ASE 389 Laboratory 1 Report
Simulating Quadrotor Dynamics

Justin Hart

January 26, 2023

1 Introduction

In order to describe quadrotor dynamics, attitude representation and Newtonian mechanics must be understood. Basic equations for quadrotor dynamics are listed in [1] and can be used to develop a simulator. Once a simulator is developed, it can be used for future work including controller development.

In this laboratory assignment, a quadrotor dynamics simulator was developed. This was done using Newtonian mechanics and using the equations generated to write MATLAB code. This simulator was then used to experiment with an input structure causing the simulated quad to fly a complete circle at a constant altitude.

2 Theoretical Analysis

2.1 Time Derivative of a Rotation Matrix

Euler's formula expresses a rotation matrix in terms of a rotation axis â and a rotation angle φ:

$$R(\hat{a}, \varphi) = cos\varphi I_{3x3} + (1 - cos\varphi)\hat{a}\hat{a}^T - sin\varphi[\hat{a}\times] \quad (1)$$

Let $C(t) = R(\hat{a}, \varphi(t))$. To show that $\frac{d}{dt}C(t) = -[\omega\times]C(t)$ one can explicitly differentiate Euler's formula. First, let $\varphi(t) = \omega t$, where ω is the angular velocity of a body about the axis of rotation $\hat{a}$ fixed to the body frame. Next, take the derivative of Euler's formula to get:

$$\frac{d}{dt}C(t) = -\omega \sin(\omega t)\, I_{3x3} + \omega \sin(\omega t)\, \hat{a}\hat{a}^T - \omega\cos(\omega t)[\hat{a}\times] \quad (2)$$

Now, let $\hat{a}\hat{a}^T = I_{3x3} + [\hat{a}\times]^2$ and substitute into (2):

$$\frac{d}{dt}C(t) = -\omega \sin(\omega t)\, I_{3x3} + \omega \sin(\omega t)\, (I_{3x3} + [\hat{a}\times]^2) - \omega\cos(\omega t)[\hat{a}\times]$$

(3)

Simplify (3) to get:

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

(4)

Then multiply $C(t)$ by $-[\omega\times]$ (which is equal to $-\omega[\hat{a}\times]$) to get:

$$-[\omega\times]C(t) = -\omega\, [\hat{a}\times]\cos(\omega t)\, I_{3x3} - \omega[\hat{a}\times](1-\cos(\omega t))\hat{a}\hat{a}^T + \omega \sin(\omega t)\, [\hat{a}\times]^2$$

(5)

Since a matrix multiplied by the identity matrix is itself, and $[\hat{a}\times]\hat{a} = 0$, the final result is:

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

(6)

Which is equal to (4) showing that $\frac{d}{dt}C(t) = -[\omega\times]C(t)$.

2.2 Relationship Between Rotation Angle and Trace of a Rotation Matrix

To show that the total rotation angle φ can be related to the trace of the rotation matrix R (expressed as $tr(R)$) by

$$tr(R) = 1 + 2cos\varphi$$

(7)

(1) is used to calculate R. Set an arbitrary unit vector $\hat{a} = \begin{matrix} x \\ y \\ z \end{matrix}$ and calculate R. Since the trace is the sum of the diagonal elements of a matrix, only those are listed in (8).

$$R(\hat{a}, \varphi) = \begin{bmatrix} cos\varphi + x^2 - x^2cos\varphi & \cdots & \ldots \\ \vdots & cos\varphi + y^2 - y^2cos\varphi & \vdots \\ \ldots & \cdots & cos\varphi + z^2 - z^2cos\varphi \end{bmatrix}$$

(8)

Calculating the trace of (8) yields $tr(R) = 3cos\varphi + (x^2 + y^2 + z^2) - (x^2 + y^2 + z^2)cos\varphi$. Since $\hat{a}$ is a unit vector, $(x^2 + y^2 + z^2) = 1$. This gives $tr(R) = 1 + 2cos\varphi$.

3 Implementation

The quadrotor dynamics simulator is designed using three levels of code. At the lowest level, an ordinary differential equation (ODE) function is created (quadOdeFunction.m) that is fed into a MATLAB ODE solver. At the second level of code, a simulation function (simulateQuadrotorDynamics.m) is written that solves the ODEs in the ODE function and returns the progression of the states through time given the appropriate inputs. The highest level of code is a top-level script (topSimulate.m) that provides the necessary inputs, runs the lower level functions, and outputs position plots and a visualization of the quadrotor's motion. See Appendix A for all code.

The quadOdeFunction.m takes as inputs time, a state vector, a vector of the constant rotor angular velocities, a disturbance vector, and a structure of parameters. The state vector is composed of a 3x1 position vector of the quadrotor's center of mass, a 3x1 velocity vector of the center of mass, a 3x3 attitude matrix (from the inertial frame to the body frame), and a 3x1 angular rate vector of the body with respect to the inertial frame expressed in the body frame. The 3x3 attitude matrix is used as a state as opposed to the vector of Euler angles in order to avoid the singularity that occurs at $\varphi = +/-\frac{\pi}{2}$. The disturbance vector is a 3x1 vector of constant disturbance forces acting on the quadrotor's center of mass. The parameters include the force and torque constants, the direction of spin of each rotor, the location of each rotor with respect to the center of mass, the mass of the quadrotor, the moment of inertia of the quadrotor, and the gravitational constant for earth. The output of the function is the derivative of the state vector. The governing state equations that relate the derivatives of the states to the states and inputs of the quadrotor are given in section 3.2.4 of [2]. These are implemented in the function as shown below:

```
% Xdot ------- Nx-by-1 time derivative of the input vector X

% rIdot
rIdot = vI;

% vIdot
Ft = 0;
for i = 1:4
    F = kF(i)*omegaVec(i)^2;
    Ft = Ft + F;
end

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

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

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

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

The simulateQuadrotorDynamics.m function takes as input a structure and outputs a structure. The input structure consists of a vector of time offsets from the initial time zero, an oversampling factor (to increase resolution of simulation results), a matrix of rotor speed inputs for each time set point, the initial states, the disturbance vector, and the quadrotor parameters/constants. The output structure has a vector of output sample time points, and the state of the quadrotor at each output sample time point. These are found by iterating through each sample time point and passing the quadOdeFunction.m into the MATLAB ode45 solver at each point. The time span is broken down further by the oversampling factor increase resolution of the results. The results are fed into output matrices that show how the state of the quadrotor evolves with time. This is implemented into simulateQuadrotorDynamics.m as follows:

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

    Xk = X0;

    for k = 1:N-1

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

        [tVeck,XMatk] =
ode45(@(t,X)quadOdeFunction(t,X,omegaMat(k,:)',distMat(k,:)',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,:)];

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

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

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

The topSimulate.m script sets all of the inputs and initial states and then runs the simulateQuadrotorDynamics.m function. The horizontal and vertical locations are plotted and a visualization of the quadrotor's motion over time is presented.

4 Results and Analysis

4.1 Attitude Representation Function Testing

Sub-functions were created to account for different attitude representations. This includes a function rotationMatrix.m that takes an angle and unit vector and calculates the rotation matrix (using (1)), a function that converts 3-1-2 Euler angles to a direction cosine matrix (euler2dcm.m), and a function that goes from the direction cosine matrix to Euler angles (dcm2euler.m).

These functions were tested to prove they work prior to implementing them into the quadrotor simulator. The first test was to show you can go from Euler angles, to the direction cosine matrix, and back to the original Euler angles. This was done using the following code in Test.m script:

```
%test dcm2euler function
clear
clc

etest=[pi()/3;-pi()/2;2*pi()/3]
Rtest = euler2dcm(etest)
enew = dcm2euler(Rtest)
```

The results of running the script are as follows:

```
etest =

   1.0472
  -1.5708
   2.0944

Rtest =

   0.7500    0.4330    0.5000
  -0.4330   -0.2500    0.8660
   0.5000   -0.8660    0.0000

enew =

   1.0472
  -1.5708
   2.0944
```

This shows you can recover the original set of Euler angles using dcm2euler.m. The singularity condition was tested as well by setting the first Euler angle $\varphi = \frac{\pi}{2}$. The results below show that the function returns an error when the conversion from the direction cosine matrix to Euler angles is singular:

etest =

```
   1.5708
  -1.5708
   2.0944
```

Rtest =

```
   0.8660    0.5000    0.0000
  -0.0000   -0.0000    1.0000
   0.5000   -0.8660    0.0000
```

Error using dcm2euler (line 31)
conversion to euler angles is singular: phi = +/- pi/2

Error in Test (line 90)
enew = dcm2euler(Rtest)

Next, the euler2dcm.m function was tested using the rotationMatrix.m function. The rotationMatrix.m function is used in the euler2dcm.m function in order to calculate the direction cosine matrix as shown below:

```
% INPUTS
%
% e ---------- 3-by-1 vector containing the Euler angles in radians:
phi =
%              e(1), theta = e(2), and psi = e(3)
%
phi = e(1);
theta = e(2);
psi = e(3);

e1 = [1;0;0];
e2 = [0;1;0];
e3 = [0;0;1];

% OUTPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
    R_BW =
rotationMatrix(e2,theta)*rotationMatrix(e1,phi)*rotationMatrix(e3,psi)
;
```

A script in Test.m compares the output of euler2dcm vs. the hand calculated 3-1-2 rotation matrix:

```
%test euler2dcm
clear
clc

phi = pi()/3;
theta = -pi()/2;
psi = 2*pi()/3;

euler = [phi;theta;psi];

e1 = [1;0;0];
e2 = [0;1;0];
e3 = [0;0;1];

R_BW = euler2dcm(euler)

c11 = cos(theta)*cos(psi) - sin(psi)*sin(theta)*sin(phi);
c12 = cos(theta)*sin(psi) + cos(psi)*sin(theta)*sin(phi);
c13 = -sin(theta)*cos(phi);
c21 = -cos(phi)*sin(psi);
c22 = cos(phi)*cos(psi);
c23 = sin(phi);
c31 = sin(theta)*cos(psi)+cos(theta)*sin(phi)*sin(psi);
c32 = sin(theta)*sin(psi)-cos(theta)*sin(phi)*cos(psi);
c33 = cos(theta)*cos(phi);

R_BW_act = [c11 c12 c13;
    c21 c22 c23;
    c31 c32 c33]
```

The results of the test show that euler2dcm.m returns the correct direction cosine matrix:

R_BW =

0.7500	0.4330	0.5000
-0.4330	-0.2500	0.8660
0.5000	-0.8660	0.0000

R_BW_act =

0.7500	0.4330	0.5000
-0.4330	-0.2500	0.8660
0.5000	-0.8660	0.0000

4.2 Quadrotor Simulator Testing

The simulateQuadrotorDynamics.m function was run using the Stest.mat structure. The output data of the simulator were compared to the data in the Ptest.mat structure which contains the desired outputs of the

simulator based on the Stest.mat input. Plots comparing the states of the Ptest.mat structure vs. the output of the simulator are shown below in Figs. 4.1- 4.4.

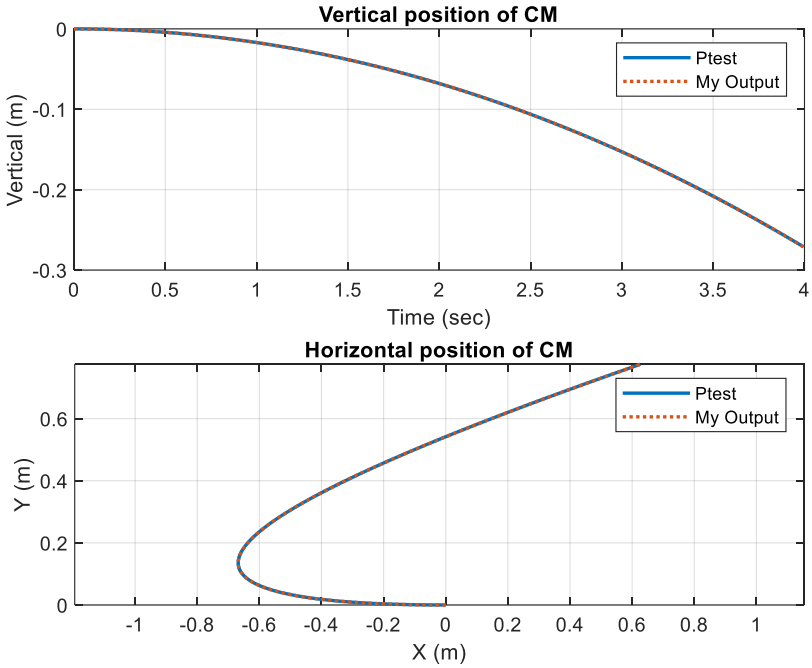

Figure 4.1: Ptest vs simulator (My Output) Quad center of mass vertical and horizontal positions over time

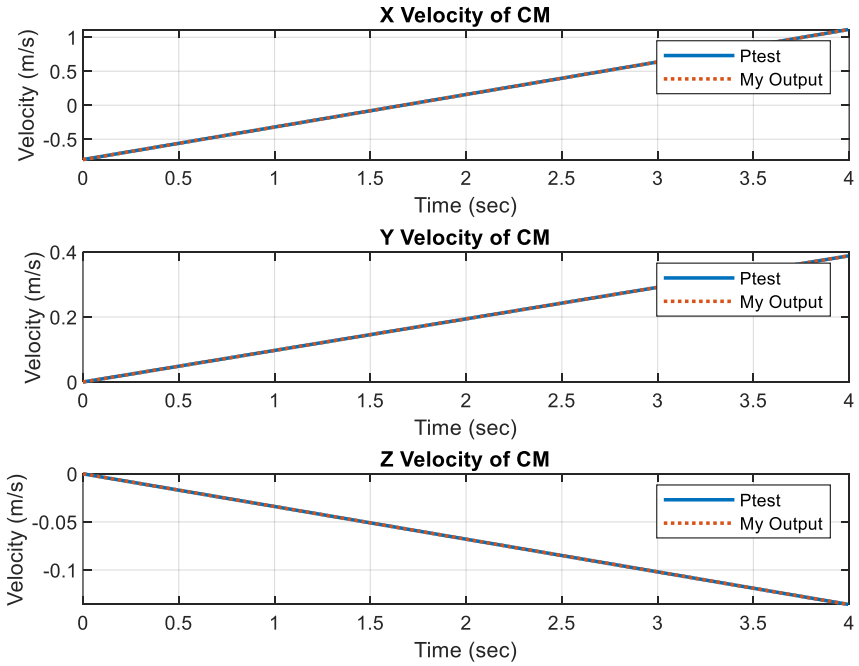

Figure 4.2: Ptest vs simulator (My Output) Quad center of mass velocity over time

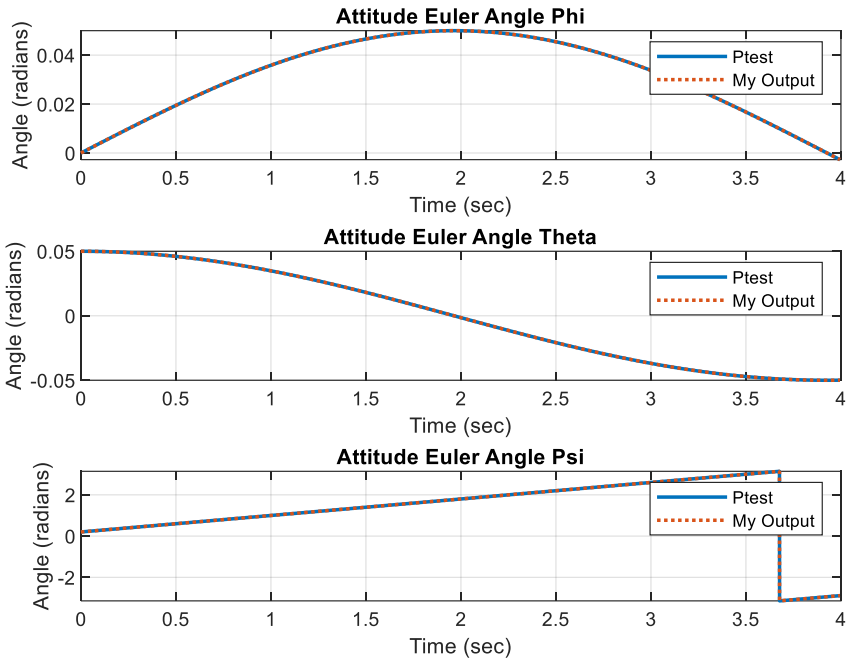

Figure 4.3: Ptest vs simulator (My Output) Quad attitude euler angles over time

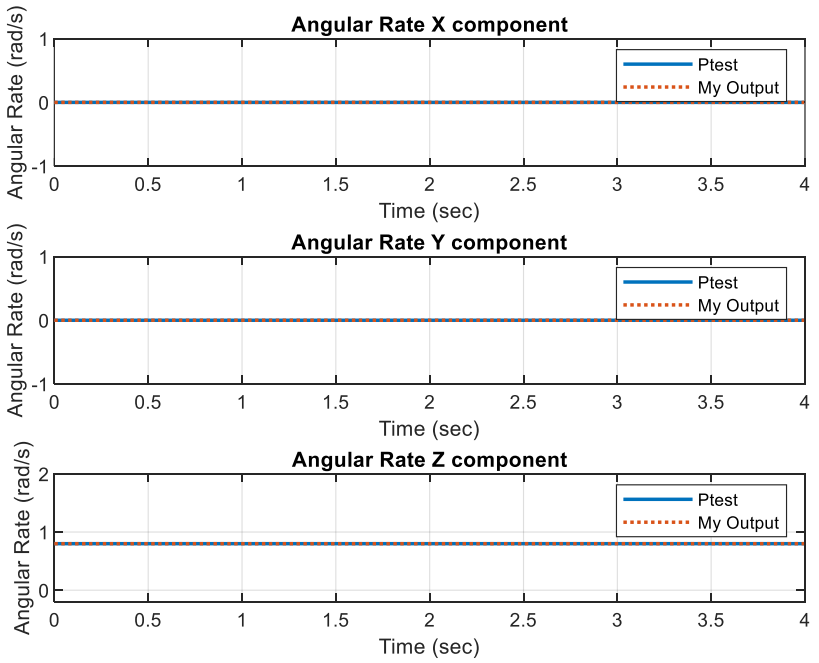

Figure 4.4: Ptest vs simulator (My Output) Quad angular rate over time

From these plots, it is shown that the simulator produces the expected output data when Stest.mat is used as the input structure.

4.3 Quadrotor Simulator Experimentation

An input structure was desired to make the simulated quad fly in a complete circle in the horizontal plane while maintaining a constant altitude. The radius of the circle was chosen as 5 meters, the period of rotation as 4 seconds, and the altitude as 10 meters. The roll Euler angle φ was chosen as 0.

First, the Euler angle θ (pitch) required was found in order to achieve the necessary centripetal acceleration. This was done by looking at the free body diagram (FBD) shown in Fig. 4.5:

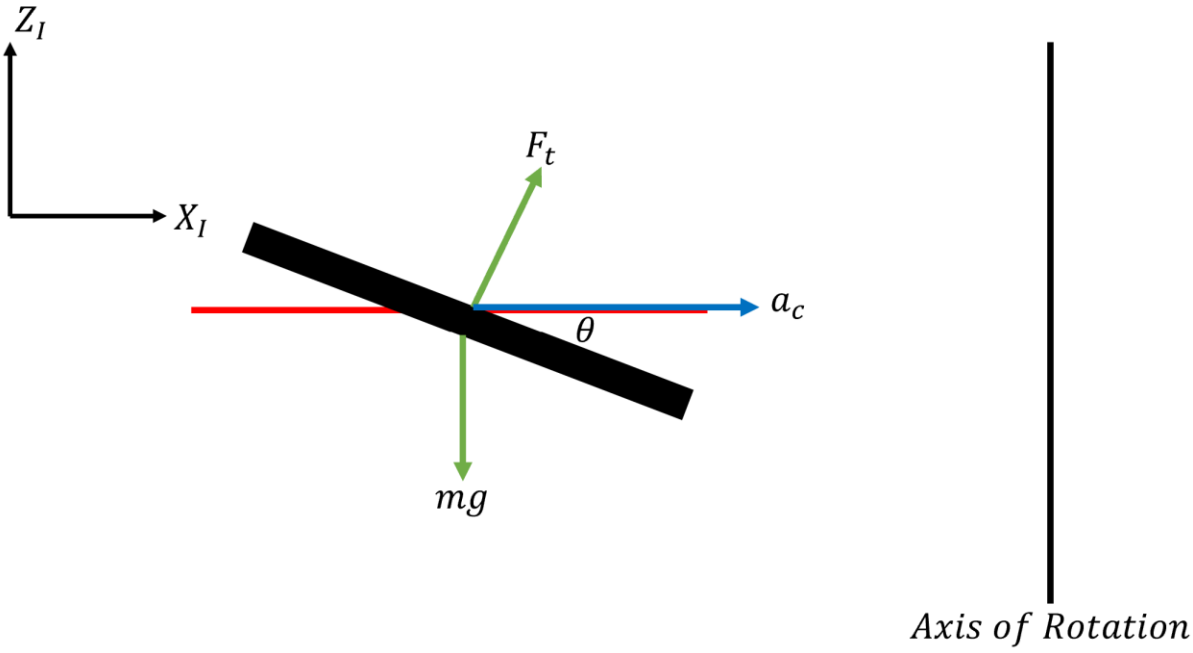

Figure 4.5: Quad FBD for circular motion

Where F_t is the total thrust from the propellers, and a_c is the centripetal acceleration. Centripetal acceleration is calculated as $\frac{v^2}{r}$ where v is the tangential velocity, and r is the radius of the circle. From the FBD the following equations were developed which allows for the pitch angle and total thrust to be solved for.

$$F_t cos\theta = mg \tag{9}$$

$$F_t sin\theta = ma_c \tag{10}$$

Next, the Euler angle rates were chosen. Since φ and θ should remain constant, their angle rates are 0. Ψ has a constant angle rate which is 2π divided by the period of rotation. Using these angle rate values, the

constant value for ω_B (angular rate of the body frame with respect to the inertial frame) was found using the rotation matrix listed in section 2.4 of [2]. Since ω_B is constant, $\dot{\omega}_B = 0$. This relationship was used to solve for the total torque N_B required to keep ω_B constant using (3.7) in [2]. Using this total torque and total thrust calculated earlier, a system of four equations was developed to solve for the four unknown rotor rates. These are shown in (11) and (12):

$$\boldsymbol{N_B} = \sum_{i=1}^{4} (\boldsymbol{N_i} + r_i \times \boldsymbol{F_i}) \tag{11}$$

$$F_t = \sum_{i=1}^{4} F_i \tag{12}$$

Where N_i is the ith propeller torque in the body frame, r_i is the location of the ith propeller in the body frame, and F_i is the external force due to the ith propeller in the body frame [2]. Using the values chosen, the four rotor rates calculated were $w1 = 742.07\frac{rad}{s}, w2 = 742.07\frac{rad}{s}, w3 = 743.09\frac{rad}{s}, w4 = 743.09\frac{rad}{s}$. The rotor orientation on the quad are shown in Fig. 4.6.

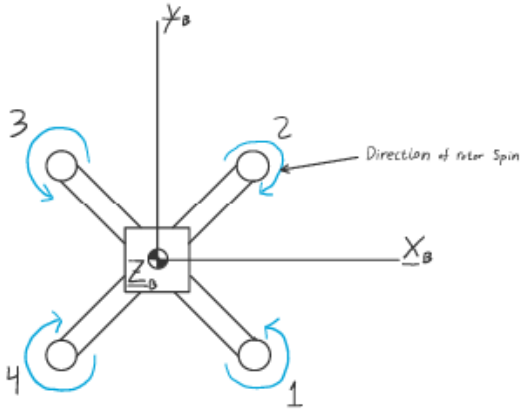

Figure 4.6: Quad rotor orientation [2]

The initial location of the quad was arbitrarily chosen as $[-5\ 0\ 10]$ for $X, Y, and\ Z$ respectively in the inertial frame. The initial attitude was set as 0 for φ since we want to keep it 0 for the entire simulation. The pitch angle θ was set to the calculated value in order to generate the desired centripetal acceleration and will remain constant for the duration of the simulation. The initial yaw angle ψ was chosen as 0 arbitrarily. The initial ω_B vector was chosen as the value calculated above based on the desired Euler angle rates and will remain constant. The initial velocity vector was set to 0 in the $X\ and\ Z$ directions, but Y was set to the tangential velocity calculated based on the circular path's radius and the period of rotation.

Using the values and initial conditions calculated, the script Experimentation.m was run in order to call the quad simulator function and plot the results of the simulation. See Appendix A for Experimentation.m. Figs. 4.7 and 4.8 show the results of the simulation.

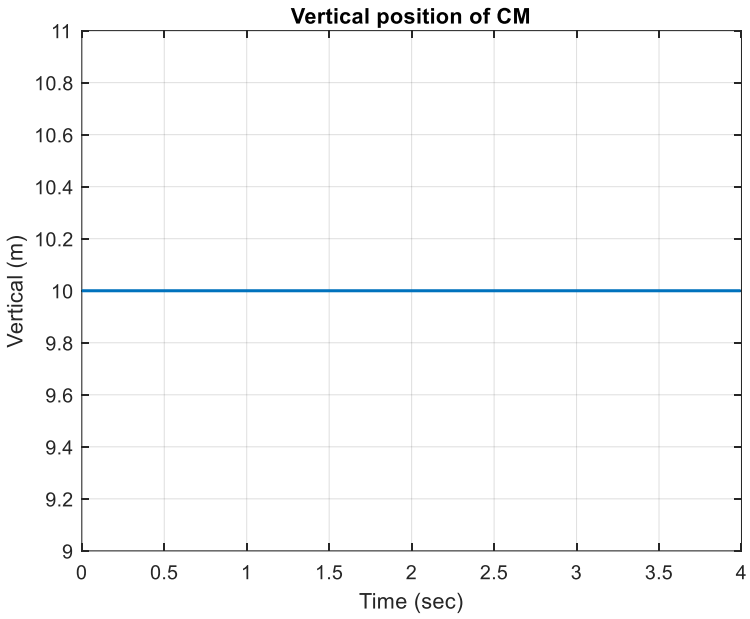

Figure 4.7: Vertical position Z_I of the quad's center of mass vs. time.

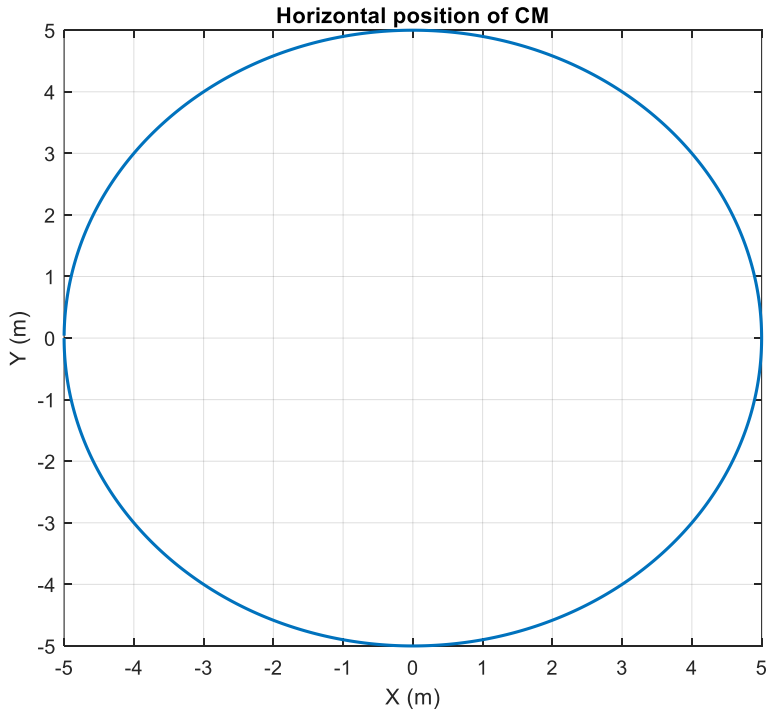

Figure 4.8: Horizontal position of the quad's center of mass in the $X_I - Y_I$ plane during the simulation.

The plots of the simulation show that the quad both maintained a constant altitude and flew in a complete circle in the horizontal plane.

5 Conclusion

A dynamics simulator for a quadrotor UAV was developed using basic Newtonian mechanics to generate state equations that could be numerically integrated. The simulator was tested against provided data, and an experiment was conducted to find the requisite rotor rates required to fly the quad in a complete circle at a constant altitude.

References

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

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

Appendix A: MATLAB Code

A.1 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.2 crossProductEquivalent.m

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

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

%
```

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

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

%
%+------------------------------------------------------------------
----------+
```

A.3 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.
%
phi = asin(R_BW(2,3));
theta = atan2(-R_BW(1,3),R_BW(3,3));
```

```
psi = atan2(-R_BW(2,1),R_BW(2,2));

if (phi == pi()/2) || (phi == -pi()/2) %**or make this if R_BW(2,3) =
+/- 1
    error('conversion to euler angles is singular: phi = +/- pi/2')
elseif(phi < -pi()/2) || (phi > pi()/2)
    error('conversion to euler angles is singular: phi is outside of
acceptable range')
elseif (theta < -pi()) || (theta >= pi())
    error('conversion to euler angles is singular: theta is outside of
acceptable range')
elseif (psi < -pi()) || (psi >= pi())
    error('conversion to euler angles is singular: psi is outside of
acceptable range')
else
    e = [phi;theta;psi];
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)
%
phi = e(1);
theta = e(2);
psi = e(3);

e1 = [1;0;0];
e2 = [0;1;0];
```

```
e3 = [0;0;1];

% OUTPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
    R_BW =
rotationMatrix(e2,theta)*rotationMatrix(e1,phi)*rotationMatrix(e3,psi)
;

end
%+------------------------------------------------------------------
----------+
```

A.5 Experimentation.m

```
%Experimentation --> complete circle
clear
clc

% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
S.quadParams = quadParams;
S.constants = constants;

%calculate constants for circular flight
radius = 5; %flight path radius in meters
period = 4; %flight period in seconds
ang_vel = 2*pi()/period; %quad angular vel in rad/s
tang_vel = radius*ang_vel; %quad tangential vel in m/s
acc_c = (tang_vel^2)/radius; %quad centripetal acc in m/s2
theta = atan(acc_c/S.constants.g); %quad pitch angle theta in radians
phi = 0; %quad roll angle phi in radians
total_thrust = S.constants.g*S.quadParams.m/cos(theta); %total thrust
required in Newtons
psidot = 2*pi()/period; %constant yaw rate (psidot) in rad/s
edot = [0;0;psidot]; %attitude euler angle rate vector in rad/s
omegaBRotMat = [cos(theta) 0 -sin(theta)*cos(phi);
    0 1 sin(phi);sin(theta) 0 cos(theta)*cos(phi)]; %omegaB rotation
matrix
omegaBconst = omegaBRotMat*edot; %constant omegaB for circular flight
path
omegaBconstcross = crossProductEquivalent(omegaBconst); %omegaB cross
product equivalent
torque = omegaBconstcross*S.quadParams.Jq*omegaBconst; %torque
required for const omegaB

%solve for rotor rates
syms w1 w2 w3 w4
```

```
W = [w1 w2 w3 w4];
N = 0;
for i = 1:4
    ri = S.quadParams.rotor_loc(:,i);
    Fi = [0;0;S.quadParams.kF(i)*W(i)^2];
    Ni = [0;0;S.quadParams.omegaRdir(i)*S.quadParams.kN(i)*W(i)^2];
    NBi = Ni + cross(ri,Fi);
    N = N+NBi;
end

eq1 = [N(1) == torque(1)];
eq2 = [N(2) == torque(2)];
eq3 = [N(3) == torque(3)];
eq4 =
[(S.quadParams.kF(1)*w1^2)+(S.quadParams.kF(2)*w2^2)+(S.quadParams.kF(
3)*w3^2)+(S.quadParams.kF(4)*w4^2) == total_thrust];
eqns = [eq1,eq2,eq3,eq4];

Sol = vpasolve(eqns,W,[0 Inf;0 Inf;0 Inf;0 Inf]);
omegaRotor =
[double(Sol.w1);double(Sol.w2);double(Sol.w3);double(Sol.w4)];

%%
%Simulation
% 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
S.omegaMat = repmat(omegaRotor',N-1,1);
% Initial position in m
S.state0.r = [-5 0 10]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 theta 0]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 -tang_vel 0]';
% Initial angular rate of body with respect to I, expressed in B, in
rad/s
S.state0.omegaB = omegaBconst;
% Oversampling factor
S.oversampFact = 10;

P = simulateQuadrotorDynamics(S);
```

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

%Horizontal Position
figure(2);clf;
plot(P.state.rMat(:,1), P.state.rMat(:,2),'LineWidth',1.5);
xlim([-5 5]);
ylim([-5 5]);
grid on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');
```

A.6 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
    rI = [X(1:3)];
%              vI = 3x1 velocity vector wrt I and in I, in meters/sec
    vI = [X(4:6)];
%             RBI = 3x3 attitude matrix from I to B frame
    RBI = [X(7:9),X(10:12),X(13:15)];
%          omegaB = 3x1 angular rate vector of body wrt I, expressed
in B
%                   in rad/sec
    omegaB = [X(16:18)];
%
```

```
% omegaVec --- 4x1 vector of rotor angular rates, in rad/sec.
omegaVec(i)
%              is the 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
    kF = P.quadParams.kF;
    kN = P.quadParams.kN;
    omegaRdir = P.quadParams.omegaRdir;
    rotor_loc = P.quadParams.rotor_loc;
    m = P.quadParams.m;
    Jq = P.quadParams.Jq;
%     Ad = P.quadParams.Ad;
%     Cd = P.quadParams.Cd;
%     taum = P.quadParams.taum;
%     cm = P.quadParams.cm;
%     eamax = P.quadParams.eamax;
%     r_rotor = P.quadParams.r_rotor;
%
%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m
%     rho = P.constants.rho;
    g = P.constants.g;
%
% OUTPUTS
%
% Xdot ------- Nx-by-1 time derivative of the input vector X

% rIdot
rIdot = vI;

% vIdot
Ft = 0;
for i = 1:4
    F = kF(i)*omegaVec(i)^2;
    Ft = Ft + F;
end

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

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

% omegaBdot
```

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

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

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

A.7 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);
```

A.8 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
% Euler's 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
%
R = cos(phi)*eye(3)+(1-cos(phi))*aHat*aHat'-
sin(phi)*crossProductEquivalent(aHat);

%+------------------------------------------------------------------
----------+
```

A.9 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.

    tVecIn = S.tVec;
    N = length(tVecIn);
%
%  oversampFact = Oversampling factor. Let dtIn = tVec(2) - tVec(1).
Then the
%                 output sample interval will be dtOut =
%                 dtIn/oversampFact. Must satisfy oversampFact >= 1.

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

    omegaMat = S.omegaMat;
%
%        state0 = State of the quad at tVec(1) = 0, expressed as a
structure
%                 with the following elements:
%
%                   r = 3x1 position in the world frame, in meters

           r0 = S.state0.r;
%
```

```
%                   e = 3x1 vector of Euler angles, in radians,
indicating the
%                       attitude

            e0 = S.state0.e;
            RBI0 = euler2dcm(e0);
%
%                   v = 3x1 velocity with respect to the world frame
and
%                       expressed in the world frame, in meters per
second.

            v0 = S.state0.v;
%
%               omegaB = 3x1 angular rate vector expressed in the body
frame,
%                       in radians per second.

            omegaB0 = S.state0.omegaB;

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

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

        distMat = S.distMat;
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m
%
%
% OUTPUTS
%
% P ---------- Structure with the following elements:
%
%          tVec = Mx1 vector of output sample time points, in seconds,
where
%                 P.tVec(1) = S.tVec(1), P.tVec(M) = S.tVec(N), and M
=
```

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

    Xk = X0;

    for k = 1:N-1

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

        [tVeck,XMatk] =
ode45(@(t,X)quadOdeFunction(t,X,omegaMat(k,:)',distMat(k,:)',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,:)];

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

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

    P.state.eMat = eMat;
    P.state.vMat = XMat(:,4:6);
    P.state.omegaBMat = XMat(:,16:18);
%+------------------------------------------------------------------------
----------+
```

A.10 Test.m

```
%test euler2dcm
clear
clc

phi = pi()/3;
theta = -pi()/2;
psi = 2*pi()/3;

euler = [phi;theta;psi];

e1 = [1;0;0];
e2 = [0;1;0];
e3 = [0;0;1];

R_BW = euler2dcm(euler)

c11 = cos(theta)*cos(psi) - sin(psi)*sin(theta)*sin(phi);
c12 = cos(theta)*sin(psi) + cos(psi)*sin(theta)*sin(phi);
c13 = -sin(theta)*cos(phi);
c21 = -cos(phi)*sin(psi);
c22 = cos(phi)*cos(psi);
c23 = sin(phi);
c31 = sin(theta)*cos(psi)+cos(theta)*sin(phi)*sin(psi);
c32 = sin(theta)*sin(psi)-cos(theta)*sin(phi)*cos(psi);
c33 = cos(theta)*cos(phi);

R_BW_act = [c11 c12 c13;
    c21 c22 c23;
    c31 c32 c33]
```

```
R_BW'*R_BW
det(R_BW)


%test dcm2euler function
clear
clc

etest=[pi()/3;-pi()/2;2*pi()/3]
Rtest = euler2dcm(etest)
enew = dcm2euler(Rtest)
```

A.11 topSimulate.m

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

% 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
S.omegaMat = 585*ones(N-1,4);
% Initial position in m
S.state0.r = [0 0 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0.05 0.2]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [-0.8 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in
rad/s
S.state0.omegaB = [0 0 0.8]';
% 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');
```

A.12 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
```