ASE 389 Laboratory 3 Report

Measurement Simulation and State Estimation

Justin Hart

February 28, 2023

1 Introduction

In Labs 1 and 2, the true, error-free state of the quad was used for simulation and control design. In practice, though, the estimated state is derived with an estimator through a series of noisy, external sensors as shown in [2].

In this laboratory assignment, simulators for the quad's sensors (GNSS receiver, camera, and inertial measurement unit (IMU)) were built. Measurement and dynamics models were derived to be used by a model-based estimator. The estimator (unscented Kalman filter based) was applied to reconstruct the state of the quad from the sensor measurements. This estimated state, along with sensor models were tested using the controller developed in Lab 2.

2 Theoretical Analysis

2.1 GNSS Measurement Simulation Baseline Vector Measurement Model

In simulating GNSS measurements, a baseline vector measurement model is developed, pointing from the primary to secondary antenna. The difference in this vs the primary antenna position measurement model is that there is no measurement error in the length of the baseline measurement model as described on page 2 of the lab 3 handout. This means that the covariance matrix of this measurement model has a particular form as shown on page 3 of the lab 3 handout. This covariance matrix, R_{bG}, has a rank of 2, forcing the addition of a slight value to its diagonal to ensure positive definiteness. To show that R_{bG} has a rank of 2, first plug in a vector $r_{bG} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ into R_{bG}. This gives:

$$R_{bG} = \begin{bmatrix} \sigma_b^2(y^2 + z^2) & -\sigma_b^2 xy & -\sigma_b^2 xz \\ -\sigma_b^2 xy & \sigma_b^2(x^2 + z^2) & -\sigma_b^2 yz \\ -\sigma_b^2 xz & -\sigma_b^2 yz & \sigma_b^2(x^2 + y^2) \end{bmatrix} \quad (1)$$

The determinant of $R_{bG} = 0$ which means the rank is not 3. To check if the rank is 2, look for a minor of order 2 that is non-zero. The minor $M_{2,3}$ is checked:

$$M_{2,3} = \det\begin{bmatrix} \sigma_b^2(y^2+z^2) & -\sigma_b^2 xy \\ -\sigma_b^2 xz & -\sigma_b^2 yz \end{bmatrix} = -\sigma_b^4(y^3z + yz^3 + x^2yz)$$

(2)

Since this minor is non-zero, the rank of $R_{bG} = 2$.

2.2 Intersection of Two Lines

A line defined by $ax + by + c = 0$ can be represented by $l = [a, b, c]^T$. It can be shown that the point defined by the intersection of two lines l_1 and l_2 can be found as $\underline{x} = l_1 \times l_2$ by first defining l_1 and l_2 as $[a_1, b_1, c_1]^T$ and $[a_2, b_2, c_2]^T$ respectively. Taking the cross product of these two lines yields:

$\begin{bmatrix} b_1c_2 - c_1b_2 \\ c_1a_2 - a_1c_2 \\ a_1b_2 - b_1a_2 \end{bmatrix}$ divide by the third element to get the 2D representation:

$$\underline{x} = \begin{bmatrix} (b_1c_2 - c_1b_2)/(a_1b_2 - b_1a_2) \\ (c_1a_2 - a_1c_2)/(a_1b_2 - b_1a_2) \end{bmatrix}$$

(3)

Cramer's rule can be used to solve for x and y in $\underline{x}$ satisfying both equations of lines (and is the intersection point of the lines). First, the equations for two lines are written in $\underline{A}\underline{x} = \underline{B}$ form:

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -c_1 \\ -c_2 \end{bmatrix}$$

(4)

x and y are found as $\frac{D_x}{D}$ and $\frac{D_y}{D}$ respectively. D is equal to the determinant of $\underline{A}$, D_x is the determinant of the combination of the first column of $\underline{A}$ with $\underline{B}$ and D_y is the determinant of the combination of the second column of $\underline{A}$ with $\underline{B}$. Plugging these results into the equations for x and y gives:

$$x = \frac{D_x}{D} = \frac{b_1c_2 - c_1b_2}{a_1b_2 - b_1a_2}$$

$$y = \frac{D_y}{D} = \frac{c_1a_2 - a_1c_2}{a_1b_2 - b_1a_2}$$

(5)

This is the same result as when taking the cross product of each line, proving the intersection of two lines can be found as $\underline{x} = l_1 \times l_2$.

2.3 Accelerometer Measurement Model

For the accelerometer measurement model, it is assumed that the accelerometer is coincident with the quad's center of mass (CM), which implies that the accelerometer lever arm, $l_B = 0$. This results in the acceleration of the quad's CM in the I frame, $a_I = \ddot{R}_I$, where $\ddot{R}_I$ is the vector pointing from the I frame

origin to the B frame origin. If the accelerometer were to be located at a constant position so that $l_B \neq 0$ in the B frame, the acceleration with respect to I of a point fixed in the B frame located at l_B in the B frame (a_I) is more complicated than just $\ddot{R}_I$.

It is known that $\dot{l_B} = \ddot{l_B} = 0$ because the accelerometer is fixed in the body frame. Also, $\dot{RBI} = -[\omega_B \times]RBI$. First the accelerometer lever arm is defined:

$$l_B = RBI * l_I \tag{6}$$

Using the product rule to find $\dot{l_B}$ yields:

$$\dot{l_B} = -\omega_B \times l_B + RBI * \dot{l_I} \tag{7}$$

Using the product rule once more to find $\ddot{l_B}$ gives:

$$\ddot{l_B} = -\dot{\omega_B} \times l_B + \left[-\omega_B \times \dot{l_B} - \omega_B \times (\omega_B \times l_B)\right] + RBI * \ddot{l_I} \tag{8}$$

Because B is translating with respect to I, the vector from the origin of the I frame to the point of interest P (accelerometer location) is $r_I = R_I + l_I$ where R_I is the vector from the origin of the I frame to the quad's CM in I, and l_I is the vector from the quad's CM to P in the I frame. Using this relationship (taking its second derivative) and (8), $\ddot{r}_I = a_I$ can be solved for:

$$a_I = \ddot{R}_I + RBI^T[\dot{\omega_B} \times l_B + \omega_B \times (\omega_B \times l_B)] \tag{9}$$

2.4 Rate Gyro

If it is assumed, as in section 2.3, that the IMU is positioned at l_B, then the rate gyros are also located at this position. It is unnecessary to account for the lever arm l_B in the rate gyro measurement model ((8) in the lab 3 handout) because angular velocity/acceleration is independent of the lever arm distance between the quad's CM and the accelerometer proof mass. Every point on the body experiences the same angular velocity/acceleration.

Since all points on a rotating rigid body have the same rotation angles according to Euler's rotation theorem, and the angular velocity is just the derivative of the rotation angle, all points on a rigid body have the same angular velocity.

3 Implementation

The implementation of the code for this lab starts at the top with a simulate control script; topSimulateControl.m (see Appendix A for all MATLAB code). This script first specifies the seed for Matlab's random number generator. A flag can be asserted to call the full estimation and control simulator; otherwise only the control simulator is called. This can be used for comparison in the control of

the quad when using estimated vs. true state inputs. A reference trajectory is created, and initial conditions are specified. Feature locations for the camera measurement model are designated and known parameters/constants are set. Results are plotted for visual reference.

Within the top-level script, a simulateQuadrotorEstimationAndControl.m function is utilized in order to simulate closed-loop estimation and control of the quad. This function takes in the reference trajectory, known parameters/constants, and the initial state, and outputs the estimated state of the quad over the simulation length. This is done by first using the sensor models to simulate noisy measurements. These measurements are fed into the unscented Kalman filter function (stateEstimatorUKF.m) which outputs the estimated state based on the measurements. The estimated state is fed into the trajectory and attitude controllers from Lab 2 to develop the control input into the quad simulator function (quadOdeFunctionHF.m) which solves for the state at each time input.

The first sensor model is the GNSS simulator (gnssMeasSimulator.m). This function takes in the state of the quad and sensor parameters, and outputs both the measured position of the quad's primary GNSS antenna, as well as the measured position of the secondary GNSS antenna in ECEF coordinates. The calculation for the outputs is shown below:

```
% rpGtilde --- 3x1 GNSS-measured position of the quad's primary GNSS
antenna,
%              in ECEF coordinates relative to the reference antenna,
in
%              meters.

    rpI = rI + RBI'*raB(:,1);
    RIG = Recef2enu(r0G);
    rpG = RIG'*rpI;
    RpG = (inv(RIG)*RpL)*inv(RIG');
    wpG = mvnrnd(zeros(3,1),RpG)';

    rpGtilde = rpG + wpG;
%
% rbGtilde --- 3x1 GNSS-measured position of secondary GNSS antenna,
in ECEF
%              coordinates relative to the primary antenna, in meters.
%              rbGtilde is constrained to satisfy norm(rbGtilde) = b,
where b
%              is the known baseline distance between the two
antennas.
%
    rsI = rI + RBI'*raB(:,2);
    rsG = RIG'*rsI;
    rbG = rsG - rpG;
    b = norm(rbG);
    eps = 10^(-8);
    rbGunit = rbG/norm(rbG);
    RbG = norm(rbG)^2*sigmab^2*(eye(3)-rbGunit*rbGunit')+eps*eye(3);
    wbG = mvnrnd(zeros(3,1),RbG)';

    rbGtilde = rbG + wbG;
    rbGtilde = rbGtilde*b/norm(rbGtilde);
```

Two key points in this code is that first, noise is added to the measurements through the use of Matlab's mvnrnd function by assuming a white Gaussian noise with zero mean and covariance matrix R_{bG}. The second is that rbGtilde is constrained by the fact that it's magnitude must equal the known baseline distance between the two antennas.

The camera measurement simulator (hdCameraSimulator.m) takes in a known location of a feature point, the state of the quad, and camera parameters, and outputs the 2x1 measured position of the feature point projection on the camera's image plane in pixels. This measured position is checked against the image plane size as well as the orientation of the image plane to confirm the feature is actually seen by the camera. If not, an empty position vector is output. This section of the function code is shown below:

```
  if (rXC(3)<=f)
        rx = [];
    elseif (rx(1)>imagePlane_pixel(1)/2)||(rx(1)<-
imagePlane_pixel(1)/2)
        rx = [];
    elseif (rx(2)>imagePlane_pixel(2)/2)||(rx(2)<-
imagePlane_pixel(2)/2)
        rx = [];
    end
```

The IMU simulator function (imuSimulator.m) takes in the state of the quad, as well as IMU sensor parameters, and outputs the specific force measured by the accelerometer and the angular rate measured by the rate gyro. The accelerometer and rate gyro models include bias, which is remembered from the previous call to the function. Before the bias is used from the previous call to the function, the bias for each model has to be initialized as shown in the code below:

```
persistent ba bg va2 vg2
if(isempty(ba))
    % Set ba's initial value
    QbaSteadyState = P.sensorParams.Qa2/(1 - P.sensorParams.alphaa^2);
    ba = mvnrnd(zeros(3,1), QbaSteadyState)';
end
if(isempty(bg))
    % Set bg's initial value
    QbgSteadyState = P.sensorParams.Qg2/(1 - P.sensorParams.alphag^2);
    bg = mvnrnd(zeros(3,1), QbgSteadyState)';
```

Within the stateEstimatorUKF.m function, a wahba solver function (wahbaSolver.m) is used to initialize the attitude state estimate on the first call to the function. Wahba's problem is solved using singular value decomposition (SVD). Least-squares weights for each pair of vectors, a matrix of vectors in I, and a matrix of the vectors in B are input into the function, and the rotation matrix RBI is output.

In the UKF estimator function, a measurement model and dynamics model for the quad are used (h_meas.m, f_dynamics.m). The measurement model relates the state of the system to the input measurements and outputs a measurement vector. It takes in the current state, measurement noise, the attitude matrix estimate, and the known visual feature locations as well as sensor parameters to output a measurement vector at that time. The dynamics model approximates forward evolution in time of the state

of the quad. It takes in the current state, the IMU measurement at that time, process noise, an attitude matrix estimate, and known parameters to output the predicted state of the quad at t+1.

4 Results and Analysis

4.1 Wahba's Problem Solver Testing

In order to test the wahbaSolver.m function, a script wahbaTester.m was developed (see Appendix A.9). Within the testing function, a random 3-1-2 rotation matrix (Rin) is generated as well as 1000 random unit vectors (vIMat). Then a set of transformed, random unit vectors is generated by multiplying the rotation matrix by the first set of unit vectors (vBMat). A least squares weights vector is assembled so that the sum of these values is equal to one (aVec). Next, the wahbaSolver.m function is called with the least squares weights and two sets of random vectors and outputs a rotation matrix, Rout. The rotation matrix Rin is compared to Rout to six decimal points. If the wahbaSolver.m function works correctly, Rin should equal Rout. This entire process was repeated 1000 times within the wahbaTester.m function, and a vector of the comparison of Rin to Rout was output. This vector has the value of 1 if Rin and Rout were equal to six decimal points, and a value of 0 if not. When the wahbaTester.m was run, the output vector was all ones (the vector was too long to place in this report).

The only way to 100% guarantee there are no bugs in the wahbaSolver.m function would be to input all possible vector combinations and confirm the output rotation matrix is equal to the input rotation matrix. This is nearly impossible, but the next best thing is to input a reasonable number of random, known vectors instead to confirm the rotation matrices are equivalent. In the strategy described above, a combination of 1000 different vectors was tested 1000 different times. Barring unlimited time and computational resources, this is a reasonable number of random combinations to prove the functionality of the wahbaSolver.m function.

4.2 Quadrotor Estimator and Controller Experimentation

To test the whether the quad could be controlled using state estimates from noisy measurements, a reference trajectory was input into the quad estimation/control simulation function. The results of the simulation are shown in Fig. 1:

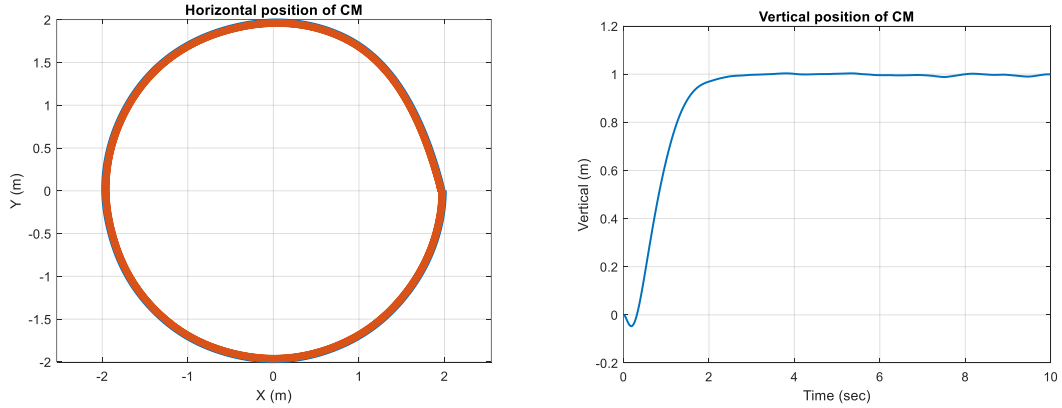

Figure 1: Left – quad's true CM position in the XI-YI plane with a vector pointing from the CM in the direction of xI. Right – quad's true CM position in the ZI direction.

From Fig. 1 it is seen that the simulated quad was ably to fly the reference trajectory of a circle with a two meter diameter and a vertical height of one meter with the intended yaw angle.

Next, the tracking performance of the quad was compared when using the simulateQuadrotorEstimationAndControl.m vs. simulateQuadrotorControl.m. The resultant plots are shown in Figs. 2-4:

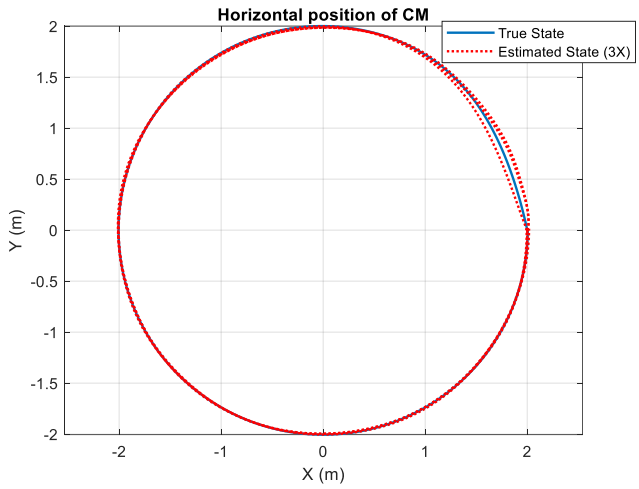

Figure 2: quad's CM position in the XI-YI plane when using true state for control vs. estimated state (estimated state was simulated three times to get variation in noise).

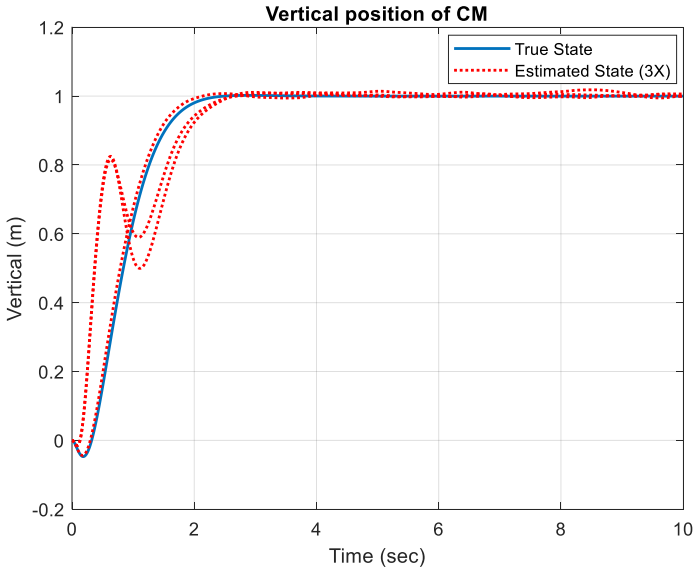

Figure 3: quad's CM position in the ZI direction when using true state for control vs. estimated state (estimated state was simulated three times to get variation in noise).

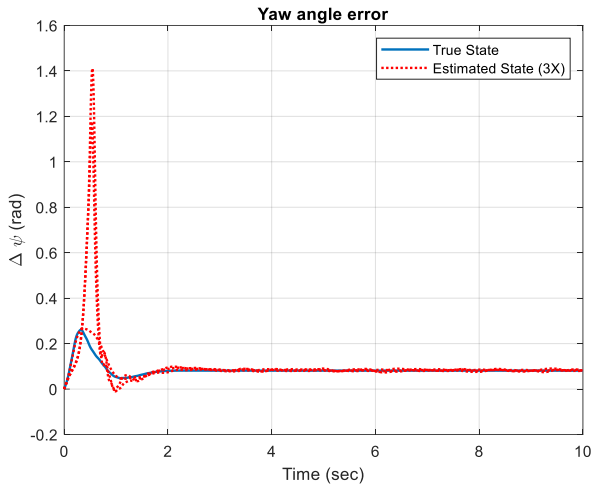

Figure 4: quad's yaw angle error when using true state for control vs. estimated state (estimated state was simulated three times to get variation in noise).

The tracking performance of the quad is better when using simulateQuadrotorControl.m. This is because the true state is fed into the controller, instead of estimated states from noisy measurements. The horizontal position is closer to the reference trajectory, the vertical position of the quad is reached quicker and has less variation in steady state, and the yaw angle error is smaller.

With the random number generator seed set to 'shuffle' the tracking performance changes enough each time the estimated simulation is run to notice variation in simulations as shown in Figs. 2-4. Looking at the yaw angle error Fig. 4 for example, it is seen that the error varies at a maximum of 1.4 radians for one simulation vs. about 0.3 radians for another. This is expected, though, as the random number generator seed is changed for each simulation, affecting any noise simulation within the function. The variation in tracking performance is not enough, though, to cause the quad to lose the ability to be controlled. Past the first two seconds of simulation, the difference in simulations is negligible.

In order to test the limits of the estimation and control simulator, the camera measurement model was turned off. The results of the simulation with the camera turned off are shown in Figs. 5-6:

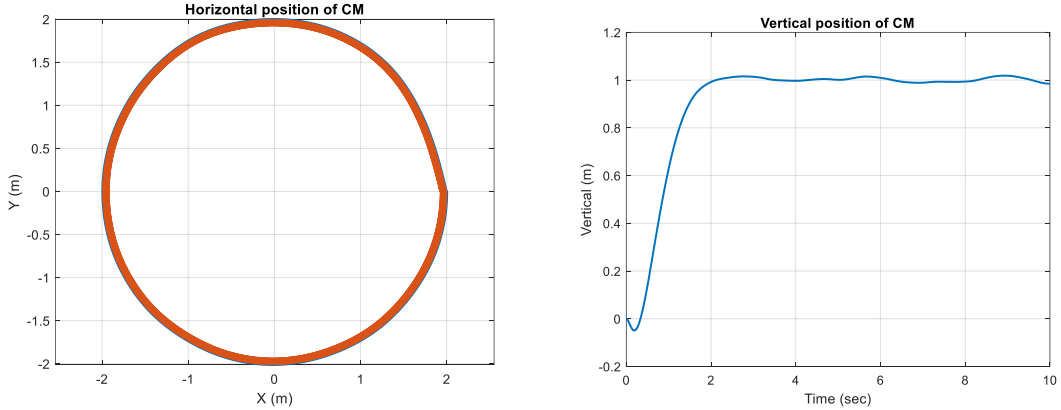

Figure 5: Left – quad's true CM position in the XI-YI plane with a vector pointing from the CM in the direction of xI with the camera turned off. Right – quad's true CM position in the ZI direction with the camera turned off.

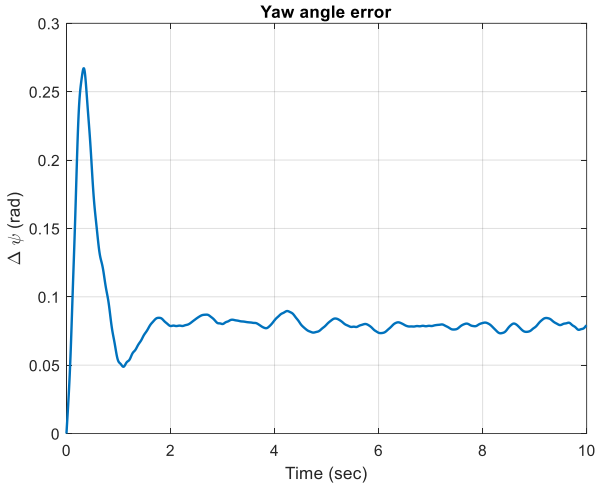

Figure 6: quad's yaw angle error with the camera turned off.

Without input from the camera, it is seen that the estimator's state estimates are sufficiently accurate for quad control. This is because when comparing Figs. 1 and 5, the differences are indiscernible. The quad still completes a circle with radius of 2 meters and is able to maintain a vertical position of 1 meter for the duration of the simulation.

5 Conclusion

Sensor measurement models were developed along with measurement and dynamics models for estimation. These were used within a model-based state estimator to reconstruct the state of the quad from the noisy sensor measurements. The quad was controlled based on the estimated state and desired trajectory input, showing the estimator's state estimates are sufficiently accurate for quad control.

References

[1] 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.

[2] Y. Bar-Shalom, X. R. Li, and T. Kirubarajan, Estimation with Applications to Tracking and Navigation. New York: John Wiley and Sons, 2001.

Appendix A: MATLAB Code

A.1 f_dynamics.m

```
function [xkp1] = f_dynamics(xk,uk,vk,delt,RBIHatk,P)
% f_dynamics : Discrete-time dynamics model for quadcopter.
%
% INPUTS
%
% xk --------- 15x1 state vector at time tk, defined as
%
%              xk = [rI', vI', e', ba', bg']'
%
%              where all corresponding quantities are identical to
those
%              defined for E.statek in stateEstimatorUKF.m and where e
is the
%              3x1 error Euler angle vector defined such that for an
estimate
%              RBIHat of the attitude, the true attitude is RBI =
%              dRBI(e)*RBIHat, where dRBI(e) is the DCM formed from
the error
%              Euler angle vector e.
% uk --------- 6x1 IMU measurement input vector at time tk, defined as
%
%              uk = [omegaBtilde', fBtilde']'
%
%              where all corresponding quantities are identical to
those
%              defined for M in stateEstimatorUKF.m.
%
% vk --------- 12x1 process noise vector at time tk, defined as
%
%              vk = [vg', vg2', va', va2']'
%
%              where vg, vg2, va, and va2 are all 3x1 mutually-
independent
%              samples from discrete-time zero-mean Gaussian noise
processes.
%              These represent, respectively, the gyro white noise
(rad/s),
%              the gyro bias driving noise (rad/s), the accelerometer
white
%              noise (m/s^2), and the accelerometer bias driving noise
%              (m/s^2).
```

```
%
% delt ------- Propagation interval, in seconds.
%
% RBIHatk ---- 3x3 attitude matrix estimate at time tk.
%
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m
        g = P.constants.g;
%
%  sensorParams = Structure containing sensor parameters, as defined
in
%                 sensorParamsScript.m
        alphaa = P.sensorParams.alphaa;
        alphag = P.sensorParams.alphag;
%
%
% OUTPUTS
%
% xkp1 ------- 15x1 state vector propagated to time tkp1
%
%+-----------------------------------------------------------------------
----------+
% References:
%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.
%
% Author: Justin Hart
%+=======================================================================
==========+

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

rIk = xk(1:3);
vIk = xk(4:6);
ek = xk(7:9);
bak = xk(10:12);
bgk = xk(13:15);
omegaBtildek = uk(1:3);
fBtildek = uk(4:6);
vgk = vk(1:3);
vg2k = vk(4:6);
```

```
vak = vk(7:9);
va2k = vk(10:12);
RBIk = euler2dcm(ek)*RBIHatk;

%edotk
omegaBk = omegaBtildek-bgk-vgk;
ct = cos(ek(2));cp = cos(ek(1));
st = sin(ek(2));sp = sin(ek(1));
Sek = 1/cp*[cp*ct 0 cp*st;sp*st cp -ct*sp;-st 0 ct];
edotk = Sek*omegaBk;

%aIk
aIk = RBIk'*(fBtildek-bak-vak)-g*[0;0;1];

%xkp1 components
rIkp1 = rIk+delt*vIk+0.5*(delt^2)*aIk;
vIkp1 = vIk+delt*aIk;
ekp1 = ek+delt*edotk;
bakp1 = alphaa*bak+va2k;
bgkp1 = alphag*bgk+vg2k;

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

A.2 gnssMeasSimulator.m

```
function [rpGtilde,rbGtilde] = gnssMeasSimulator(S,P)
% gnssMeasSimulator : Simulates GNSS measurements for quad.
%
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure
with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
        rI = S.statek.rI;
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
        RBI = S.statek.RBI;
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
        raB = P.sensorParams.raB;
        r0G = P.sensorParams.r0G;
        RpL = P.sensorParams.RpL;
```

```
        sigmab = P.sensorParams.sigmab;
%
% OUTPUTS
%
% rpGtilde --- 3x1 GNSS-measured position of the quad's primary GNSS
antenna,
%              in ECEF coordinates relative to the reference antenna,
in
%              meters.

    rpI = rI + RBI'*raB(:,1);
    RIG = Recef2enu(r0G);
    rpG = RIG'*rpI;
    RpG = (inv(RIG)*RpL)*inv(RIG');
    wpG = mvnrnd(zeros(3,1),RpG)';

    rpGtilde = rpG + wpG;
%
% rbGtilde --- 3x1 GNSS-measured position of secondary GNSS antenna,
in ECEF
%              coordinates relative to the primary antenna, in meters.
%              rbGtilde is constrained to satisfy norm(rbGtilde) = b,
where b
%              is the known baseline distance between the two
antennas.
%
    rsI = rI + RBI'*raB(:,2);
    rsG = RIG'*rsI;
    rbG = rsG - rpG;
    b = norm(rbG);
    eps = 10^(-8);
    rbGunit = rbG/norm(rbG);
    RbG = norm(rbG)^2*sigmab^2*(eye(3)-rbGunit*rbGunit')+eps*eye(3);
    wbG = mvnrnd(zeros(3,1),RbG)';

    rbGtilde = rbG + wbG;
    rbGtilde = rbGtilde*b/norm(rbGtilde);
```

A.3 h_meas.m

```
function [zk] = h_meas(xk,wk,RBIBark,rXIMat,mcVeck,P)
% h_meas : Measurement model for quadcopter.
%
% INPUTS
%
% xk --------- 15x1 state vector at time tk, defined as
%
%              xk = [rI', vI', e', ba', bg']'
%
%              where all corresponding quantities are identical to
those
```

```
%              defined for E.statek in stateEstimatorUKF.m and where e
is the
%              3x1 error Euler angle vector defined such that for an
estimate
%              RBIBar of the attitude, the true attitude is RBI =
%              dRBI(e)*RBIBar, where dRBI(e) is the DCM formed from
the error
%              Euler angle vector e.
    rI = xk(1:3);
    vI = xk(4:6);
    e = xk(7:9);
% wk --------- nz-by-1 measurement noise vector at time tk, defined as
%
%              wk = [wpIk', wbIk', w1C', w2C', ..., wNfkC']'
%
%              where nz = 6 + Nfk*3, with Nfk being the number of
features
%              measured by the camera at time tk, and where all 3x1
noise
%              vectors represent additive noise on the corresponding
%              measurements.
%
% RBIBark ---- 3x3 attitude matrix estimate at time tk.
%
% rXIMat ----- Nf-by-3 matrix of coordinates of visual features in the
%              simulation environment, expressed in meters in the I
%              frame. rXIMat(i,:)' is the 3x1 vector of coordinates of
the ith
%              feature.
%
% mcVeck ----- Nf-by-1 vector indicating whether the corresponding
feature in
%              rXIMat is sensed by the camera: If mcVeck(i) is true
(nonzero),
%              then a measurement of the visual feature with
coordinates
%              rXIMat(i,:)' is assumed to be made by the camera.
mcVeck
%              should have Nfk nonzero values.
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m
%
%  sensorParams = Structure containing sensor parameters, as defined
in
%                 sensorParamsScript.m
```

```
        rocB = P.sensorParams.rocB;
        f = P.sensorParams.f;
        RCB = P.sensorParams.RCB;
        raB = P.sensorParams.raB;
%
% OUTPUTS
%
% zk --------- nz-by-1 measurement vector at time tk, defined as
%
%              zk = [rpItilde', rbItildeu', v1Ctildeu', ...,
vNfkCtildeu']'
%
%              where rpItilde is the 3x1 measured position of the
primary
%              antenna in the I frame, rbItildeu is the 3x1 measured
unit
%              vector pointing from the primary to the secondary
antenna,
%              expressed in the I frame, and viCtildeu is the 3x1 unit
vector,
%              expressed in the camera frame, pointing toward the ith
3D
%              feature, which has coordinates rXIMat(i,:)'.
    RBIk = euler2dcm(e)*RBIBark;

    N = size(rXIMat,1);
    rXI = [];
    for j = 1:N
        if mcVeck(j) == 0
            continue
        else
            rXI = [rXI;rXIMat(j,:)];
        end
    end

    Nf = size(rXI,1);
    rCI = rI+RBIk'*rocB;
    v_IMat = zeros(Nf,3);
    for k = 1:Nf
        xI = rXI(k,:)';
        v_I = xI-rCI;
        v_Iu = v_I/norm(v_I);
        v_IMat(k,:) = v_Iu';
    end

    rpItilde = rI+RBIk'*raB(:,1);
    rbB = raB(:,2)-raB(:,1);
    rbBu = rbB/norm(rbB);
    rbIutilde = RBIk'*rbBu;

    hk = [rpItilde;rbIutilde];
    for n = 1:Nf
```

```
        v_IC = RCB*RBIk*v_IMat(n,:)';
        hk = [hk;v_IC];
    end

    zk = hk+wk;
```

A.4 hdCameraSimulator.m

```
function [rx] = hdCameraSimulator(rXI,S,P)
% hdCameraSimulator : Simulates feature location measurements from the
%                     quad's high-definition camera.
%
%
% INPUTS
%
% rXI -------- 3x1 location of a feature point expressed in I in
meters.
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure
with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
        rI = S.statek.rI;
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
        RBI = S.statek.RBI;
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
    rocB = P.sensorParams.rocB;
    RCB = P.sensorParams.RCB;
    Rc = P.sensorParams.Rc;
    pixelSize = P.sensorParams.pixelSize;
    f = P.sensorParams.f;
    K = P.sensorParams.K;
    imagePlaneSize = P.sensorParams.imagePlaneSize;

% OUTPUTS
%
% rx --------- 2x1 measured position of the feature point projection
on the
%             camera's image plane, in pixels.  If the feature point
is not
%             visible to the camera (the ray from the feature to the
camera
%             center never intersects the image plane, or the feature
is
```

```
%               behind the camera), then rx is an empty matrix.
    tB = RBI*rI+rocB;
    tC = -(RCB*tB);
    RCI = RCB*RBI;
    I2Cmat = [RCI,tC;0 0 0 1];
    ProjMat = [K,[0;0;0]]*I2Cmat;
    x = ProjMat*[rXI;1];
    xc = [x(1)/x(3);x(2)/x(3)];
    wc = mvnrnd(zeros(2,1),Rc)';
    rXC = I2Cmat*[rXI;1];
    rXC = rXC(1:3);
    imagePlane_pixel = imagePlaneSize/pixelSize;
    rx = xc/pixelSize + wc;

    if (rXC(3)<=f)
        rx = [];
    elseif (rx(1)>imagePlane_pixel(1)/2)||(rx(1)<-
imagePlane_pixel(1)/2)
        rx = [];
    elseif (rx(2)>imagePlane_pixel(2)/2)||(rx(2)<-
imagePlane_pixel(2)/2)
        rx = [];
    end

  end
```

A.5 imuSimulator.m

```
function [ftildeB,omegaBtilde] = imuSimulator(S,P)
% imuSimulator : Simulates IMU measurements of specific force and
%                body-referenced angular rate.
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure
with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
        rI = S.statek.rI;
%                  vI = 3x1 velocity of CM with respect to the I frame
and
%                       expressed in the I frame, in meters per
second.
        vI = S.statek.vI;
%                  aI = 3x1 acceleration of CM with respect to the I
frame and
%                       expressed in the I frame, in meters per
second^2.
```

```
        aI = S.statek.aI;
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
        RBI = S.statek.RBI;
%              omegaB = 3x1 angular rate vector expressed in the body
frame,
%                       in radians per second.
        omegaB = S.statek.omegaB;
%           omegaBdot = 3x1 time derivative of omegaB, in radians per
%                       second^2.
        omegaBdot = S.statek.omegaBdot;
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
        IMUdelt = P.sensorParams.IMUdelt;
        lB = P.sensorParams.lB;
        Sa = P.sensorParams.Sa;
        Qa = P.sensorParams.Qa;
        taua = P.sensorParams.taua;
        sigmaa = P.sensorParams.sigmaa;
        alphaa = P.sensorParams.alphaa;
        Qa2 = P.sensorParams.Qa2;
        Sg = P.sensorParams.Sg;
        Qg = P.sensorParams.Qg;
        taug = P.sensorParams.taug;
        sigmag = P.sensorParams.sigmag;
        alphag = P.sensorParams.alphag;
        Qg2 = P.sensorParams.Qg2;

%     constants = Structure containing constants used in simulation
and
%                 control, as defined in constantsScript.m
        g = P.constants.g;
% OUTPUTS
%
persistent ba bg va2 vg2
if(isempty(ba))
    % Set ba's initial value
    QbaSteadyState = P.sensorParams.Qa2/(1 - P.sensorParams.alphaa^2);
    ba = mvnrnd(zeros(3,1), QbaSteadyState)';
end
if(isempty(bg))
    % Set bg's initial value
    QbgSteadyState = P.sensorParams.Qg2/(1 - P.sensorParams.alphag^2);
    bg = mvnrnd(zeros(3,1), QbgSteadyState)';
end
if(isempty(va2))
    % Set va2's initial value
    va2 = mvnrnd(zeros(3,1),Qa2)';
end
if(isempty(vg2))
```

```
    % Set vg2's initial value
    vg2 = mvnrnd(zeros(3,1),Qg2)';
end

% ftildeB ---- 3x1 specific force measured by the IMU's 3-axis
accelerometer
    e3 = [0;0;1];
    va = mvnrnd(zeros(3,1),Qa)';
    ba = alphaa*ba+va2;
    va2 = mvnrnd(zeros(3,1),Qa2)';

    ftildeB = RBI*(aI+g*e3)+ba+va;
% omegaBtilde  3x1 angular rate measured by the IMU's 3-axis rate gyro
    vg = mvnrnd(zeros(3,1),Qg)';
    bg = alphag*bg+vg2;
    vg2 = mvnrnd(zeros(3,1),Qg2)';

    omegaBtilde = omegaB+bg+vg;
```

A.6 topSimulateControl.m

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

% 'clear all' is needed to clear out persistent variables from run to
run
clear all; clc;
% Seed Matlab's random number: this allows you to simulate with the
same noise
% every time (by setting a nonnegative integer seed as argument to
rng) or
% simulate with a different noise realization every time (by setting
% 'shuffle' as argument to rng).
%rng('shuffle');
rng('shuffle');
% Assert this flag to call the full estimation and control simulator;
% otherwise, only the control simulator is called
estimationFlag = 1;
% Total simulation time, in seconds
Tsim = 10;
% Update interval, in seconds
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec=[0:N-1]'*delt;
% Angular rate of orbit, in rad/sec
n = 2*pi/10;
% Radius of circle, in meters
r = 2;
% Populate reference trajectory
R.tVec = tVec;
```

```
R.rIstar = [r*cos(n*tVec),r*sin(n*tVec),ones(N,1)];
R.vIstar = [-r*n*sin(n*tVec),r*n*cos(n*tVec),zeros(N,1)];
R.aIstar = [-r*n*n*cos(n*tVec),-r*n*n*sin(n*tVec),zeros(N,1)];
% The desired xI points toward the origin. The code below also
normalizes
% each row in R.xIstar.
R.xIstar = diag(1./vecnorm(R.rIstar'))*(-R.rIstar);
% Matrix of disturbance forces acting on the body, in Newtons,
expressed in I
S.distMat = 0*randn(N-1,3);
% Initial position in m
S.state0.r = [r 0 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 pi]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in
rad/s
S.state0.omegaB = [0 0 0]';
% Oversampling factor
S.oversampFact = 2;
% Feature locations in the I frame
S.rXIMat = [0,0,0; 0,0,0.7];
%S.rXIMat = [];
% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
sensorParamsScript;
P.quadParams = quadParams;
P.constants = constants;
P.sensorParams = sensorParams;

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

S2.tVec = Q.tVec;
S2.rMat = Q.state.rMat;
S2.eMat = Q.state.eMat;
S2.plotFrequency = 20;
S2.makeGifFlag = false;
S2.gifFileName = 'testGif.gif';
S2.bounds=2.5*[-1 1 -1 1 -0.1 1];
%visualizeQuad(S2);

%get vector from CM in direction of xI projected onto XY plane
lg = length(Q.state.eMat);
xIvec = [];
for mm = 1:lg
    RBIk = euler2dcm(Q.state.eMat(mm,:)');
```

```
    xk = RBIk'*[1;0;0];
    xIvec = [xIvec;xk'];
end

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

figure(3);clf;
psiError = unwrap(n*Q.tVec + pi - Q.state.eMat(:,3));
meanPsiErrorInt = round(mean(psiError)/2/pi);
plot(Q.tVec,psiError - meanPsiErrorInt*2*pi,'LineWidth',1.5);
grid on;
xlabel('Time (sec)');
ylabel('\Delta \psi (rad)');
title('Yaw angle error');

figure(2);clf;
plot(Q.state.rMat(:,1), Q.state.rMat(:,2),'LineWidth',1.5);
axis equal; grid on;hold on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');
quiver(Q.state.rMat(:,1), Q.state.rMat(:,2),xIvec(:,1),xIvec(:,2));

%calculate performance
xerror = Q.state.rMat(1:2:end,1)-R.rIstar(:,1);
meanxerr = round(mean(xerror));
yerror = Q.state.rMat(1:2:end,2)-R.rIstar(:,2);
meanyerr = round(mean(yerror));
zerror = Q.state.rMat(1:2:end,3)-R.rIstar(:,3);
meanzerr = round(mean(zerror));
figure(4);clf;
plot(tVec,xerror-meanxerr,tVec,yerror-meanyerr,tVec,zerror-
meanzerr,'LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Error (m)');
title('Tracking Error Relative To Desired Trajectory');
legend('x','y','z');
```

A.7 topSimulateControl_new.m

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

% 'clear all' is needed to clear out persistent variables from run to
run
clear all; clc;
```

```
% Seed Matlab's random number: this allows you to simulate with the
same noise
% every time (by setting a nonnegative integer seed as argument to
rng) or
% simulate with a different noise realization every time (by setting
% 'shuffle' as argument to rng).
rng('shuffle');
%rng(1234);
% Total simulation time, in seconds
Tsim = 10;
% Update interval, in seconds
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec=[0:N-1]'*delt;
% Angular rate of orbit, in rad/sec
n = 2*pi/10;
% Radius of circle, in meters
r = 2;
% Populate reference trajectory
R.tVec = tVec;
R.rIstar = [r*cos(n*tVec),r*sin(n*tVec),ones(N,1)];
R.vIstar = [-r*n*sin(n*tVec),r*n*cos(n*tVec),zeros(N,1)];
R.aIstar = [-r*n*n*cos(n*tVec),-r*n*n*sin(n*tVec),zeros(N,1)];
% The desired xI points toward the origin. The code below also
normalizes
% each row in R.xIstar.
R.xIstar = diag(1./vecnorm(R.rIstar'))*(-R.rIstar);
% Matrix of disturbance forces acting on the body, in Newtons,
expressed in I
S.distMat = 0*randn(N-1,3);
% Initial position in m
S.state0.r = [r 0 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 pi]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in
rad/s
S.state0.omegaB = [0 0 0]';
% Oversampling factor
S.oversampFact = 2;
% Feature locations in the I frame
S.rXIMat = [0,0,0; 0,0,0.7];
%S.rXIMat = [];
% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
sensorParamsScript;
P.quadParams = quadParams;
P.constants = constants;
P.sensorParams = sensorParams;
```

```
%when controller is fed true state
Q = simulateQuadrotorControl(R,S,P);

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

figure(2);clf;
plot(Q.state.rMat(:,1), Q.state.rMat(:,2),'LineWidth',1.5);
axis equal; grid on;hold on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');

figure(3);clf;
psiError = unwrap(n*Q.tVec + pi - Q.state.eMat(:,3));
meanPsiErrorInt = round(mean(psiError)/2/pi);
plot(Q.tVec,psiError - meanPsiErrorInt*2*pi,'LineWidth',1.5);
grid on;hold on;
xlabel('Time (sec)');
ylabel('\Delta \psi (rad)');
title('Yaw angle error');

%add performance with estimated states
for jj = 1:3
    Q = simulateQuadrotorEstimationAndControl(R,S,P);

    figure(1)
    plot(Q.tVec,Q.state.rMat(:,3),':r','LineWidth',1.5);

    figure(2)
    plot(Q.state.rMat(:,1), Q.state.rMat(:,2),':r','LineWidth',1.5);

    figure(3)
    psiError = unwrap(n*Q.tVec + pi - Q.state.eMat(:,3));
    meanPsiErrorInt = round(mean(psiError)/2/pi);
    plot(Q.tVec,psiError - meanPsiErrorInt*2*pi,':r','LineWidth',1.5);
end

figure(1)
legend('True State','Estimated State (3X)');
figure(2)
legend('True State','Estimated State (3X)');
figure(3)
legend('True State','Estimated State (3X)');
```

A.8 wahbaSolver.m

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

    [U,S,V] = svd(B);
    %testsvd = U*S*V';
    M = [1 0 0;0 1 0;0 0 det(U)*det(V)];

    RBI = U*M*V';
```

A.9 wahbaTester.m

```
%test wahbaSolver
clear;clc;

n = 1000;
```

```
testMat = zeros(n,1);
for jj = 1:n
    clear aVec vIMat vBMat;
    %generate random rotation matrix
    phi = (pi()/2+pi()/2)*rand-pi()/2;
    theta = ((pi()-.00001)+pi())*rand-pi();
    psi = ((pi()-.00001)+pi())*rand-pi();
    e = [phi;theta;psi];
    Rin = euler2dcm(e);
    %create random set of unit vectors vI
    N = 1000;
    vIMat=[];
    for ii = 1:N
        vIi = randn(3,1);
        vIiu = vIi/norm(vIi);
        vIMat = [vIMat;vIiu'];
    end

    %create random set of transformed unit vectors vB
    vBMat = [];
    for kk = 1:N
        vIk = vIMat(kk,:)';
        vBk = Rin*vIk;
        vBMat = [vBMat;vBk'];
    end
    %create weighting factor so all a values add up to 1
    a = 1/N;
    aVec = a*ones(N,1);

    %test wahbaSolver to see if Rout=Rin
    Rout = wahbaSolver(aVec,vIMat,vBMat);
    %test to 6 decimal points
    roundRin = fix(Rin*1e6)/1e6;
    roundRout = fix(Rout*1e6)/1e6;
    if isequal(roundRin,roundRout)
        testMat(jj)=1;
    else
        testMat(jj)=0;
    end
end
```