ASE 479W Laboratory 1 Report
Simulating Quadrotor Dynamics

Harrison Qiu Jin

February 8, 2022

1 Introduction

Unmanned aerial vehicles, or UAVs, have many different applications in a variety of industries. UAVs pose many challenges that are still being widely researched. As researchers develop different decision-making policies and algorithms, it is important for them to be able to easily test and iterate those models. While testing can certainly be done on physical hardware UAVs, any mistakes in the software or faulty decision-making can lead to costly crashes. Thus, it is important for researchers to be able to test their work on a simulated UAV with a robust dynamic model. This paper explores the development of a rudimentary simulator for a quadrotor drone.

Simulating the motion of a quadrotor drone requires a robust kinematic and dynamic model of the drone. In particular, tracking the kinematic relationships between the reference frame of the drone (the "body" frame) and the inertial frame (the "world" frame) is especially crucial. The governing equations of these relationships, which are described in [1], are foundational in all UAV simulators. These relationships are used to model the state of a quadrotor and how that state changes over time in response to the angular rate of the rotors and any external disturbance forces. The simulator described in this paper implements a dynamics simulator of a quadrotor drone in MATLAB. Finally, an open-loop control strategy is developed to have the simulated quadrotor fly a circular path at nearly-constant altitude.

2 Theoretical Analysis

2.1 Attitude Representations

The attitude of the body frame with respect to the world frame can be represented in a number of different ways. The simulator detailed in this paper uses both Euler angles and rotation matrices. Euler angles are used for simplicity in presentation and input by the user, but Euler angles are not unique representations of attitude. Therefore, rotation matrices are used to iterate the state of the quadrotor over time.

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

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

Let the attitude of the body frame be represented by the Euler angles ϕ, θ, and ψ. By convention, ϕ represents rotation about the x-axis, θ represents rotation about the y-axis, and ψ represents rotation about the z-axis. These rotations are expressed in radians and constrained to the following domains (by convention):

$$\frac{-\pi}{2} < \phi < \frac{\pi}{2},\ -\pi \leq \theta \leq \pi,\ -\pi \leq \psi \leq \pi$$

The axes of rotation (Euler axes) can be represented with the following unit column vectors:

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ e_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ e_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

To convert between attitude representations, Euler's formula can be applied to successively rotate about each Euler axis. For the purposes of this paper, these rotations are applied in a 3-1-2 order. The final rotation matrix C then becomes the following:

$$C(\phi, \theta, \psi) = R(e_2, \theta)R(e_1, \phi)R(e_3, \psi) \tag{2}$$

This rotation matrix C can cast vectors expressed in the world reference frame into the coordinates of the body reference frame. Using the fully expanded form of C (omitted for brevity), the rotation matrix can then be converted back into Euler equations with the following set of equations:

$$\phi = \arcsin(C_{2,3}) \tag{3}$$

$$\theta = \text{atan2}(-C_{2,1}, C_{2,2}) \tag{4}$$

$$\psi = \text{atan2}(-C_{1,3}, C_{3,3}) \tag{5}$$

Here, $C_{i,j}$ represents the element of C located in the i^{th} row and the j^{th} column of C. As mentioned earlier, Euler angles are not unique. Therefore, these equations only hold given that the attitude represented by C is not singular. This singularity occurs if and only if $\phi = \frac{\pi}{2} \pm n\pi$ for $n = 0, 1, 2, 3 \ldots$

2.2 Time Derivative of a Rotation Matrix

To modelRecall Euler's formula from 1. Let $C(t) = R(\hat{\boldsymbol{a}}, \phi(t))$. To show that

$$\frac{d}{dt}C(t) = -\left[\boldsymbol{\omega}\times\right]C(t)$$

one can explicitly differential Euler's formula. Assume that the body is rotating about a vector $\hat{\boldsymbol{a}}$ at a rate of ω relative to the world frame, i.e., the rotation angle $\phi(t) = \omega t$ and the angular rate vector $\boldsymbol{\omega} = \omega\hat{\boldsymbol{a}}$. First, apply the chain rule of differentiation.

$$\frac{d}{dt}C(t) = \frac{dC}{d\phi}\frac{d\phi}{dt} = (-\sin(\omega t)I_{3\times3} + \sin(\omega t)\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^\intercal - \cos(\omega t)\left[\hat{\boldsymbol{a}}\times\right])\omega$$

Next, substitute the following identity: $\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^\intercal = I_{3\times3} + \left[\hat{\boldsymbol{a}}\times\right]^2$.

$$\frac{d}{dt}C(t) = (-\sin(\omega t)I_{3\times3} + \sin(\omega t)(I_{3\times3} + \left[\hat{\boldsymbol{a}}\times\right]^2) - \cos(\omega t)\left[\hat{\boldsymbol{a}}\times\right])\omega$$

$$\frac{d}{dt}C(t) = (-\sin(\omega t)I_{3\times3} + \sin(\omega t)I_{3\times3} + \sin(\omega t)\left[\hat{\boldsymbol{a}}\times\right]^2 - \cos(\omega t)\left[\hat{\boldsymbol{a}}\times\right])\omega$$

$$\frac{d}{dt}C(t) = (\sin(\omega t)\,[\hat{\boldsymbol{a}}\times]^2 - \cos(\omega t)\,[\hat{\boldsymbol{a}}\times])\omega$$

Use the fact that scalar multiplication of a matrix is commutative:

$$\frac{d}{dt}C(t) = \omega\,[\hat{\boldsymbol{a}}\times]\,(\sin(\omega t)\,[\hat{\boldsymbol{a}}\times] - \cos(\omega t))$$

Substitute $\omega\,[\hat{\boldsymbol{a}}\times] = [\boldsymbol{\omega}\times]$ and $C(t) = \cos(\omega t)I_{3\times 3} + (1-\cos(\omega t))\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^{\intercal} - \sin(\omega t)\,[\hat{\boldsymbol{a}}\times]$

$$\frac{d}{dt}C(t) = -\,[\boldsymbol{\omega}\times]\,(C(t) - (1-\cos(\omega t))\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^{\intercal}) \tag{6}$$

The $\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^{\intercal}$ term appears to be extraneous. Further investigation reveals that $-\,[\boldsymbol{\omega}\times]\,\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^{\intercal} = \underline{0}$

$$-\,[\boldsymbol{\omega}\times]\,\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^{\intercal} = \begin{bmatrix} 0 & -\omega a_3 & \omega a_2 \\ \omega a_3 & 0 & -\omega a_1 \\ -\omega a_2 & \omega a_1 & 0 \end{bmatrix} \begin{bmatrix} a_1^2 & a_1 a_2 & a_1 a_3 \\ a_1 a_2 & a_2^2 & a_2 a_3 \\ a_1 a_3 & a_2 a_3 & a_3^2 \end{bmatrix}$$

$$-\,[\boldsymbol{\omega}\times]\,\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^{\intercal} = \begin{bmatrix} -\omega a_1 a_2 a_3 + \omega a_1 a_2 a_3 & -\omega a_2^2 a_3 + \omega a_2^2 a_3 & -\omega a_2 a_3^2 + \omega a_2 a_3^2 \\ \omega a_1^2 a_3 - \omega a_1^2 a_3 & \omega a_1 a_2 a_3 - \omega a_1 a_2 a_3 & \omega a_1 a_3^2 - \omega a_1 a_3^2 \\ -\omega a_1^2 a_2 + \omega a_1^2 a_2 & -\omega a_1 a_2^2 + \omega a_1 a_2^2 & -\omega a_1 a_2 a_3 + \omega a_1 a_2 a_3 \end{bmatrix}$$

$$-\,[\boldsymbol{\omega}\times]\,\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^{\intercal} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{7}$$

Substitute the identity found in (7) into (6):

$$\frac{d}{dt}C(t) = -\,[\boldsymbol{\omega}\times]\,C(t) \tag{8}$$

Equation (8) is the kinematic equation for C and is an important piece of simulating a quadrotor's motion over time.

3 Implementation

3.1 Converting Between Attitude Representations

The first piece of the simulator involves switching between the world frame and the body frame. This is accomplished with the function `euler2dcm`, which takes a set of Euler angles as input and outputs a rotation matrix R_{BI} that casts vectors expressed in the inertial frame to the body frame. The function `dcm2euler` can be used to convert R_{BI} back into Euler angles, although it will throw an error in the singular case (see Section 2.1). To verify that the functions `dcm2euler` and `euler2dcm` are working properly, the script below is executed. In addition, the script verifies that the output of `euler2dcm` produces a correct rotation matrix.

```
for i = -0.49:0.05:0.49
    for j = -1:0.1:1
        for k = -1:0.1:1
            % test that euler angles can be converted to R and back
            e = [pi*i;pi*j;pi*k];
            R = euler2dcm(e);
            e2 = dcm2euler(R);
            if norm(e2-e) > 0.0001
                e
                e2
                error("Did not convert properly");
            end
            % test that rotationMatrix(e) == R
            R2 = rotationMatrix([0;1;0], e(2))*rotationMatrix([1;0;0], ...
                e(1))*rotationMatrix([0;0;1], e(3));
            if norm(R2-R) > 0.0001
                error("Did not get same transformation");
            end
        end
    end
end
```

3.2 State Variables and Derivatives

With these functions, the simulator can store the state of quadrotor. The full state of the quadrotor drone in this simulator includes the following:

r_I - The position of the center of mass of the drone, expressed in the inertial reference frame.

v_I - The velocity of the center of mass of the drone, expressed in the inertial reference frame.

e - The attitude of the body frame with respect to the inertial frame, expressed as 3-1-2 Euler angles.

ω_B - The angular velocity of the body frame relative to the inertial frame expressed in the body frame.

The state of the quadrotor can then be numerically integrated using MATLAB's `ode45` function, which is an ODE solver that implements the Runge-Kutta method. To do this, time derivatives for each of the state variables must be known. As mentioned, e is converted into a rotation matrix R_{BI} for differentiation and numerical integration purposes. The governing equations for the time derivatives of each of the state variables is shown below.

$$\dot{r_I} = v_I \tag{9}$$

$$m\dot{v_I} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_{BI}^{\intercal} \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ F_i \end{bmatrix} + d_I, \text{ where } F_i \text{ is the force produced by the } i^{th} \text{ rotor} \tag{10}$$

$$\dot{\omega}_B = J^{-1}(N_B - [\omega_B \times] J\omega_B), \text{ where } N_B \text{ is the total torque produced by all rotors} \tag{11}$$

$$\dot{R}_{BI} = -[\omega_B \times] R_{BI} \tag{12}$$

Equation (9) is derived from the definition of velocity. Equation (10) is the force balance equation obtained from evaluating Newton's second law on the drone. Equation (11) is obtained from Euler's equation. Finally, Equation (12) is derived in Section 2.2. The implementation of the state derivative in the MATLAB function `quadOdeFunction` is shown below (See Appendix A for full simulator implementation).

```matlab
% Initialize Xdot
Xdot = zeros(18,1);
% rI dot
Xdot(1:3) = vI;
% vI dot
weight = [0;0;P.quadParams.m * -1 * P.constants.g];
rotorThrust = R_BI' * [0;0;rotorForces];
Xdot(4:6) = (weight + rotorThrust + distVec)/P.quadParams.m;
% RBI dot
RBI_dot = -1*crossProductEquivalent(omegaB)*R_BI;
Xdot(7:9) = RBI_dot(1:3,1);
Xdot(10:12) = RBI_dot(1:3,2);
Xdot(13:15) = RBI_dot(1:3,3);
% omegaB dot
torque = [0;0;rotorTorques];
for k = 1:4
    % Torque due to thrust
    torque = torque + crossProductEquivalent(P.quadParams.rotor_loc(1:3,k)) ...
             * (P.quadParams.kF(k)*[0;0;omegaVec(k)^2]);
end
Xdot(16:18) = inv(P.quadParams.Jq) * ...
               (torque-crossProductEquivalent(omegaB)*P.quadParams.Jq*omegaB);
```

3.3 Numerical Integration

With the state variables and derivatives, the simulator can now model the dynamics of the quadrotor over time. The state of the quadrotor drone in each time-step can be found using numerical integration. As mentioned, the simulator described in this paper uses MATLAB's `ode45` function for numerical integration. The simulator's integration loop is shown below, where `omegaVec` is a vector containing the angular rates of each rotor and `distVec` is a vector containing any external disturbance forces acting on the drone. Each iteration of the loop represents one time step. For the purposes of this paper, a time step of 0.0005 seconds was used.

```
for k = 1:N-1
    tspan = [S.tVec(k):dtOut:S.tVec(k+1)]';
    omegaVec = S.omegaMat(k,:)';
    distVec = S.distMat(k,:)';
    % Build X_big
    RBI = euler2dcm(eK);
    Xk = [rK;vK;RBI(1:3,1);RBI(1:3,2);RBI(1:3,3);omegaB_K];
    % Run ODE solver on segment
    [tVecK,XMatk] = ode45(@(t,X)quadOdeFunction(t,X,omegaVec, distVec, params), ...
        tspan, Xk);
    % Store outputs
    tVecOut = [tVecOut; tVecK(1:end-1)];
    rMat = [rMat; XMatk(1:end-1, 1:3)];
    vMat = [vMat; XMatk(1:end-1, 4:6)];
    for row = 1:size(XMatk,1)-1
        R = [XMatk(row,7:9); XMatk(row,10:12); XMatk(row,13:15)]';
        eMat = [eMat;(dcm2euler(R))'];
    end
    omegaBMat = [omegaBMat;XMatk(1:end-1, 16:18)];
    % Prep for next iteration
    rK = XMatk(end, 1:3)';
    vK = XMatk(end, 4:6)';
    eK = dcm2euler([XMatk(end,7:9); XMatk(end,10:12); XMatk(end,13:15)]');
    omegaB_K = XMatk(end, 16:18)';
end
```

With an initial quadrotor state and parameters such as mass and moment of inertia, the simulator is now complete.

4 Results and Analysis

4.1 Testing the Simulator

The `simulateQuadrotorDynamics` function was run on the provided test data `Stest`. Table 1 shows sample values for the results of `simulateQuadrotorDynamics` and the expected values from the provided data `Ptest`.

t (sec)	0.0	0.5	1.0	1.5	2.0	2.5
Ptest Position	$\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}-0.3401\\0.0121\\-0.0042\end{bmatrix}$	$\begin{bmatrix}-0.5605\\0.0485\\-0.0170\end{bmatrix}$	$\begin{bmatrix}-0.6612\\0.1092\\-0.0382\end{bmatrix}$	$\begin{bmatrix}-0.6421\\0.1942\\-0.0679\end{bmatrix}$	$\begin{bmatrix}-0.5032\\0.3034\\-0.1062\end{bmatrix}$
PSimulated Position	$\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}-0.3401\\0.0121\\-0.0042\end{bmatrix}$	$\begin{bmatrix}-0.5605\\0.0485\\-0.0170\end{bmatrix}$	$\begin{bmatrix}-0.6612\\0.1092\\-0.0382\end{bmatrix}$	$\begin{bmatrix}-0.6421\\0.1942\\-0.0679\end{bmatrix}$	$\begin{bmatrix}-0.5032\\0.3034\\-0.1062\end{bmatrix}$
Error norm	0	2.6021e-17	8.7013e-17	3.7115e-16	1.0463e-15	2.2400e-15

Table 1: Comparison between the expected position of the drone to the simulated position of the drone.

Table 1 shows that the error norm between the expected values and simulated values is extremely low. This suggests that the simulator is behaving as expected.

4.2 Uniform Circular Motion

For the purposes of this experiment, the following assumptions are made:

1. The drone starts at a position of $(0, 1, 1.5)$.
2. The drone will fly in a clockwise direction.
3. The circular path has a radius R.
4. The time it takes the drone to complete one circle is the period T
5. The acceleration due to gravity is g.
6. The drone has a mass m.
7. The drone has moment of inertia matrix J with only non-zero entries on the diagonal.

The first step is to determine what the initial conditions of the drone are. The initial velocity is simply the tangential velocity needed to achieve uniform circular motion with the given parameters. The given starting position means that the initial velocity only has a non-zero component in the positive x direction. With these conditions, the initial velocity can be found as the following:

$$v_0 = \begin{bmatrix} \frac{2\pi R}{T} \\ 0 \\ 0 \end{bmatrix} \tag{13}$$

To achieve uniform circular motion at a constant altitude, the total force applied by rotors must satisfy two conditions:

1. The force must counteract the drone's acceleration due to gravity g.
2. The force must provide a centripetal acceleration directed towards the center of the circle $a_c = \frac{4\pi^2 R}{T^2}$.

Using these constraints, the initial roll ϕ of the drone can be found to be $\phi = \arctan(\frac{a_c}{g})$. The initial pitch and yaw of the drone are both 0.

Flying in uniform circular motion requires the drone to be turning at a constant rate, i.e., the body frame has a constant angular rate relative to the inertial frame. In the inertial frame, the pitch and roll rates will be 0, while the yaw rate $\dot{\psi} = \frac{2\pi R}{T}$, where r is the radius of the circle and T is the period. 3-1-2 Euler angle rotations can then be used to obtain the initial conditions for ω_B.

$$\boldsymbol{\omega_B} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta\cos\phi \\ 0 & 1 & \sin\phi \\ \sin\theta & 0 & \cos\theta\cos\phi \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{14}$$

Next, the angular rates for each rotor must be determined. Using Euler's equation and the fact that ω_B is constant over time, 3 constraints can be found on the angular rates of the rotors.

$$\dot{\boldsymbol{\omega}}_{\boldsymbol{B}} = J^{-1}(N_B - [\boldsymbol{\omega}_{\boldsymbol{B}}\times]\, J\boldsymbol{\omega}_{\boldsymbol{B}}) = 0$$

$$N_B - [\boldsymbol{\omega}_{\boldsymbol{B}}\times]\, J\boldsymbol{\omega}_{\boldsymbol{B}} = 0$$

$$N_B = \sum_{i=1}^{4} N_i + r_i \times F_i = [\boldsymbol{\omega}_{\boldsymbol{B}}\times]\, J\boldsymbol{\omega}_{\boldsymbol{B}}\ ,\ N_i = \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4}(-1)^i k_N \omega_i^2 \end{bmatrix}\ ,\ F_i = \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} k_F \omega_i^2 \end{bmatrix}$$

$$\sum_{i=1}^{4} \begin{bmatrix} k_F \omega_i^2 r_{iy} \\ -k_F \omega_i^2 r_{ix} \\ (-1)^i k_N \omega_i^2 \end{bmatrix} = \begin{bmatrix} \omega_{B2}\omega_{B3}(J_{3,3} - J_{2,2}) \\ \omega_{B1}\omega_{B3}(J_{1,1} - J_{3,3}) \\ \omega_{B1}\omega_{B2}(J_{2,2} - J_{1,1}) \end{bmatrix}$$

Because there are four rotors, this system of 3 equations is insufficient to determine the rotor speeds. The last constraint needed comes from the fact that the total thrust force $F_T = m\sqrt{a_c^2 + g^2}$. The final system to solve for the rotor rates then becomes the following.

$$\begin{bmatrix} k_F r_{1y} & k_F r_{2y} & k_F r_{3y} & k_F r_{4y} \\ -k_F r_{1x} & -k_F r_{2x} & -k_F r_{3x} & -k_F r_{4x} \\ -k_N & k_N & -k_N & k_N \\ k_F & k_F & k_F & k_F \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \begin{bmatrix} \omega_{B2}\omega_{B3}(J_{3,3} - J_{2,2}) \\ \omega_{B1}\omega_{B3}(J_{1,1} - J_{3,3}) \\ \omega_{B1}\omega_{B2}(J_{2,2} - J_{1,1}) \\ F_T \end{bmatrix} \tag{15}$$

Because none of the variables in (15) vary with time, the angular rates of each rotor are also constant with time. This makes sense because the motion of the drone is uniform. With (13), (14), and (15), as well as the drone's starting coordinates, the initial state of the drone for uniform circular motion can be fully evaluated. For the purposes of experimentation, the following parameters were chosen.

$R = 1m$

$T = 3s$

$g = 9.8m/s$

$m = 0.78kg$

$$J = 1 \times 10^{-9} \begin{bmatrix} 1756500 & 0 & 0 \\ 0 & 3572300 & 0 \\ 0 & 0 & 4713400 \end{bmatrix} kg\ m^2$$

Figure 1 shows the horizontal position of the drone in the simulator for the parameters and initial state described above.

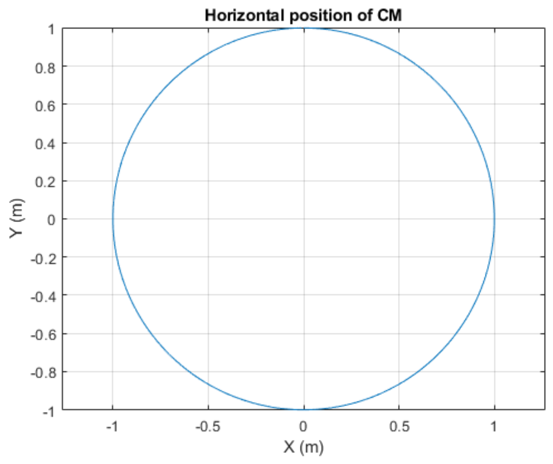

Figure 1: The horizontal position of the drone's center of mass during flight.

Figure 1 shows that the flight path of the drone was very circular. This demonstrates that the chosen initial conditions were correct for uniform circular motion.

Figure 2 shows the vertical position of the drone in the simulator for the same flight.

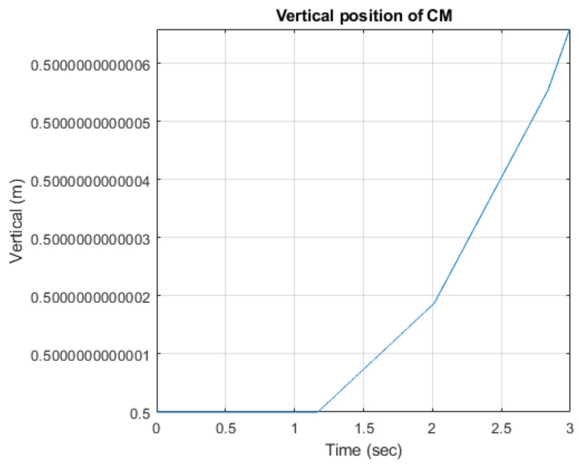

Figure 2: The horizontal position of the drone's center of mass during flight.

The very small changes in vertical position of the drone demonstrate that the simulator is working as intended and that the chosen initial conditions were correct. These changes could be due to errors in either the numerical integration or floating point operations.

5 Conclusion

An open-loop control strategy was developed to generate the rotor angular rates required to cause the simulated quadrotor to fly in a complete circle on a horizontal plane. The strategy was based on satisfying the constraints necessary for a rigid body to move in uniform circular motion. By relating these constraints to the state of the drone, the requisite angular rates of each rotor were determined. The control strategy successfully completed a circular flight path in the MATLAB simulator, which was developed using numerical integration and attitude dynamic and kinematic equations.

References

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

Appendices

A Full MATLAB Implementation

A.1 topSimulate.m

```
% Top-level script for calling simulateQuadrotorDynamics
clear; clc;
period = 3; % in seconds
radius = 1;
% Total simulation time, in seconds
Tsim = period;
% 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);
% Initial position in m
S.state0.r = [0 1 0.5]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [atan(4*pi^2*radius/period^2/9.8) 0 0]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [(2*pi*radius/period) 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in rad/s
eDot = [0 0 -1*2*pi*radius/period]';
phi = S.state0.e(1);
theta = S.state0.e(2);
psi = S.state0.e(3);
S.state0.omegaB = [cos(theta) 0 -sin(theta)*cos(phi);...
                   0 1 sin(phi);...
                   sin(theta) 0 cos(theta)*cos(phi)]*eDot;
% Oversampling factor
S.oversampFact = 10;
% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
S.quadParams = quadParams;
S.constants = constants;
% Rotor speeds at each time, in rad/s
kF = S.quadParams.kF;
kN = S.quadParams.kN;
r = S.quadParams.rotor_loc;
J = S.quadParams.Jq;
m = S.quadParams.m;
g = S.constants.g;
omegaB = S.state0.omegaB;
A = [kF(1)*r(2,1) kF(2)*r(2,2) kF(3)*r(2,3) kF(4)*r(2,4);
     -kF(1)*r(1,1) -kF(2)*r(1,2) -kF(3)*r(1,3) -kF(4)*r(1,4);
     -kN' .* S.quadParams.omegaRdir;
     kF(1) kF(2) kF(3) kF(4)];
```

```
b = [omegaB(2)*omegaB(3)*(J(3,3)-J(2,2));
     omegaB(1)*omegaB(3)*(J(1,1)-J(3,3));
     omegaB(2)*omegaB(1)*(J(2,2)-J(1,1));
     m*sqrt((2*pi*radius/period)^4+g^2)];
S.omegaMat = (((A\b).^(1/2))*ones(1, N-1))';
% load('Stest.mat');
P = simulateQuadrotorDynamics(S);
% load('Ptest.mat');
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');
```

A.2 simulateQuadrotorDynamics.m

```
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) 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: Harrison Jin
%+==============================================================================+

% Load parameters
params.quadParams = S.quadParams;
params.constants = S.constants;
% Number of iterations to run
N = length(S.tVec);
% Sampling rate
dtOut = (S.tVec(2)-S.tVec(1))/S.oversampFact;
% Create empty storage vectors
tVecOut = [];
rMat = [];
vMat = [];
eMat = [];
omegaBMat = [];

% Set initial states
rK = S.state0.r;
vK = S.state0.v;
eK = S.state0.e;
omegaB_K = S.state0.omegaB;

% Iterate
for k = 1:N-1
    tspan = [S.tVec(k):dtOut:S.tVec(k+1)]';
    omegaVec = S.omegaMat(k,:)';
    distVec = S.distMat(k,:)';
    % Build X_big
    RBI = euler2dcm(eK);
    Xk = [rK;vK;RBI(1:3,1);RBI(1:3,2);RBI(1:3,3);omegaB_K];
    % Run ODE solver on segment
    [tVecK,XMatk] = ode45(@(t,X)quadOdeFunction(t,X,omegaVec, distVec, params), ...
        tspan, Xk);
    % Store outputs
    tVecOut = [tVecOut; tVecK(1:end-1)];
    rMat = [rMat; XMatk(1:end-1, 1:3)];
    vMat = [vMat; XMatk(1:end-1, 4:6)];
    for row = 1:size(XMatk,1)-1
        R = [XMatk(row,7:9); XMatk(row,10:12); XMatk(row,13:15)]';
        eMat = [eMat;(dcm2euler(R))'];
    end
    omegaBMat = [omegaBMat;XMatk(1:end-1, 16:18)];
    % Prep for next iteration
    rK = XMatk(end, 1:3)';
    vK = XMatk(end, 4:6)';
    eK = dcm2euler([XMatk(end,7:9); XMatk(end,10:12); XMatk(end,13:15)]');
    omegaB_K = XMatk(end, 16:18)';
end
% Store the final state of the final segment
tVecOut = [tVecOut; tVecK(end,:)];
rMat = [rMat; rK'];
vMat = [vMat; vK'];
```

```
eMat = [eMat; eK'];
omegaBMat = [omegaBMat; omegaB_K'];

% Store into output structure
state.rMat = rMat;
state.vMat = vMat;
state.eMat = eMat;
state.omegaBMat = omegaBMat;
P.tVec = tVecOut;
P.state = state;
end
```

A.3 quadOdeFunction.m

```
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:
% Lecture Notes
% dxdtODE.m
% Author: Harrison Jin
%+==============================================================================+
% Extract data from input X
vI = X(4:6);
R_BI = zeros(3,3);
R_BI(1:3,1) = X(7:9);
R_BI(1:3,2) = X(10:12);
R_BI(1:3,3) = X(13:15);
omegaB = X(16:18);
% Get forces and torques from rotors
torqueDirs = -1*P.quadParams.kN' .* P.quadParams.omegaRdir;
rotorForces = P.quadParams.kF' *(omegaVec.^2);
rotorTorques = torqueDirs *(omegaVec.^2);
% Initialize Xdot
Xdot = zeros(18,1);
% rI dot
Xdot(1:3) = vI;
% vI dot
weight = [0;0;P.quadParams.m * -1 * P.constants.g];
rotorThrust = R_BI' * [0;0;rotorForces];
Xdot(4:6) = (weight + rotorThrust + distVec)/P.quadParams.m;
% RBI dot
RBI_dot = -1*crossProductEquivalent(omegaB)*R_BI;
Xdot(7:9) = RBI_dot(1:3,1);
Xdot(10:12) = RBI_dot(1:3,2);
Xdot(13:15) = RBI_dot(1:3,3);
% omegaB dot
torque = [0;0;rotorTorques];
for k = 1:4
    % Torque due to thrust
    torque = torque + crossProductEquivalent(P.quadParams.rotor_loc(1:3,k)) ...
              * (P.quadParams.kF(k)*[0;0;omegaVec(k)^2]);
end
Xdot(16:18) = ...
    inv(P.quadParams.Jq)*(torque-crossProductEquivalent(omegaB)*P.quadParams.Jq*omegaB);
end
```

A.4 euler2dcm.m

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

```
% INPUTS
%
% e ---------- 3-by-1 vector containing the Euler angles in radians: phi =
%              e(1), theta = e(2), and psi = e(3)
%
%
% OUTPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
%+------------------------------------------------------------------------------+
% References:
% Lecture notes
%
% Author: Harrison Jin
%+==============================================================================+
    R_BW = rotationMatrix([0;1;0], e(2))*rotationMatrix([1;0;0], ...
        e(1))*rotationMatrix([0;0;1], e(3));
end
```

A.5 dcm2euler.m

```
function [e] = dcm2euler(R_BW)
% dcm2euler : Converts a direction cosine matrix R_BW to Euler angles phi =
%             e(1), theta = e(2), and psi = e(3) (in radians) for a 3-1-2
%             rotation. If the conversion to Euler angles is singular (not
%             unique), then this function issues an error instead of returning
%             e.
%
% Let the world (W) and body (B) reference frames be initially aligned.  In a
% 3-1-2 order, rotate B away from W by angles psi (yaw, about the body Z
% axis), phi (roll, about the body X axis), and theta (pitch, about the body Y
% axis).  R_BW can then be used to cast a vector expressed in W coordinates as
% a vector in B coordinates: vB = R_BW * vW
%
% INPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
%
% OUTPUTS
%
% e ---------- 3-by-1 vector containing the Euler angles in radians: phi =
%              e(1), theta = e(2), and psi = e(3).  By convention, these
%              should be constrained to the following ranges: -pi/2 ≤ phi ≤
%              pi/2, -pi ≤ theta < pi, -pi ≤ psi < pi.
%
%+------------------------------------------------------------------------------+
% References:
% Lecture notes
%
% Author: Harrison Jin
%+==============================================================================+
% Singular case: phi == pi/2 or multiple
    if abs(R_BW(2,3) - 1) < 0.001
        error('Input matrix is singular case, unable to convert to euler angles');
    end
% Nominal case
    phi = asin(R_BW(2,3));
    psi = atan2(-1*R_BW(2,1), R_BW(2,2));
    theta = atan2(-1*R_BW(1,3), R_BW(3,3));
    e = [phi;theta;psi];
end
```

A.6 rotationMatrix.m

```
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
% Eulers formula for a rotation matrix.
%
% INPUTS
%
% aHat ------- 3-by-1 unit vector constituting the axis of rotation.
%
% phi -------- Angle of rotation, in radians.
%
%
% OUTPUTS
%
% R ---------- 3-by-3 rotation matrix
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author: Harrison Jin
%+==============================================================================+
    R = cos(phi)*eye(3,3) + (1-cos(phi))*(aHat*aHat') - ...
        sin(phi)*crossProductEquivalent(aHat);
end
```

A.7 crossProductEquivalent.m

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

```
%
%
% OUTPUTS
%
% uCross ----- 3-by-3 skew-symmetric cross-product equivalent matrix
%
%+------------------------------------------------------------------------------+
% References:
% Lecture Notes
%
% Author: Harrison Jin
%+==============================================================================+
    uCross = [      0   -1*u(3)     u(2);
                 u(3)        0  -1*u(1);
              -1*u(2)     u(1)       0];
end
```

A.8 visualizeQuad.m

```
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);

% RBG scaled on [0,1] for the color orange
rgbOrange=[1 .4 0];

% 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',rgbOrange);
    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',rgbOrange);
    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,:),...
        'black');
    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,:),...
        'black');

    % 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',rgbOrange);
        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',rgbOrange);
        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,:), 'black');
        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,:), 'black');

        % 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
```

A.9 constantsScript.m

```
% A script that loads constants used in the Aerial Robotics course.

% Acceleration due to gravity, in m/s^2
constants.g = 9.8;
% Mass density of moist air, in kg/m^3
constants.rho = 1.225;
```

A.10 quadParamsScript.m

```
% quadParamsScript.m
%
% Loads quadrotor parameters into the structure quadParams

% kF(i) is the rotor thrust constant for the ith rotor, in N/(rad/s)^2
quadParams.kF = 6.11e-8*(0.104719755)^-2*ones(4,1);
% kN(i) is the rotor counter-torque constant for the ith rotor, in N-m/(rad/s)^2
quadParams.kN = 1.5e-9*(0.104719755)^-2*ones(4,1);
% omegaRdir(i) indicates the ith rotor spin direction: 1 for a rotor angular
% rate vector aligned with the body z-axis, -1 for the opposite direction.
% Note that the torque vector caused by the ith rotor's twisting against the
% air is *opposite* the spin direction: Ni =
% [0;0;-kN*omegaRdir(i)*omegaVec(i)^2], where omegaVec is the 4x1 vector of
% rotor angular rates.
quadParams.omegaRdir = [1 -1  1 -1];
% rotor_loc(:,i) holds the 3x1 coordinates of the ith rotor in the body frame,
% in meters
quadParams.rotor_loc  =  ...
    0.21*[ 1  1 -1 -1; ...
          -1  1  1 -1; ...
           0  0  0  0];
% Mass of the quad, in kg
quadParams.m = 0.78;
% The quad's moment of inertia, expressed in the body frame, in kg-m^2
quadParams.Jq = diag(1e-9*[1756500; 3572300; 4713400]);
% The circular-disk-equivalent area of the quad's body, in m^2
quadParams.Ad = 0.01;
% The quad's coefficient of drag (unitless)
quadParams.Cd = 0.3;
% taum(i) is the time constant of the ith rotor, in seconds. This governs how
% quickly the rotor responds to input voltage.
quadParams.taum = (1/20)*ones(4,1);
% cm(i) is the factor used to convert motor voltage to motor angular rate
% in steady state for the ith motor, with units of rad/sec/volt
quadParams.cm = 200*ones(4,1);
% Maximum voltage that can be applied to any motor, in volts
quadParams.eamax = 12;
% r_rotor(i) is the radius of the ith rotor, in meters
quadParams.r_rotor = 0.06*ones(4,1);
%----------------------------------------------------------------------------

% The parameters below are for a more detailed model of the quad that includes
% a detailed aerodynamics model and handling of blade flexure.  You do not
% need to use these parameters for the Aerial Robotics course unless you'd
% like to make an especially high-fidelity simulator.
%
% The more detailed model is described in
%
% G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin, "Quadrotor
% helicopter flight dynamics and control: Theory and experiment," in Proc.of
% the AIAA Guidance, Navigation, and Control Conference, vol.  2, p. 4, 2007.

% Jm(i) is the moment of inertia about the main axis of rotation for the ith
% rotor, in kg-m^2.
quadParams.Jm = 0.0124*ones(4,1);
% Lock number, nondimensional
quadParams.gamma = 1e-3*quadParams.r_rotor.^4./quadParams.Jm;
% Offset of "hinge" from rotor hub, as a fraction of full rotor length.  See
% reference [2] of Lab Assignment 1.
quadParams.efConst = 1e-3*ones(4,1);
% Ratio of kB (parameter modeling rotor stiffness) to the rotor's moment of
% inertia.  Units of (rad/s)^2
quadParams.kBIb = 1e-4*ones(4,1);
% Average induced velocity ratio, unitless. See reference [1].
quadParams.lambdaI = 1.3772e-4*ones(4,1);
% Average pitch angle of the blade, radians
quadParams.thetaAvg = pi/12*ones(4,1);
% Linearized lift coefficients of the body, expressed in the body frame.
% Units of N/(m/s).
quadParams.Clmat=[0 0 0; 0 0 0; .001 .001 0];
% Linearized drag coefficients of the body, expressed in the body frame.
% Units of N/(m/s).
quadParams.Cdmat=diag([0.007 0.002 0.01]);
% Second-order drag coefficients of the body, expressed in the body frame.
% Units of N/(m/s)^2.
quadParams.Cd2mat=diag([0.0001 0.0001 0.0005]);
% Linearized coefficients of body roll/pitch/yaw.  Units of N-m/(rad/s).
quadParams.Cpqr=zeros(3,3);
```