag{12}$$

The $\boldsymbol{\omega}_\mathrm{B}$ above is the true angular rate in the body frame of the quadrotor obtained from the dynamics model, discussed in the next section. The gyro bias $\boldsymbol{b}_\mathrm{g}$ and noise vector $\boldsymbol{v}_\mathrm{g}$ are calculated for subsequently. Unlike the previous accelerometer measurement model, the lever arm $\boldsymbol{l}_\mathrm{B}$ need not be taken into account for (12), assuming both the accelerometer and the gyros are located at $\boldsymbol{l}_\mathrm{B}$. This is due to the fact that by definition angular rate already accounts for the distance away from the point of reference. For the quadrotor angular rate $\boldsymbol{\omega}_\mathrm{B}$ which represents the angular velocity of the body frame relative to the inertial frame, every point on the quadrotor has the same angular rate. At any $\boldsymbol{l}_\mathrm{B}$ for which the IMU sits on the quadcopter, the angular rate will be the same $\boldsymbol{\omega}_\mathrm{B}$ at that moment in time. For this reason, it is unnecessary to consider the lever arm here.

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 euler rotation theorem. 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 3: Cross product equivalent function

The application above outputs 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 4: 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.

By utilizing the script in Fig. 4, 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.

$$C(\psi, \phi, \theta) = R_2(\boldsymbol{e}_2, \theta)R_1(\boldsymbol{e}_1, \phi)R_3(\boldsymbol{e}_3, \psi) \tag{13}$$

$$\boldsymbol{e}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \boldsymbol{e}_2 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \boldsymbol{e}_3 = \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{13.1}$$

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.

```
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 5: Euler angles to rotation matrix

The formula in Fig. 5 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. 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 6: Rotation matrix to euler angles

The function in Fig. 6 does the inverse operation from Fig. 5, 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. 7.

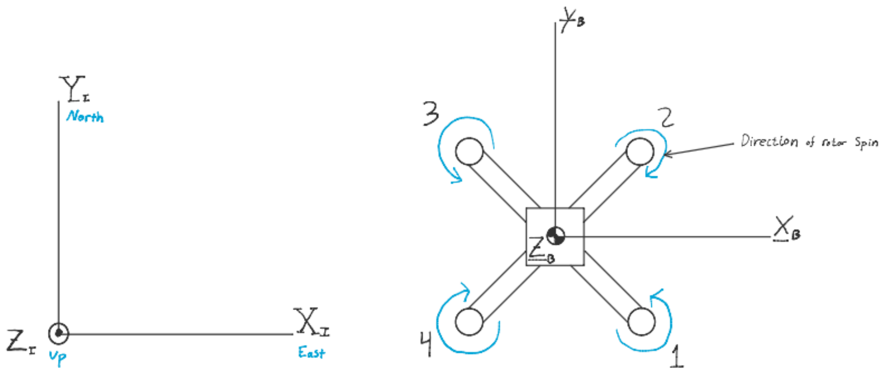


Figure 5: Quadcopter visualization

First it is important to understand the definition of state of the quadrotor, seen in (14).

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{r}_\mathrm{I} \\ \boldsymbol{e} \\ \boldsymbol{v}_\mathrm{I} \\ \boldsymbol{\omega}_\mathrm{B} \end{bmatrix} \tag{14}$$

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 Ẋ, the following equations will need to be computed for each state input.

$$\dot{\boldsymbol{r}}_{\mathrm{I}} = \boldsymbol{v}_{\mathrm{I}} \tag{15}$$

$$m\dot{\boldsymbol{v}}_{\mathrm{I}} = m\ddot{\boldsymbol{r}}_{\mathrm{I}} = -mg\boldsymbol{z}_{\mathrm{I}} + \sum_{i=1}^{4} \boldsymbol{F}_{i\mathrm{I}} + \boldsymbol{d}_{\mathrm{I}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_{\mathrm{BI}}^{\mathrm{T}} \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ F_i \end{bmatrix} + \boldsymbol{d}_{\mathrm{I}} \tag{16}$$

$$\dot{\boldsymbol{\omega}}_{\mathrm{B}} = J^{-1}\left(\boldsymbol{N}_{\mathrm{B}} - [\boldsymbol{\omega}_{\mathrm{B}}\times]J\boldsymbol{\omega}_{\mathrm{B}}\right) \quad \text{where} \quad \boldsymbol{N}_{\mathrm{B}} = \sum_{i=1}^{4}\left(\boldsymbol{N}_{i\mathrm{B}} + \boldsymbol{r}_{i\mathrm{B}} \times \boldsymbol{F}_{i\mathrm{B}}\right) \tag{17}$$

$$\dot{R}_{\mathrm{BI}} = -[\boldsymbol{\omega}_{\mathrm{B}}\times]R_{\mathrm{BI}} \tag{18}$$

Functions (15) and (18) are explicit in that (15) simply defines the rate of change of position as velocity and (18) 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.

$$m\ddot{\boldsymbol{r}}_{\mathrm{I}} = -d_a\boldsymbol{v}_{\mathrm{I}}^{u} - mg\boldsymbol{Z}_{\mathrm{I}} + \sum_{i=1}^{4} \boldsymbol{F}_{i\mathrm{I}} + \boldsymbol{d}_{\mathrm{I}} \tag{19}$$

The translation equation of motion (16) is updated to account for the drag force d_{a} in the direction opposite to velocity $-\boldsymbol{v}_{\mathrm{I}}^{\mathrm{u}}$. 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 (17) is defined because the euler angles from the initial state matrix will be transformed, using the function in Fig. 5, 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 $\dot{R}_\mathrm{BI}$ (unpacked column by column) to provide a now 22x1 state matrix. An overview of the process can be seen in Fig. 7.

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

% 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.
FMat = [zeros(2,4);(P.quadParams.kF.*(omegaVec.^2))'];
NMat = [zeros(2,4);(P.quadParams.kN.*(omegaVec.^2).*(-P.quadParams.omegaRdir)')'];

omegaBx = crossProductEquivalent(omegaB);
zI = RBI(3,:)';

% Calculate drag coefficient
epsilon_vI = 1e-3;
da = 0; vIu = [1;0;0];
if(norm(vI) > epsilon_vI)
 vIu = vI/norm(vI);
 fd = abs(zI'*vIu)*norm(vI)^2;
 da = 0.5*P.quadParams.Cd*P.quadParams.Ad*P.constants.rho*fd;
end

% Find derivatives of state elements
rIdot = vI;
vIdot = ([0;0;-mq*gE] + RBI'*sum(FMat,2) + distVec - da*vIu)/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);
omegaVecdot = (eaVec.*P.quadParams.cm - omegaVec)./P.quadParams.taum;

% Output state derivative vector
Xdot = [rIdot;vIdot;RBIdot(:);omegaBdot;omegaVecdot];
```

Figure 7: Quadrotor ordinary differential equation function

The output of this function is the derivative state vector $\dot{X}$, like (14) in construct, with the addition of the rotor angular velocity rate of change. This function provides the true state, which will be used to model the noisy sensor measurements that must be used to estimate the state. Though this system may seem counter-intuitive, the application of modeling noisy sensor measurements allows for a more realistic simulation of UAV flight. As a result, the entire system of true state calculation, sensor measurement modeling, and noisy measurement state estimation can be compiled into the quadrotor dynamics plant denoted in the feedback schematic below.

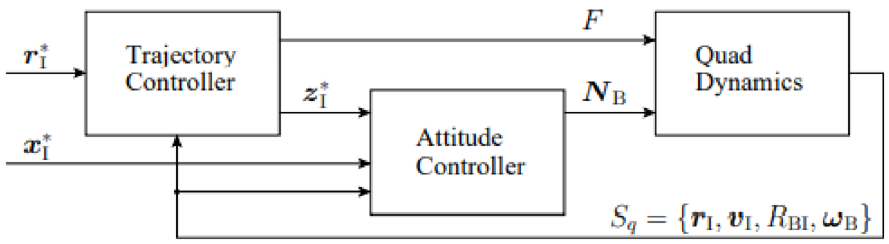


Figure 8: Top-level quadrotor control architecture

The next phase of implementation is to create the two control schemes seen in Fig. 8. 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).

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

This process is laid out in Fig. 9 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 9: Trajectory controller function

To create the attitude controller evident in Fig. 8, we instead substitute $-\omega_B$ for $\dot{e}_E$ because of the complexity of deriving $\dot{e}_E$ from the attitude controller inputs $\boldsymbol{z}_I^*$ and $\boldsymbol{x}_I^*$. 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) for the attitude controller.

```
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 10: Attitude control function

The final piece in the control loop is a hidden function from Fig 8. 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.

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

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.

With the control architecture developed, the primary alteration of this experiment can blossom. For ease of understanding, the model simulator is depicted. Given an arbitrary initial state structure, the Runge Kutta ordinary differential equation solver in Matlab allows for visualization of the quadrotor dynamics across a preset sample time. The basic implementation of the control system in Fig. 8 is presented in the Fig. 11 simulator function.

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


for kk=1:N-1
 % True state output
 statek.rI = Xk(1:3);
 statek.RBI(:) = Xk(7:15);
 statek.vI = Xk(4:6);
 statek.omegaB = Xk(16:18);
 statek.aI = Xdotk(4:6);
 statek.omegaBdot = Xdotk(16:18);
 Sm.statek = statek;

 % Simulate measurements
 M.tk=dtIn*(kk-1);
 [M.rpGtilde,M.rbGtilde] = gnssMeasSimulator(Sm,P);
 M.rxMat = [];
 for ii=1:Nf
   rx = hdCameraSimulator(S.rXIMat(ii,:)',Sm,P);
   if(isempty(rx))
     M.rxMat(ii,:) = [NaN,NaN];
   else
     M.rxMat(ii,:) = rx';
   end
 end
 [M.ftildeB,M.omegaBtilde] = imuSimulator(Sm,P);

 % Call estimator
 E = stateEstimatorUKF(Se,M,P);
 if(~isempty(E.statek))
   % Call trajectory and attitude controllers
   Rtc.rIstark = R.rIstar(kk,:)';
   Rtc.vIstark = R.vIstar(kk,:)';
   Rtc.aIstark = R.aIstar(kk,:)';
   Rac.xIstark = R.xIstar(kk,:)';
   distVeck = S.distMat(kk,:)';
   Sc.statek = E.statek;
   [Fk,Rac.zIstark] = trajectoryController(Rtc,Sc,P);
   NBk = attitudeController(Rac,Sc,P);
   % Convert commanded Fk and NBk to commanded voltages
   eaVeck = voltageConverter(Fk,NBk,P);
```

```
  else
    % Apply no control if the state estimator's output is empty.
    eaVeck = zeros(4,1);
    distVeck = [0;0;P.quadParams.m*P.constants.g];
  end

  tspan = [R.tVec(kk):dtOut:R.tVec(kk+1)]';
  [tVeck,XMatk] = ode45(@(t,X) quadOdeFunctionHF(t,X,eaVeck,distVeck,P),tspan,Xk);

  Xk = XMatk(end,:)';
  Xdotk = quadOdeFunctionHF(tVeck(end),Xk,eaVeck,distVeck,P);

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.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 11: Quadcopter dynamics 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. As evident from the simulation model, the true state using the function in Fig. 5 is determined. The true state generated, as mentioned before, is used to simulate noisy sensor measurements to *estimate* the quadrotor state. This estimated state is fed into the control loop, and using the reference trajectory, the trajectory controllers determine the next state to minimize the difference. Upon the next iteration, this state is fed into the high fidelity dynamics function to derive the next true state for which this process continues in the simulator above.

To make this system functional, the GNSS sensor model is depicted below.

```
Unset
function [rpGtilde,rbGtilde] = gnssMeasSimulator(S,P)

%% Primary
% Transform the inertial ECEF frame to the ENU frame, where vEnu = R*vEcef
```

```
RLG = Recef2enu(r0G);

% Find RpG
RpG = inv(RLG)*RpL*inv(RLG');

% Finding coordinates of primary antenna in I
rpI = rI + (RBI')*ra1B;

% Finding coordinates of primary antenna in G
rpG = (RLG')*rpI;

% Simulate noise vector
k = 1;
j = 1;
covarMatrix = RpG*eq(k,j);
w = mvnrnd(zeros(3,1), covarMatrix)';

% Finding rpGtilde
rpGtilde = rpG + w;

%% Baseline
% Finding coordinates of secondary antenna in I
rbI = rI + (RBI')*ra2B;

% Finding coordinates of secondary antenna in G
rbG = (RLG')*rbI;

% Find RbG
epsilon = 10^(-9);
rubG = rbG./norm(rbG);
RbG = ((norm(rbG)^2)*(sigmab^2)).*(eye(3) - (rubG*(rubG'))) + epsilon.*eye(3);

% Simulate noise vector
k = 1;
j = 1;
covarMatrix = RbG*eq(k,j);
w = mvnrnd(zeros(3,1), covarMatrix)';

% Finding rbGtilde with constraint norm(rbGtilde) = norm(rbG)
rbGtildeNoMag = rbG + w;
rbGtildeUnit = rbGtildeNoMag./norm(rbGtildeNoMag);
rbGtilde = norm(rbG).*rbGtildeUnit;
```

Figure 12: Noisy GNSS sensor measurement model

The responsibility of the GNSS model is to derive measurement values for the position of the primary and secondary antennas. The method for the primary and baseline measurement can be shown in (1) (2) and

(3). However, for the primary measurement, the covariance matrix R_{pG} is found using (30) instead of (4).

$$R_{pL} = R_{LG} R_{pG} R_{LG}^{T} \tag{30}$$

In solving for R_{pG} the value of R_{LG} is the known rotation matrix from G to the L frame, where G is the Earth Centered Earth Fixed (ECEF) frame and L is approximately the Inertial I frame; the value of R_{pL} is the known error covariance matrix for either primary or secondary antenna expressed in the L frame. Further calculations viewed in Fig. 12.

Unset

```
function [ftildeB,omegaBtilde] = imuSimulator(S,P)

% Initializing input vectors
rI = S.statek.rI;
vI = S.statek.vI;
aI = S.statek.aI;
RBI = S.statek.RBI;
omegaB = S.statek.omegaB;
omegaBdot = S.statek.omegaBdot;

%% Acceleration
% Find RI double dot
RIdd = aI;

% Find noise vector
k = 1;
j = 1;
covarMatrix1 = Qa*eq(k,j);
covarMatrix2 = Qa2*eq(k,j);
va = mvnrnd(zeros(3,1), covarMatrix1)';
va2 = mvnrnd(zeros(3,1), covarMatrix2)';

% Find accelerometer bias, update upon each call to the model persistent ba;

if (isempty(ba))
    % Set ba’s initial value
    QbaSteadyState = Qa2/(1 - alphaa^2);
    ba0 = mvnrnd(zeros(3,1), QbaSteadyState)';
    ba = [ba, ba0];
else
    bak1 = alphaa.*ba(:,end) + va2;
    ba = [ba, bak1];
end

% Find fBtilde
```

```
e3 = [0 0 1]';
ftildeB = RBI*(RIdd + g.*e3) + ba(:,end) + va;


%% Angular Rates
% Find noise vector
k = 1;
j = 1;
covarMatrix1 = Qg*eq(k,j);
covarMatrix2 = Qg2*eq(k,j);
vg = mvnrnd(zeros(3,1), covarMatrix1)';
vg2 = mvnrnd(zeros(3,1), covarMatrix2)';

% Find gyroscope bias, update upon each call to the model persistent bg;
bgk1 = alphag.*bg(:,end) + vg2;
bg = [bg, bgk1];

% Find omegaBtilde
omegaBtilde = omegaB + bg(:,end) + vg;
```
Figure 13: Noisy IMU sensor measurement model

The crux of the IMU sensor measurement model is (8) and (12). Thus, the output of the model becomes the IMU measurement of specific force and quadrotor angular rate. The values are updated upon each call to the function using persistent variable types in Matlab.

Unset
```
function [rx] = hdCameraSimulator(rXI,S,P)

% Solving for RCI
RCI = RCB * RBI;

% Solving for t
t = -RCI*(rI + ((RBI')*rocB));

% Solving for projection matrix P
zeroMatrix = [0 0 0];
K = [K,zeroMatrix'];
PMat = K*[RCI t; zeroMatrix 1];

% Solving for x using PX
xh = PMat*X;

% If feature is not in camera image plane, rx = []
```

```
rx = [];
if xh(3) <= 0
    return % If x(3) is negative, the feature is behind the camera (z-plane), end
script here
end

% Finding xc
x = xh(1)/xh(3);
y = xh(2)/xh(3);
xc = [x y]';

% Finding noise vector wc
k = 1;
j = 1;
covarMatrix = Rc*eq(k,j);
w = mvnrnd(zeros(2,1), covarMatrix)';

% Finding xctilde
xctilde = (1/pixelSize).*xc + w;

% Check if xctilde is inside the camera detection plane
imagePlaneX = imagePlaneSize(1)/2;
imagePlaneY = imagePlaneSize(2)/2;
featureCameraX = abs(xctilde(1));
featureCameraY = abs(xctilde(2));

if featureCameraX <= imagePlaneX && featureCameraY <= imagePlaneY
    rx = xctilde;
else
    rx = [];
end
```

Figure 14: Noisy camera measurement model

For the last of the sensor models, the camera outputs the measured position of the feature point projection on the camera's image plane. These features are predefined in 3D space and, if not in the camera's field of view, do not contribute to the estimation of the quadrotor state. Using the homogenous point projection discussed in section 2.2, the noisy position vector is determined. Given some set of homogeneous coordinates X = [X, Y, Z, 1]^T in the I frame, it is possible to map the coordinates to the camera image plane $\boldsymbol{x}_{ci}$ using (31).

$$\underbrace{\begin{bmatrix} fX_c \\ fY_c \\ Z_c \end{bmatrix}}_{\boldsymbol{x}_{ci}} = \left[\begin{array}{c|c} \underbrace{\begin{matrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{matrix}}_{K} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \end{array}\right] \begin{bmatrix} R_{\mathrm{CI}} & \boldsymbol{t}_{\mathrm{C}} \\ 0_{1\times 3} & 1 \end{bmatrix} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\boldsymbol{X}_{i\mathrm{I}}} \tag{31}$$

The above equation uses the focal length of the camera f, the rotation matrix from the inertial to the camera frame R_{CI}, and the translation vector $t_C = R_{CI}t_I$ expressed in the camera frame C. The implementation and derivation of further values, such as t_I, can be found in Fig. 14. After finding $\boldsymbol{x}_{ci}$, the 3D vector was converted to the 2D image plane using the $[x/z,\ y/z]^T$ method previously mentioned. Then to create the noisy measurement of feature position in the C frame, the following equation was implemented.

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix} = \frac{1}{p_s}\begin{bmatrix} x \\ y \end{bmatrix} + \underbrace{\begin{bmatrix} w_x \\ w_y \end{bmatrix}}_{\boldsymbol{w}_c} \tag{32}$$

The pixel size p_s is used to scale the true projection and a generated noise vector, using a predefined covariance matrix and similar methods as the previous sensor models, is used to slightly alter it.

With the creation of the sensor models, combining the highly nonlinear systems can prove extremely complex. The sensor measurements are used to reconstruct the state and to do this- a provided function for an unscented Kalman filter (UKF) is used for estimation. To initialize the UKF, a function created to solve Wahba’s problem is used to get a reasonable first estimate of attitude. The application can be seen in Fig. 15.

```
Unset
function [RBI] = wahbaSolver(aVec,vIMat,vBMat)

% Find B
B = zeros(3);
for i=1:length(aVec)
    a = aVec(i);
    V = vIMat(i,:)';
    U = vBMat(i,:)';
    B = B + a.*U*V';
end

% Find U and V from B = USV' given B
[U,S,V] = svd(B);

% Initilize M
M = [1 0 0; 0 1 0; 0 0 det(U)*det(V)];

% Solve for
RBI = U*M*V';
```

Figure 15: Wabha’s Problem solver

Next, to create the state estimator, a measurement function is created which is used to output a measurement matrix of the GNSS and camera sensor using the previously defined functions. The calculation of (33) can be seen in the following figure.

$$\boldsymbol{z}(k) = \boldsymbol{h}[\boldsymbol{x}(k)] + \boldsymbol{w}(k) \tag{33}$$

Where $\boldsymbol{z}(k)$ is the vector of measurements and $\boldsymbol{h}[\boldsymbol{x}(k)]$ is a conversion function of the true state.

```
Unset
function [zk] = h_meas(xk,wk,RBIBark,rXIMat,mcVeck,P)

% Find predicted RBI matrix
RBI = euler2dcm(er)*RBIBark;

% Set location of C frame origin in I
rcI = rI + (RBI')*rocB;

% Solve for vector pointing from primary to secondary
rbB = ra2B - ra1B;
rubB = rbB./norm(rbB);

% Solve for first two vectors of z(k)
zk = [rI + (RBI')*ra1B; (RBI')*rubB] + wk(1:6);

% Solve for the feature vectors in z(k)
for i = 1:length(mcVeck)
    % If detected (nonzero), assess
    if mcVeck(i) ~= 0
        % Find vector pointing from camera center to the ith 3D feature in I
        viI = rXIMat(i,:)' - rcI;

        % Normalize for attitude calculation
        vuiI = viI./norm(viI);

        % Finding 3x1 noise vector from wk
        wki = wk(startIndex:endIndex);

        % Solve for vector in C frame
        vuiC = RCB*RBI*vuiI + wki;

        % Update zk for features
        zk = [zk;vuiC];
    end
end
```

Figure 16: Sensor measurement model

Similarly, a dynamics model is created to derive the propagated state from the IMU measurements $\boldsymbol{u}(k)$, current state $\boldsymbol{x}(k)$, and noise measurements $\boldsymbol{v}(k)$ functions.

$$\boldsymbol{x}(k+1) = \boldsymbol{f}\left[\boldsymbol{x}(k), \boldsymbol{u}(k), \boldsymbol{v}(k)\right] \tag{34}$$

In expanding (34) to define the vector-valued function $\boldsymbol{f}$, the equation (35) is applied to the following function to propagate the measured state.

$$\underbrace{\begin{bmatrix} \boldsymbol{r}_{\mathrm{I}}(k+1) \\ \boldsymbol{v}_{\mathrm{I}}(k+1) \\ \boldsymbol{e}(k+1) \\ \boldsymbol{b}_a(k+1) \\ \boldsymbol{b}_g(k+1) \end{bmatrix}}_{\boldsymbol{x}(k+1)} = \underbrace{\begin{bmatrix} \boldsymbol{r}_{\mathrm{I}}(k) + \Delta t \boldsymbol{v}_{\mathrm{I}}(k) + \frac{1}{2}(\Delta t)^2 \boldsymbol{a}_{\mathrm{I}}(k) \\ \boldsymbol{v}_{\mathrm{I}}(k) + \Delta t \boldsymbol{a}_{\mathrm{I}}(k) \\ \boldsymbol{e}(k) + \Delta t \dot{\boldsymbol{e}}(k) \\ \alpha_a \boldsymbol{b}_a(k) + \boldsymbol{v}_{a2}(k) \\ \alpha_g \boldsymbol{b}_g(k) + \boldsymbol{v}_{g2}(k) \end{bmatrix}}_{\boldsymbol{f}[\boldsymbol{x}(k), \boldsymbol{u}(k), \boldsymbol{v}(k)]} \tag{35}$$

The estimated state is defined as the position in the I frame $\boldsymbol{r}_{\mathrm{I}}$, velocity in the I frame $\boldsymbol{v}_{\mathrm{I}}$, euler angles $\boldsymbol{e}$, accelerometer bias $\boldsymbol{b}_{\mathrm{a}}$, and gyro bias $\boldsymbol{b}_{\mathrm{g}}$, as shown above. These state values, alongside the output of the function in Fig. 16 and the Wahba problem solver drive the UKF function.

```
Unset
function [xkp1] = f_dynamics(xk,uk,vk,delt,RBIHatk,P)

% Find aIk
ge3 = [0 0 g]';
aIk = ((RBIk')*(fBtildek - bak - vak)) - ge3;

% Find edotk
omegabk = omegaBtildek - bgk - vgk;
S = (1/cos(phik)).*[cos(phik)*cos(thetak), 0, cos(phik)*sin(thetak);
                    sin(phik)*sin(thetak), cos(phik), -cos(thetak)*sin(phik);
                    -sin(thetak), 0, cos(thetak)];
edotk = S*omegabk;

% Put together output vectors
rIkp1 = rIk + deltat.*vIk + 0.5*(deltat^2)*aIk;
vIkp1 = vIk + deltat.*aIk;
ekp1 = ek + deltat.*edotk;
bakp1 = alphaa*bak + va2k;
bgkp1 = alphag*bgk + vg2k;
```

```
% Output
xkp1 = [rIkp1;vIkp1;ekp1;bakp1;bgkp1];
```

Figure 17: Measurement-based dynamics model

With the application of the function in Fig. 17, the provided UKF function for this experiment is functional.

In integrating these functions, a top level function to call the simulator is employed to assess the control application in this experiment. A protocol generating the target matrices can be seen in Fig. 18.

```
Unset
rng('shuffle');

% Total simulation time, in seconds
Tsim = 10;
% Update interval, in seconds
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec=[0:N-1]'*delt;
% Angular rate of orbit, in rad/sec
n = 2*pi/10;
% Radius of circle, in meters
r = 2;

% Populate reference trajectory
R.tVec = tVec;
R.rIstar = [r*cos(n*tVec),r*sin(n*tVec),ones(N,1)];
R.vIstar = [-r*n*sin(n*tVec),r*n*cos(n*tVec),zeros(N,1)];
R.aIstar = [-r*n*n*cos(n*tVec),-r*n*n*sin(n*tVec),zeros(N,1)];
% The desired xI points toward the origin. The code below also normalizes
% each row in R.xIstar.
R.xIstar = diag(1./vecnorm(R.rIstar'))*(-R.rIstar);

% Matrix of disturbance forces acting on the body, in Newtons, expressed in I
S.distMat = 0*randn(N-1,3);
% Initial position in m
S.state0.r = [r 0 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 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 = 2;
% Feature locations in the I frame
S.rXIMat = [0,0,0; 0,0,0.7];
%S.rXIMat = [];
```

Figure 18: Quadrotor top level simulation script

These functions set up the foundation for conducting in-depth 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 and theory are tested to verify accurate implementation.

4.1 Wahba Solver Testing

To test the wahba's problem solver depicted in Fig. 15, a R_{BI} matrix is initialized using a random euler angle vector as input into the function from Fig. 5. A vector of least squares is also initialized to a N by 1 matrix of ones. Using random input, N unit vectors are defined to represent the body frame vectors. With the generated R_{BI} matrix, these vectors are transformed into the I frame. The body frame and inertial frame vectors are then used to construct the input matrices into the Wahba's Problem solver of size N by 3, respectively. The resulting matrix is then compared to the initial R_{BI} matrix, for which there should be an exact match.

Upon further testing to iron out any bugs within the function, noise is introduced to the inertial frame vectors after the transformation from the body frame is conducted. The magnitude of the added noise is small, between 0 and 0.1, however, this is a more realistic test of the functions utility. The same process as before proceeds and the resulting R_{BI} is compared to the original. The test results of both cases can be found in Table 2.

	Generated Rotation Matrix	Calculated Rotation Matrix

No Noise Test	
	```
	0.0000        0   -1.0000
	     0   1.0000        0
	1.0000        0    0.0000
	```
		```
		 0.0000  -0.0000  -1.0000
		-0.0000   1.0000  -0.0000
		 1.0000   0.0000   0.0000
		```

Noise Added Test	0.0000 0 -1.0000
	0 1.0000 0
	1.0000 0 0.0000
		0.0578 0.0736 -0.9956
		-0.0391 0.9967 0.0714
		0.9976 0.0348 0.0605

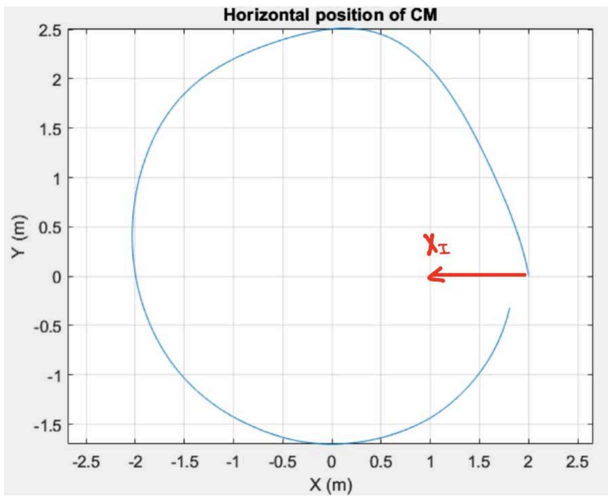
Table 2: Testing results of Wabha's Problem solver function using euler angles $\boldsymbol{e} = [0\ \pi/2\ 0]^T$

In analyzing Table 2, the generated rotation matrix is the true rotation matrix of the initialized euler angles $\boldsymbol{e}$ and the calculated rotation matrices are the outputs from the Wahba's problem solver function from Fig. 15. In the no noise test, the calculated R_{BI} matrix is an exact match. When noise of known magnitude was added to the inertial frame vectors, or the 'vIMat' function input, variations under the magnitude of 0.08 were observed- which marks passing. Note that this number would decrease as N increases. It was also found that increasing the magnitude of noise increased the error of the calculated rotation matrix, however, noise on the order of 0 to 0.1 is a safe assumed value for what can be expected during the full model simulation.

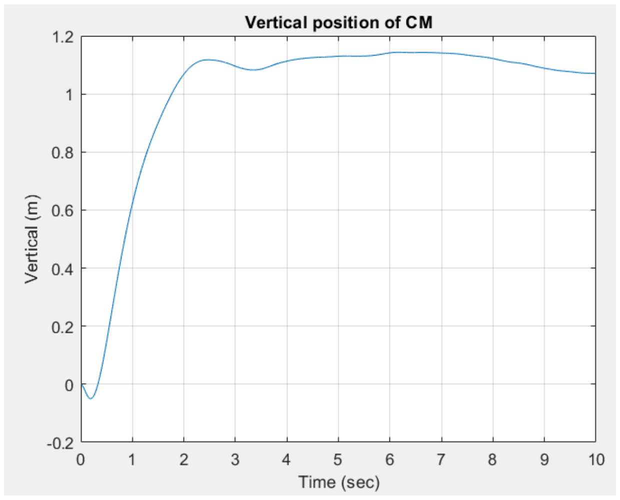
This test is likely good enough to catch discrepancies in the function because it accurately models the use case the function was developed for. As a result of the successful results above, it can be concluded that the function is working appropriately. A detailed view of the testing procedure can be found in the appendix.

4.2 Experimentation

Upon feeding the desired trajectory, the simulation function infers the state from noisy measurements. For this experiment, the desired trajectory was a circular path with an orbital period of ten seconds and a diameter of four meters at constant altitude. The following figure displays the simulated results.



A. Quadrotor horizontal position



B. Quadrotor vertical position

Figure 19: Simulation results using sensor measurement models

As evident from Fig. 19, the developed controller operates successfully given an estimated, noisy state. Furthermore, the quadrotor is able to fly the intended path with the intended yaw angle as shown below.

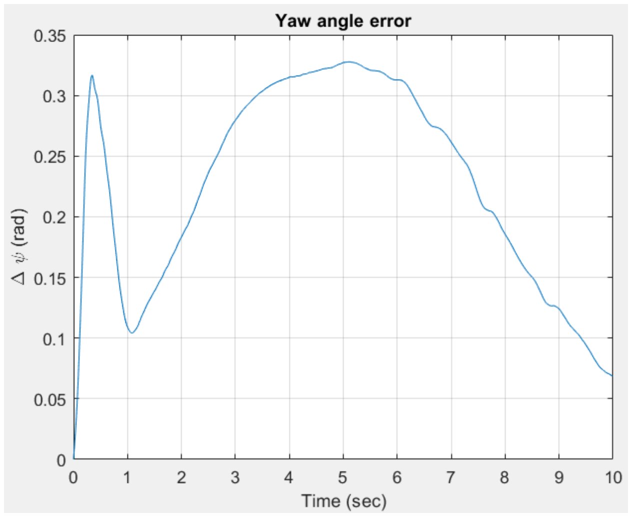
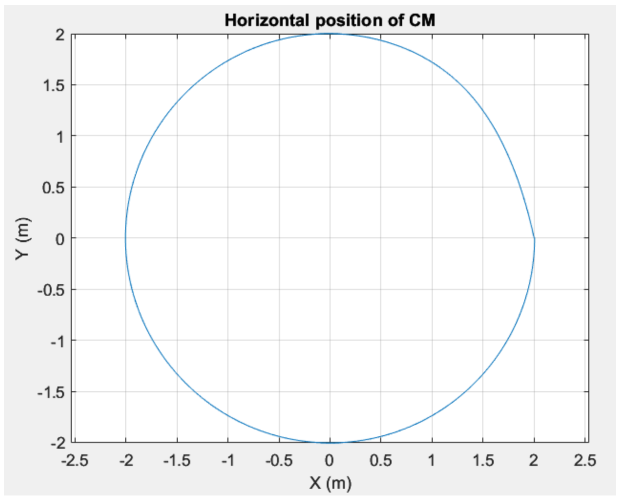
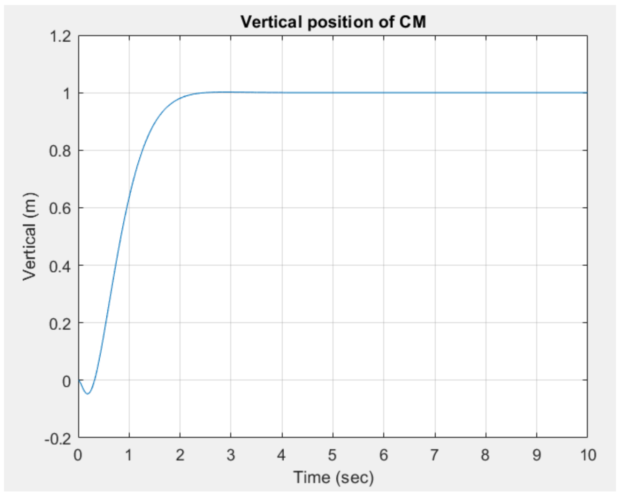


Figure 20: Yaw angle error during quadrotor simulation

The yaw angle, although high at the start, lends itself to minimization as the simulation continues. The result from Fig. 20 can be compared to the previously developed simulation function that only propagates the true state. Similar graphs for the old simulator function are produced to assess tracking performance of both simulation methods.



A. Quadrotor horizontal position



B. Quadrotor vertical position

Figure 21: Simulation results only using true state

In conducting an empirical analysis, the trajectory developed using the old simulation method in Fig. 21 is much better in terms of accuracy. The created horizontal trajectory looks much more like the reference trajectory and the vertical position settles to one much quicker than in Fig. 19. Furthermore, after the initial maneuver from the misplaced initial position, the yaw angle error settles much quicker and to a lower value than the state estimation simulator. This can be shown below.

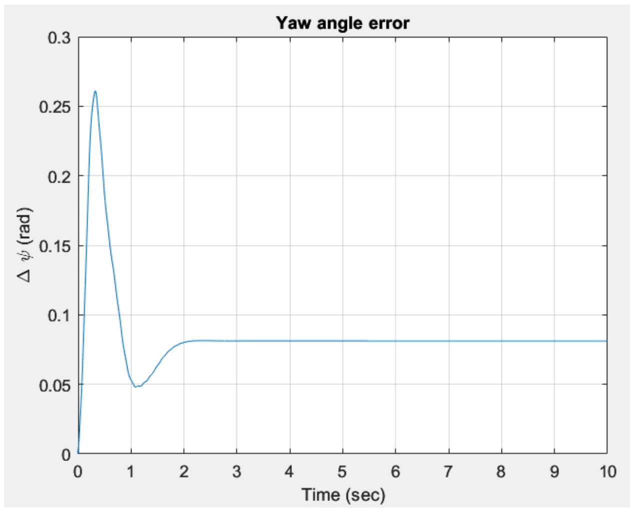
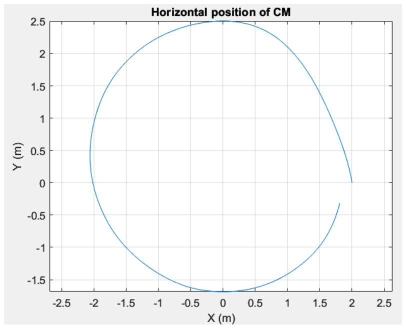
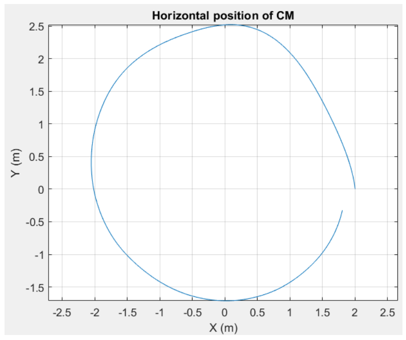
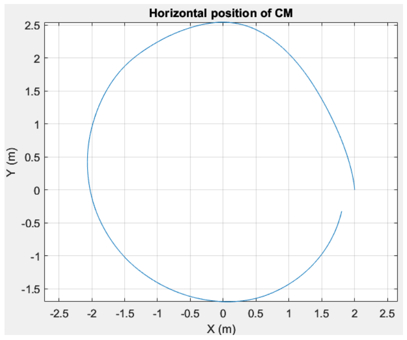


Figure 22: Yaw angle error during quadrotor simulation

Overall, the tracking performance of the previous simulation method is better. However, like mentioned before, the previous simulation model is not as realistic as the simulator developed in this experiment due to the implementation of sensor measurement noise. This effect is randomly generated using a monte carlo method in the top level simulation script detailed in Fig. 18. As a result, the tracking performance can vary from test to test as evident in Table 1.

Test 1	Test 2	Test 3
		

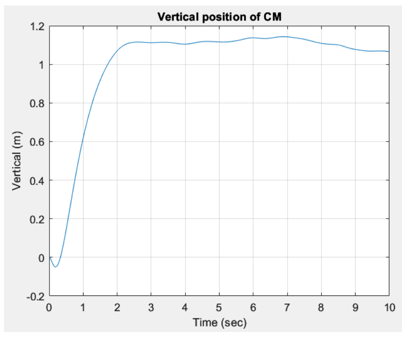
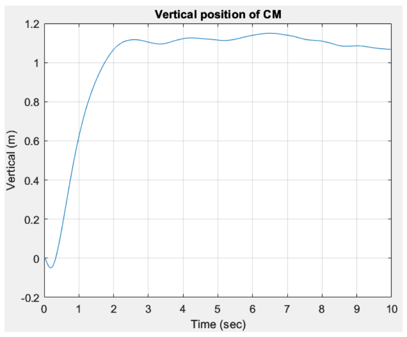
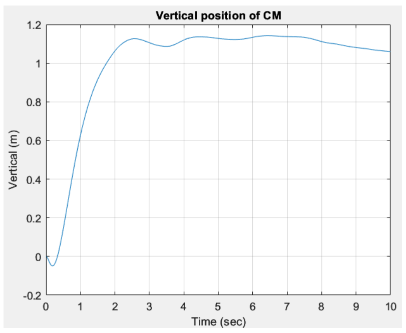
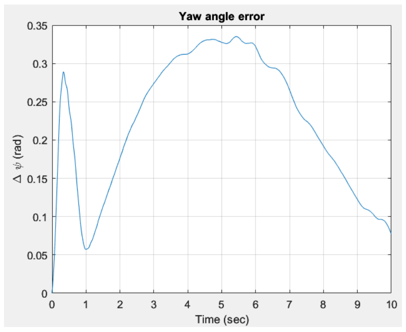
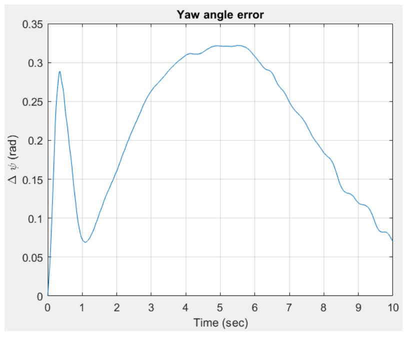
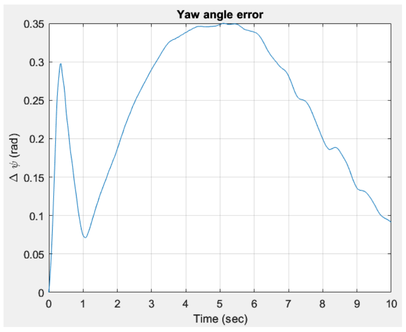
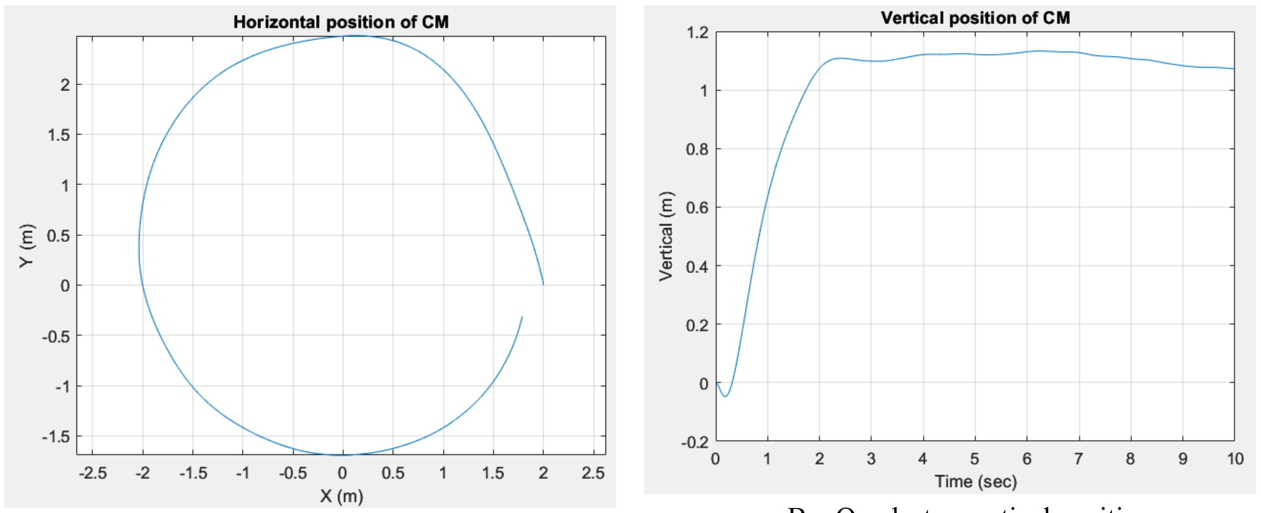
		
		

Table 1: Variation of tracking performance across simulation iterations

The table denotes the variation in simulation due to the randomization of noise. The fluctuation in tracking performance is clear in the yaw angle error plots of each test above. In Test 3, the yaw angle error peaks much higher than in Test 1, which too peaks higher than Test 2. In terms of yaw angle error as the marker of tracking performance, one could observe by how much the tracking performance changes each time the simulation function is run. However, the variation value across tests is hardly noticeable in the vertical and horizontal trajectory plots from Table 1.

A final test was performed in comparing the developed simulation with the camera “on” and “off.” This was done by defining the feature array as empty in the top level script from Fig. 18. Without the camera input, the trajectory plots are presented below.



A. Quadrotor horizontal position

B. Quadrotor vertical position

Figure 23: Simulation results using only IMU and GNSS sensor measurement models

From Fig. 23, one can remark that removing the camera had little to no effect on the accuracy of the trajectory. The initial maneuver from the starting position slightly past $\pi/2$ radians is increasingly incorrect but the quadrotor quickly fixes the difference from the reference trajectory. In analyzing Fig. 24, it is clear that, in removing the camera, the estimator states are still sufficiently accurate to control the quadrotor.

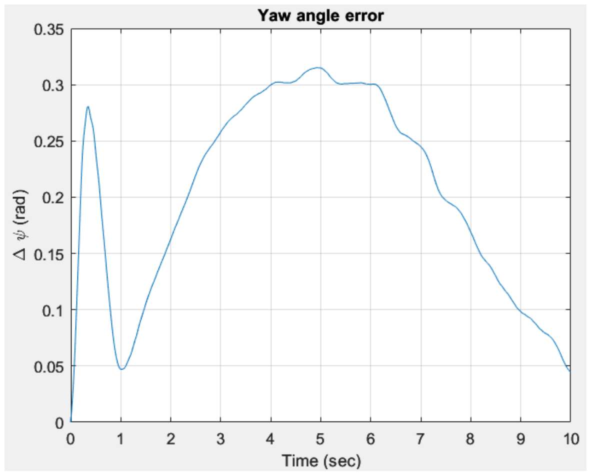


Figure 24: Yaw angle error during quadrotor simulation without camera sensor

From the plot in Fig. 24, the peak yaw angle error is almost the same as the previous tests run, possibly less than some. This occurs most likely for two reasons. The first reason is inherent redundant risk mitigation. If no features are in the camera field of view, the camera sensor is essentially "off" which may be a scenario more often than not. As a result, it is crucial that the IMU and GNSS noisy sensor measurements alone are enough for the state estimator to construct an accurate state. Furthermore, for this specific maneuver, where the quadcopter is just rotating at a fixed altitude to create the observed circular trajectory, the IMU and GNSS offer enough information to construct an accurate state. The presence of detected mapped features from the camera do not hinder the accuracy of the trajectory due to the simplicity of the attempted maneuver.

5 Conclusion

A closed loop control strategy was developed to simulate a quadcopter conducting a specific circular trajectory. The method 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. Furthermore, models for a GNSS antenna, IMU, and camera sensor were integrated into the control loop. The models represented noisy sensor measurements, based on the true dynamics of the quadcopter. These measurements were then reconstructed to estimate the current state. This estimation was used over the true state for simulated control since pragmatic systems apply the same concept. As a result, a simulation of higher fidelity was developed and tested against previous methods. It was found that the control loop is able to use the sensors' estimated state to *accurately* conduct the target circular trajectory.

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] “Rank of a Matrix - Definition: How to Find the Rank of the Matrix?” Cuemath, www.cuemath.com/algebra/rank-of-a-matrix/. Accessed 4 Mar. 2024.

[3] “Homogeneous Coordinates and Vanishing Points.” UCSD, CSE Web, cseweb.ucsd.edu/classes/sp06/cse152/hw1sol.pdf. Accessed 4 Mar. 2024.

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);

% 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);
zI = RBI(3,:)';

% Calculate drag coefficient
epsilon_vI = 1e-3;
da = 0; vIu = [1;0;0];
if(norm(vI) > epsilon_vI)
  vIu = vI/norm(vI);
  fd = abs(zI'*vIu)*norm(vI)^2;
  da = 0.5*P.quadParams.Cd*P.quadParams.Ad*P.constants.rho*fd;
end

% Find derivatives of state elements
rIdot = vI;
```

```
vIdot = ([0;0;-mq*gE] + RBI'*sum(FMat,2) + distVec - da*vIu)/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);
omegaVecdot = (eaVec.*P.quadParams.cm - omegaVec)./P.quadParams.taum;

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

% 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 high fidelity ordinary differential equation 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 6: Quadcopter control simulation function

```
Unset
function [Q] = simulateQuadrotorEstimationAndControl(R,S,P)
% simulateQuadrotorEstimationAndControl : Simulates closed-loop estimation and
%                                         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 I 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).
%
%        rXIMat = Nf-by-3 matrix of coordinates of visual features in the
%                 simulation environment, expressed in meters in the I
%                 frame. rXIMat(i,:)' is the 3x1 vector of coordinates of
%                 the ith feature.
%
% 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);
omegaVec0 = zeros(4,1);
Xk = [S.state0.r;S.state0.v;RBIk(:);S.state0.omegaB;omegaVec0];
Xdotk = zeros(length(Xk),1);
statek.RBI = zeros(3,3);
[Nf,~] = size(S.rXIMat);
Se.rXIMat = S.rXIMat;
Se.delt = dtIn;
XMat = []; tVec = [];

for kk=1:N-1
  % Simulate measurements
  statek.rI = Xk(1:3);
  statek.RBI(:) = Xk(7:15);
  statek.vI = Xk(4:6);
  statek.omegaB = Xk(16:18);
  statek.aI = Xdotk(4:6);
  statek.omegaBdot = Xdotk(16:18);
  Sm.statek = statek;
  % Simulate measurements
  M.tk=dtIn*(kk-1);
  [M.rpGtilde,M.rbGtilde] = gnssMeasSimulator(Sm,P);
  M.rxMat = [];
  for ii=1:Nf
    rx = hdCameraSimulator(S.rXIMat(ii,:)',Sm,P);
    if(isempty(rx))
      M.rxMat(ii,:) = [NaN,NaN];
    else
      M.rxMat(ii,:) = rx';
    end
  end
  [M.ftildeB,M.omegaBtilde] = imuSimulator(Sm,P);
  % Call estimator
  E = stateEstimatorUKF(Se,M,P);
  if(~isempty(E.statek))
    % Call trajectory and attitude controllers
    Rtc.rIstark = R.rIstar(kk,:)';
    Rtc.vIstark = R.vIstar(kk,:)';
    Rtc.aIstark = R.aIstar(kk,:)';
    Rac.xIstark = R.xIstar(kk,:)';
    distVeck = S.distMat(kk,:)';
    Sc.statek = E.statek;
    [Fk,Rac.zIstark] = trajectoryController(Rtc,Sc,P);
    NBk = attitudeController(Rac,Sc,P);
    % Convert commanded Fk and NBk to commanded voltages
    eaVeck = voltageConverter(Fk,NBk,P);
  else
```

```
    % Apply no control if state estimator's output is empty.  Set distVeck to
    % apply a normal force in the vertical direction that exactly offsets the
    % acceleration due to gravity.
    eaVeck = zeros(4,1);
    distVeck = [0;0;P.quadParams.m*P.constants.g];
  end
  tspan = [R.tVec(kk):dtOut:R.tVec(kk+1)]';
  [tVeck,XMatk] = ...
      ode45(@(t,X) quadOdeFunctionHF(t,X,eaVeck,distVeck,P),tspan,Xk);
  if(length(tspan) == 2)
    % Deal with S.oversampFact = 1 case
    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,:)';
  Xdotk = quadOdeFunctionHF(tVeck(end),Xk,eaVeck,distVeck,P);
  % Ensure that RBI remains orthogonal
  if(mod(kk,100) == 0)
   RBIk(:) = Xk(7:15);
   [UR,~,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.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 and estimation simulation function

Unset

```
% Top-level script for calling simulateQuadrotorControl or
% simulateQuadrotorEstimationAndControl

% 'clear all' is needed to clear out persistent variables from run to run
clear all; clc;
% Seed Matlab's random number: this allows you to simulate with the same noise
% every time (by setting a nonnegative integer seed as argument to rng) or
% simulate with a different noise realization every time (by setting
% 'shuffle' as argument to rng).
rng('shuffle');
%rng(1234);
% Assert this flag to call the full estimation and control simulator;
% otherwise, only the control simulator is called
estimationFlag = 1;
% Total simulation time, in seconds
Tsim = 10;
% Update interval, in seconds
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec=[0:N-1]'*delt;
% Angular rate of orbit, in rad/sec
n = 2*pi/10;
% Radius of circle, in meters
r = 2;
% Populate reference trajectory
R.tVec = tVec;
R.rIstar = [r*cos(n*tVec),r*sin(n*tVec),ones(N,1)];
R.vIstar = [-r*n*sin(n*tVec),r*n*cos(n*tVec),zeros(N,1)];
R.aIstar = [-r*n*n*cos(n*tVec),-r*n*n*sin(n*tVec),zeros(N,1)];
% The desired xI points toward the origin. The code below also normalizes
% each row in R.xIstar.
R.xIstar = diag(1./vecnorm(R.rIstar'))*(-R.rIstar);
% Matrix of disturbance forces acting on the body, in Newtons, expressed in I
S.distMat = 0*randn(N-1,3);
% Initial position in m
S.state0.r = [r 0 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 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 = 2;
% Feature locations in the I frame
%S.rXIMat = [0,0,0; 0,0,0.7];
S.rXIMat = [];
% Quadrotor parameters and constants
```

```
quadParamsScript;
constantsScript;
sensorParamsScript;
P.quadParams = quadParams;
P.constants = constants;
P.sensorParams = sensorParams;

if(estimationFlag)
  Q = simulateQuadrotorEstimationAndControl(R,S,P);
else
  Q = simulateQuadrotorControl(R,S,P);
end

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=2.5*[-1 1 -1 1 -0.1 1];
visualizeQuad(S2);

figure(2);clf;
plot(Q.tVec,Q.state.rMat(:,3)); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('Vertical position of CM');

figure(3);clf;
psiError = unwrap(n*Q.tVec + pi - Q.state.eMat(:,3));
meanPsiErrorInt = round(mean(psiError)/2/pi);
plot(Q.tVec,psiError - meanPsiErrorInt*2*pi);
grid on;
xlabel('Time (sec)');
ylabel('\Delta \psi (rad)');
title('Yaw angle error');

figure(5);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 8: 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 9: Simulation visualization function

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

```
Unset
function [rpGtilde,rbGtilde] = gnssMeasSimulator(S,P)
% gnssMeasSimulator : Simulates GNSS measurements for quad.
%
```

```matlab
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
%
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
%
%
% OUTPUTS
%
% rpGtilde --- 3x1 GNSS-measured position of the quad's primary GNSS antenna,
%              in ECEF coordinates relative to the reference antenna, in
%              meters.
%
% rbGtilde --- 3x1 GNSS-measured position of secondary GNSS antenna, in ECEF
%              coordinates relative to the primary antenna, in meters.
%              rbGtilde is constrained to satisfy norm(rbGtilde) = b, where b
%              is the known baseline distance between the two antennas.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+

rI = S.statek.rI;
RBI = S.statek.RBI;

RpL = P.sensorParams.RpL;
sigmab = P.sensorParams.sigmab;
r0G = P.sensorParams.r0G;
ra1B = P.sensorParams.raB(:,1); % primary antenna 3x1 position in B
ra2B = P.sensorParams.raB(:,2); % secondary antenna 3x1 position in B

%% Primary
```

```
% Transform the inertial ECEF frame to the ENU frame, where vEnu = R*vEcef
RLG = Recef2enu(r0G);

% Find RpG
RpG = inv(RLG)*RpL*inv(RLG');

% Finding coordinates of primary antenna in I
rpI = rI + (RBI')*ra1B;

% Finding coordinates of primary antenna in G
rpG = (RLG')*rpI;

% Simulate noise vector
k = 1;
j = 1;
covarMatrix = RpG*eq(k,j);
w = mvnrnd(zeros(3,1), covarMatrix)';

% Finding rpGtilde
rpGtilde = rpG + w;

%% Baseline

% Finding coordinates of secondary antenna in I
rbI = rI + (RBI')*ra2B;

% Finding coordinates of secondary antenna in G
rbG = (RLG')*rbI;

% Find RbG
epsilon = 10^(-9);
rubG = rbG./norm(rbG);
RbG = ((norm(rbG)^2)*(sigmab^2)).*(eye(3) - (rubG*(rubG'))) + epsilon.*eye(3);

% Simulate noise vector
k = 1;
j = 1;
covarMatrix = RbG*eq(k,j);
w = mvnrnd(zeros(3,1), covarMatrix)';

% Finding rbGtilde with constraint norm(rbGtilde) = norm(rbG)
rbGtildeNoMag = rbG + w;
rbGtildeUnit = rbGtildeNoMag./norm(rbGtildeNoMag);
rbGtilde = norm(rbG).*rbGtildeUnit;
```

Figure 13: GNSS sensor model

Unset

```
function [ftildeB,omegaBtilde] = imuSimulator(S,P)
% imuSimulator : Simulates IMU measurements of specific force and
%                body-referenced angular rate.
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
%
%                  vI = 3x1 velocity of CM with respect to the I frame and
%                       expressed in the I frame, in meters per second.
%
%                  aI = 3x1 acceleration of CM with respect to the I frame and
%                       expressed in the I frame, in meters per second^2.
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
%
%              omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
%           omegaBdot = 3x1 time derivative of omegaB, in radians per
%                       second^2.
%
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
% OUTPUTS
%
% ftildeB ---- 3x1 specific force measured by the IMU's 3-axis accelerometer
%
% omegaBtilde  3x1 angular rate measured by the IMU's 3-axis rate gyro
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+
```

```matlab
% Initializing input vectors
rI = S.statek.rI;
vI = S.statek.vI;
aI = S.statek.aI;
RBI = S.statek.RBI;
omegaB = S.statek.omegaB;
omegaBdot = S.statek.omegaBdot;

% Initializing sensor parameters
lB = P.sensorParams.lB;
Sa = P.sensorParams.Sa;
Qa = P.sensorParams.Qa;
sigmaa = P.sensorParams.sigmaa;
alphaa = P.sensorParams.alphaa;
Qa2 = P.sensorParams.Qa2;
Sg = P.sensorParams.Sg;
Qg = P.sensorParams.Qg;
sigmag = P.sensorParams.sigmag;
alphag = P.sensorParams.alphag;
Qg2 = P.sensorParams.Qg2;

% Initializing constants
g = P.constants.g;

%% Acceleration

% Find RI double dot
RIdd = aI;

% Find noise vector
k = 1;
j = 1;
covarMatrix1 = Qa*eq(k,j);
covarMatrix2 = Qa2*eq(k,j);
va = mvnrnd(zeros(3,1), covarMatrix1)';
va2 = mvnrnd(zeros(3,1), covarMatrix2)';

% Find accelerometer bias, update upon each call to the model
persistent ba;

if (isempty(ba))
    % Set ba’s initial value
    QbaSteadyState = Qa2/(1 - alphaa^2);
    ba0 = mvnrnd(zeros(3,1), QbaSteadyState)';
    ba = [ba, ba0];
else
    bak1 = alphaa.*ba(:,end) + va2;
    ba = [ba, bak1];
end

% Find fBtilde
e3 = [0 0 1]';
ftildeB = RBI*(RIdd + g.*e3) + ba(:,end) + va;

%% Angular Rates

% Find noise vector
k = 1;
j = 1;
covarMatrix1 = Qg*eq(k,j);
covarMatrix2 = Qg2*eq(k,j);
vg = mvnrnd(zeros(3,1), covarMatrix1)';
vg2 = mvnrnd(zeros(3,1), covarMatrix2)';

% Find gyroscope bias, update upon each call to the model
persistent bg;

if(isempty(bg))
    % Set bg’s initial value
    QbgSteadyState = Qg2/(1 - alphag^2);
    bg0 = mvnrnd(zeros(3,1), QbgSteadyState)';
    bg = [ba, bg0];
else
    bgk1 = alphag.*bg(:,end) + vg2;
    bg = [bg, bgk1];
end

% Find omegaBtilde
omegaBtilde = omegaB + bg(:,end) + vg;
```

Figure 14: IMU sensor model

Unset

```
function [rx] = hdCameraSimulator(rXI,S,P)
% hdCameraSimulator : Simulates feature location measurements from the
%                     quad's high-definition camera.
%
%
% INPUTS
%
% rXI -------- 3x1 location of a feature point expressed in I in meters.
%
% S ---------- Structure with the following elements:
```

```
%
%        statek = State of the quad at tk, expressed as a structure with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
%
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
%
% OUTPUTS
%
% rx --------- 2x1 measured position of the feature point projection on the
%              camera's image plane, in pixels.  If the feature point is not
%              visible to the camera (the ray from the feature to the camera
%              center never intersects the image plane, or the feature is
%              behind the camera), then rx is an empty matrix.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+

rI = S.statek.rI;
RBI = S.statek.RBI;

rocB = P.sensorParams.rocB;
RCB = P.sensorParams.RCB;
Rc = P.sensorParams.Rc;
pixelSize = P.sensorParams.pixelSize;
K = P.sensorParams.K;
imagePlaneSize = P.sensorParams.imagePlaneSize;

X = [rXI;1];

% Solving for RCI
RCI = RCB * RBI;

% Solving for t
t = -RCI*(rI + ((RBI')*rocB));

% Solving for projection matrix PMat
```

```
zeroMatrix = [0 0 0];
K = [K,zeroMatrix'];
PMat = K*[RCI t; zeroMatrix 1];

% Solving for x using PX
xh = PMat*X;

% If feature is not in camera image plane, rx = []
rx = [];
if xh(3) <= 0
    return % If x(3) is negative, the feature is behind the camera (z-plane), end
script here
end

% Finding xc
x = xh(1)/xh(3);
y = xh(2)/xh(3);
xc = [x y]';

% Finding noise vector wc
k = 1;
j = 1;
covarMatrix = Rc*eq(k,j);
w = mvnrnd(zeros(2,1), covarMatrix)';

% Finding xctilde
xctilde = (1/pixelSize).*xc + w;

% Check if xctilde is inside the camera detection plane
imagePlaneX = imagePlaneSize(1)/2;
imagePlaneY = imagePlaneSize(2)/2;
featureCameraX = abs(xctilde(1));
featureCameraY = abs(xctilde(2));

if featureCameraX <= imagePlaneX && featureCameraY <= imagePlaneY
    rx = xctilde;
else
    rx = [];
end
```

Figure 15: Camera sensor model

Unset

```
function [zk] = h_meas(xk,wk,RBIBark,rXIMat,mcVeck,P)
% h_meas : Measurement model for quadcopter.
%
```

```
% INPUTS
%
% xk --------- 15x1 state vector at time tk, defined as
%
%              xk = [rI', vI', e', ba', bg']'
%
%              where all corresponding quantities are identical to those
%              defined for E.statek in stateEstimatorUKF.m and where e is the
%              3x1 error Euler angle vector defined such that for an estimate
%              RBIBar of the attitude, the true attitude is RBI =
%              dRBI(e)*RBIBar, where dRBI(e) is the DCM formed from the error
%              Euler angle vector e.
%
% wk --------- nz-by-1 measurement noise vector at time tk, defined as
%
%              wk = [wpIk', wbIk', w1C', w2C', ..., wNfkC']'
%
%              where nz = 6 + Nfk*3, with Nfk being the number of features
%              measured by the camera at time tk, and where all 3x1 noise
%              vectors represent additive noise on the corresponding
%              measurements.
%
% RBIBark ---- 3x3 attitude matrix estimate at time tk.
%
% rXIMat ----- Nf-by-3 matrix of coordinates of visual features in the
%              simulation environment, expressed in meters in the I
%              frame. rXIMat(i,:)' is the 3x1 vector of coordinates of the ith
%              feature.
%
% mcVeck ----- Nf-by-1 vector indicating whether the corresponding feature in
%              rXIMat is sensed by the camera: If mcVeck(i) is true (nonzero),
%              then a measurement of the visual feature with coordinates
%              rXIMat(i,:)' is assumed to be made by the camera.  mcVeck
%              should have Nfk nonzero values.
%
% 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
```

```
%
% zk --------- nz-by-1 measurement vector at time tk, defined as
%
%              zk = [rpItilde', rbItildeu', v1Ctildeu', ..., vNfkCtildeu']'
%
%              where rpItilde is the 3x1 measured position of the primary
%              antenna in the I frame, rbItildeu is the 3x1 measured unit
%              vector pointing from the primary to the secondary antenna,
%              expressed in the I frame, and viCtildeu is the 3x1 unit vector,
%              expressed in the camera frame, pointing toward the ith 3D
%              feature, which has coordinates rXIMat(i,:)'.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+

% Unpack state vector
rI = xk(1:3);
vI = xk(4:6);
er = xk(7:9);
ba = xk(10:12);
bg = xk(13:15);

% Initialize needed parameters
rocB = P.sensorParams.rocB;
ra1B = P.sensorParams.raB(:,1);
ra2B = P.sensorParams.raB(:,2);
RCB = P.sensorParams.RCB;

% Find predicted RBI matrix
RBI = euler2dcm(er)*RBIBark;

% Set location of C frame origin in I
rcI = rI + (RBI')*rocB;

% Solve for vector pointing from primary to secondary
rbB = ra2B - ra1B;
rubB = rbB./norm(rbB);

% Solve for first two vectors of z(k)
zk = [rI + (RBI')*ra1B; (RBI')*rubB] + wk(1:6);

% Initializing indexes for noise to parse
startIndex = 6;
endIndex = 8;
```

```
% Solve for the feature vectors in z(k)
for i = 1:length(mcVeck)
    % If detected (nonzero), assess
    if mcVeck(i) ~= 0
        % Find vector pointing from camera center to the ith 3D feature in I
        viI = rXIMat(i,:)' - rcI;

        % Normalize for attitude calculation
        vuiI = viI./norm(viI);

        % Finding 3x1 noise vector from wk
        wki = wk(startIndex:endIndex);

        % Solve for vector in C frame
        vuiC = RCB*RBI*vuiI + wki;

        % Update zk for features
        zk = [zk;vuiC];
    end
    % Update noise array indexes
    startIndex = startIndex + 3;
    endIndex = endIndex + 3;
end
```

Figure 16: Sensor measurement model

```
Unset
function [xkp1] = f_dynamics(xk,uk,vk,delt,RBIHatk,P)
% f_dynamics : Discrete-time dynamics model for quadcopter.
%
% INPUTS
%
% xk --------- 15x1 state vector at time tk, defined as
%
%              xk = [rI', vI', e', ba', bg']'
%
%              where all corresponding quantities are identical to those
%              defined for E.statek in stateEstimatorUKF.m and where e is the
%              3x1 error Euler angle vector defined such that for an estimate
%              RBIHat of the attitude, the true attitude is RBI =
%              dRBI(e)*RBIHat, where dRBI(e) is the DCM formed from the error
%              Euler angle vector e.
%
% uk --------- 6x1 IMU measurement input vector at time tk, defined as
%
```

```
%               uk = [omegaBtilde', fBtilde']'
%
%               where all corresponding quantities are identical to those
%               defined for M in stateEstimatorUKF.m.
%
% vk --------- 12x1 process noise vector at time tk, defined as
%
%               vk = [vg', vg2', va', va2']'
%
%               where vg, vg2, va, and va2 are all 3x1 mutually-independent
%               samples from discrete-time zero-mean Gaussian noise processes.
%               These represent, respectively, the gyro white noise (rad/s),
%               the gyro bias driving noise (rad/s), the accelerometer white
%               noise (m/s^2), and the accelerometer bias driving noise
%               (m/s^2).
%
% delt ------- Propagation interval, in seconds.
%
% RBIHatk ---- 3x3 attitude matrix estimate at time tk.
%
%
% 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
%
% xkp1 ------- 15x1 state vector propagated to time tkp1
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+

if(abs(delt - P.sensorParams.IMUdelt) > 1e-9)
  error('Propagation time must be same as IMU measurement time');
end
```

```
% Initializing input vectors
rIk = xk(1:3); % 3x1
vIk = xk(4:6);
ek = xk(7:9);
bak = xk(10:12);
bgk = xk(13:15);
omegaBtildek = uk(1:3);
fBtildek = uk(4:6);
vgk = vk(1:3);
vg2k = vk(4:6);
vak = vk(7:9);
va2k = vk(10:12);
RBIk = euler2dcm(ek)*RBIHatk;
phik = ek(1);
thetak = ek(2);
psik = ek(3);

% Initializing sensor parameters
deltat = P.sensorParams.IMUdelt;
alphaa = P.sensorParams.alphaa;
alphag = P.sensorParams.alphag;
g = P.constants.g;

% Find aIk
ge3 = [0 0 g]';
aIk = ((RBIk')*(fBtildek - bak - vak)) - ge3;

% Find edotk
omegabk = omegaBtildek - bgk - vgk;
S = (1/cos(phik)).*[cos(phik)*cos(thetak), 0, cos(phik)*sin(thetak);
                    sin(phik)*sin(thetak), cos(phik), -cos(thetak)*sin(phik);
                    -sin(thetak), 0, cos(thetak)];
edotk = S*omegabk;

% Put together output vectors
rIkp1 = rIk + deltat.*vIk + 0.5*(deltat^2)*aIk;
vIkp1 = vIk + deltat.*aIk;
ekp1 = ek + deltat.*edotk;
bakp1 = alphaa*bak + va2k;
bgkp1 = alphag*bgk + vg2k;

% Output
xkp1 = [rIkp1;vIkp1;ekp1;bakp1;bgkp1];
```
Figure 17: Measurement-based dynamics model

Unset

```
function [E] = stateEstimatorUKF(S,M,P)
% stateEstimatorUKF : Unscented-Kalman-filter-based estimation of the state of
%                     quadcopter from sensor measurements.
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%        rXIMat = Nf-by-3 matrix of coordinates of visual features in the
%                 simulation environment, expressed in meters in the I
%                 frame. rXIMat(i,:)' is the 3x1 vector of coordinates of
%                 the ith feature.
%
%          delt = Measurement update interval, in seconds.
%
% M ---------- Structure with the following elements:
%
%            tk = Time at which all measurements apply, in seconds.
%
%      rpGtilde = 3x1 GNSS-measured position of the quad's primary GNSS
%                 antenna, in ECEF coordinates relative to the reference
%                 antenna, in meters.
%
%      rbGtilde = 3x1 GNSS-measured position of secondary GNSS antenna, in
%                 ECEF coordinates relative to the primary antenna, in meters.
%                 rbGtilde is constrained to satisfy norm(rbGtilde) = b, where b
%                 is the known baseline distance between the two antennas.
%
%         rxMat = Nf-by-2 matrix of measured positions of feature point
%                 projections on the camera's image plane, in
%                 pixels. rxMat(i,:)' is the 2x1 image position measurement of
%                 the 3D feature in rXIMat(i,:)'.  If the ith feature point is
%                 not visible to the camera (the ray from the feature to the
%                 camera center never intersects the image plane), then
%                 rxMat(i,:) = [NaN, NaN].
%
%       ftildeB = 3x1 specific force measured by the IMU's 3-axis
%                 accelerometer, in meters/sec^2
%
%   omegaBtilde = 3x1 angular rate measured by the IMU's 3-axis rate gyro,
%                 in rad/sec.
%
%
% 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
%
% E ---------- Structure with the following elements:
%
%        statek = Estimated state of the quad at tk, expressed as a structure
%                 with the following elements.
%
%                  rI = 3x1 position of CM in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude
%
%                  vI = 3x1 velocity of CM 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.
%
%                  ba = 3x1 bias of accelerometer expressed in the
%                       accelerometer frame in meter/second^2.
%
%                  bg = 3x1 bias of rate gyro expressed in the body frame in
%                       rad/sec.
%
%            Pk = nx-by-nx error covariance matrix for the estimator state
%                 vector xk that applies at time tk, which is defined as
%
%                 xk = [rI', vI', e', ba', bg']'
%
%                 where all corresponding quantities are identical to those
%                 defined for E.statek and where e is the 3x1 error Euler
%                 angle vector defined such that for an estimate RBIHat of the
%                 attitude, the true attitude is RBI = dRBI(e)*RBIHat, where
%                 dRBI(e) is the DCM formed from the error Euler angle vector
%                 e, expressed in radians.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author: Todd Humphreys
```

```
%+==============================================================================+

%----- Setup
persistent xBark PBark RBIBark
RIG = Recef2enu(P.sensorParams.r0G);
rbB = P.sensorParams.raB(:,2) - P.sensorParams.raB(:,1); rbBu = rbB/norm(rbB);
rpB = P.sensorParams.raB(:,1);  e3 = [0;0;1];
epsilon = 1e-8; nx = 15; nv = 12;
alphaUKF = 1e-3; betaUKF = 2; kappaUKF = 0;
lambda_p = alphaUKF^2*(kappaUKF + nx + nv) - nx - nv;
c_p = sqrt(nx+nv+lambda_p);
Wm0_p = lambda_p/(nx + nv + lambda_p);
Wmi_p = 1/(2*(nx + nv + lambda_p));
Wc0_p = Wm0_p + 1 - alphaUKF^2 + betaUKF; Wci_p = Wmi_p;

%----- Convert GNSS measurements to I frame
rpItilde = RIG*M.rpGtilde;
rbItilde = RIG*M.rbGtilde; rbItildeu = rbItilde/norm(rbItilde);

%----- Initialize state estimate on first call to this function
if(isempty(xBark))
  vIMat = [rbItildeu'; e3']; vBMat = [rbBu'; e3']; aVec = ones(2,1);
  RBIBark = wahbaSolver(aVec,vIMat,vBMat);
  rIBark = rpItilde - RBIBark'*rpB;
  xBark = [rIBark; zeros(12,1)];
  QbaSteadyState = P.sensorParams.Qa2/(1 - P.sensorParams.alphaa^2);
  QbgSteadyState = P.sensorParams.Qg2/(1 - P.sensorParams.alphag^2);
  PBark = diag([2*diag(P.sensorParams.RpL);
                0.001*ones(3,1); ...
                2*P.sensorParams.sigmab^2*ones(3,1); ...
                diag(QbaSteadyState); diag(QbgSteadyState)]);
end

%----- Assemble measurements
zk = [rpItilde; rbItildeu];
RcCCellArray = {}; jj = 1; Nfk = 0;
[Nf,~] = size(M.rxMat);
mcVeck = zeros(Nf,1);
for ii=1:Nf
  if(~isnan(M.rxMat(ii,1)))
    mcVeck(ii) = 1;
    viCtilde = [P.sensorParams.pixelSize*M.rxMat(ii,:)';P.sensorParams.f];
    norm_viCtilde = norm(viCtilde);
    viCtildeu = viCtilde/norm_viCtilde;
    % viCtildeu is the measured unit vector pointing from the camera (C) frame
    % origin to the 3D feature point, expressed in C.
    zk = [zk; viCtildeu];
    % Error covariance matrix for the unit vector viCtildeu
    sigmac = sqrt(P.sensorParams.Rc(1,1))*P.sensorParams.pixelSize/norm_viCtilde;
    RcC = sigmac^2*(eye(3) - viCtildeu*viCtildeu') + epsilon*eye(3);
    RcCCellArray{jj} = RcC;
    jj = jj + 1;
    Nfk = Nfk + 1;
  end
end

%----- Perform measurement update
% Form measurement error covariance matrix Rk
RpI = P.sensorParams.RpL;
rbIu = RBIBark'*rbBu;
RbI = P.sensorParams.sigmab^2*(eye(3)-rbIu*rbIu') + epsilon*eye(3);
Rk = blkdiag(RpI,RbI);
for ii=1:Nfk
  Rk = blkdiag(Rk,RcCCellArray{ii});
end
nz = length(zk);
lambda_u = alphaUKF^2*(kappaUKF + nx + nz) - nx - nz;
c_u = sqrt(nx+nz+lambda_u);
Wm0_u = lambda_u/(nx + nz + lambda_u);
Wmi_u = 1/(2*(nx + nz + lambda_u));
Wc0_u = Wm0_u + 1 - alphaUKF^2 + betaUKF;
Wci_u = Wmi_u;
% Form augmented a priori state and error covariance matrix
xBarAugk = [xBark; zeros(nz,1)];
PBarAugk = blkdiag(PBark,Rk);
SxBar = chol(PBarAugk)';
% Assemble sigma points and push these through the measurement function
sp0 = xBarAugk;
spMat = zeros(nx+nz, 2*(nx+nz));  zpMat = zeros(nz,2*(nx+nz));
zp0 = h_meas(sp0(1:nx),sp0(nx+1:end),RBIBark,S.rXIMat,mcVeck,P);
for ii=1:2*(nx+nz)
  jj = ii; pm = 1;
  if(ii > (nx + nz)) jj = ii - nx - nz; pm = -1; end
  spMat(:,ii) = sp0 + pm*c_u*SxBar(:,jj);
  zpMat(:,ii) =
h_meas(spMat(1:nx,ii),spMat(nx+1:end,ii),RBIBark,S.rXIMat,mcVeck,P);
end
% Recombine sigma points
zBark = sum([Wm0_u*zp0, Wmi_u*zpMat],2);
Pzz = Wc0_u*(zp0 - zBark)*(zp0 - zBark)';
Pxz = Wc0_u*(sp0(1:nx) - xBark)*(zp0 - zBark)';
for ii=1:2*(nx+nz)
  Pzz = Pzz + Wci_u*(zpMat(:,ii) - zBark)*(zpMat(:,ii) - zBark)';
  Pxz = Pxz + Wci_u*(spMat(1:nx,ii) - xBark)*(zpMat(:,ii) - zBark)';
end
% Perform LMMSE measurement update
```

```
PzzInv = inv(Pzz);
xHatk = xBark + Pxz*PzzInv*(zk - zBark);
Pk = PBark - Pxz*PzzInv*Pxz';

%----- Package output state
statek.rI = xHatk(1:3); statek.vI = xHatk(4:6); ek = xHatk(7:9);
statek.RBI = euler2dcm(ek)*RBIBark;
statek.ba = xHatk(10:12); statek.bg = xHatk(13:15);
statek.omegaB = M.omegaBtilde - statek.bg;
PkDiag = diag(Pk);
E.statek = statek;
E.Pk = Pk;

if(0)
% Testing section
tk = M.tk
[statek.rI - statekTrue.rI]
[statek.vI - statekTrue.vI]
dcm2euler(statekTrue.RBI' * statek.RBI)*180/pi
sqrt(PkDiag(7:9))*180/pi
pause;
clc;
end

%----- Propagate state to time tkp1
RBIHatk = statek.RBI; xHatk(7:9) = zeros(3,1);
Qk = blkdiag(P.sensorParams.Qg,P.sensorParams.Qg2,...
  P.sensorParams.Qa,P.sensorParams.Qa2);
xHatAugk = [xHatk; zeros(nv,1)];
PAugk = blkdiag(Pk,Qk);
Sx = chol(PAugk)';
% Assemble sigma points and push these through the dynamics function
sp0 = xHatAugk;
xpMat = zeros(nx,2*(nx+nv));
uk = [M.omegaBtilde;M.ftildeB];
xp0 = f_dynamics(sp0(1:nx),uk,sp0(nx+1:end),S.delt,RBIHatk,P);
for ii=1:2*(nx+nv)
  jj = ii; pm = 1;
  if(ii > (nx + nv)) jj = ii - nx - nv; pm = -1; end
  spii = sp0 + pm*c_p*Sx(:,jj);
  xpMat(:,ii) = f_dynamics(spii(1:nx),uk,spii(nx+1:end),S.delt,RBIHatk,P);
end
% Recombine sigma points
xBarkp1 = sum([Wm0_p*xp0, Wmi_p*xpMat],2);
PBarkp1 = Wc0_p*(xp0 - xBarkp1)*(xp0 - xBarkp1)';
for ii=1:2*(nx+nv)
  PBarkp1 = PBarkp1 + ...
            Wci_p*(xpMat(:,ii) - xBarkp1)*(xpMat(:,ii) - xBarkp1)';
```

```
end
ekp1 = xBarkp1(7:9);
RBIBarkp1 = euler2dcm(ekp1)*RBIHatk;
xBarkp1(7:9) = zeros(3,1);
% Set k = kp1 in preparation for next iteration
RBIBark = RBIBarkp1; xBark = xBarkp1; PBark = PBarkp1;
```

Figure 18: Unscented Kalman filter state estimator function

Unset

```
function [RBI] = wahbaSolver(aVec,vIMat,vBMat)
% wahbaSolver : Solves Wahba's problem via SVD.  In other words, this
%               function finds the rotation matrix RBI that minimizes the
%               cost Jw:
%
%                     N
%    Jw(RBI) = (1/2) sum ai*||viB - RBI*viI||^2
%                    i=1
%
%
% INPUTS
%
% aVec ------- Nx1 vector of least-squares weights.  aVec(i) is the weight
%              corresponding to the ith pair of vectors
%
% vIMat ------ Nx3 matrix of 3x1 unit vectors expressed in the I frame.
%              vIMat(i,:)' is the ith 3x1 vector.
%
% vBMat ------ Nx3 matrix of 3x1 unit vectors expressed in the B
%              frame. vBMat(i,:)' is the ith 3x1 vector, which corresponds to
%              vIMat(i,:)';
%
% OUTPUTS
%
% RBI -------- 3x3 direction cosine matrix indicating the attitude of the
%              B frame relative to the I frame.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+

% Find B
B = zeros(3);
```

```
for i=1:length(aVec)
    a = aVec(i);
    V = vIMat(i,:)';
    U = vBMat(i,:)';
    B = B + a.*U*V';
end

% Find U and V from B = USV' given B
[U,S,V] = svd(B);

% Initilize M
M = [1 0 0; 0 1 0; 0 0 det(U)*det(V)];

% Solve for
RBI = U*M*V';
```

Figure 19: Wahba's Problem solver function

Unset

```
% Create a random euler angle vector
e = [0, pi/2, 0]';

% Create the resulting RBI
RBI = euler2dcm(e);

% Initializing weights
aVec = ones(3);

%% No Noise Test
% Create random body frame unit vectors for function input
VB1 = rand(3,1); % 3X1
VB1 = VB1/norm(VB1);
VB2 = rand(3,1);
VB2 = VB2/norm(VB2);
VB3 = rand(3,1);
VB3 = VB3/norm(VB3);

% Calculate I frame vectors from body frame rotation using RBI
VI1 = RBI'*VB1; % 3x1
VI2 = RBI'*VB2;
VI3 = RBI'*VB3;

% Using predefined vectors to create VIMat and VBMat
VBMat = [VB1'; VB2'; VB3'];
VIMat = [VI1'; VI2'; VI3'];
```

```
% Attempt Wahba solver to get original RBI matrix
RBItest = wahbaSolver(aVec, VIMat, VBMat);

% disp(VBMat) % To show that the matrix values are changing
disp(RBItest)
disp(RBI)

%% Noise Test
% Create random body frame unit vectors for function input
VB1 = rand(3,1); % 3X1
VB1 = VB1/norm(VB1);
VB2 = rand(3,1);
VB2 = VB2/norm(VB2);
VB3 = rand(3,1);
VB3 = VB3/norm(VB3);

% Calculate I frame vectors from body frame rotation using RBI
VI1 = RBI'*VB1; % 3x1
VI2 = RBI'*VB2;
VI3 = RBI'*VB3;

% Add noise to the inertial frame vectors, after transformation
noiseVector1 = 0 + (0.5-0).*rand(3,1);
noiseVector2 = 0 + (0.5-0).*rand(3,1);
noiseVector3 = 0 + (0.5-0).*rand(3,1);
VI1 = VI1 + noiseVector1;
VI2 = VI2 + noiseVector2;
VI3 = VI3 + noiseVector3;

% Using predefined vectors to create VIMat and VBMat
VBMat = [VB1'; VB2'; VB3']; % NX3
VIMat = [VI1'; VI2'; VI3'];

% Attempt Wahba solver to get original RBI matrix
RBItest = wahbaSolver(aVec, VIMat, VBMat);

disp(RBItest)
disp(RBI)
```

Figure 20: Wahba’s Problem solver testing script