Final Project

Greg Anders

May 4, 2017

1 Introduction

For this project, a Monte Carlo simulation with $n = 100$ runs was performed to evaluate a designed attitude determination and control system. The simulation parameters are described below.

- **Orbit parameters**
  - **Semi-major axis:** $a = 6778\,\mathrm{km}$
  - **Eccentricity:** $e = 0$
  - **Inclination:** $I = 30^\circ$
  - **Longitude of the ascending node:** $\Omega = 0^\circ$
  - **Orbit period:** $T = 92.425\,\mathrm{min}$
- **Simulated trajectory:** One-half of the orbit
  - **Simulation time:** $t_f = \frac{T}{2} = 46.213\,\mathrm{min}$
  - **Initial position:** $\boldsymbol{r}_0 = [a, 0, 0]^T$
  - **Final position:** $\boldsymbol{r}_f = [-a, 0, 0]^T$
- **Simulation frequency:** $50\,\mathrm{Hz}$
- **Flight computer frequency:** $10\,\mathrm{Hz}$
- **Satellite inertia matrix:** $\mathbf{J} = \begin{bmatrix} 90 & 0 & 0 \\ 0 & 70 & 0 \\ 0 & 0 & 60 \end{bmatrix}$

At the beginning of the simulation, the satellite is facing earth and is in an Earth-pointing trajectory, that is, at time $t = t_0$,

$$\hat{\boldsymbol{b}}_1 = -\hat{\boldsymbol{i}}_1 \qquad \hat{\boldsymbol{b}}_2 = \cos I \cdot \hat{\boldsymbol{i}}_2 + \sin I \cdot \hat{\boldsymbol{i}}_3 \qquad \hat{\boldsymbol{b}}_3 = \sin I \cdot \hat{\boldsymbol{i}}_2 - \cos I \cdot \hat{\boldsymbol{i}}_3$$

In quaternion form, the attitude is represented as

$$\boldsymbol{q}_0 = \begin{bmatrix} 0 \\ 0.9659 \\ 0.2588 \\ 0 \end{bmatrix}$$

The reference angular velocity is $\bar{\omega} = [0, 0, -n]^T$ where n is the orbital angular velocity and is defined as

$$n = \sqrt{\frac{\mu}{a^3}} \left[1 + \frac{3}{2} \left(\frac{r_E}{a} \right)^2 J_2 \left(1 - 3\cos^2 I \right) \right] \text{rad/sec} \tag{1}$$

where μ is the gravitational constant of Earth, a is the semi-major axis of the satellite orbit, r_E is the radius of Earth, J_2 is a constant representing the J2 obliquity perturbations, and I is the orbit inclination.

2 Perturbations and tumbling motion

The perturbations modeled in the simulation include the gravity gradient, J2 perturbations, and the Earth's magnetic field.

2.1 Gravity gradient

The torque applied by the gravity gradient is modeled as:

$$\boldsymbol{\tau}_{gg} = 3n^2 \hat{\boldsymbol{r}} \times \mathbf{J} \hat{\boldsymbol{r}} \tag{2}$$

where n is the mean motion of the satellite defined in (1), $\mathbf{J}$ is the satellite inertia matrix, and $\hat{\boldsymbol{r}}$ is the unit vector pointing from the satellite toward the Earth in the satellite body frame.

2.2 Magnetic perturbations

Earth's magnetic field is shown in Figure 1. The torque acting on the satellite from the magnetic field is

$$\boldsymbol{\tau}_{mag} = \boldsymbol{\mu} \times \boldsymbol{B} \tag{3}$$

where $\boldsymbol{\mu}$ is the magnetic moment of the satellite (not to be confused with the gravitational constant in (2)) and $\boldsymbol{B}$ is the magnetic field vector at the satellite.

2.3 Tumbling motion

The tumbling motion of the satellite due to these perturbations, along with the reference angular velocity, is shown in Figure 2.

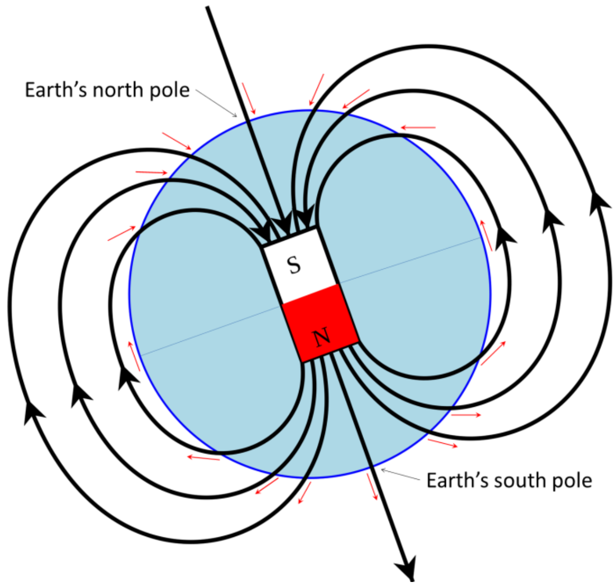


Figure 1: Earth's magnetic field

3 Noise analysis

The statistics of the noise in the simulation are given in Table 1.

Source	Standard deviation
Sun sensor	$0.05°$
Horizon sensor	$0.015°$
Magnetometer	$0.5°$
Gyro noise (ARW)	$0.45\,°/\sqrt{\mathrm{h}}$
Gyro bias rate noise	$4\,°/\mathrm{h}$
Actuator noise (σ_{RCS})	$0.05\,\mathrm{N\,m}$
Initial attitude noise (σ_θ)	$5°$
Initial bias estimate noise (σ_β)	$0.02°$

Table 1: Noise statistics

4 Sensor models

The simulation employs a sun sensor, an Earth horizon sensor, and a magnetometer. Each sensor generates a unit vector in the body frame affected by noise (see Table 1). The body frame measurements are simulated using a reference vector. The reference vector for the sun sensor is arbitrarily chosen as $[1, 0, 0]$ and is assumed to be constant over the simulation interval. The reference vector for the horizon sensor is the vector pointing from the satellite to the Earth, and the reference vector for the magnetometer is the magnetic field vector at the satellite location in the inertial frame.

The body frame measurements and the reference measurements are used to calculate a measurement

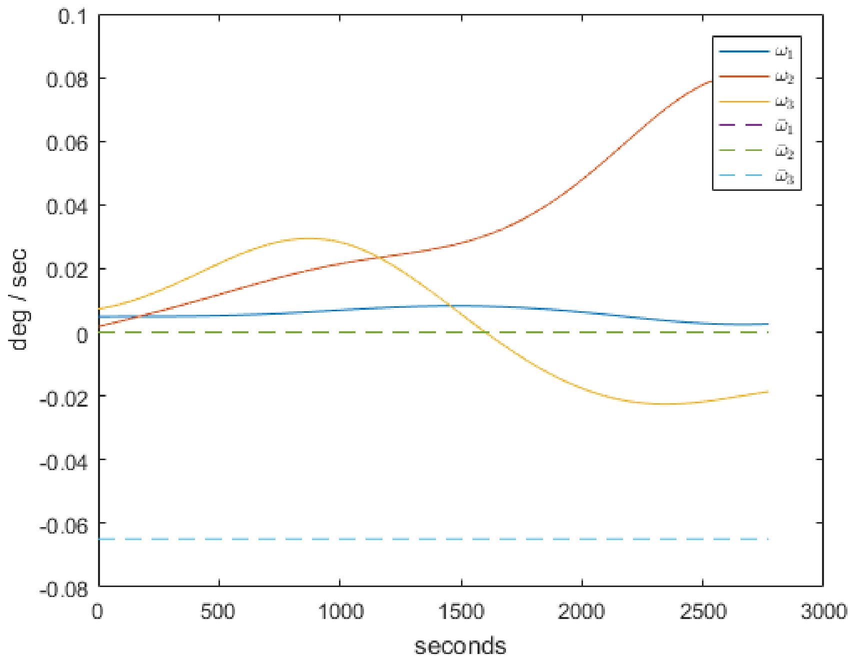


Figure 2: Tumbling motion

quaternion, $\tilde{\boldsymbol{q}}$ using the SVD method to solve Wahba's problem.

A gyroscope is used to measure the angular velocity. The gyroscope measurement model is

$$\tilde{\boldsymbol{\omega}} = \boldsymbol{\omega} + \boldsymbol{\beta} + \boldsymbol{\eta}_1 \tag{4}$$

where $\boldsymbol{\beta}$ is the gyro bias and $\boldsymbol{\eta}_1$ is some random noise characterized as an angular random walk (ARW) (see Table 1). The gyro bias is not constant and evolves as

$$\dot{\boldsymbol{\beta}} = \boldsymbol{\eta}_2$$

where $\boldsymbol{\eta}_2$ is a random variable characterized as gyro bias rate noise (see Table 1).

The measurements $\tilde{\boldsymbol{\omega}}$ and $\tilde{\boldsymbol{q}}$ are fed into a Multiplicative Extended Kalman Filter (MEKF) that estimates the current angular velocity $\hat{\boldsymbol{\omega}}$ and current attitude $\hat{\boldsymbol{q}}$.

The initial covariance matrix used in the MEKF is

$$\mathbf{P}_0 = \begin{bmatrix} \mathbf{P}_{\theta\theta} & \mathbf{0} \\ \mathbf{0} & \mathbf{P}_{\beta\beta} \end{bmatrix}$$

where $\mathbf{P}_{\theta\theta} = \sigma_\theta^2 I$ and $\mathbf{P}_{\beta\beta} = \sigma_\beta^2 I$, where σ_θ^2 and σ_β^2 are the initial attitude and bias estimate variances defined in Table 1.

5 Actuation

A reaction control system (RCS) was used to control the trajectory. The actuator was modeled as having random noise with σ_{RCS} given in Table 1. The phase plane plot for the RCS is given in Figure 3. The control torque generated by the RCS, given a control u, is modeled as $\mathbf{J}\boldsymbol{u}$, where $\mathbf{J}$ is the inertia matrix of the satellite. The control $\boldsymbol{u}$ generated by the controller is designed such that the control torque is constant in each dimension, i.e.

$$\boldsymbol{u} = \mathbf{J}^{-1}\boldsymbol{\tau}$$

where $\boldsymbol{\tau}$ was chosen to be $0.5\,\mathrm{N\,m}$ in each dimension.

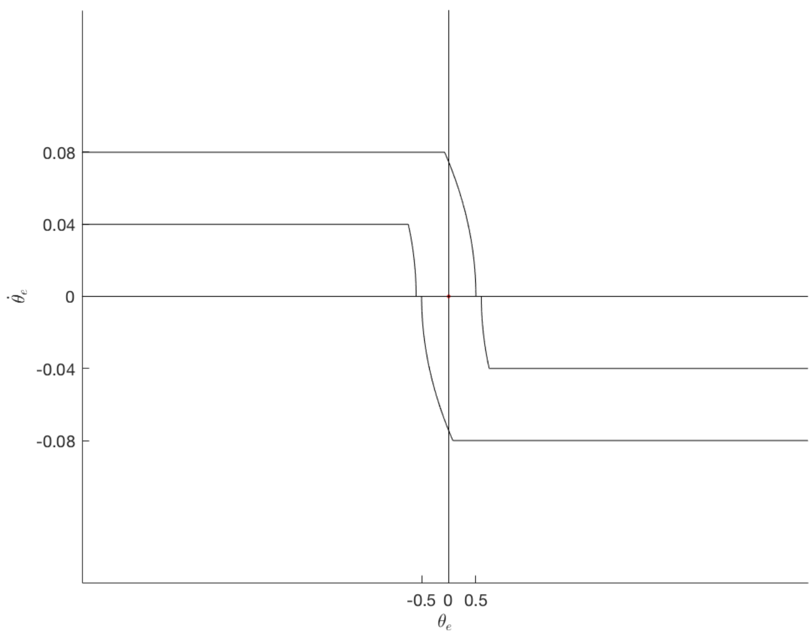


Figure 3: Phase plane of RCS

6 Results

Figure 4 shows the bias estimate error for a single simulation. The red-dashed lines represent the $\pm 3\sigma$ of the covariance $\mathbf{P}$. Figure 4 shows that the bias estimate converges very quickly.

Figure 5 shows the true angular velocity and attitude versus the reference angular velocity and attitude for one simulation. The sharp spikes in the true angular velocity come from the thrust from the RCS. Figure 7 shows the phase plane controller time histories for one simulation. The red dots indicate points where the control law was non-zero. Figure 6 shows the time history of the thruster actuation for one simulation.

For each Monte Carlo run, the gyro bias was initiated to some random value, and the initial attitude was given some random error. The average bias estimate error and attitude pointing error over all Monte Carlo

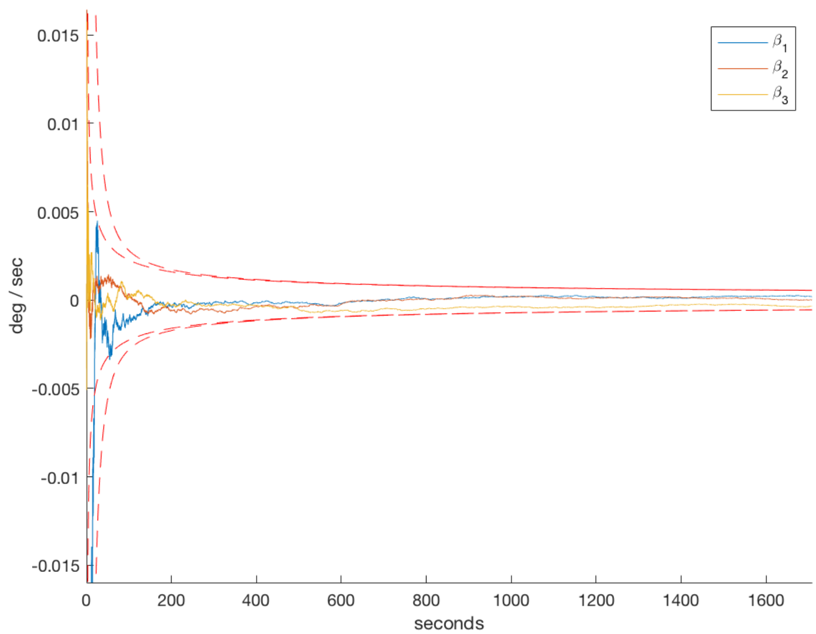
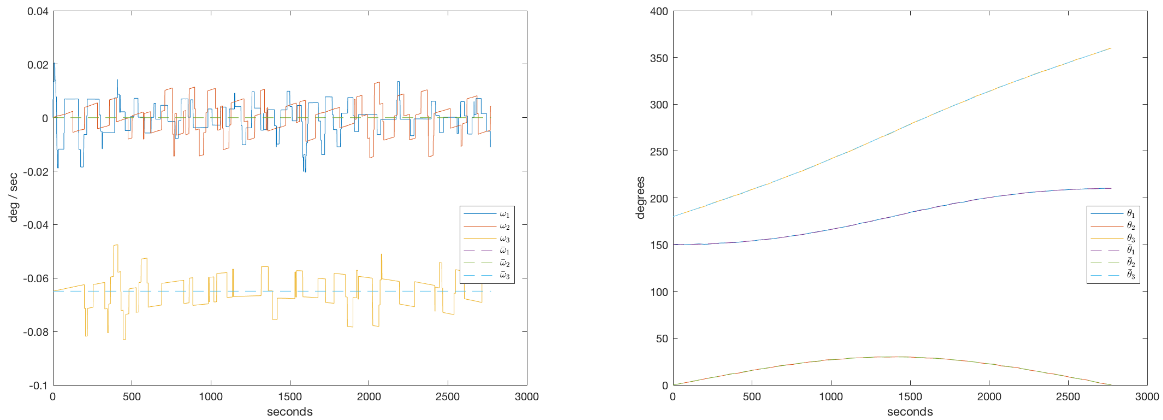


Figure 4: Bias estimate error for one simulation

runs are shown in Figure 8. Over all 100 Monte Carlo runs, the average amount of time that the thrusters fired over the simulation interval was **0.191%**.



(a) (b)

Figure 5: (a) True angular velocity vs reference angular velocity (b) True attitude vs reference attitude

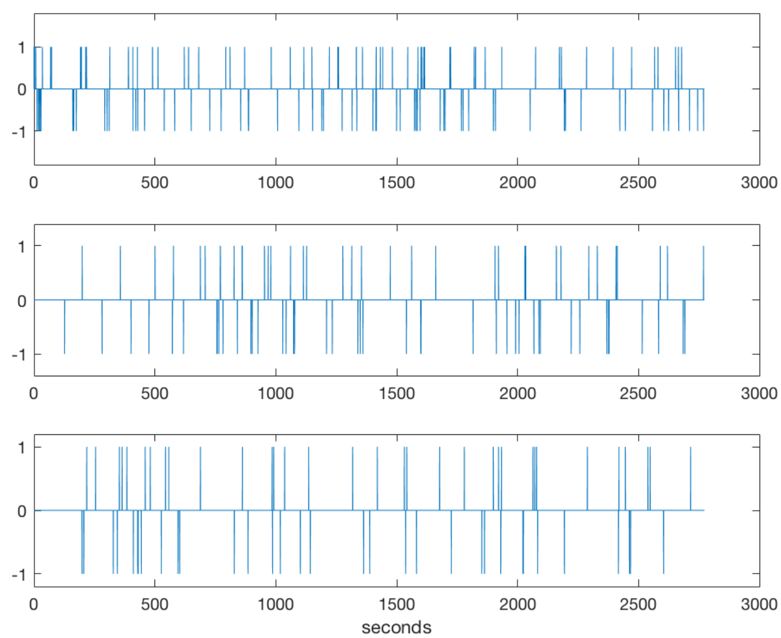
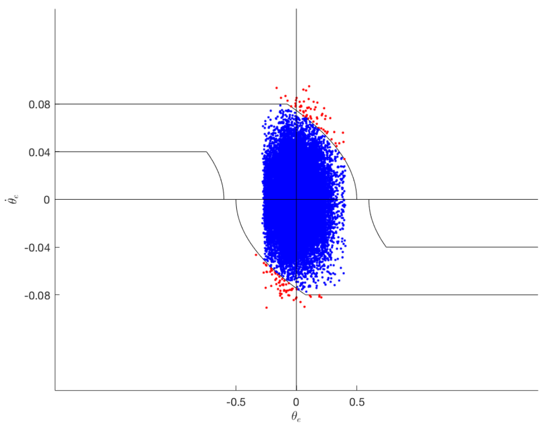
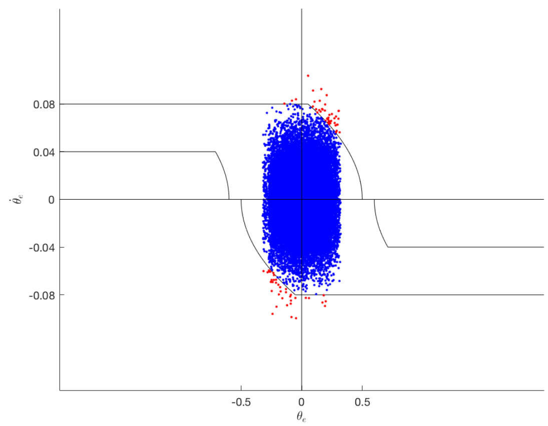


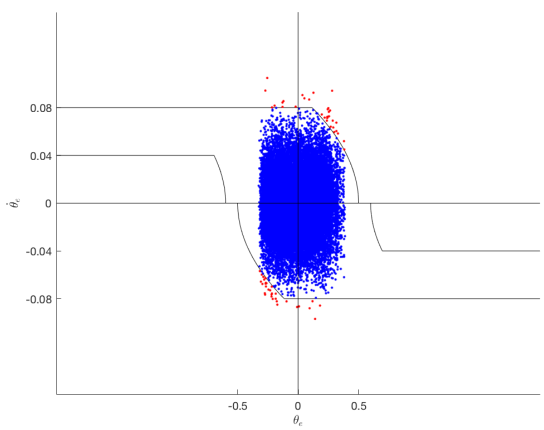
Figure 6: Thruster time history



(a)

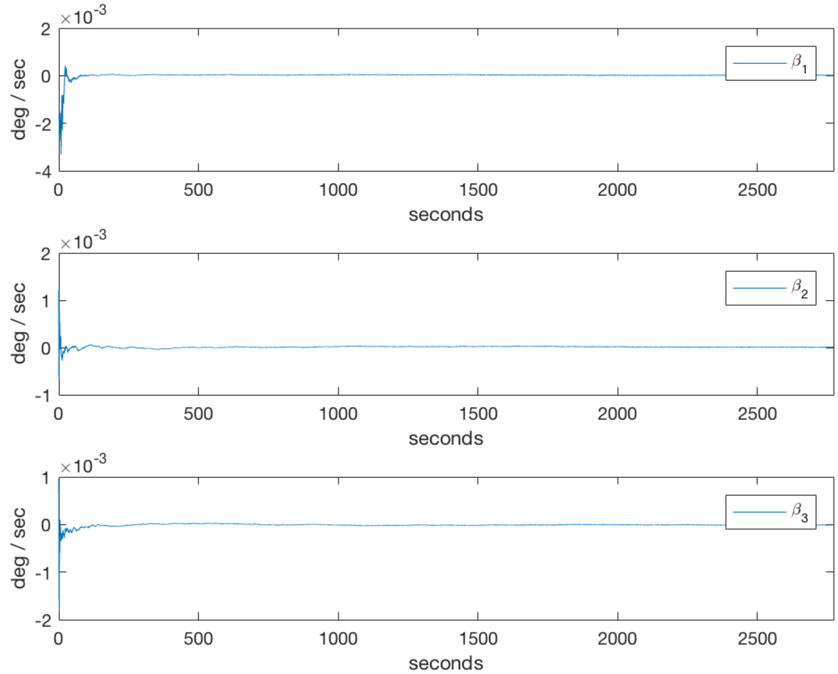


(b)

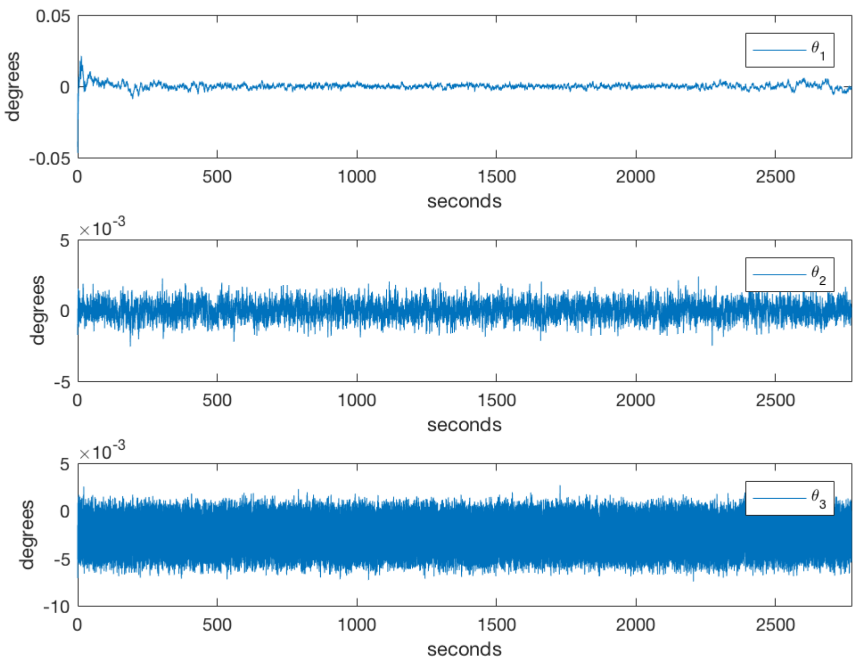


(c)

Figure 7: Phase plane plots for one simulation



(a)



(b)

Figure 8: Results of the Monte Carlo simulation (a) average bias estimate error (b) average attitude pointing error

Appendix

A Source Code

```
%% Simulation

%% Setup
%
% Define constants and utility functions
%
J2 = 1.082e-3;  % J2 constant
rE = 6378e3;    % radius of earth
GM = 3.986004415e14;    % gravitational constant
a = rE + 400e3; % semi-major axis of satellite orbit
n = sqrt(GM / a^3) * (1 + (3/2) * (rE/a)^2 * J2 * (1 - 3*cos(I)^2));   % mean motion of sat
I = pi/6;   % orbit inclination
orbitPeriod = 2*pi / n; % satellite orbit period
J = diag([90 70 60]);  % satellite inertia matrix
fSim = 50;  % simulation sampling frequency
fCom = 10;  % flight computer sampling frequency
magmoment = skew([1 0 0]); % satellite magnetic moment
numTrials = 100;

dt = 1/fSim;
t0 = 0;                                % initial time
tf = 0.5 * orbitPeriod;                  % final time
tVec = 0:dt:tf-dt;                     % time vector
Nt = length(tVec);

% Sun reference measurement (assumed constant)
rSun = [1 0 0]';

% Position as a function of time
x = @(t) a*cos(n*t);
y = @(t) a*sin(n*t)*cos(I);
z = @(t) a*sin(n*t)*sin(I);
r = [x(tVec') y(tVec') z(tVec')];

%
% Configure simulation components
%
```

```
opts = struct( ...
    ... % Simulation sampling frequency
    'SimulationFreq', fSim, ...
    ... % Flight computer sampling frequency
    'ComputerFreq', fCom, ...
    ... % Standard deviation of measurement noise (sun, horizon, magnetometer)
    'MeasurementNoise', [deg2rad(0.05), deg2rad(0.015) deg2rad(0.5)], ...
    ... % Gyro measurement noise - angle random walk (ARW) (deg / sqrt(sec))
    'GyroNoise', deg2rad(0.45)/60, ...
    ... % Gyro bias rate (deg / sec / sec)
    'GyroBiasRateNoise', deg2rad(4)/3600/1000, ...
    ... % Initial attitude error covariance
    'AttitudeError', deg2rad(5), ...
    ... % Initial bias error covariance
    'GyroBiasError', deg2rad(0.02), ...
    ... % Actuator type
    'Actuator', 'rcs', ...
    ... % Actuator noise (Newton-meters)
    'ActuatorNoise', 0.05, ...
    ... % Configure RCS
    'Rcs', struct( ...
        'Thrust', 0.5 ./ diag(J), ...
        'UpperLim', 0.08, ...
        'LowerLim', 0.04, ...
        'Deadband', 1 ...
    ), ...
    'RelTol', 1e-9, ...
    'AbsTol', 1e-9 ...
);

%
% Initialize and preallocate variables
%

% Reference angular velocity
wRef = [0 0 -n]';

% Reference quaternion (pointing toward Earth)
qRef0 = quat2vec(qTrue(1));
[~, y] = ode45(@(~, q) quat2vec(qmult(quat(wRef/2), quat(q)))', tVec, qRef0, opts);
qRef = quat(y);
```

```matlab

% Track time history of theta and theta dot for phase plane plot
dtheta = zeros(Nt, 3);
dthetadot = zeros(Nt, 3);

% Calculate matrices used in linearized dynamics
A = J \ (skew(J*wRef) - skew(wRef)*J);
phiA = @(t, t0) expm(A * (t - t0));
Ad = phiA(dt, 0);
Bd = integral(@(u) phiA(dt, u), 0, dt, 'ArrayValued', true) / J;

attError = zeros(Nt, 3, numTrials);
biasEstError = zeros(Nt, 3, numTrials);
onTimeRatio = zeros(numTrials, 1);

%
%% Main simulation loop
%
for jj = 1:numTrials
    % Initialize true angular velocity
    wTrue = zeros(Nt, 3);
    wTrue(1, :) = wRef;

    % Angular velocity estimate
    wEst = zeros(Nt, 3);
    wEst(1, :) = wRef;

    % Initalize true quaternion
    qTrue = repmat(quat([0 0 0]), Nt, 1);
    qTrue(1) = eul2quat(-I, pi, 0);

    % Initialize gyro estimate (initial estimate is zero)
    betaEst = zeros(Nt, 3);
    betaEst(1, :) = [0 0 0];

    % Initialize quaternion estimate with some random error
    qEst = repmat(quat([0 0 0]), Nt, 1);
    dtheta0 = opts.AttitudeError * rand([3 1]);
    qEst(1) = qnormalize(qmult(quat([dtheta0/2; 1]), qTrue(1)));

    % Initialize true gyro bias
```

```matlab
    betaTrue = zeros(Nt, 3);
    betaTrue(1, :) = opts.GyroBiasError * randn([3 1]);

    % Track time history of control law
    u = zeros(Nt, 3);

    % Initialize covariance
    P = zeros(6, 6, Nt);
    P(:, :, 1) = blkdiag(opts.AttitudeError^2 * eye(3), opts.GyroBiasError^2 * eye(3));

    for ii = 1:Nt
        % True values
        t = tVec(ii);
        w = wTrue(ii, :)';
        q = qTrue(ii);
        beta = betaTrue(ii, :)';
        rI = r(ii, :)';
        T = quat2rotm(q)';

        % Calculate magnetic field at time t
        rMag = magfield(rI);

        % Should the computer fire?
        if ~mod(t - t0 - dt, 1/opts.ComputerFreq)
            % Calculate reference horizon sensor measurement
            rEarth = -rI / norm(rI);

            % Put all measurements into a matrix and normalize
            ref = normc([rSun rEarth rMag]);

            % Get sensor measurements
            [wTilde, qTilde, R] = sensors(w, q, ref, beta, opts);

            % Estimate angular velocity and attitude
            [wHat, qHat, P(:, :, ii)] = nav( ...
                wTilde, ...
                qTilde, ...
                R, ...
                qEst(ii-1), ...
                wEst(ii-1, :)', ...
                betaEst(ii-1, :)', ...
```

```
                P(:, :, ii-1), ...
                opts ...
            );

            % Calculate control
            [u(ii, :), dtheta(ii, :), dthetadot(ii, :)] = controller(wHat, qHat, wRef, qRef(ii), opts);

            % Update estimate time history
            wEst(ii, :) = wHat';
            betaEst(ii, :) = (wTilde - wHat)';
            qEst(ii) = qHat;
        else
            % If computer doesn't fire, control is zero
            u(ii, :) = zeros(1, 3);

            if ii > 1
                % Computer doesn't update, so use same estimate as last epoch
                wEst(ii, :) = wEst(ii-1, :);
                betaEst(ii, :) = betaEst(ii-1, :);
                qEst(ii) = qEst(ii-1);
                P(:, :, ii) = P(:, :, ii-1);
            end
        end

        if ii < Nt
            % Convert control law into a torque
            tau = actuator(u(ii, :)', J, opts);

            % Calculate external torques in body frame
            M = sum([ ...
                gravgrad(J, T*rI, n), ...           % gravity gradient
                magmoment * (T*rMag), ...           % magnetic torque
                tau ...                                     % control torque
                ], 2);

            % Propagate angular velocity using linearized dynamics
            dw = Ad*(w - wRef) + Bd*M;
            wTrue(ii+1, :) = (wRef + dw)';

            % Propagate attitude quaternion
```

```
            qTrue(ii+1) = qpropagate(q, w, dt);

            % Update bias random walk
            betaTrue(ii+1, :) = beta + (opts.GyroBiasRateNoise*dt) * randn([3 1]);

            % Calculate attitude estimate error
            qe = qmult(qTrue(ii), qconj(qEst(ii)));
            attError(ii, :, jj) = 2*qe.v / qe.s;
        end
    end

    % Thruster on-time percentage
    onTimeRatio(jj) = sum(any(u, 2)) / Nt * 100;

    % Bias estimate error for this MC run
    biasEstError(:, :, jj) = betaTrue - betaEst;
end

%% Plot results
close all;

%% Plot true bias and bias estimate
figure;
plot(tVec, rad2deg(betaTrue), tVec, rad2deg(betaEst), '--');
h = legend('$\beta_1$', '$\beta_2$', '$\beta_3$', '$\hat{\beta}_1$', '$\hat{\beta}_2$', '$\hat{\beta}_3$');
set(h, 'Interpreter', 'latex');
xlabel('seconds');
ylabel('deg / sec');
title('Gyro bias (true vs. estimated)');

% Plot bias estimate error
figure;
hold on;
plot(tVec, rad2deg(betaTrue - betaEst));
sigma = zeros(Nt, 3);
for ii = 1:Nt
    d = diag(P(4:6, 4:6, ii));
    sigma(ii, :) = sqrt(d);
end
plot(tVec, rad2deg(3*sigma), 'r--', tVec, rad2deg(-3*sigma), 'r--');
```

```
xlabel('seconds');
ylabel('deg / sec');
title('Bias estimate error');
legend('\beta_1', '\beta_2', '\beta_3');

%% Plot true angular velocity vs reference
figure;
plot(tVec, rad2deg(wTrue), [tVec(1) tVec(end)], rad2deg([wRef wRef]'), '--');
h = legend('$\omega_1$', '$\omega_2$', '$\omega_3$', '$\bar{\omega}_1$', '$\bar{\omega}_2$',
     '$\bar{\omega}_3$');
set(h, 'Interpreter', 'latex');
xlabel('seconds');
ylabel('deg / sec');
title('True angular velocity vs reference');

%% Plot true attitude vs reference
% Convert to angles
eulTrue = zeros(Nt, 3);
eulRef = zeros(Nt, 3);
attE = zeros(Nt, 3);
sigma = zeros(Nt, 3);
for ii = 1:Nt
    [eulTrue(ii, 1), eulTrue(ii, 2), eulTrue(ii, 3)] = quat2eul(qTrue(ii));
    [eulRef(ii, 1), eulRef(ii, 2), eulRef(ii, 3)] = quat2eul(qRef(ii));
    dq = qmult(qTrue(ii), qconj(qEst(ii)));
    attE(ii, :) = 2*dq.v / dq.s;
    d = diag(P(1:3, 1:3, ii));
    sigma(ii, :) = sqrt(d);
end

% Plot true attitude vs reference
figure;
plot(tVec, rad2deg(unwrap(eulTrue)), tVec, rad2deg(unwrap(eulRef)), '--');
xlabel('seconds');
ylabel('degrees');
h = legend('$\theta_1$', '$\theta_2$', '$\theta_3$', '$\bar{\theta}_1$', '$\bar{\theta}_2$',
     '$\bar{\theta}_3$');
set(h, 'Interpreter', 'latex');
title('True attitude vs. reference attitude');

% Plot attitude error
```

```
figure;
for k = 1:3
    subplot(3, 1, k), ...
        hold on, ...
        plot(tVec, rad2deg(attE(:, k))), ...
        plot(tVec, rad2deg(3*sigma(:, k)), 'r--', tVec, rad2deg(-3*sigma(:, k)), 'r--'), ...
        legend(['\theta_' num2str(k)]), xlabel('seconds'), ylabel('degrees');
    if k == 1
        title('Attitude Error (true vs. estimate)');
    end
end

%% Plot phase plane
plotphaseplane(rad2deg(dtheta(:, 1)), rad2deg(dthetadot(:, 1)), opts.Rcs.UpperLim, opts.Rcs.LowerLim, opts.Rcs.Deadband, u(:, 1));
plotphaseplane(rad2deg(dtheta(:, 2)), rad2deg(dthetadot(:, 2)), opts.Rcs.UpperLim, opts.Rcs.LowerLim, opts.Rcs.Deadband, u(:, 2));
plotphaseplane(rad2deg(dtheta(:, 3)), rad2deg(dthetadot(:, 3)), opts.Rcs.UpperLim, opts.Rcs.LowerLim, opts.Rcs.Deadband, u(:, 3));

%% Plot thruster time history
figure;
subplot(3, 1, 1);
plot(tVec, u(:, 1));
subplot(3, 1, 2);
plot(tVec, u(:, 2));
subplot(3, 1, 3);
plot(tVec, u(:, 3));
```

```
function [ M ] = actuator( u, J, opts )
%ACTUATOR Actuate a control into a torque.
%   M = ACTUATOR(U) creates a torque M from the control U

if any(u)
    % Only add noise if the actuator actually does something
    switch (opts.Actuator)
        case 'rcs'
            % In a reaction control system, the control U represents the desired
            % angular acceleration. The control torque is therefore J*U, where J
            % is the inertia matrix of the spacecraft
            M = J*u + (u ~= 0) .* (opts.ActuatorNoise * randn([3 1]));
```

```
        otherwise
            M = zeros(3, 1);
    end
else
    M = zeros(3, 1);

end
```

```
function [ u, dtheta, dthetadot ] = controller( w, q, wref, qref, opts )
%CONTROLLER Calculate control law.
%   U = CONTROLLER(W,Q,WREF,QREF,OPTS) calculates the control law U based on
%   the current angular velocity W and attitude Q and the reference angular
%   velocity WREF and reference attitude QREF. OPTS is a struct containing
%   controller parameters.
%
%   [U,DTHETA,DTHETADOT] = CONTROLLER(W,Q,WREF,QREF,OPTS) also returns the
%   calculated attitude error and its derivative.

dw = w - wref;
dq = qmult(q, qconj(qref));
dtheta = 2*dq.v';
dthetadot = cross(-wref, dtheta) + dw;
u = rcs(dtheta, dthetadot, opts.Rcs);

end
```

```
function [ M ] = gravgrad( J, r, n )
%GRAVGRAD Gravity gradient in body frame, assuming circular orbit.
%   GRAVGRAD(J,R) computes the torque M acting on a satellite body
%   with inertia matrix J at a position R with attitude represented by the
%   quaternion Q. The position R must be in the body frame.

if nargin < 3
    mu = 3.986005e14;
    n = sqrt(mu / norm(r)^3);
end

rHat = r / norm(r);
M = 3 * n^2 * skew(rHat) * J * rHat;

end
```

```
function [ b ] = magfield( r )
%MAGFIELD Calculate magnetic field vector.
%   MAGFIELD(R) calculates the magnetic field vector at the point R in the
%   inertial frame.

B0 = 3.12e-5;
rE = 6378e3;

a = norm(r);
rx = r(1);
ry = r(2);
rz = r(3);

el = acos(rz / a);
az = atan2(ry, rx);

R = [sin(el)*cos(az)    cos(el)*cos(az) -sin(az); ...
     sin(el)*sin(az)    cos(el)*sin(az) cos(az); ...
     cos(el)           -sin(el)        0];

br = -2 * B0 * (rE/a)^3 * cos(el);
bel = -B0 * (rE/a)^3 * sin(el);

b = reshape(R * [br bel 0]', size(r));

end
```

```
function [ wHat, qHat, P ] = nav( wTilde, qTilde, R, qPrev, wPrev, betaPrev, PPrev, opts )
%NAV Estimate angular velocity and attitude.
%   [WHAT,QHAT,P] = NAV(WTILDE,QTILDE,R,QPREV,WPREV,BETAPREV,PPREV,OPTS) is a
%   Multiplicative Extended Kalman Filter (MEKF) that estimates the angular
%   velocity WHAT, attitude quaternion QHAT, and state covariance matrix P given
%   an angular velocity measurment WTILDE, measurement quaternion QTILDE and
%   measurement covariance R, the previous quaternion estimate PREV, the
%   previous gyro bias estimate BETAPREV, and the previous error covariance
%   matrix PPREV. OPTS contains options that define simulation and estimator
%   parameters.
%
%   See also SENSORS, PHIF, CONTROLLER.

% Integration time step
```

```
dt = 1/opts.ComputerFreq;

% Propagate quaternion reference from last time step to current time step
qHat = qpropagate(qPrev, wPrev, dt);

dq = qmult(qTilde, qconj(qHat));
da = 2*dq.v' / dq.s; % Use Gibbs vector parameterization

% Define sensitivity matrix
H = eye(3, 6);

% Define process noise
Q = blkdiag(opts.GyroNoise^2 * eye(3), opts.GyroBiasRateNoise^2 * eye(3));

% Predict P
G = blkdiag(-eye(3), eye(3));
F = phiF(wPrev, dt);
P = F*PPrev*F' + G*(Q*dt)*G';

% Calculate Kalman gain
S = H*P*H' + R;
K = P*H' / S;

dx = K*da;
da = dx(1:3);
dbeta = dx(4:6);
dq = quat(1/sqrt(4 + norm(da)^2) * [da; 2]);
beta = betaPrev + dbeta;

wHat = wTilde - beta;
qHat = qnormalize(qmult(dq, qHat));
P = (eye(6) - K*H)*P;

end
```

```
function [ M ] = phiF( w, dt )
%PHIF Calculate state transition matrix of F
%   M = PHIF(W,DT) calculates the state transition matrix M of the matrix F used
%   in propagating the covariance matrix in the MEKF in NAV. W is the angular
%   velocity and DT is the propagation interval.
%
```

```
%   See also NAV.

% The state transition matrix below was calculated symbolically and copied for
% performance.
alpha = (-w(1)^2 - w(2)^2 - w(3)^2)^(1/2);
beta = alpha^3;
gamma = alpha^5;
delta = norm(w)^2;
c1 = exp(dt*alpha);
c2 = 1/c1;
M = real([ ...
    (w(2)^2*(c1 + c2) + w(3)^2*(c1 + c2) + 2*w(1)^2)/(2*delta), ((w(3)^7*c2)/2 + w(1)*w(2)
        ^3*beta + w(1)^3*w(2)*beta + (w(1)^6*w(3)*c2)/2 + (w(2)^6*w(3)*c2)/2 + (3*w(1)^2*w
        (3)^5*c2)/2 + (3*w(1)^4*w(3)^3*c2)/2 + (3*w(2)^2*w(3)^5*c2)/2 + (3*w(2)^4*w(3)^3*c2)
        /2 - (w(3)*c1*delta^3)/2 + (w(1)*w(2)*c1*gamma)/2 + (w(1)*w(2)*c2*gamma)/2 + 3*w(1)
        ^2*w(2)^2*w(3)^3*c2 + w(1)*w(2)*w(3)^2*beta + (3*w(1)^2*w(2)^4*w(3)*c2)/2 + (3*w(1)
        ^4*w(2)^2*w(3)*c2)/2)/(- w(1)^2 - w(2)^2 - w(3)^2)^(7/2), -(c2*(c1 - 1)*(2*w(1)^2*w
        (2)^3 + 2*w(2)^3*w(3)^2 + w(1)^4*w(2) + w(2)*w(3)^4 + w(2)^5*c1 + w(2)^5 + w(1)^4*w
        (2)*c1 + w(2)*w(3)^4*c1 + 2*w(1)^2*w(2)*w(3)^2 + 2*w(1)^2*w(2)^3*c1 + 2*w(2)^3*w(3)
        ^2*c1 + w(1)*w(3)*beta - w(1)*w(3)*c1*beta + 2*w(1)^2*w(2)*w(3)^2*c1))/(2*gamma), (
        c2*(w(1)^2*w(2)^2 + w(1)^2*w(3)^2 + 2*w(2)^2*w(3)^2 - w(2)^4*exp(2*dt*alpha) - w(3)
        ^4*exp(2*dt*alpha) + w(2)^4 + w(3)^4 - w(1)^2*w(2)^2*exp(2*dt*alpha) - w(1)^2*w(3)
        ^2*exp(2*dt*alpha) - 2*w(2)^2*w(3)^2*exp(2*dt*alpha) + 2*dt*w(1)^2*c1*beta))/(2*
        gamma), -(2*w(3)^3*beta + 2*w(1)^2*w(3)*beta + 2*w(2)^2*w(3)*beta + w(1)*w(2)^5*c1 +
         w(1)^5*w(2)*c1 - w(1)*w(2)^5*c2 - w(1)^5*w(2)*c2 + 2*w(1)^3*w(2)^3*c1 - 2*w(1)^3*w
        (2)^3*c2 + w(3)*c1*gamma + w(3)*c2*gamma + 2*dt*w(1)*w(2)^3*beta + 2*dt*w(1)^3*w(2)*
        beta + w(1)*w(2)*w(3)^4*c1 - w(1)*w(2)*w(3)^4*c2 + 2*w(1)*w(2)^3*w(3)^2*c1 + 2*w(1)
        ^3*w(2)*w(3)^2*c1 - 2*w(1)*w(2)^3*w(3)^2*c2 - 2*w(1)^3*w(2)*w(3)^2*c2 + 2*dt*w(1)*w
        (2)*w(3)^2*beta)/(2*(- w(1)^2 - w(2)^2 - w(3)^2)^(7/2)), (2*w(2)^3*beta + 2*w(1)^2*w
        (2)*beta + 2*w(2)*w(3)^2*beta - w(1)*w(3)^5*c1 - w(1)^5*w(3)*c1 + w(1)*w(3)^5*c2 + w
        (1)^5*w(3)*c2 - 2*w(1)^3*w(3)^3*c1 + 2*w(1)^3*w(3)^3*c2 + w(2)*c1*gamma + w(2)*c2*
        gamma - 2*dt*w(1)*w(3)^3*beta - 2*dt*w(1)^3*w(3)*beta - w(1)*w(2)^4*w(3)*c1 + w(1)*w
        (2)^4*w(3)*c2 - 2*w(1)*w(2)^2*w(3)^3*c1 - 2*w(1)^3*w(2)^2*w(3)*c1 + 2*w(1)*w(2)^2*w
        (3)^3*c2 + 2*w(1)^3*w(2)^2*w(3)*c2 - 2*dt*w(1)*w(2)^2*w(3)*beta)/(2*(- w(1)^2 - w(2)
        ^2 - w(3)^2)^(7/2));
    -(c2*(c1 - 1)*(w(3)*beta - w(1)*w(2)^3 - w(1)^3*w(2) - w(1)*w(2)*w(3)^2 + w(1)*w(2)^3*c1
         + w(1)^3*w(2)*c1 + w(3)*c1*beta + w(1)*w(2)*w(3)^2*c1))/(2*delta^2), (w(1)^2*c1 + w
        (1)^2*c2 + w(3)^2*c1 + w(3)^2*c2 + 2*w(2)^2)/(2*delta), -(c2*(c1 - 1)*(w(1)*w(2)^2 +
         w(1)*w(3)^2 + w(1)^3*c1 + w(1)^3 + w(1)*w(2)^2*c1 + w(1)*w(3)^2*c1 + w(2)*w(3)*
        alpha - w(2)*w(3)*c1*alpha))/(2*beta), -(c2*(w(3)^3*beta - 2*w(1)^3*w(2)^3 - w(1)*w
        (2)^5 - w(1)^5*w(2) + w(1)^2*w(3)*beta + w(2)^2*w(3)*beta + w(3)^3*exp(2*dt*alpha)*
```

```
    beta - w(1)*w(2)*w(3)^4 + w(1)*w(2)^5*exp(2*dt*alpha) + w(1)^5*w(2)*exp(2*dt*alpha)
    - 2*w(1)*w(2)^3*w(3)^2 - 2*w(1)^3*w(2)*w(3)^2 + 2*w(1)^3*w(2)^3*exp(2*dt*alpha) + 2*
    w(3)*c1*gamma + w(1)^2*w(3)*exp(2*dt*alpha)*beta + w(2)^2*w(3)*exp(2*dt*alpha)*beta
    + w(1)*w(2)*w(3)^4*exp(2*dt*alpha) + 2*w(1)*w(2)^3*w(3)^2*exp(2*dt*alpha) + 2*w(1)
    ^3*w(2)*w(3)^2*exp(2*dt*alpha) - 2*dt*w(1)*w(2)*c1*gamma))/(2*(- w(1)^2 - w(2)^2 - w
    (3)^2)^(7/2)), (c2*(w(1)^2*w(2)^2 + 2*w(1)^2*w(3)^2 + w(2)^2*w(3)^2 - w(1)^4*exp(2*
    dt*alpha) - w(3)^4*exp(2*dt*alpha) + w(1)^4 + w(3)^4 - w(1)^2*w(2)^2*exp(2*dt*alpha)
     - 2*w(1)^2*w(3)^2*exp(2*dt*alpha) - w(2)^2*w(3)^2*exp(2*dt*alpha) + 2*dt*w(2)^2*c1*
    beta))/(2*gamma), (c2*(3*w(2)^3*w(3)^5 + 3*w(2)^5*w(3)^3 + w(2)*w(3)^7 + w(2)^7*w(3)
     - w(1)*w(2)^2*gamma + w(1)*w(3)^4*beta - w(2)*w(3)^7*exp(2*dt*alpha) - w(2)^7*w(3)*
    exp(2*dt*alpha) + w(1)^3*w(3)^2*beta + 2*w(1)^2*w(2)*w(3)^5 + 2*w(1)^2*w(2)^5*w(3) +
     w(1)^4*w(2)*w(3)^3 + w(1)^4*w(2)^3*w(3) - 3*w(2)^3*w(3)^5*exp(2*dt*alpha) - 3*w(2)
    ^5*w(3)^3*exp(2*dt*alpha) + 4*w(1)^2*w(2)^3*w(3)^3 - 4*w(1)^2*w(2)^3*w(3)^3*exp(2*dt
    *alpha) + 2*w(1)*w(2)^2*c1*gamma + 2*w(1)*w(3)^2*c1*gamma - w(1)*w(2)^2*exp(2*dt*
    alpha)*gamma + w(1)*w(3)^4*exp(2*dt*alpha)*beta + w(1)*w(2)^2*w(3)^2*beta + w(1)^3*w
    (3)^2*exp(2*dt*alpha)*beta - 2*w(1)^2*w(2)*w(3)^5*exp(2*dt*alpha) - 2*w(1)^2*w(2)^5*
    w(3)*exp(2*dt*alpha) - w(1)^4*w(2)*w(3)^3*exp(2*dt*alpha) - w(1)^4*w(2)^3*w(3)*exp
    (2*dt*alpha) + w(1)*w(2)^2*w(3)^2*exp(2*dt*alpha)*beta + 2*dt*w(2)*w(3)^3*c1*gamma +
     2*dt*w(2)^3*w(3)*c1*gamma))/(2*(w(2)^2 + w(3)^2)*(- w(1)^2 - w(2)^2 - w(3)^2)^(7/2)
    );
(c2*(c1 - 1)*(w(2)*beta + w(1)*w(3)^3 + w(1)^3*w(3) + w(1)*w(2)^2*w(3) - w(1)*w(3)^3*c1
    - w(1)^3*w(3)*c1 + w(2)*c1*beta - w(1)*w(2)^2*w(3)*c1))/(2*delta^2), (c2*(c1 - 1)*(w
    (1)*w(2)^2 + w(1)*w(3)^2 + w(1)^3*c1 + w(1)^3 + w(1)*w(2)^2*c1 + w(1)*w(3)^2*c1 - w
    (2)*w(3)*alpha + w(2)*w(3)*c1*alpha))/(2*beta), (w(1)^2*c1 + w(1)^2*c2 + w(2)^2*c1 +
     w(2)^2*c2 + 2*w(3)^2)/(2*delta),  (c2*(2*w(1)^3*w(3)^3 + w(2)^3*beta + w(1)*w(3)^5
    + w(1)^5*w(3) + w(1)^2*w(2)*beta + w(2)*w(3)^2*beta + w(2)^3*exp(2*dt*alpha)*beta +
    w(1)*w(2)^4*w(3) - w(1)*w(3)^5*exp(2*dt*alpha) - w(1)^5*w(3)*exp(2*dt*alpha) + 2*w
    (1)*w(2)^2*w(3)^3 + 2*w(1)^3*w(2)^2*w(3) - 2*w(1)^3*w(3)^3*exp(2*dt*alpha) + 2*w(2)*
    c1*gamma + w(1)^2*w(2)*exp(2*dt*alpha)*beta + w(2)*w(3)^2*exp(2*dt*alpha)*beta - w
    (1)*w(2)^4*w(3)*exp(2*dt*alpha) - 2*w(1)*w(2)^2*w(3)^3*exp(2*dt*alpha) - 2*w(1)^3*w
    (2)^2*w(3)*exp(2*dt*alpha) + 2*dt*w(1)*w(3)*c1*gamma))/(2*(- w(1)^2 - w(2)^2 - w(3)
    ^2)^(7/2)), -(c2*(w(1)*w(2)^4*beta - 3*w(2)^5*w(3)^3 - w(2)*w(3)^7 - w(2)^7*w(3) -
    3*w(2)^3*w(3)^5 - w(1)*w(3)^2*gamma + w(2)*w(3)^7*exp(2*dt*alpha) + w(2)^7*w(3)*exp
    (2*dt*alpha) + w(1)^3*w(2)^2*beta - 2*w(1)^2*w(2)*w(3)^5 - 2*w(1)^2*w(2)^5*w(3) - w
    (1)^4*w(2)*w(3)^3 - w(1)^4*w(2)^3*w(3) + 3*w(2)^3*w(3)^5*exp(2*dt*alpha) + 3*w(2)^5*
    w(3)^3*exp(2*dt*alpha) - 4*w(1)^2*w(2)^3*w(3)^3 + 4*w(1)^2*w(2)^3*w(3)^3*exp(2*dt*
    alpha) + 2*w(1)*w(2)^2*c1*gamma + 2*w(1)*w(3)^2*c1*gamma + w(1)*w(2)^4*exp(2*dt*
    alpha)*beta - w(1)*w(3)^2*exp(2*dt*alpha)*gamma + w(1)*w(2)^2*w(3)^2*beta + w(1)^3*w
    (2)^2*exp(2*dt*alpha)*beta + 2*w(1)^2*w(2)*w(3)^5*exp(2*dt*alpha) + 2*w(1)^2*w(2)^5*
    w(3)*exp(2*dt*alpha) + w(1)^4*w(2)*w(3)^3*exp(2*dt*alpha) + w(1)^4*w(2)^3*w(3)*exp
    (2*dt*alpha) + w(1)*w(2)^2*w(3)^2*exp(2*dt*alpha)*beta - 2*dt*w(2)*w(3)^3*c1*gamma -
```

```
         2*dt*w(2)^3*w(3)*c1*gamma))/(2*(w(2)^2 + w(3)^2)*(- w(1)^2 - w(2)^2 - w(3)^2)^(7/2)
        ), (c2*(2*w(1)^2*w(2)^2 + w(1)^2*w(3)^2 + w(2)^2*w(3)^2 - w(1)^4*exp(2*dt*alpha) - w
        (2)^4*exp(2*dt*alpha) + w(1)^4 + w(2)^4 - 2*w(1)^2*w(2)^2*exp(2*dt*alpha) - w(1)^2*w
        (3)^2*exp(2*dt*alpha) - w(2)^2*w(3)^2*exp(2*dt*alpha) + 2*dt*w(3)^2*c1*beta))/(2*
        gamma);
    0, 0, 0, 1, 0, 0;
    0, 0, 0, 0, 1, 0;
    0, 0, 0, 0, 0, 1;
    ]);

end
```

```
function [ r ] = qpropagate( q, w, tspan )
%QPROPAGATE Propagate a quaternion over time.
%   R = QPROPAGATE(Q,W,TSPAN) propagates the quaternion Q over the interval
%   TSPAN = [T0 TFINAL] given an angular velocity vector W. If TSPAN is a
%   scalar, T0 is assumed to be 0 and the quaternion is propagated over the
%   interval [0 TSPAN].

r = q;
if isquat(q)
    q = quat2vec(q);
end

if length(tspan) == 2
    dt = diff(tspan);
elseif length(tspan) == 1
    dt = tspan;
else
    error('Invalid argument')
end

% The following is the state transition matrix for the kinematic equation of the
% quaternion. The matrix was calculated symbolically and has been copied here
% for improved performance.
alpha = (-w(1)^2 - w(2)^2 - w(3)^2)^(1/2);
M = [ ...
    cosh((dt*alpha)/2),  (w(3)*sinh((dt*alpha)/2))/alpha, -(w(2)*sinh((dt*alpha)/2))/alpha,
        (w(1)*sinh((dt*alpha)/2))/alpha;
    -(w(3)*sinh((dt*alpha)/2))/alpha, cosh((dt*alpha)/2), (w(1)*sinh((dt*alpha)/2))/alpha, (
        w(2)*sinh((dt*alpha)/2))/alpha;
```

```
    (w(2)*sinh((dt*alpha)/2))/alpha, -(w(1)*sinh((dt*alpha)/2))/alpha, cosh((dt*alpha)/2), (
        w(3)*sinh((dt*alpha)/2))/alpha;
    -(w(1)*sinh((dt*alpha)/2))/alpha, -(w(2)*sinh((dt*alpha)/2))/alpha, -(w(3)*sinh((dt*
        alpha)/2))/alpha, cosh((dt*alpha)/2);
    ];

q(:) = M * q(:);

if isquat(r)
    r = quat(real(q));
else
    r(:) = real(q);
end

end
```

```
function [ u ] = rcs( dtheta, dthetadot, opts )
%RCS Summary of this function goes here
%   Detailed explanation goes here

c1 = opts.UpperLim;
c2 = opts.LowerLim;
a3 = -opts.Deadband/2;
a6 = opts.Deadband/2;
a2 = a3 - 0.1;
a7 = a6 + 0.1;

thrust = opts.Thrust;

u = [0 0 0];
for i = 1:3
    k1 = 1/(2*thrust(i));
    a1 = -k1 * c2^2 + a2;
    a8 = k1 * c2^2 + a7;
    theta = rad2deg(dtheta(i));
    thetadot = rad2deg(dthetadot(i));
    if thetadot >= c1
        u(i) = -thrust(i);
    elseif theta >= (-k1*thetadot^2 + a6) && thetadot > 0 && thetadot < c1
        u(i) = -thrust(i);
    elseif thetadot >= 0 && theta >= a6
```

```
            u(i) = -thrust(i);
        elseif theta >= (k1*thetadot^2 + a7) && thetadot > -c2 && thetadot < 0
            u(i) = -thrust(i);
        elseif theta >= a8 && thetadot > -c2
            u(i) = -thrust(i);
        elseif thetadot <= -c1
            u(i) = thrust(i);
        elseif thetadot <= c2 && theta <= a1
            u(i) = thrust(i);
        elseif theta <= (-k1*thetadot^2 + a2) && thetadot < c2 && thetadot > 0
            u(i) = thrust(i);
        elseif thetadot <= 0 && theta <= a3
            u(i) = thrust(i);
        elseif theta <= (k1*thetadot^2 + a3) && thetadot < 0 && thetadot > -c1
            u(i) = thrust(i);
        elseif thetadot <= -c1
            u(i) = thrust(i);
        end
    end

end
```

```
function [ wTilde, qTilde, R ] = sensors( w, q, r, bias, opts )
%SENSORS Simulate sensors.
%   [WTILDE,QTILDE,R] = SENSORS(W,Q,R,BIAS,OPTS) simulates a gyro sensor
%   measurement with bias BIAS of the angular velocity WTILDE and a quaternion
%   attitude measurement QTILDE with measurement covariance matrix R given the
%   true (simulated) value of the angular velocity W, the true (simulated)
%   attitude quaternion Q, a 3-by-N weighted set of reference measurement
%   vectors R.
%
%   OPTS must be a struct containing parameters that define the sensors.
%   The following parameters are required:
%       - 'MeasurementNoise' : a N-by-1 vector representing the standard
%                              deviation of each corresponding
%                              measurement
%       - 'GyroNoise'        : standard deviation of gyro noise

if ~isstruct(opts)
    error('Final argument must be a struct');
elseif ~isfield(opts, 'MeasurementNoise')
```

```
    error('Missing required option: MeasurementNoise');
elseif ~isfield(opts, 'GyroNoise')
    error('Missing required option: GyroNoise');
end

dt = 1/opts.ComputerFreq;

T = quat2rotm(q)';
b = zeros(size(r));

% Normalize measurements
r = normc(r);

% Calculate noisy measurements in body frame
sigma = opts.MeasurementNoise;
heta = bsxfun(@times, randn([3 length(sigma)]), sigma);
for ii = 1:size(r, 2)
    b(:, ii) = T * r(:, ii) + heta(:, ii);
end

% Normalize after adding noise
b = normc(b);

% Calculate attitude measurement
weights = 1./(opts.MeasurementNoise).^2;
[T, B] = svdatt(b, r, weights);

wTilde = w + bias + (opts.GyroNoise / sqrt(dt)) * randn([3 1]);
qTilde = rotm2quat(T');
R = inv(-B*T' + trace(B*T')*eye(3));

end
```

```
function [ A, B, t ] = svdatt( b, r, w )
%SVDATT Find optimal attitude matrix using SVD method.
%   A = SVDATT(B, R, W) finds the optimal 3-by-3 attitude matrix A from a
%   N-by-3 set of body-frame measurements B and N-by-3 reference frame
%   measurements R and a N-by-1 vector of weights W that minimizes the cost
%   function in Wahba's problem.
%
%   [A, Y] = SVDATT(B, R, W) also returns the computed measurement matrix Y.
```

```
%
%   See also SVD, DAVQ, QUEST, FOAM, ESOQ.

if size(b, 2) < 2
    error('Minimum of 2 measurements required');
end

tic;
B = bsxfun(@times, w, b) * r';
[U, ~, V] = svd(B);
Up = U * diag([1 1 det(U)]);
Vp = V * diag([1 1 det(V)]);

A = Up * Vp';
t = toc;

end
```