# ASE 479W Laboratory 1 Report
# Simulating Quadrotor Dynamics

Aaron Pandian

January 26, 2024

## 1 Introduction

Quadcopters are an emerging technology for Unmanned Aerial Vehicle (UAV) implementation. The dissertation *Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment* mentions "surveillance, search and rescue, and mobile sensor networks" to be growing applications. To better understand quadrotor control design, a simple simulation using translational and attitude dynamics can be found on (1). However, Euler's concept alone fails to account for effects such as aerodynamic drag, motor response rates, propeller flexure, and rotor angular momentum. Compiling a simulator based on these principles can model a horizon period at higher fidelity. These concepts prelude the expanding development of quadrotor control design.

For this laboratory assignment, a simplified quadcopter dynamics simulator was designed from the theory of Newtonian dynamics and implemented in Matlab code. Fabricated in modeling, testing, and control steps- a quadrotor dynamics equation was simulated through Runge Kutta numerical integration and experimented to operate a specified control maneuver. The concept, implementation, and experiment results are discussed in the following report.

## 2 Theoretical Analysis

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

### 2.1 Relating Rotation Matrix to Rotation Angle

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

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

The $I_{3\times3}$ represents an identity matrix and $[\hat{\boldsymbol{a}}\times]$ denotes the skew-symmetric cross product equivalent, which is a varied notation of a cross product operation depicted in (2).

$$u \times v = [u\times]v \tag{2}$$

The profound result of (1) states that every rotation can be expressed as a rotation about a *single* axis, or the eigenvector of rotation $\hat{\boldsymbol{a}}$ denoted above. Expanding the general formula above produces the proper orthogonal matrix below:

$$R(\hat{a}, \phi) = \begin{bmatrix} cos\phi + \hat{a}_x^2(1 - cos\phi) & \hat{a}_x\hat{a}_y(1 - cos\phi) - \hat{a}_z sin\phi & \hat{a}_x\hat{a}_z(1 - cos\phi) + \hat{a}_y sin\phi \\ \hat{a}_y\hat{a}_x(1 - cos\phi) + \hat{a}_z sin\phi & cos\phi + \hat{a}_y^2(1 - cos\phi) & \hat{a}_y\hat{a}_z(1 - cos\phi) - \hat{a}_x sin\phi \\ \hat{a}_z\hat{a}_x(1 - cos\phi) - \hat{a}_y sin\phi & \hat{a}_z\hat{a}_y(1 - cos\phi) + \hat{a}_x sin\phi & cos\phi + \hat{a}_z^2(1 - cos\phi) \end{bmatrix} \quad (3)$$

The matrix present in (3) can be verified as proper orthogonal by the realization (4).

$$\det(R(\hat{\boldsymbol{a}}, \phi)) = 1 \quad (4)$$

Furthermore, an important concept to note is the trace of matrix $R(\hat{\boldsymbol{a}}, \phi)$. The trace can be defined as the sum of all diagonal elements. From (1), the term sin $\phi$ $[\hat{\boldsymbol{a}}\times]$ is skew-symmetric, as mentioned earlier. Thus the trace of this term equates to zero since the trace of any skew-symmetric matrix equals zero. The derivation of (5) in (5.1) shows that the total rotation angle ϕ can be related to the trace of R.

$$\text{tr}(R) = 1 + 2\cos\phi \quad (5)$$

$$\text{tr}(\text{R}) = \text{tr}(\cos(\phi)I_{3\times3}) + \text{tr}((1 - \cos\phi)(\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T)) = (3)\cos(\phi) + (1 - \cos\phi)(1) - (0) \quad (5.1)$$

This derivation is key to understanding the rotation transformations present in quadrotor maneuvers, which will all be done about an axis vector using a rotation matrix R.

## 2.2 Time Derivative of Rotation matrix

Another integral component of quadrotor analysis is finding an algorithm for how the rotation matrix varies through time. The derivative of the rotation matrix provides information on how the body frame relates to the inertial frame across a timestep. For evaluation, certain variables need to be defined prior. Thus, given a quadcopter spinning at a constant rate ω about an axis **â** fixed in the body frame, the angle of rotation, the body-frame-referenced angular rate vector, and the time-varying rotation matrix can be shown in (6), (7), and (8) respectively.

$$\phi(t) = \omega t \quad (6)$$

$$\dot{\phi} = \boldsymbol{\omega} = \omega\hat{\boldsymbol{a}} \quad (7)$$

$$C(t) = R(\hat{\boldsymbol{a}}, \phi(t)) \quad (8)$$

The angle of rotation in (6) corresponds to its time derivative $\dot{\phi}$ in (7). Furthermore, the angle of rotation function in (6) is used to represent C(t) in (8) as a time-varying rotation matrix. By denoting these values as they vary with time, the calculation of how these values vary with respect to time, or the rate of change, is enabled. This concept was already utilized in (7) to represent the angular speed at the rate of change of the angle of rotation.

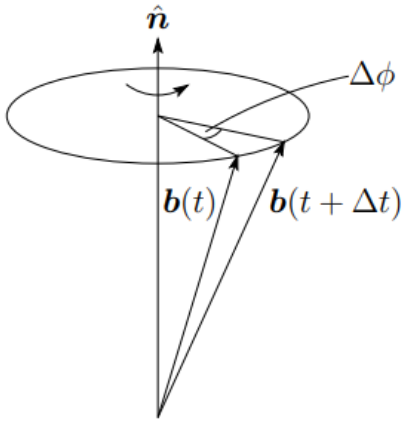


Figure 1: Rigid body rotating around unit vector $\hat{\mathbf{n}}$

Now given the physical system in Fig. 1, note that $\mathbf{b}(t)$ is a vector fixed in the body frame and $\Delta\phi$ is the angle through which the body rotates about $\hat{\mathbf{n}}$ in time $\Delta t$. Using (7) the following equation can be identified:

$$\dot{\boldsymbol{b}}(t) = \omega \times \boldsymbol{b}(t) \tag{9}$$

$$\Delta\boldsymbol{b} \triangleq \boldsymbol{b}(t+\Delta t) - \boldsymbol{b}(t) = \Delta\phi(\hat{\boldsymbol{n}} \times \boldsymbol{b}) \tag{9.1}$$

The derivation in (9.1) allows the $\Delta\phi\hat{\mathbf{n}}$ term to equate to the angular rate $\omega$, and, in utilizing the concept of a derivative mentioned before, the term $\Delta\mathbf{b}$ could change to $\dot{\mathbf{b}}(t)$, representing the value continuously. Now utilizing (8) to transform b(t) from the body to the inertial frame we arrive at (10).

$$\boldsymbol{b}(t)_B = C(t)\boldsymbol{b}(t)_I \tag{10}$$

Then by differentiating both sides, we obtain the subsequent derivation to find the kinematic equation for C(t). It is important to realize that $\dot{\boldsymbol{b}}(t)_B = 0$ because the vector is already fixed in the body frame.

$$\frac{d}{dt}C(t) = -([\omega_B \times]C(t)) \tag{11}$$

$$0 = \dot{\boldsymbol{b}}(t)_B = \dot{C}(t)\boldsymbol{b}(t)_I + C(t)\dot{\boldsymbol{b}}(t)_I \tag{11.1}$$

$$0 = \dot{C}(t)\boldsymbol{b}(t)_I + C(\omega_I \times \boldsymbol{b}(t)_I) \tag{11.2}$$

$$0 = \dot{C}(t)\boldsymbol{b}(t)_I + (\omega_B \times C(t))\boldsymbol{b}(t)_I \tag{11.3}$$

Due to the fact that C(t) is proper orthogonal, it can be distributed across the cross product seen in (11.2). Doing so allows the substitution of $\omega_B$ for $C(t)\omega_I$ which represents the same transformation as in (10). The equation in (11.3) must hold for any vector $\boldsymbol{b}(t)_I$ found in the inertial frame, thus simplifying to the time-varying rotation matrix equation (11).

This formula, however, can be derived in a much simpler manner using (1). Implementing the simplification (8), the Euler equation can be transformed to the following:

$$C(t) = \cos(\phi(t))I_{3\times3} + (1 - \cos(\phi(t))\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T - \sin(\phi(t))[\hat{\boldsymbol{a}}\times] \tag{12}$$

Paying heed to the identity $\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T = I_{3\times3} + [\hat{\boldsymbol{a}}\times]^2$, the ensuing derivation can take place.

$$\frac{d}{dt}C(t) = -\sin(\phi(t))\frac{d\phi}{dt}I_{3\times3} + \sin(\phi(t))\frac{d\phi}{dt}\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T - \cos(\phi(t))\frac{d\phi}{dt}[\hat{\boldsymbol{a}}\times] \tag{12.1}$$

Implementing the identity and factoring out the angular rate of change as $\omega$.

$$= -\omega(\sin(\phi(t))I_{3\times3} - \sin(\phi(t))(I_{3\times3} + [\hat{\boldsymbol{a}}\times]^2) + \cos(\phi(t))[\hat{\boldsymbol{a}}\times]) \tag{12.2}$$

$$= -\omega(\sin(\phi(t))I_{3\times3} - \sin(\phi(t)I_{3\times3} - \sin(\phi(t)[\hat{\boldsymbol{a}}\times]^2 + \cos(\phi(t))[\hat{\boldsymbol{a}}\times]) \tag{12.3}$$

$$= -[\omega\times](-\sin(\phi(t)[\hat{\boldsymbol{a}}\times]^2 + \cos(\phi(t))[\hat{\boldsymbol{a}}\times]) \tag{12.4}$$

Upon reaching (12.4) in the derivation, to achieve (11) the $(1 - \cos(\phi(t))\hat{\mathbf{a}}\hat{\mathbf{a}}^T$ term must be analyzed. By theoretically removing the term from inside the factorization in (12), the resulting term is (12.5).

$$\frac{d\phi}{dt}[\hat{\boldsymbol{a}}\times](1 - \cos(\phi(t))\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T \tag{12.5}$$

Where $[\hat{\boldsymbol{a}}\times]\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T = 0$, causing the term to equate to zero as well. With this, equation (12.4) is the same by adding the term, allowing the simplification below.

$$= -[\omega\times](-\sin(\phi(t)[\hat{\boldsymbol{a}}\times]^2 + (1 - \cos(\phi(t))\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T + \cos(\phi(t))[\hat{\boldsymbol{a}}\times]) \tag{12.6}$$

Whereby equation (12.6) equals the end formulation (11) by substituting (12) for the term after $-[\omega\times]$.

# 3 Implementation

To implement the 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 algorithm for the quadrotor vehicle.

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

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

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

end
```

Figure 2: Cross product equivalent function

The application above is an expansion of (2) by outputting the cross-product-equivalent matrix uCross for an arbitrary 3-by-1 vector u. This function eases evaluating the cross product between two matrices, finding the vector that is perpendicular to both, which is critical when working with multiple directions and reference frames.

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

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

end
```

Figure 3: Rotation matrix function

To complicate the previous function, a rotation matrix function was developed. This function generates a rotation matrix through an angle about a specified axis. This function is a direct application of (3).

By utilizing the script in Fig. 3, 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. This specific rotation sequence can be simplified using (13) and (3).

```
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 4: Euler angles to rotation matrix

The formula in Fig. 4 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 previously shown in (10). The rotation matrices R1, R2, and R3, are expressions of (1) for the specified axis. As a result, similar properties follow, such as the trace of all three matrices verify (5). 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 5: Rotation matrix to euler angles

The function in Fig. 5 does the inverse operation from Fig. 4, 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 model is shown in Fig. 5.

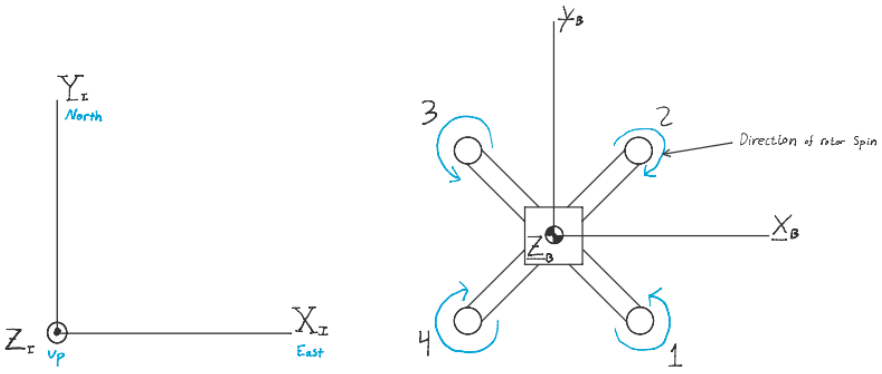


Figure 5: Quadcopter visualization

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 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 matrix is as defined in Fig. 3 and $\omega_B$ is the angular velocity in the body frame. Considering the 3-dimensional size of each input matrix, the state matrix is a 12x1 matrix with all variables listed out- instead of demarketing four input matrices within the state matrix. To find the derivative of the state $\dot{X}$, the following equations will need to be computed for each state input.

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

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

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

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

Functions (15) and (18) are explicit in that (15) simply defines the rate of change of position as the velocity and (18) models the derivation of (11). The latter equations are translational and rotational applications of Newton's second law of motion. Obscure terms are better understood as $F_{iI}$ equates to the force applied by rotor one through four

The formula in (17) is defined because the euler angles from the initial state matrix will be transformed, using the function in Fig. 3, to the equivalent rotation matrix. This is done to avoid the singularity error spurred by working with euler angles. Thus, the derivative state vector is formed, where e is replaced with the elements of $R_{BI}$ (unrolled column by column) to provide a now 18x1 state matrix. An overview of the process can be seen in Fig. 6.

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

% Finding rI_dot
rI_dot = vI;

% Finding vI_dot
FzI = [0 0 -m*g]';
F1 = [0 0 KF(1,1)*((wi(1,1))^2)]'; % Zeroed to the z-axis due to formula
.
.
rotor_force_term = F1 + F2 + F3 + F4;
vI_dot = (1/m) * (FzI +  (transpose(RBI)*rotor_force_term) + dI);
```

```
% Finding w_dot
r1 = [ri(1,1) ri(2,1) ri(3,1)]';
.
.
N1 = -1*[0 0 KN(1)*((wi(1,1))^2)]'; % The 1 and 3 propellor have negative values
N2 =  1*[0 0 KN(2)*((wi(2,1))^2)]'; % The 2 and 4 propellor have positive values
.
.
NB1 = N1 + crossProductEquivalent(r1)*F1;
.
.
NB = NB1 + NB2 + NB3 + NB4;
w_dot = (inv(J)) * (NB - crossProductEquivalent(wB)*J*wB);

% Finding RBI_dot
RBI_dot = -(crossProductEquivalent(wB))*RBI;
```

Figure 6: Quadrotor ordinary differential equation function

The output of this function is a state vector, like (14), in construct, but represents its derivative $\dot{X}$.

Conclusively, the Runge Kutta ordinary differential equation solver in Matlab supports the simulation of the model above, as mentioned before. Given an arbitrary initial state variable, the simulator allows for visualization of the quadrotor dynamics across a preset sample time. The basic implementation is presented in Fig. 7.

```
Unset
function [P] = simulateQuadrotorDynamics(S)

% oversampFact is the oversampling factor
N = length(tVec);
dtIn = tVec(2) - tVec(1);
dtOut = dtIn/oversampFact;
% Create empty storage vectors
tVecOut = [];
XMat = [];

% Set initial state
Xk = X0;

% Starting the ode45 method
for k = 1:N-1
   distVec = distMat(k,:)';
   omegaVec = [omegaMat(k,1) omegaMat(k,2) omegaMat(k,3) omegaMat(k,4)]';
```

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

    [tVeck,XMatk] = ode45(@(t,X)quadOdeFunction(t,X,omegaVec,distVec,S),tspan,Xk);

    tVecOut = [tVecOut; tVeck(1:end-1)];
    XMat = [XMat; XMatk(1:end-1,:)];

    Xk = XMatk(end,:)';
end

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

Figure 7: Runge Kutta simulated numerical solver

This function outputs a structure containing matrix values to define all states through every iteration of time, till the defined sampling time. With this, the aforementioned visualization is processed. Moreover, this simulator is employed to assess the control application in this experiment. To do so, a structure is created to enforce a level circle made by the quadcopter in said simulation. A protocol was created to automate testing values, which can be seen in Fig. 8.

Unset

```
r = [0.5 0 0.5]'; % initial position
v = [0 -1 0]'; % arbitrary value, must always be in the Y direction only
radius = 1; % arbitrary value

% find euler angles
forceRatio = -(m*(v(2)^2))/(m*g*radius);
theta = atan(forceRatio);

% find wB matrix
yawRate = (2*pi)/((2*pi*radius)/v(2));
angularRates = [0 0 yawRate]';
conRotOmegaMat = [cos(phi)*cos(theta) 0 cos(phi)*cos(theta);
                  sin(phi)*sin(theta) cos(phi) -cos(theta)*sin(phi);
                  -sin(theta) 0 cos(theta)];
wB = conRotOmegaMat*angularRates;

% find NB
NB = crossProductEquivalent(wB)*J*wB;

% find w12 using w34, where wb = w34 is less than wa = w12
wb = % random input for testing
appliedForce = -(m*(v(2)^2))/(radius*sin(theta));
Fz = appliedForce*cos(theta);
Fzwa = 2*kF*(wb^2)*cos(theta);
Fzwb = Fz - Fzwa;
```

```
wa = sqrt(Fzwb/(2*kF*cos(theta)));
```

Figure 8: Quadrotor circular flight control structure derivation function

From the function above, a circular flight path controlled from just an initial state can be derived. The method applied in the function can primarily relate to equations (15) through (18). Additionally, to find the angular rate in the body frame given a calculated angular rate vector, calculate the sum of the angular rates contributed by each of the Euler angles. This relationship is shown in (19).

$$\omega_B = \begin{bmatrix} c\theta & 0 & -s\theta c\phi \\ 0 & 1 & s\phi \\ s\theta & 0 & c\theta c\phi \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{19}$$

These functions set up the foundation for conducting dynamical analysis on a quadcopter for various initial conditions. A more detailed overview of the input structure development to incur a level circle flight is provided in the following section.

# 4 Results and Analysis

In this section, the results of the simulation experiment are discussed. Furthermore, the implemented code is tested and validated against test sets, provided or derived.

Prior to conducting an analysis on the quadrotor simulation, there are two checkpoints for which the code must pass. A simple error in trigonometric calculation when moving between euler angles and rotation matrices can cause a catastrophic shift in simulated data. As such, below is verification that the functions shown in Fig. 4 and 5 do not misconstrue each other.

| Input Euler Angles | euler2dcm() Output | dcm2euler() Output |
|---|---|---|
| [0 0.5 0.2] | [0.8601 0.1743 -0.4794<br>-0.1987 0.9801 0<br>0.4699 0.0952 0.8776] | [0 0.5 0.2] |
| [π/6 0 -π/3] | [0.5000 -0.8660 0<br>0.7500 0.4330 0.5000<br>-0.4330 -0.2500 0.8660] | [π/6 0 -π/3] |

Table 1: Euler to rotation matrix to euler angle cross-check

To further verify the accurate rotation of matrices and vectors from one frame to another, another step to ensure the euler2dcm function is shown in Table 2. Here, the rotationMatrix function, depicted in Fig. 3, is

used to mimic the 3-1-2 rotation sequence calculated in euler2dcm. Both methods of deriving the direction cosine matrix are provided with the same input euler angle vector.

| Input Euler Angles | euler2dcm() Output | rotationMatrix() Method Output |
|---|---|---|
| [0 0.5 0.2] | [0.8601 0.1743 -0.4794<br>-0.1987 0.9801 0<br>0.4699 0.0952 0.8776] | [0.8601 0.1743 -0.4794<br>-0.1987 0.9801 0<br>0.4699 0.0952 0.8776] |
| [π/6 0 -π/3] | [0.5000 -0.8660 0<br>0.7500 0.4330 0.5000<br>-0.4330 -0.2500 0.8660] | [0.5000 -0.8660 0<br>0.7500 0.4330 0.5000<br>-0.4330 -0.2500 0.8660] |

Table 2: Function verification using individual calculation

The simple code schematic used to conduct the test in Table 2 is presented in Fig. 9.

```
Unset
% Compare matrix with matrixp
e = % a random 3 by 1 matrix
matrix = euler2dcm(e);
R2 = rotationMatrix([0 1 0]',e(2));
R1 = rotationMatrix([1 0 0]',e(1));
R3 = rotationMatrix([0 0 1]',e(3));
matrixp = R2*R1*R3;
```

Figure 9: Rotation matrix calculation using various methods

By understanding the accuracy of the functions developed, creating the model to simulate the quadrotor parameters as a function of time becomes simpler. Now implementing the full code, featured in the Appendix Fig. 5 and 6, a test structure was provided to certify the output of the simulation was working as it should. To conduct this test, a random number was generated between 1 and 7991. These represent all indexes of the simulated output state matrices. Once the number was generated, the value of all state matrices were collected at that index, as well as five indices ahead. These values were then compared to the test structure state matrices at the same index. The presentation of said test is visible in the subsequent tables.

| Index Row | Representative State Euler Values | Test State Euler Values |
|---|---|---|
| 944 | [0.0184 0.0465 0.5768] | [0.0184 0.04649 0.577] |
| 945 | [0.0184 0.0465 0.5772] | [0.0184 0.04649 0.5772] |

| | | |
|---|---|---|
| 946 | [0.0184 0.0465 0.5776] | [0.0184 0.0465 0.5776] |
| 947 | [0.0185 0.0465 0.5780] | [0.0185 0.0465 0.5780] |
| 948 | [0.0185 0.0465 0.5784] | [0.0185 0.0465 0.5784] |

Table 3: Comparison of test and simulated euler angle values

It is apparent after looking at Table 3 that the fidelity of the simulation is daily high in reference to the provided test values. However, one aspect of state alone should not pass the system without checking other variables.

| Index Row | Representative State Position Values | Test State Position Values |
|---|---|---|
| 4471 | [-0.5917 0.2425 -0.0848] | [-0.5917 0.2425 -0.0848] |
| 4472 | [-0.5916 0.2426 -0.0849] | [-0.5916 0.2426 -0.0849] |
| 4473 | [-0.5915 0.2427 -0.0849] | [-0.5915 0.2427 -0.0849] |
| 4474 | [-0.5913 0.2428 -0.0850] | [-0.5913 0.2428 -0.0850] |
| 4475 | [-0.5912 0.2429 -0.0850] | [-0.5912 0.2429 -0.0850] |

Table 4: Comparison of test and simulated position vector values

| Index Row | Representative State Velocity Values | Test State Velocity Values |
|---|---|---|
| 2613 | [-0.1745 0.1268 -0.044] | [-0.1745 0.1268 -0.044] |
| 2614 | [-0.1742 0.1268 -0.044] | [-0.1742 0.1268 -0.044] |
| 2615 | [-0.1740 0.1269 -0.044] | [-0.1740 0.1268 -0.044] |
| 2616 | [-0.1738 0.1269 -0.044] | [-0.1738 0.1269 -0.044] |
| 2617 | [-0.1735 0.1270 -0.044] | [-0.1735 0.1270 -0.044] |

Table 5: Comparison of test and simulated velocity vector values

As evident from Table 4 and 5 as well, the simulated model meets the standards of accuracy this model is capable of achieving.

Once the simulation was verified as accurate, testing for the experiment control goal took place. The crux of the problem to have a quadrotor rotate about a circle on a fixed altitude plane comes down to the speeds of the rotors.

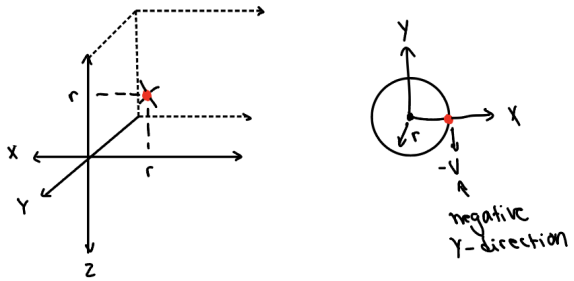


Figure 10: Quadrotor orientation for circular movement

Based on Fig. 10, the initial position of the rotorcraft is along the X-Z plane. This is done to simplify calculations as it can be assumed this is the point in the circular path where velocity aligns with the negative Y direction. Initializing an arbitrary position vector [r 0 r] for simplicity, where r is greater than zero, and velocity for the velocity vector [0 -v 0], it is possible to assess the forces acting on the UAV. Due to the fact that the only acting force in the Z axis direction is gravity, the following equation can be utilized.

$$F cos(\theta) = mg \tag{20}$$

$$F sin(\theta) = -(mv^2 r^{-1}) \tag{21}$$

Furthermore, the only acting force in the X axis direction is the necessary centripetal force to keep the quadcopter in circular motion. Thus, the two equations above are formed. Since all values are known except for the force F, a ratio can be created to solve for angle theta, which represents the pitch into the circle- as will be shown since the pitch is calculated to be negative.

Once these values are calculated, the next step is to calculate the angular rates for us in (19) to find the $\omega_B$ vector. For this experiment, a level altitude needs to be maintained. For that reason, there should be no change in roll or pitch from the initial state as either will cause the UAV to spin downwards. However, the yaw of the aircraft needs to vary with time to ensure the front, the side with rotor numbers 3 and 4 evident from Fig. 5, stays facing the center of the circle, or, in this case the origin. To find the rate of change of the yaw angle ψ, it is easiest to understand the value in terms of the period. The period is defined as the amount of time it takes to make one full revolution around a circle. Additionally, the length of a circle in radians is $2\pi$, thus formula (22) can be implemented to find the yaw angle rate of change.

$$(2\pi v)(2^{-1}\pi^{-1}r^{-1}) = v/r \tag{22}$$

The angular rates can be formed into a vector [0 0 v/r] and input into (19) to find $\omega_B$. With $\omega_B$ it is now possible to adjust for the body frame torques with (17). Noting that $\omega_B$ should be constant to maintain constant centripetal force, the omega dot term $\tilde{\dot{\omega}}_B$ equates to zero, allowing the evaluation of $N_B$.

Now given the body frame forces and torques for the necessary orientation to maintain a level and constant angular rate circular path, the derivation for the angular speeds of the rotor blades can be worked out. With the subsequent equations, the specific force and torque each rotor is responsible for can be determined.

$$F_i = k_F \omega_i^2 \tag{23}$$

$$N_i = k_N \omega_i^2 \tag{24}$$

Where in both (23) and (24), $i$ relates to the specific rotor and the constants $k_F$ and $k_N$ are provided. While solving for the $\omega_i$ values, it is important to keep in mind the orientation of the UAV. Due to the fact that rotors 3 and 4, according to Fig. 5, are those closest to the circle, the force output will be less than the rotors in the back, facing outside the circle. Furthermore, the rotors inside the circle should be near equal to prevent excess torque, and the same speaks for the outer rotors. While undergoing trial and error to test various combinations of rotor speeds, it is key to note that those force and torques generated by each propeller should match the applied forces derived earlier whilst adhering to the limitations just described.

After personal trial and error, the following figures depict the quadrotors center of mass on the X-Y plane and separately on the Z-axis with respect to time.

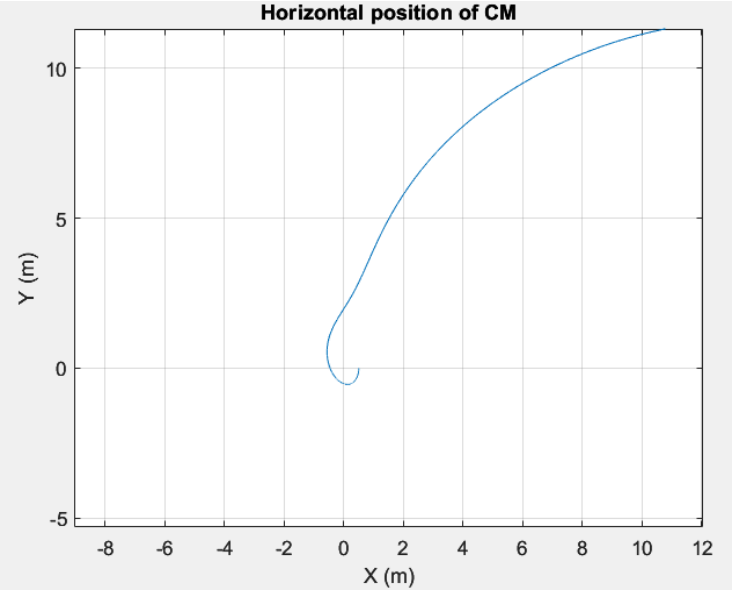


Figure 11: Quadrotor horizontal center of mass position across simulation

The final values used for the rotor speeds and initial states can be found in Appendix Fig. 9. Based on the trial and error conducted, the quadrotor for the schematic in this experiment completed three fourths of a circle before flying off trajectory. The center of mass likely gained additional torque in another direction since the circle created appears to be fairly unsymmetrical.

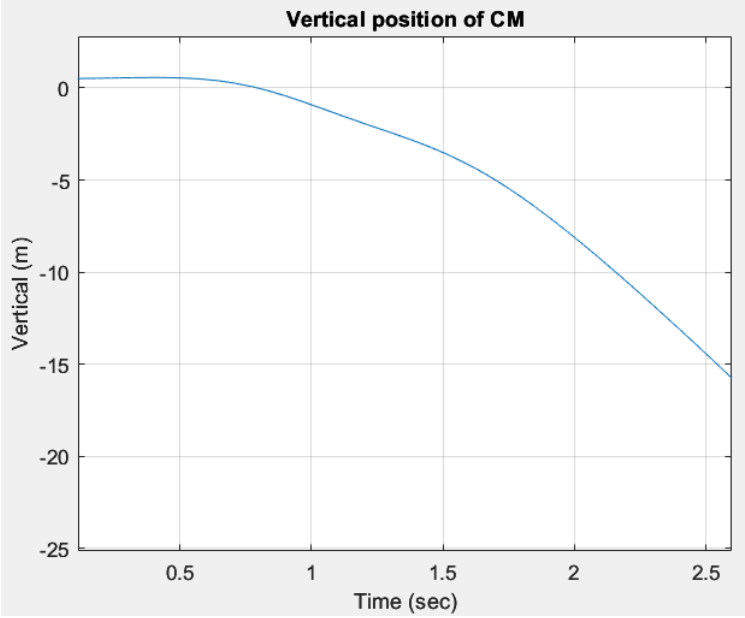

Figure 12: Quadrotor center of mass altitude across simulation

Furthermore, based on the information in Fig. 12, it is evident that the quadrotor began to spiral downward and away from the center after maintaining altitude for about 0.75 seconds. To accommodate for this variation from the theoretical approach, a more holistic testing of different rotor speed configurations would need to be conducted to find the sweet spot for the quadrotor to maintain altitude and thus continue on a symmetrical circular path.

# 5 Conclusion

An open-loop control schematic was developed to generate the rotor angular rates required to cause the simulated quadcopter to fly in a complete circle on a horizontal plane. The strategy is based on the ratio between the necessary force to oppose gravity and the induced centripetal force. By initializing an initial position and velocity, this force fraction enables the calculation of quadrotor pitch towards the center of the circle. Consequently, the angular rate of the vehicle in the body frame can lead to the body torques in each axis. Using the necessary torques and forces as references, the controller now has enough information to vary rotor speeds until those conditions are satisfied. As it turns out, the ideal scenario is when the two rotors facing into the circle have rotor speeds that are less than the two facing out of the circle; however, this variation should be minimized to obtain a steady flight path.

## References


[1] Hoffmann, G., Huang, H., Waslander, S., & Tomlin, C. (n.d.). *Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment*. Stanford. https://ai.stanford.edu/~gabeh/papers/Quadrotor_Dynamics_GNC07.pdf

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

[3] Hammen, D., & Selene Routley, S. (1961, July 1). *Rotation matrix of Euler's equations of rotation relative to inertial reference frame. Physics Stack Exchange*. https://physics.stackexchange.com/questions/196769/rotation-matrix-of-eulers-equations-of-rotation-relative-to-inertial-reference

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

## Appendix

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

```
Unset
function [uCross] = crossProductEquivalent(u)
% crossProductEquivalent : Outputs the cross-product-equivalent matrix uCross
% such that for arbitrary 3-by-1 vectors u and v,
% cross(u,v) = uCross*v.
%
% INPUTS
%
% u ---------- 3-by-1 vector
%
%
% OUTPUTS
%
% uCross ----- 3-by-3 skew-symmetric cross-product equivalent matrix
%
%+-------------------------------------------------------------------------------+
% References: None
%
%
% Author: Aaron Pandian
%+===============================================================================+
u1 = u(1,1); % extracted values from row 1, column 1
u2 = u(2,1);
u3 = u(3,1);
uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];
end
```

Figure 1: Cross vector equivalent function

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

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

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

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

Figure 2: Rotation matrix development function

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

```
% INPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
%
% OUTPUTS
%
% e ---------- 3-by-1 vector containing the Euler angles in radians: phi =
%              e(1), theta = e(2), and psi = e(3).  By convention, these
%              should be constrained to the following ranges: -pi/2 <= phi <=
%              pi/2, -pi <= theta < pi, -pi <= psi < pi.
%
%+--------------------------------------------------------------------------+
% References: None
%
%
% Author: Aaron Pandian
%+==========================================================================+

phi = asin(R_BW(2,3));
% assert() throws an error if condition is false
assert(sin(phi)~=-pi/2 && sin(phi)~=pi/2,'Conversion is singular.');
theta = atan2(-(R_BW(1,3)),R_BW(3,3));
if theta == pi
   theta = -pi;
end
psi = atan2(-(R_BW(2,1)),R_BW(2,2));
if psi == pi
   psi = -pi;
end
e = [phi; theta; psi]';
end
```

Figure 3: Direction cosine matrix to euler angles function

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

```
% a vector in B coordinates: vB = R_BW * vW
%
% INPUTS
%
% e ---------- 3-by-1 vector containing the Euler angles in radians: phi =
%              e(1), theta = e(2), and psi = e(3)
%
% OUTPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
%+-------------------------------------------------------------------------------+
% References: Attitude Transformations. VectorNav. (n.d.).
%https://www.vectornav.com/resources/inertial-navigation-primer/math-fundamentals/
%math-attitudetran
%
% Author: Aaron Pandian
%+===============================================================================+

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

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

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

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

Figure 4: Euler angles to direction cosine matrix

Unset

```
function [Xdot] = quadOdeFunction(t,X,omegaVec,distVec,P)
% quadOdeFunction : Ordinary differential equation function that models
%                   quadrotor dynamics.  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']'
%
%              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 constant rotor speed 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: Aaron Pandian
%+================================================================================+

% Initializing matrices and variables
m = P.quadParams.m;
g = P.constants.g;
KF = P.quadParams.kF;
KN = P.quadParams.kN;
J = P.quadParams.Jq;
ri = P.quadParams.rotor_loc; % 3 X 4
dI = distVec; % 3 x 1
rI = [X(1,1) X(2,1) X(3,1)]'; % 3 x 1, not needed for subsequent calculations
vI = [X(4,1) X(5,1) X(6,1)]'; % 3 x 1
RBI = [X(7,1) X(10,1) X(13,1); % 3 x 3
      X(8,1) X(11,1) X(14,1);
      X(9,1) X(12,1) X(15,1)];
wB = [X(16,1) X(17,1) X(18,1)]'; % 3 x 1
wi = omegaVec; % 4 x 1

% Finding rI_dot
rI_dot = vI;

% Finding vI_dot
FzI = [0 0 -m*g]';
F1 = [0 0 KF(1,1)*((wi(1,1))^2)]'; % Zeroed to the z-axis due to formula
F2 = [0 0 KF(2,1)*((wi(2,1))^2)]';
F3 = [0 0 KF(3,1)*((wi(3,1))^2)]';
F4 = [0 0 KF(4,1)*((wi(4,1))^2)]';
rotor_force_term = F1 + F2 + F3 + F4;
vI_dot = (1/m) * (FzI +  (transpose(RBI)*rotor_force_term) + dI);

% Finding w_dot
r1 = [ri(1,1) ri(2,1) ri(3,1)]'; % 3 X 1
r2 = [ri(1,2) ri(2,2) ri(3,2)]';
r3 = [ri(1,3) ri(2,3) ri(3,3)]';
r4 = [ri(1,4) ri(2,4) ri(3,4)]';
N1 = -1*[0 0 KN(1)*((wi(1,1))^2)]'; % The 1 and 3 propellor have negative values
N2 =  1*[0 0 KN(2)*((wi(2,1))^2)]'; % The 2 and 4 propellor have positive values
N3 = -1*[0 0 KN(3)*((wi(3,1))^2)]';
N4 =  1*[0 0 KN(4)*((wi(4,1))^2)]';
```

```
NB1 = N1 + crossProductEquivalent(r1)*F1;
NB2 = N2 + crossProductEquivalent(r2)*F2;
NB3 = N3 + crossProductEquivalent(r3)*F3;
NB4 = N4 + crossProductEquivalent(r4)*F4;
NB = NB1 + NB2 + NB3 + NB4;
w_dot = (inv(J)) * (NB - crossProductEquivalent(wB)*J*wB); % Replace inv() method

% Finding RBI_dot
RBI_dot = -(crossProductEquivalent(wB))*RBI;

% Initializing output
Xdot = [rI_dot(1,1) rI_dot(2,1) rI_dot(3,1) vI_dot(1,1) vI_dot(2,1) vI_dot(3,1)
RBI_dot(1,1) RBI_dot(2,1) RBI_dot(3,1) RBI_dot(1,2) RBI_dot(2,2) RBI_dot(3,2)
RBI_dot(1,3) RBI_dot(2,3) RBI_dot(3,3) w_dot(1,1) w_dot(2,1) w_dot(3,1)]';
```

Figure 5: Quadcopter ordinary differential equation function

```
Unset
function [P] = simulateQuadrotorDynamics(S)
% simulateQuadrotorDynamics : Simulates the dynamics of a quadrotor aircraft.
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%          tVec = Nx1 vector of uniformly-sampled time offsets from the
%                 initial time, in seconds, with tVec(1) = 0.
%
%  oversampFact = Oversampling factor. Let dtIn = tVec(2) - tVec(1). Then the
%                 output sample interval will be dtOut =
%                 dtIn/oversampFact. Must satisfy oversampFact >= 1.
%
%      omegaMat = (N-1)x4 matrix of rotor speed inputs.  omegaMat(k,j) >= 0 is
%                 the constant (zero-order-hold) rotor speed setpoint for the
%                 jth rotor over the interval from tVec(k) to tVec(k+1).
%
%        state0 = State of the quad at tVec(1) = 0, expressed as a structure
%                 with the following elements:
%
%                   r = 3x1 position in the world frame, in meters
%
%                   e = 3x1 vector of Euler angles, in radians, indicating the
%                       attitude
%
%                   v = 3x1 velocity with respect to the world frame and
```

```
%                       expressed in the world frame, in meters per second.
%
%              omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
%       distMat = (N-1)x3 matrix of disturbance forces acting on the quad's
%                 center of mass, expressed in Newtons in the world frame.
%                 distMat(k,:)' is the constant (zero-order-hold) 3x1
%                 disturbance vector acting on the quad from tVec(k) to
%                 tVec(k+1).
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
%
% OUTPUTS
%
% P ---------- Structure with the following elements:
%
%          tVec = Mx1 vector of output sample time points, in seconds, where
%                 P.tVec(1) = S.tVec(1), P.tVec(M) = S.tVec(N), and M =
%                 (N-1)*oversampFact + 1.
%
%
%         state = State of the quad at times in tVec, expressed as a structure
%                 with the following elements:
%
%                rMat = Mx3 matrix composed such that rMat(k,:)' is the 3x1
%                       position at tVec(k) in the world frame, in meters.
%
%                eMat = Mx3 matrix composed such that eMat(k,:)' is the 3x1
%                       vector of Euler angles at tVec(k), in radians,
%                       indicating the attitude.
%
%                vMat = Mx3 matrix composed such that vMat(k,:)' is the 3x1
%                       velocity at tVec(k) with respect to the world frame
%                       and expressed in the world frame, in meters per
%                       second.
%
%           omegaBMat = Mx3 matrix composed such that omegaBMat(k,:)' is the
%                       3x1 angular rate vector expressed in the body frame in
%                       radians, that applies at tVec(k).
%
%
```

```
%+--------------------------------------------------------------------------+
% References:
%
%
% Author: Aaron Pandian
%+==========================================================================+

% Initializing vectors
tVec = S.tVec;
oversampFact = S.oversampFact;
omegaMat = S.omegaMat;
state0 = S.state0;
distMat = S.distMat;
r = state0.r;
e = state0.e;
v = state0.v;
omegaB = state0.omegaB;

% counting all bases by the subsequent,for creating X0
rI = r;
vI = v;
RBI = euler2dcm(e); % transformation
X0 = [rI(1,1), rI(2,1), rI(3,1), vI(1,1), vI(2,1), vI(3,1), RBI(1,1), RBI(2,1),
RBI(3,1), RBI(1,2), RBI(2,2), RBI(3,2), RBI(1,3), RBI(2,3), RBI(3,3), omegaB(1,1),
omegaB(2,1), omegaB(3,1)]';

% Oversampling causes the ODE solver to produce output at a finer time
% resolution than dtIn. oversampFact is the oversampling factor.
N = length(tVec);
dtIn = tVec(2) - tVec(1);
dtOut = dtIn/oversampFact;

% Create empty storage vectors
tVecOut = [];
XMat = [];

% Set initial state
Xk = X0;

% Starting the ode45 method
for k = 1:N-1
    % Create instances of the  quad motor dynamics function parameters
   distVec = distMat(k,:)';
   omegaVec = [omegaMat(k,1) omegaMat(k,2) omegaMat(k,3) omegaMat(k,4)]';

   % Build the time vector for kth segment.  We oversample by a factor
   % oversampFact relative to the coarse timing of each segment.
   tspan = [tVec(k):dtOut:tVec(k+1)]';
```

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

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

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

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

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

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

Figure 6: Quadcopter numerical simulation function

```
Unset
quadParamsScript;
constantsScript;

r = [0.5 0 0.5]'; % not editable
v = [0 -1 0]'; % must always be in the Y direction only
radius = 1;
m = quadParams.m;
g = constants.g;

% set euler matrix
forceRatio = -(m*(v(2)^2))/(m*g*radius);
theta = atan(forceRatio);
phi = 0;
psi = 0;
e = [phi, theta, psi]; % e(1) = phi, e(2) = theta, e(3) = psi

% find wB matrix
yawRate = (2*pi)/((2*pi*radius)/v(2));
angularRates = [0 0 yawRate]'; % negative yaw for clockwise rotation around origin
conRotOmegaMat = [cos(phi)*cos(theta) 0 cos(phi)*cos(theta);
                  sin(phi)*sin(theta) cos(phi) -cos(theta)*sin(phi);
                  -sin(theta) 0 cos(theta)];
wB = conRotOmegaMat*angularRates;

% find NB
J = quadParams.Jq;
NB = crossProductEquivalent(wB)*J*wB;

% find w12 using w34, where wb = w34 is less than wa = w12
wb = 585;
kF = quadParams.kF(1);
appliedForce = -(m*(v(2)^2))/(radius*sin(theta)); % put negative because  %
centripetal in negative x dir

Fz = appliedForce*cos(theta);
Fx = appliedForce*sin(theta);
Fzwa = 2*kF*(wb^2)*cos(theta);
Fzwb = Fz - Fzwa;
wa = sqrt(Fzwb/(2*kF*cos(theta)));
fprintf('wB = %f\n',wB')
fprintf('e = %f\n',e')
fprintf('w12 = %f, w34 = %f\n',wa,wb)
Fxapproximation = 2*kF*(wa^2)*sin(theta) + 2*kF*(wb^2)*sin(theta);
fprintf('Fx = %f, Fxp = %f\n',Fx,Fxapproximation)
```

Figure 7: Finding circle structure parameters script

Unset

```
function P = visualizeQuad(S)
% visualizeQuad : Takes in an input structure S and visualizes the resulting
%                 3D motion in approximately real-time.  Outputs the data
%                 used to form the plot.
%
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%           rMat = 3xM matrix of quad positions, in meters
%
%           eMat = 3xM matrix of quad attitudes, in radians
%
%           tVec = Mx1 vector of times corresponding to each measurement in
%                  xevwMat
%
%  plotFrequency = The scalar number of frames of the plot per each second of
%                  input data.  Expressed in Hz.
%
%         bounds = 6x1, the 3d axis size vector
%
%    makeGifFlag = Boolean (if true, export the current plot to a .gif)
%
%    gifFileName = A string with the file name of the .gif if one is to be
%                  created.  Make sure to include the .gif exentsion.
%
%
% OUTPUTS
%
% P ---------- Structure with the following elements:
%
%          tPlot = Nx1 vector of time points used in the plot, sampled based
%                  on the frequency of plotFrequency
%
%          rPlot = 3xN vector of positions used to generate the plot, in
%                  meters.
%
%          ePlot = 3xN vector of attitudes used to generate the plot, in
%                  radians.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:  Nick Montalbano
%+==============================================================================+
 % Important params
```

```
figureNumber = 42; figure(figureNumber); clf;
fcounter = 0; %frame counter for gif maker
m = length(S.tVec);
% UT colors
burntOrangeUT = [191, 87, 0]/255;
darkGrayUT = [51, 63, 72]/255;
% Parameters for the rotors
rotorLocations=[0.105 0.105 -0.105 -0.105
   0.105 -0.105 0.105 -0.105
   0 0 0 0];
r_rotor = .062;
% Determines the location of the corners of the body box in the body frame,
% in meters
bpts=[ 120  120 -120 -120  120  120 -120 -120
   28  -28   28  -28   28  -28   28  -28
   20   20   20   20   -30   -30   -30   -30 ]*1e-3;
% Rectangles representing each side of the body box
b1 = [bpts(:,1) bpts(:,5) bpts(:,6) bpts(:,2) ];
b2 = [bpts(:,1) bpts(:,5) bpts(:,7) bpts(:,3) ];
b3 = [bpts(:,3) bpts(:,7) bpts(:,8) bpts(:,4) ];
b4 = [bpts(:,1) bpts(:,3) bpts(:,4) bpts(:,2) ];
b5 = [bpts(:,5) bpts(:,7) bpts(:,8) bpts(:,6) ];
b6 = [bpts(:,2) bpts(:,6) bpts(:,8) bpts(:,4) ];
% Create a circle for each rotor
t_circ=linspace(0,2*pi,20);
circpts=zeros(3,20);
for i=1:20
   circpts(:,i)=r_rotor*[cos(t_circ(i));sin(t_circ(i));0];
end
% Plot single epoch if m==1
if m==1
   figure(figureNumber);

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

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

```
       'color',darkGrayUT);
   hold on
   rotor3_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,3)*ones(1,20));
   rotor3plot = plot3(rotor3_circle(1,:), rotor3_circle(2,:),
rotor3_circle(3,:),...
       'color',darkGrayUT);
   hold on
   rotor4_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,4)*ones(1,20));
   rotor4plot = plot3(rotor4_circle(1,:), rotor4_circle(2,:),
rotor4_circle(3,:),...
       'color',darkGrayUT);

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

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

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

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

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

   % Interpolate to regularize times
```

```
   [t2unique, indUnique] = unique(S.tVec);
   rPlot = (interp1(t2unique, S.rMat(indUnique,:), tPlot))';
   ePlot = (interp1(t2unique, S.eMat(indUnique,:), tPlot))';

   figure(figureNumber);

   % Iterate through points
   for i=1:tPlotLen

       % Start timer
       tic

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

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

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

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

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

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

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

       % Translate, rotate, and plot body axes
```

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

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

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

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

P.tPlot = tPlot;
P.ePlot = ePlot;
```

```
   P.rPlot = rPlot;
end
end
```

Figure 8: Simulation visualization function

```
Unset
% Total simulation time, in seconds
Tsim = 4;
% Update interval, in seconds.  This value should be small relative to the
% shortest time constant of your system.
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
S.tVec = [0:N-1]'*delt;
% Matrix of disturbance forces acting on the body, in Newtons, expressed in I
S.distMat = zeros(N-1,3);
% Rotor speeds at each time, in rad/s
rotorSpeeds = [660 660 650 650];
S.omegaMat = ones(N-1,1)*rotorSpeeds;
% Initial position in m
S.state0.r = [.5 0 .5]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 -0.387523805780279 0]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 -2 0]';
% Initial angular rate of body with respect to I, expressed in B, in rad/s
S.state0.omegaB = [-1.851695287390397 0 -1.851695287390397]';
% Oversampling factor
S.oversampFact = 10;
% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
S.quadParams = quadParams;
S.constants = constants;
P = simulateQuadrotorDynamics(S);
S2.tVec = P.tVec;
S2.rMat = P.state.rMat;
S2.eMat = P.state.eMat;
S2.plotFrequency = 20;
S2.makeGifFlag = false;
S2.gifFileName = 'testGif.gif';
S2.bounds=1*[-1 1 -1 1 -1 1];
visualizeQuad(S2);
figure(1);clf;
```

```
plot(P.tVec,P.state.rMat(:,3)); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('Vertical position of CM');
figure(2);clf;
plot(P.state.rMat(:,1), P.state.rMat(:,2));
axis equal; grid on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');
```

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