ASE 479W: Simulating Quadrotor Dynamics

Victor Branea

January 26, 2023

1 Introduction

This laboratory assignment was focused on creating a simulator for a quadrotor drone based on Newtonian dynamics. The focus was on transforming basic theoretical knowledge of attitude kinematics and dynamics into simple ordinary differential equations that could be then solved using MATLAB's ode45 function to predict the motion of the drone.

All the relevant information that was used for this assignment was taken from Dr. Humpherys' class notes as well as from MATLAB's documentation page. Special thanks also to my colleague Jeremy for helping me troubleshoot aspects of syntax and logic regarding the ode45 function.

2 Theoretical Analysis

Trace of a rotation matrix

A rotation matrix can be expressed using Euler's formula as follows:

$$R(\hat{\boldsymbol{a}}, \emptyset) = \cos(\emptyset)\, I_3 + (1 - \cos(\emptyset))\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T - \sin(\emptyset)[\hat{\boldsymbol{a}} \times] \tag{1}$$

From equation (1) it follows that by taking the trace of both sides we get:

$$tr(R) = tr(\cos(\emptyset)\, I_3 + (1 - \cos(\emptyset))\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T - \sin(\emptyset)\,[\hat{\boldsymbol{a}} \times]) \tag{2}$$

And knowing the trace property:

$$tr(A + B) = tr(A) + tr(B)$$

As well as the individual traces (computed by summing the diagonal elements of the matrix):

$$tr(I_3) = 3; tr(\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T) = a_1^2 + a_2^2 + a_3^2 = |a|^2 = 1; tr([\hat{\boldsymbol{a}} \times]) = 0$$

It follows that:

$$tr\big(R(\hat{\boldsymbol{a}}, \emptyset)\big) = 3\cos(\emptyset) + (1 - \cos(\emptyset)) = 1 + 2\cos(\emptyset) \tag{3}$$

Time derivative of a rotation matrix

By taking the first derivative with respect to time of the left-hand side of equation (1) we get:

$$\frac{d}{dt}C(t) = -\dot{\emptyset}\sin(\emptyset)\, I_3 + \dot{\emptyset}\sin(\emptyset)\, \hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T - \dot{\emptyset}\cos(\emptyset)\,[\hat{\boldsymbol{a}} \times] \tag{4}$$

After some simplification, this becomes:

$$\frac{d}{dt}C(t) = -\omega(\sin(\emptyset)\, I_3 - \sin(\emptyset)\, \hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T + \cos(\emptyset)\, [\hat{\boldsymbol{a}}\, \times]) \tag{5}$$

By working on the right-hand side of the equation we get:

$$-[\hat{\boldsymbol{\omega}}\, \times]C(t) = -\omega[\hat{\boldsymbol{a}}\, \times]C(t) =$$

$$= -\omega(\cos(\emptyset)\, [\hat{\boldsymbol{a}}\, \times] + (1 - \cos(\emptyset))[\hat{\boldsymbol{a}}\, \times]\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T - \sin(\emptyset)\, [\hat{\boldsymbol{a}}\, \times]^2) \tag{6}$$

But knowing that:

$$I_3 + [\hat{\boldsymbol{a}}\, \times]^2 = \hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T \; and \; [\hat{\boldsymbol{a}}\, \times]\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T = \mathbf{0}$$

The left-hand side becomes:

$$-[\hat{\boldsymbol{\omega}}\, \times]C(t) = -\omega(\cos(\emptyset)\, [\hat{\boldsymbol{a}}\, \times] - \sin(\emptyset)\, (\hat{\boldsymbol{a}}\hat{\boldsymbol{a}}^T - I_3) \tag{7}$$

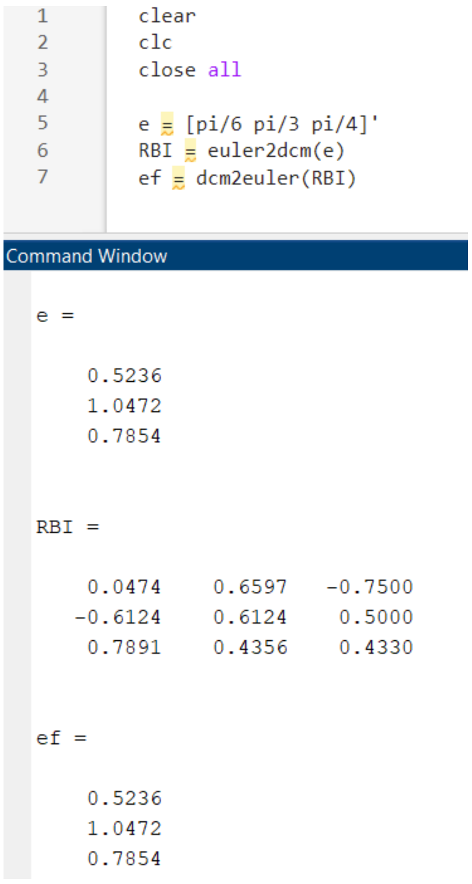
Finally, it can be noticed that equation (5) equals to (7), proving what was asked.

3 Results

Figure 1 shows the results of passing a vector e to the function euler2dcm to create a rotation matrix, and then passing this rotation matrix to the function dcm2euler to recover the original vector e. This shows that the two functions can go back and forth without a problem.

The second prompt was to use the function rotationMatrix to test euler2dcm, but considering that the code for the function euler2dcm calls upon the function rotationMatrix, it is obvious that any test performed using the rotationMatrix function will be passed by euler2dcm.

Table 1 presents values picked from the P structure outputted by the simulateQuadrotorDynamics function and compares them to values from the Ptest.mat file. As can be seen, the value match exactly, meaning the simulator runs successfully.



```
clear
clc
close all

e = [pi/6 pi/3 pi/4]'
RBI = euler2dcm(e)
ef = dcm2euler(RBI)
```

```
e =

    0.5236
    1.0472
    0.7854


RBI =

    0.0474    0.6597   -0.7500
   -0.6124    0.6124    0.5000
    0.7891    0.4356    0.4330


ef =

    0.5236
    1.0472
    0.7854
```

Figure 1 Testing euler2dcm and dcm2euler functions

	Results from simulator			Ptest.mat		
Index	tVec	eMat 1	vMat 1	tVec	eMat 1	vMat 1
500	0.2495	0.0099	-0.6804	0.2495	0.0099	-0.6804
1500	0.7495	0.0282	-0.4410	0.7495	0.0282	-0.4410
4500	2.2495	0.0486	0.2774	2.2495	0.0486	0.2774
7000	3.4995	0.0167	0.8761	3.4995	0.0167	0.8761

Table 1 Comparison between Ptest.mat and results from the coded simulator

To make the drone simulation move in a circle, I calculated the centripetal acceleration based on a radius of 0.5m and a period of 4 seconds. The period determines an angular velocity of magnitude $\pi/2$. From this a centripetal force of $-\omega^2 r$ can be computed to equal $\pi^2/8$. Knowing that the thrust of the drone must overcome the centripetal force and the force due to gravity, the following can be said:

Roll angle is given by: $$\text{atan}\left(\frac{\frac{\pi^2}{8}}{g}\right) = 0.1252\, rad \qquad (8)$$

Thrust magnitude: $$F_T = m\sqrt{(\frac{\pi^2}{8})^2 + g^2}\, N \qquad (9)$$

Therefore, rotor angle rates are all: $$\sqrt{\frac{F_T}{4k_F}} = 587.9569\, rad/s \qquad (10)$$

Looking at figures (2) and (3), it can be seen that the horizontal position of the center of mass forms a perfect circle, and the vertical position of the centre of mass does not move significantly.

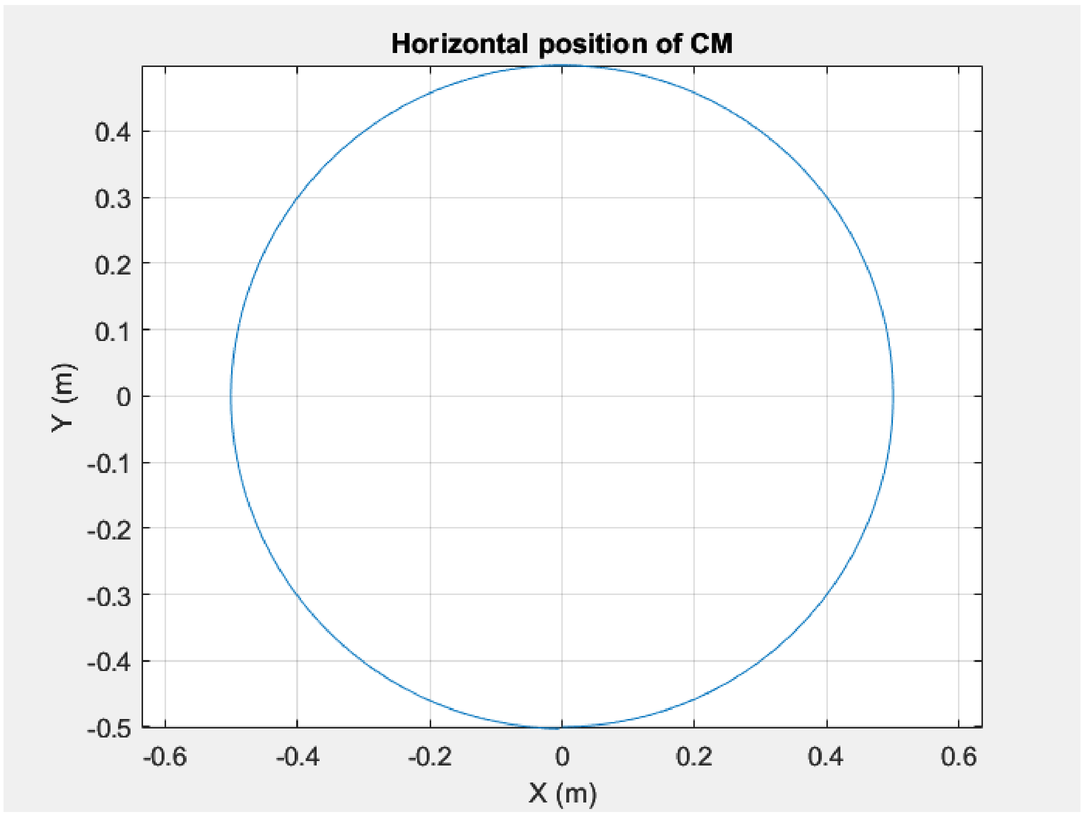


Figure 2 Horizontal position of the center of mass

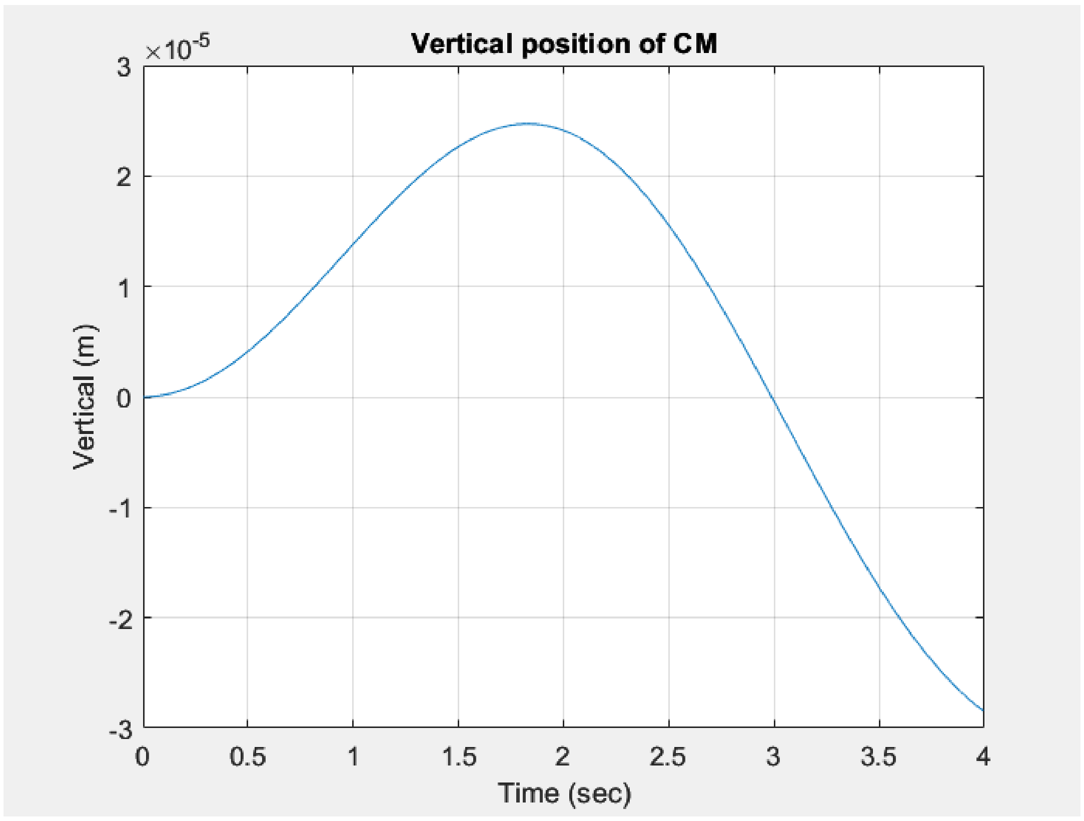


Figure 3 Vertical position of the center of mass

4 Conclusion

A simulator based on Newtonian dynamics was programmed and shown to work through a series of tests. By coding a series of functions, a simulated drone was able to fly in a perfect circle given appropriate inputs.

5 Annexe (Code)

```
function uX = uCross(u)

% uCross : 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
%
%+------------------------------------------------------------------------------+

    %Creating the uCross matrix based on the u vector
    uX = [0 -u(3) u(2); u(3) 0 -u(1); -u(2) u(1) 0];
end


function R = rotationMatrix(a, 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
%
%+------------------------------------------------------------------------------+
    %Creating identity matrix
    I3 = [1 0 0; 0 1 0; 0 0 1];
    %Creating a matrix by multiplying axis of rotation vector by its
    %transpose
    A = a*transpose(a);
    %Final rotation matrix using Euler's formula
    R = cos(phi)*I3 + (1-cos(phi))*A - sin(phi)*uCross(a);
end


function R = 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
%
%+------------------------------------------------------------------------------+
    %Rotation matrix for 3-1-2 rotation
    R = rotationMatrix([0; 1; 0], e(2))*rotationMatrix([1; 0; 0],
e(1))*rotationMatrix([0; 0; 1], e(3));
end


function e = dcm2euler(R)
% 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.
%
%+------------------------------------------------------------------------------+
    %Check if the requirements for a singularity are met for a 3-1-2
    %rotation
    if R(2,3) == 1 || R(2,3) == -1
        error('Singularity')
    end
    %Finding phi
    e1 = asin(R(2,3));
    %Finding theta and psi if phi is positive
    if e1 > 0
        if asin(-R(1,3)/cos(e1)) < 0
```

```
            e2 = -acos(R(3,3)/cos(e1));
        else
            e2 = acos(R(3,3)/cos(e1));
        end
        if asin(-R(2,1)/cos(e1)) < 0
            e3 = -acos(R(2,2)/cos(e1));
        else
            e3 = acos(R(2,2)/cos(e1));
        end
    %Finding theta and psi if phi is negative
    else
        if asin(-R(1,3)/cos(e1)) < 0
            e2 = -acos(R(3,3)/cos(e1));
        else
            e2 = acos(R(3,3)/cos(e1));
        end
        if asin(-R(2,1)/cos(e1)) < 0
            e3 = -acos(R(2,2)/cos(e1));
        else
            e3 = acos(R(2,2)/cos(e1));
        end
    end
    %Creating the output vector
    e = [e1; e2; e3];
end


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
%
%+------------------------------------------------------------------------------+

%Unpack inputs
rI = X(1:3);
vI = X(4:6);
RBI = zeros(3,3);
RBI(:) = X(7:15);
omegaB = X(16:18);

%Calculate sum of Forces
sumF = zeros(3,1);
for i = 1:4
    sumF = sumF + P.quadParams.kF(i)*omegaVec(i)^2*[0; 0; 1];
end

%Calculate sum of Torques
sumN = zeros(3,1);
for i = 1:4
    sumN = sumN + P.quadParams.omegaRdir(i)*P.quadParams.kN(i)*omegaVec(i)^2*[0;
0; 1];
end

%Calculate the r cross F components of torque

sumNF = zeros(3,1);
for i = 1:4
    sumNF = sumNF +
P.quadParams.kF(i)*omegaVec(i)^2*uCross(P.quadParams.rotor_loc(:,i))*[0; 0; 1];
end

%Calculate the derivatives
rIdot = vI;
vIdot = [0; 0; -P.constants.g] + transpose(RBI)*sumF/P.quadParams.m +
distVec/P.quadParams.m;
RBIdot = -uCross(omegaB)*RBI;
RBIdotVec = RBIdot(:);
omegaBdot = transpose(P.quadParams.Jq)*(sumN + sumNF -
uCross(omegaB)*P.quadParams.Jq*omegaB);

%Make the final output
Xdot = [rIdot; vIdot; RBIdotVec; omegaBdot];
end


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).
%
%
%+------------------------------------------------------------------------------+
%Create parameters structure A
A.quadParams = S.quadParams;
A.constants = S.constants;

tVecSZ = size(S.tVec);

dtIn = S.tVec(2) - S.tVec(1);
dtOut = dtIn/S.oversampFact;

P.state.rMat(1,:) = S.state0.r(:);
P.state.eMat(1,:) = S.state0.e(:);
P.state.vMat(1,:) = S.state0.v(:);
P.state.omegaBMat(1,:) = S.state0.omegaB(:);

P.tVec(1) = S.tVec(1);

for i = 1:(tVecSZ-1)
    %Define tspan for each iteration
    tspan = [S.tVec(i):dtOut:S.tVec(i+1)]';

    %Define step count
    k = (i-1)*S.oversampFact + 1;

    %Unpack omegaVec and distVec for each iteration
    omegaVec = transpose(S.omegaMat((i),:));
    distVec = transpose(S.distMat((i),:));

    %Unpack previous state
    rI = transpose(P.state.rMat((k),:));
    e = transpose(P.state.eMat((k),:));
    RBI = euler2dcm(e);
    RBIVec = RBI(:);
    vI = transpose(P.state.vMat((k),:));
    omegaB = transpose(P.state.omegaBMat((k),:));

    %Put together previous state vector
    Xi = [rI; vI; RBIVec; omegaB];

    %Make function handle
    Xdot = @(t, X)quadOdeFunction(t, X, omegaVec, distVec, A);

    %Call ode45
    [tVecTemp, stateMatTemp] = ode45(Xdot, tspan, Xi);

    %Save new time variables in P.tVec
    P.tVec(end) = [];
    P.tVec = [P.tVec; tVecTemp];
```

```matlab
    %Save new states
    rIf = stateMatTemp(:,1:3);
    vIf = stateMatTemp(:,4:6);
    omegaBf = stateMatTemp(:,16:18);

    tempSZ = size(tVecTemp);
    for j = 1:tempSZ
        RBIf = zeros(3,3);
        RBIf(:) = stateMatTemp(j,7:15);
        ef = dcm2euler(RBIf);
        P.state.eMat(k+j-1,:) = transpose(ef);
    end
    P.state.rMat(end,:) = [];
    P.state.rMat = [P.state.rMat; rIf];
    P.state.vMat(end,:) = [];
    P.state.vMat = [P.state.vMat; vIf];
    P.state.omegaBMat(end,:) = [];
    P.state.omegaBMat = [P.state.omegaBMat; omegaBf];
end


% 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 = 587.9569*ones(N-1,4);
% Initial position in m
S.state0.r = [0 -0.5 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [-0.1252 0 0]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [pi/4 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in rad/s
S.state0.omegaB = [0 -0.1962 1.5585]';
% 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');
```