CHAPTER 12

Spacecraft

Spacecraft pointing control is an essential technology for all robotic and manned spacecraft. A control system consists of sensing, actuation, and the dynamics of the spacecraft itself. Spacecraft control systems are of many types, but this chapter is only concerned with three-axis pointing. Reaction wheels are used for actuation.

Reaction wheels are used for control through the conservation of angular momentum. The torque on the reaction wheel causes it to spin one way and the spacecraft to spin in the opposite direction. Momentum removed from the spacecraft is absorbed in the wheel. Reaction wheels are classified as *momentum exchange devices*. You can reorient the spacecraft using wheels and without any external torques. Before reaction wheels were introduced, thrusters were often used for orientation control. This would consume propellant, which is undesirable since when you run out of propellant, the spacecraft can no longer be used.

The spacecraft is modeled as a rigid body except for the presence of three reaction wheels that rotate about orthogonal (perpendicular) axes. The shaft of the motor attached to the rotor of the wheel is attached to the spacecraft. Torque applied between the wheel and spacecraft cause the wheel and spacecraft to move in opposite angular directions. We will assume that we have attitude sensors that measure the orientation of the spacecraft. We will also assume that our wheels are ideal with just viscous damping friction.

12-1. Creating a Dynamic Model of the Spacecraft

Problem

The spacecraft is a rigid body with three wheels. Each wheel is connected to the spacecraft, as shown in Figure 12-1.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-0559-4_12) contains supplementary material, which is available to authorized users.

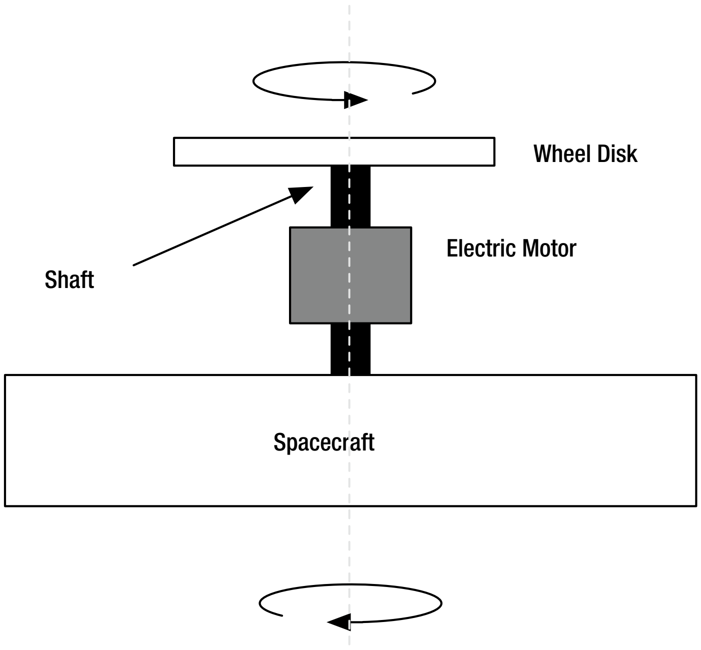

Figure 12-1. *A reaction wheel. The reaction wheel platter spins in one direction and the spacecraft spins in the opposite direction*

Solution

The equations of motion are written using angular momentum conservation. This produces a dynamical model known as the Euler equations with the addition of the spinning wheels. This is sometimes known as a *gyrostat model.*

How It Works

The spacecraft model can be partitioned into dynamics, including the dynamics of the reaction wheels, and the kinematics of the spacecraft. If you assume that the wheels are perfectly symmetric, are aligned with the three body axes, and have a diagonal inertia matrix, you can model the spacecraft dynamics with the following coupled first-order differential equations.

$$I\dot{\omega}+\omega^{\times}\left(I\omega+I_w\left(\omega_w+\omega\right)\right)+I_w\left(\dot{\omega}_w+\dot{\omega}\right)=T \tag{12.1}$$

$$I_w\left(\dot{\omega}_w+\dot{\omega}\right)=T_w \tag{12.2}$$

I is the 3 × 3 inertia matrix; it does not include the inertia of the wheels.

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \tag{12.3}$$

The matrix is symmetric, so $I_{xy} = I_{yx}, I_{xz} = I_{zx}, I_{zy} = I_{yz}$. ω is the angular rate vector for the spacecraft seen in the spacecraft frame.

$$\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{12.4}$$

ω_w is the angular rate of the reaction wheels for wheels 1, 2 and 3.

$$\omega_w = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \tag{12.5}$$

1 is aligned with *x*, 2 with *y*, and 3 with *z*. In this way, the reaction wheels form an orthogonal set and can be used for three-axis control. *T* is the external torque on the spacecraft, which can include external disturbances such as solar pressure or aerodynamic drag, and thruster or magnetic torquer coil torques. T_w is the internal torque on the wheels. I_w is the scalar inertia of the wheels (we assume that they all have the same inertia). You can substitute the second equation into the first to simplify the equations.

$$I\dot{\omega} + \omega^{\times}\left(I\omega + I_w\left(\omega_w + \omega\right)\right) + T_w = T \tag{12.6}$$

$$I_w\left(\dot{\omega}_w + \dot{\omega}\right) = T_w \tag{12.7}$$

This term

$$T_e = \omega^{\times}\left(I\omega + I_w\left(\omega_w + \omega\right)\right) \equiv \omega \times h \tag{12.8}$$

is known as the Euler torque. If the angular rates are small, you can set this term to zero and the equations simplify to

$$I\dot{\omega} + T_w = T \tag{12.9}$$

$$I_w\left(\dot{\omega}_w + \dot{\omega}\right) = T_w \tag{12.10}$$

For kinematics, we use quaternions. A quaternion is a four-parameter representation of the orientation of the spacecraft with respect to the inertial frame. We could use angles since we really only need three states to specify the orientation. The problem with angles is that they have singularities—that is, certain orientations where an angle is undefined, and therefore, are not suitable for simulations. The derivative of the quaternion from the inertial frame to the body frame is

$$\dot{q} = \frac{1}{2}\begin{bmatrix} 0 & \omega^T \\ -\omega & \omega^{\times} \end{bmatrix} \tag{12.11}$$

The term $\omega^{\times}$ is the skew symmetric matrix that is the equivalent of the cross product and is

$$\omega^{\times} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{12.12}$$

The wheel torque is a combination of friction torque and control torque. Reaction wheels are usually driven by DC motors in which the back electromotive force is cancelled by current feedback within the motor electronics. The total reaction wheel torque is therefore

$$T_w = T_c + T_f \tag{12.13}$$

where T_c is the commanded reaction wheel torque and T_f is the friction torque. A simple friction model is

$$T_f = -k_d \omega_k \tag{12.14}$$

k_d is the damping coefficient and ω_k is the angular rate of the kth wheel. If k_d is large, you may have to feedforward it to the controller. This requires careful calibration of the wheel to determine the damping coefficient.

First, you define the data structure for the model that is returned by the dynamics right-hand-side function if there are no inputs. The name of the function is `RHSSpacecraftWithRWA.m`. Use RWA to mean Reaction Wheel Assembly. We say "assembly" because the reaction wheel is assembled from bearings, wheel, shaft, support structure, and power electronics. Spacecraft are built of assemblies.

The default unit vectors for the wheel are along orthogonal axes, such as x, y, and z. The default inertia matrix is the identity matrix, making the spacecraft a sphere. The default reaction wheel inertias are 0.001. All of the non-spinning parts of the wheels are lumped in with the inertia matrix.

```
% Default data structure
if( nargin == 0 )
   xDot = struct('inr',eye(3), 'torque',[0;0;0],'inrRWA', 0.001*[1;1;1],...
                 'torqueRWA',[0;0;0],'uRWA',eye(3), 'damping',[0;0;0]);
   return
end
```

The dynamical equations for the spacecraft are given in the following lines of code. You need to compute the total wheel torque because it is applied both to the spacecraft and the wheels. We use the backslash operator to multiply the equations by the inverse of the inertia matrix. The inertia matrix is positive definite symmetric, so specialized routines can be used to speed computation. It is a good idea to avoid computing inverses, as they can be ill-conditioned—meaning that small errors in the matrix can result in large errors in the inverse.

Save the elements of the state vector as local variables with meaningful names to make reading the code easier. This also eliminates unnecessary multiple extraction of submatrices.

You will notice that the `omegaRWA` variable reads from element 8 to the end of the vector using the `end` keyword. This allows the code to handle any number of reaction wheels. You might just want to control one axis with a wheel or have more than three wheels for redundancy. Be sure that the inputs in `d` match the number of wheels. Since we also input unit vectors, the wheels do not have to be aligned with *x*, *y*, and *z*. Note the use of the backslash operator to solve the set of linear equations for $\dot{\omega}$, `omegaDotCore`.

```
% Save as local variables
q        = x(1:4);
omega    = x(5:7);
omegaRWA = x(8:end);

% Total body fixed angular momentum
h = d.inr*omega + d.uRWA*(d.inrRWA.*(omegaRWA + d.uRWA'*omega));

% Total wheel torque
tRWA = d.torqueRWA - d.damping.*omegaRWA;

% Core angular acceleration
omegaDotCore = d.inr\(d.torque - d.uRWA*tRWA - Cross(omega,h));
```

Note that uRWA is an array of the reaction wheel unit vectors; that is, the spin vectors. In computing h, you have to transform ω into the wheel frame using the transform of uRWA, and then transform back before adding the wheel component to the core component, *I*ω. The wheel dynamics are given in these lines.

```
% Wheel angular acceleration
omegaDotWheel      = tRWA./d.inrRWA - d.uRWA'*omegaDotCore;
```

The total state derivative is in these lines:

```
% State derivative
sW   = [        0 -omega(3) omega(2);...
         omega(3)        0 -omega(1);...
-omega(2) omega(1) 0]; % skew symmetric matrix qD = 0.5*[0, omega';-omega,-sW];
xDot = [qD*q;omegaDotCore;omegaDotWheel];
```

The total inertial angular momentum is an auxiliary output. In the absence of external torques, it should be conserved so it is a good test of the dynamics. A simple way to test angular momentum conservation is to run a simulation with angular rates for all the states, and then rerun it with a smaller timestep. The change in angular momentum should decrease as the timestep is decreased.

```
% Output the inertial angular momentum
if ( nargout > 1 )
   hECI = QTForm( q, h );
end
```

12-2. Computing Angle Errors from Quaternions

Problem

You want to point the spacecraft to a new target attitude (orientation) with the three reaction wheels or maintain the attitude given an external torque on the spacecraft.

Solution

Make 3 PD controllers, one for each axis. You need a function to take two quaternions and compute the small angles between them as input to these controllers.

How It Works

If you are pointing at an inertial target and wish to control about that orientation, you can simplify the rate equations by approximating ω as $\dot{\theta}$, which is valid for small angles when the order of rotation doesn't matter and the Euler angles can be treated as a vector.

$$\dot{\theta} = \omega \tag{12.15}$$

You will also multiply both sides of the Euler equation (equation 12.9) by I^{-1} to solve for the derivatives. Note that T_w, the torque from the wheels, is equivalent to Ia_w, where a is acceleration. Our system equations now become

$$\ddot{\theta} + a_w = a \tag{12.16}$$

$$I_w\left(\dot{\omega}_w + \dot{\omega}\right) = -T_w \tag{12.17}$$

The first equation is now three decoupled second-order equations, just as in Chapter 6. You can stabilize this system with our standard PD controller.

You need attitude angles as input to the PD controllers to compute the control torques. Our examples will only be for small angular displacements from the nominal attitude. You can pass the control code a target quaternion and it will compute Δ angles or you can impose a small disturbance torque.

In these cases, the attitude can be treated as a vector where the order of the rotations doesn't matter. A quaternion derived from small angles is

$$q_\Delta \approx \begin{bmatrix} 1 \\ \theta_1/2 \\ \theta_2/2 \\ \theta_3/2 \end{bmatrix} \tag{12.18}$$

You find the required error quaternion, q_Δ, by multiplying the target quaternion, q_T, with the transpose of the current quaternion

$$q_\Delta = q^T q_T \tag{12.19}$$

This algorithm to compute the angles is implemented in the following code. The quaternion multiplication is made a subfunction. This makes the code cleaner and easier to see how it relates to the algorithm. `QMult` is written to handle multiple quaternions at once, so the function is easy to vectorize. `QPose` finds the transpose of the quaternion. Both of these functions would normally be separate functions, but in this chapter they are only associated with the error computation code, so they are in the same file.

```
function deltaAngle = ErrorFromQuaternion( q, qTarget )

deltaQ       = QMult( QPose(q), qTarget );
deltaAngle   = -2.0*deltaQ(2:4);

%% Multiply two quaternions.
% Q2 transforms from A to B and Q1 transforms from B to C
% so Q3 transforms from A to C.
function Q3 = QMult( Q2,Q1 )

Q3 = [Q1(1,:).*Q2(1,:) - Q1(2,:).*Q2(2,:) - Q1(3,:).*Q2(3,:) - Q1(4,:).*Q2(4,:);...
     Q1(2,:).*Q2(1,:) + Q1(1,:).*Q2(2,:) - Q1(4,:).*Q2(3,:) + Q1(3,:).*Q2(4,:);...
     Q1(3,:).*Q2(1,:) + Q1(4,:).*Q2(2,:) + Q1(1,:).*Q2(3,:) - Q1(2,:).*Q2(4,:);...
     Q1(4,:).*Q2(1,:) - Q1(3,:).*Q2(2,:) + Q1(2,:).*Q2(3,:) + Q1(1,:).*Q2(4,:)];

%% Transpose of a quaternion
% The transpose requires changing the sign of the angle terms.
function q = QPose(q)

q(2:4,:) = -q(2:4,:);
```

The control system is implemented in the simulation loop with the following code.

```
% Find the angle error
angleError = ErrorFromQuaternion( x(1:4), qTarget );
if (controlIsOn )
   u = [0;0;0];
   for j = 1:3
      [u(j), dC(j)] = PDControl('update',angleError(j),dC(j));
   end
else
   u = [0;0;0];
end
```

12-3. Simulating the Controlled Spacecraft

Problem

You want to test the attitude controller and see how it performs.

Solution

The solution is to build a MATLAB script in which you design the PD controller matrices, and then simulate the controller in a loop, applying the calculated torques until the desired quaternion is attained or until the disturbance torque is cancelled.

How It Works

Build a simulation script for the controller, `SpacecraftSim`. The first thing to do with the script is check angular momentum conservation by running the simulation for 300 seconds at timesteps of 0.1 and 1 second, and comparing the magnitude of the angular momentum in the two test cases. The control is turned off by setting the `controlIsOn` flag to false. In the absence of external torques, if our equations are programmed correctly, the momentum should be constant. You will, however, see growth in the momentum due to error in the numerical integration. The growth should be much lower in the first case than the second case, as the smaller timestep makes the integration more exact. Note that we give the spacecraft random initial rates in both `omega` and `omegaRWA` and a nonspherical inertia, to help catch any bugs in the dynamics code.

```
tEnd                    = 300;
dT                      = 0.1;
controlIsOn             = false;
qECIToBody              = [1;0;0;0];
omega                   = [0.01;0.02;-0.03]; % rad/sec
omegaRWA                = [5;-3;2]; % rad/sec
d.inr                   = [3 0 0;0 10 0;0 0 5]; % kg-m^2
```

Figure 12-2 shows the results of the tests. Momentum growth is four orders of magnitude lower in the test with a 0.1-second timestep indicating that the dynamical equations conserve angular momentum, as they should. The shape of the growth does not change and will depend on the relative magnitudes of the various angular rates.

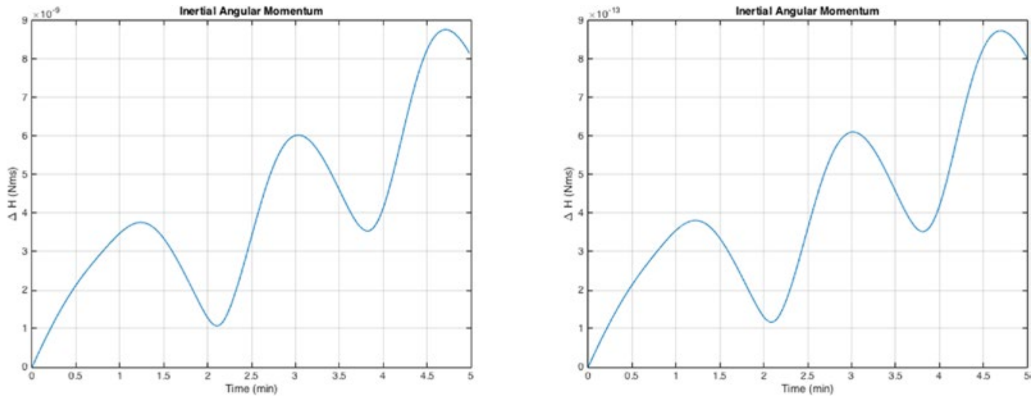

Figure 12-2. *Angular momentum conservation for 1 second and 0.1 second time steps. The growth is four orders of magnitude lower in the 0.1 second test, to le^{-13} from le^{-9}*

You initialize the script by using the data structure feature of the RHS function. This is shown next with parameters for a run with the control system on. The rates are now initialized to zero and you use the timestep of 1 second, which showed sufficiently small momentum growth in the previous test.

```
%% Data structure for the right hand side
d            = RHSSpacecraftWithRWA;
%% User initialization
% Initialize duration, delta time states and inertia
tEnd         = 600;
dT           = 1;
controlIsOn  = true;
qECIToBody   = [1;0;0;0];
omega        = [0;0;0]; % rad/sec
omegaRWA     = [0;0;0]; % rad/sec
d.inr        = [3 0 0;0 10 0;0 0 5]; % kg-m^2
qTarget      = QUnit([1;0.004;0.0;0]); % Normalize
d.torque     = [0;0;0]; % Disturbance torque
```

The control system is designed here. Note the small value of wN and the unit damping ratio. The frequency of the disturbances on a spacecraft are quite low, and the wheels have torque limits, leading to a wN much smaller than the robotics example. All three controllers are identical.

```
%% Control design
% Design a PD controller
dC            = PDControl( 'struct' );
dC(1).zeta    = 1;
dC(1).wN      = 0.02;
dC(1).wD      = 5*dC(1).wN;
dC(1).tSamp   = dT;
dC(1)         = PDControl( 'initialize', dC(1) );

% Make all 3 axis controllers identical
dC(2)         = dC(1);
dC(3)         = dC(1);
```

The simulation loop follows. As always, initialize the plotting array with zeros. The first step in the loop is finding the angular error between the current state and the target attitude. Next, the control acceleration is calculated or set to zero, depending on the value of the control flag. The control torque is calculated by multiplying the control acceleration by the spacecraft inertia. Compute the momentum for plotting purposes, and finally, integrate one timestep.

```
%% Simulation
% State vector
x = [qECIToBody;omega;omegaRWA];

% Plotting and number of steps
n  = ceil(tEnd/dT);
xP = zeros(length(x)+7,n);

% Find the initial angular momentum
[~,hECI0] = RHSSpacecraftWithRWA(0,x,d);

% Run the simulation
for k = 1:n
    % Find the angle error
    angleError = ErrorFromQuaternion( x(1:4), qTarget );
    if (controlIsOn )
        u = [0;0;0];
       for j = 1:3
          [u(j), dC(j)] = PDControl('update',angleError(j),dC(j));
       end
    else
       u = [0;0;0];
    end

  % Wheel torque is on the left hand side
  d.torqueRWA = d.inr*u;
  % Get the delta angular momentum
  [~,hECI]   = RHSSpacecraftWithRWA(0,x,d);
  dHECI      = hECI - hECI0;
  hMag       = sqrt(dHECI'*dHECI);

  % Plot storage
  xP(:,k)    = [x;d.torqueRWA;hMag;angleError];

  % Right hand side
  x          = RungeKutta(@RHSSpacecraftWithRWA,0,x,dT,d);

end
```

The output is entirely two-dimensional plots. We break them up into pages with one to three plots per page. This makes them easily readable on most computer displays.

```
%% Plotting
[t,tL] = TimeLabl((0:(n-1))*dT);

yL      = {'q_s', 'q_x', 'q_y', 'q_z'};
PlotSet( t, xP(1:4,:), 'x␣label', tL, 'y␣label', yL,...
  'plot␣title', 'Attitude', 'figure␣title', 'Attitude');

yL      = {'\omega_x', '\omega_y', '\omega_z'};
PlotSet(t, xP(5:7,:), 'x␣label', tL, 'y␣label', yL,...
  'plot␣title', 'Body␣Rates', 'figure␣title', 'Body␣Rates');

yL      = {'\omega_1', '\omega_2', '\omega_3'};
PlotSet( t, xP(8:10,:), 'x␣label', tL, 'y␣label', yL,...
  'plot␣title', 'RWA␣Rates', 'figure␣title', 'RWA␣Rates');

yL      = {'T_x␣(Nm)', 'T_y␣(Nm)', 'T_z␣(Nm)'};
PlotSet( t, xP(11:13,:), 'x␣label', tL, 'y␣label', yL,...
  'plot␣title', 'Control␣Torque', 'figure␣title', 'Control␣Torque');

yL      = {'\Delta␣H␣(Nms)'};
PlotSet( t, xP(14,:), 'x␣label', tL, 'y␣label', yL,...
  'plot␣title', 'Inertial␣Angular␣Momentum', 'figure␣title', 'Inertial␣Angular␣Momentum');

yL      = {'\theta_x␣(rad)', '\theta_y␣(rad)', '\theta_z␣(rad)'};
PlotSet( t, xP(15:17,:), 'x␣label', tL, 'y␣label', yL,...
  'plot␣title', 'Angular␣Errors', 'figure␣title', 'Angular␣Errors');
```

Note how `PlotSet` makes plotting much easier to set up and its code easier to read than using MATLAB's built-in `plot` and supporting functions. You do lose some flexibility. The *y*-axis labels use LaTeX notation. This provides limited LaTeX syntax. You can set the plotting to full LaTeX mode to get access to all LaTeX commands.

Note that we compute the angle error directly from the target and true quaternion. This represents our attitude sensor. In a real spacecraft, attitude estimation is quite complicated. Multiple sensors, such as combinations of magnetometers, GPS, and earth and sun sensors are used, and often rate-integrating gyros are employed to smooth the measurements. Star cameras or trackers are popular for three-axis sensing and require converting images in a camera to attitude estimates. You can't use gyros by themselves because they do not provide an initial orientation with respect to the inertial frame.

Run two tests. The first shows that our controllers can compensate for a body fixed disturbance torque. The second is to show that the controller can reorient the spacecraft.

The following is the initialization code for the disturbance torque test. The initial and target attitudes are the same, a unit quaternion, but there is a small disturbance torque in `d.torque`.

```
% Initialize duration, delta time states and inertia
tEnd        = 600;
dT          = 1;
controlIsOn = true;
qECIToBody  = [1;0;0;0];
omega       = [0;0;0]; % rad/sec
omegaRWA    = [0;0;0]; % rad/sec
d.inr       = [3 0 0;0 10 0;0 0 5]; % kg-m^2
qTarget     = QUnit([1;0;0.0;0]);
d.torque    = [0;0.0001;0]; % Disturbance torque (N)
```

We are running the simulations to 600 seconds to see the transients settle out. The disturbance torque is very small, which is typical for spacecraft. We make the torque single-axis to make the responses clearer. Figure 12-3 shows the complete set of output plots.

The disturbance causes a change in attitude around the *y* axis. This offset is expected with a PD controller. The control torque eventually matches the disturbance and the angular error reaches its maximum.

The *y* wheel rate grows linearly, as it has to absorb all the momentum produced by the torque. We don't limit the maximum wheel rate. In a real spacecraft, the wheel would soon saturate, reaching its maximum allowed speed. Our control system would need to have other actuators to desaturate the wheel. The inertial angular momentum also grows linearly as is expected with a constant external torque.

We now do an attitude correction around the *x* axis. The following is the initialization code.

```
% Initialize duration, delta time states and inertia
tEnd        = 600;
dT          = 1;
controlIsOn = true;
qECIToBody  = [1;0;0;0];
omega       = [0;0;0]; % rad/sec
omegaRWA    = [0;0;0]; % rad/sec
d.inr       = [3 0 0;0 10 0;0 0 5]; % kg-m^2
qTarget     = QUnit([1;0.004;0.0;0]); % Normalize
d.torque    = [0;0;0]; % Disturbance torque
```

We command a small attitude offset around the *x* axis, which is done by changing the second element in the quaternion. We unitize the quaternion to prevent numerical issues. Figure 12-4 shows the output plots.

In this case, the angular error around the *x* axis is reduced to zero. The inertial angular momentum remains "constant," although it jumps around a bit due to truncation error in the numerical integration. This is expected and it is good to keep checking the angular momentum with the control system running. If it doesn't remain nearly constant, the simulation has issues. Internal torques do not change the inertial angular momentum. This is why reaction wheels are called *momentum exchange devices*. They exchange momentum with the spacecraft body, but aren't added to the inertial angular momentum total.

The attitude rates remain small in both cases so that the Euler coupling torques are small. This justifies our earlier decision to treat the spacecraft as three double integrators. It also justifies our quaternion error to small angle approximation.

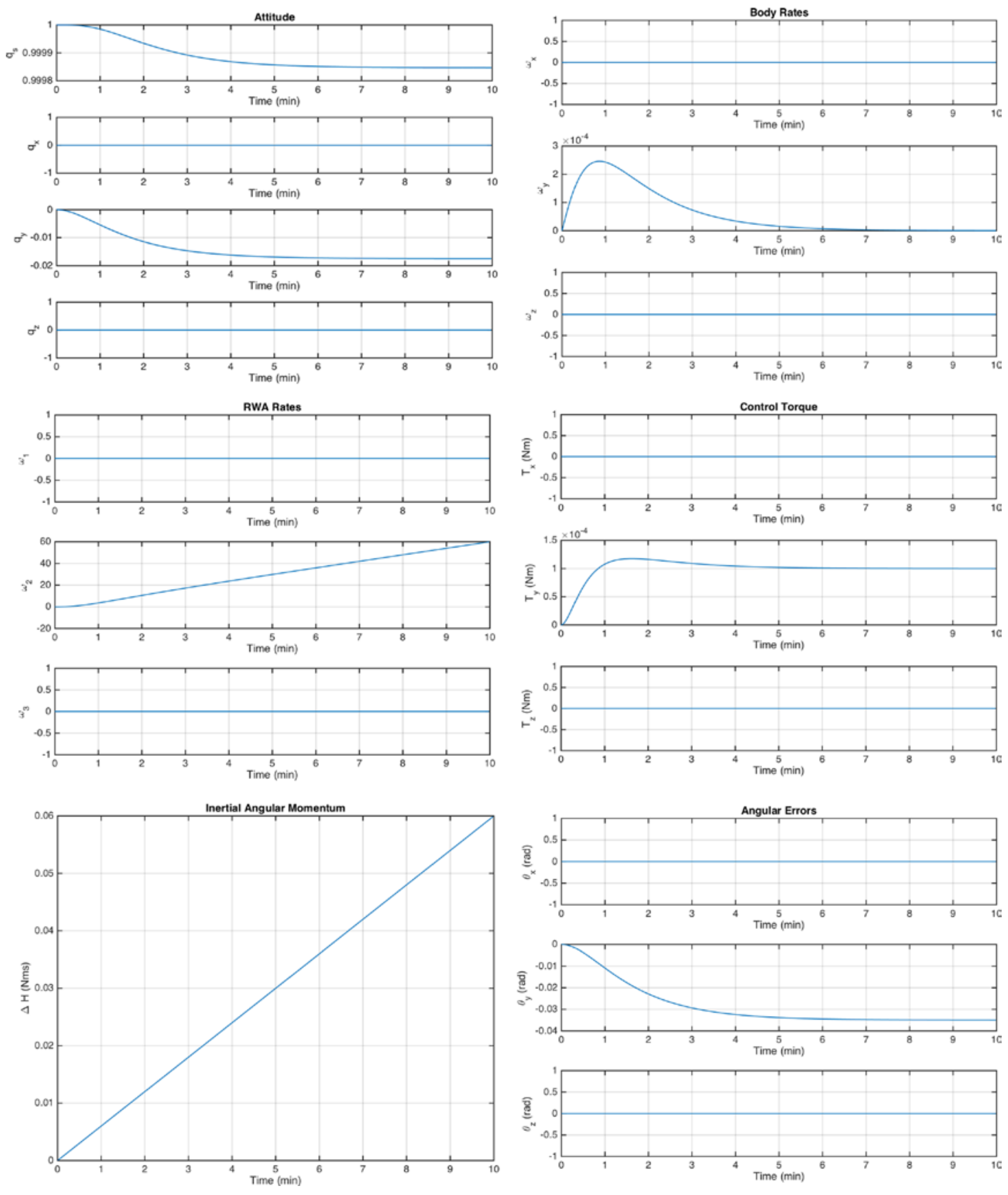

Figure 12-3. *Controlling a suddenly applied external torque*

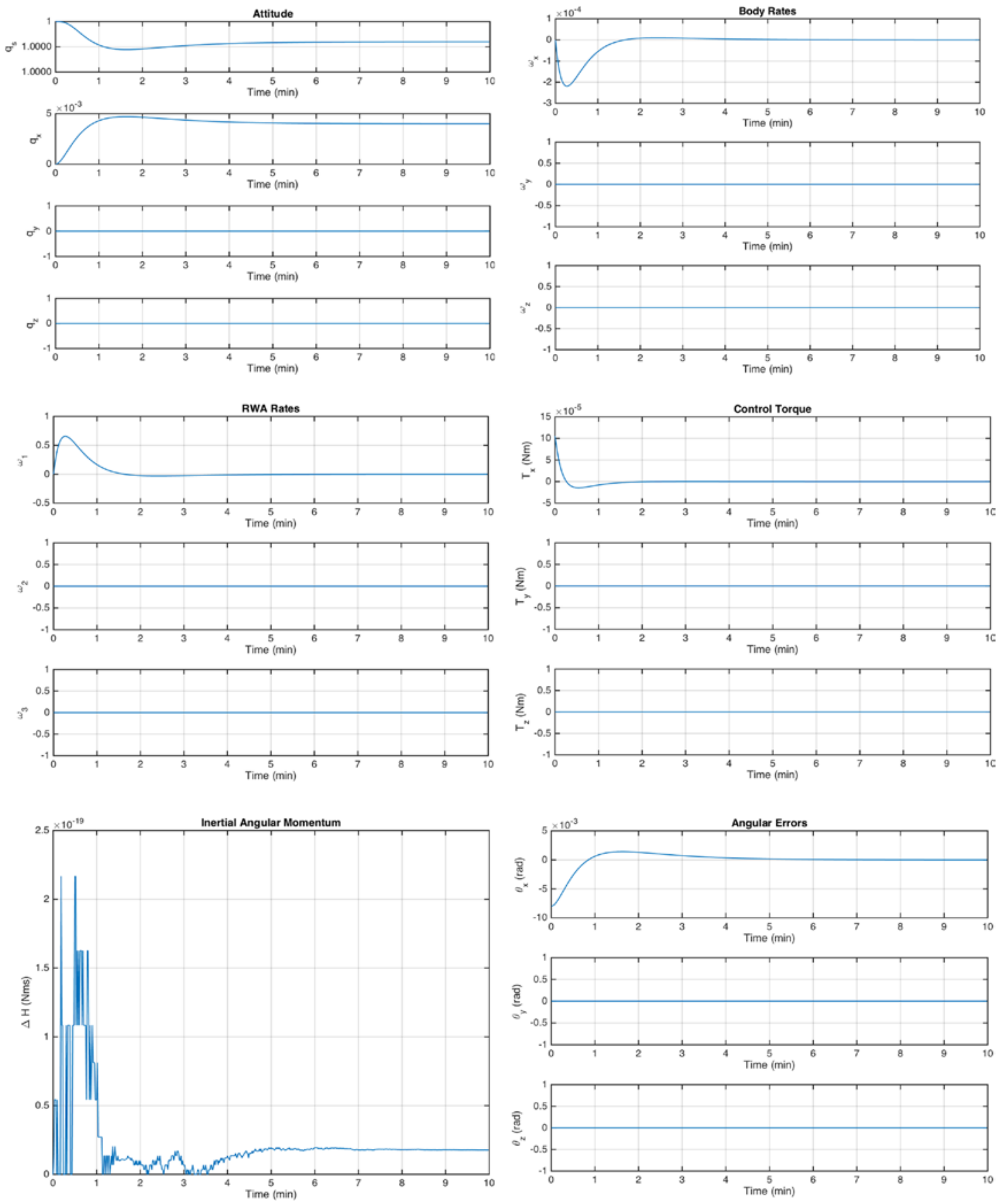

Figure 12-4. *Commanding a small attitude offset about the x-axis*

12-4. Performing Batch Runs of a Simulation

Problem

You've used the simulation script to verify momentum conservation and test the controller, but note how you have to change lines at the top by hand for each case. This is fine for development, but can make it very difficult to reproduce results; you don't know the initial conditions that generated any particular plot. You may want to run the simulation for a whole set of inputs and do Monte Carlo analysis.

Solution

Create a new function based on the script with inputs for the critical parameters. A new data structure will store both the inputs and the outputs, so you can save individual runs to mat-files. This makes it possible to replot the results of any run in the future, or redo runs from the stored inputs; for example, if you find and fix a bug in the controller.

How It Works

Start from the simulation script copied into a new file. Add a function signature. Replace the initialization variables with an input structure. Perform the simulation, and then save the input structure along with your generated output. The resulting function header is shown next. The `input` structure includes the RHS data, controller data, and simulation timing data.

```
%% SpacecraftSimFunction Spacecraft reaction wheel simulation function
% Perform a simulation of a spacecraft with reaction wheels given a
% particular initial state.
%% Form
% d = SpacecraftSimFunction( x0, qTarget, input )
%% Inputs
% x0      (7+n,1) Initial state
% qTarget (4,1) Target quaternion
% input      (.) Data structure
%                  .rhs (.) RHS data
%                  .pd (:) Controllers
%                  .dT (1,1) Timestep
%                  .tEnd (1,1) Duration
%                  .controlIsOn Flag
%% Outputs
%     d      (.) Data structure
%                  .input
%                  .x0
%                  .qTarget
%                  .xPlot
%                  .dPlot
%                  .tPlot
%                  .yLabel
%                  .dLabel
%                  .tLabel
```

Now, you can write a script that calls the simulation function in a loop. The possibilities are endless: you can test different targets, vary the initial conditions for a Monte Carlo simulation, or apply different disturbance torques. You can perform statistical analysis on your results, or identify and plot individual runs based on some criteria. In this example, you will find the maximum control torque applied in each run.

```
%% Multiple runs of spacecraft simulation
% Perform runs of SpacecraftSimBatch in a loop with varying initial
% conditions. Find the max control torque applied for each case.
%% See also
% SpacecraftSimFunction

sim = struct;
%% Control design
% Design a PD controller
dC            = PDControl( 'struct' );
dC(1).zeta    = 1;
dC(1).wN      = 0.02;
dC(1).wD      = 5*dC(1).wN;
dC(1).tSamp   = dT;
dC(1)         = PDControl( 'initialize', dC(1) );

% Make all 3 axis controllers identical
dC(2)         = dC(1);
dC(3)         = dC(1);

sim.pd = dC;

%% Spacecraft model
% Make the spacecraft nonspherical; no disturbances
rhs     = RHSSpacecraftWithRWA;
rhs.inr       = [3 0 0;0 10 0;0 0 5]; % kg-m^2
rhs.torque    = [0;0;0]; % Disturbance torque
sim.rhs = rhs;

%% Initialization
% Initialize duration, delta time states and inertia
sim.tEnd        = 600;
sim.dT          = 1;
sim.controlIsOn = true;

% Spacecraft state
qECIToBody  = [1;0;0;0];
omega       = [0;0;0]; % rad/sec
omegaRWA    = [0;0;0]; % rad/sec
x0 = [qECIToBody;omega;omegaRWA];

% Target quaternions
qTarget     = QUnit([1;0.004;0.0;0]); % Normalize
```

```
%% Simulation loop
clear d;
for k = 1:10;
  % change something in your initial conditions and simulate
  x0(5)  = 1e-3*k;
  thisD  = SpacecraftSimFunction( x0, qTarget, sim );

  % save the run results as a mat-file
  thisDir = fileparts(mfilename('fullpath'));
  fileName = fullfile(thisDir,'Output',sprintf('Run%d',k));
  save(fileName,'-struct','thisD');

  % store the run output
  d(k) = thisD;
end

%% Perform statistical analysis on results
% ... as you wish
for k = 1:length(d)
  tMax(k) = max(max(d(k).dPlot(2:4,:)));
end
figure;
plot (1:length(d),tMax);
xlabel ('Run')
ylabel ('Torque␣(Nm)')
title ('Maximum␣Control␣Torque');

% Plot a single case
kPlot = 4;
PlotSpacecraftSim( d(4) );
```

Figure 12-5 shows the maximum torque results.

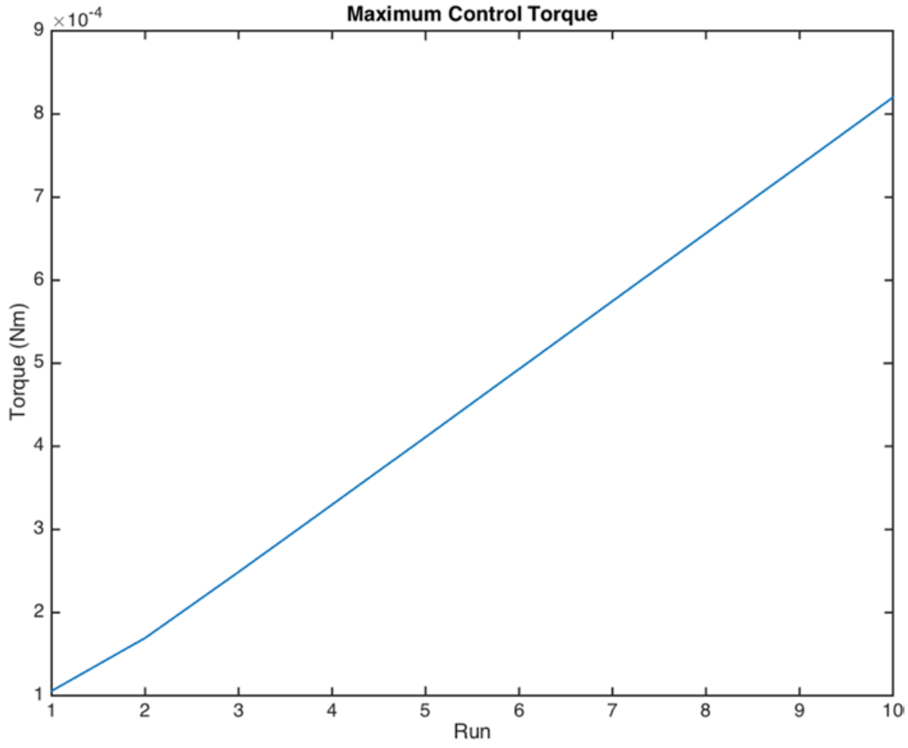

Figure 12-5. *Maximum control torque over 10 simulation runs*

An individual run's output is shown here.

```
>> d(1)
ans =

      input:   [1x1 struct]
      x0:      [10x1 double]
      qTarget: [4x1 double]
      xPlot:   [10x600 double]
      dPlot:   [7x600 double]
      tPlot:   [1x600 double]
      tLabel:  'Time (min)'
      yLabel:   {1x10 cell}
      dLabel:   {1x7 cell}
```

As another interesting example, you can give the spacecraft a higher initial rate and see how the controller responds. From the command line, change the initial rate around the *x* axis to 0.2 rad/sec and call the simulation function with no outputs, so that it generates the full suite of plots. You see that the response takes a long time, over 20 minutes, but the rate does eventually damp out.

The full simulation function is shown next. The built-in demo performs an open-loop simulation of the default spacecraft model with no control, as with the momentum conservation test performed in the previous recipe (Figure 12-2).

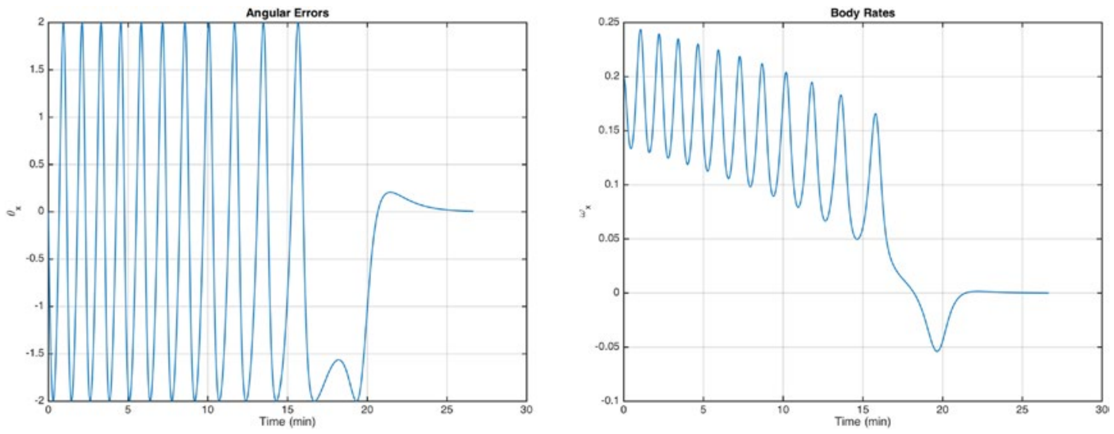

Figure 12-6. *Control response to a large rate in x. The rate does damp out, eventually!*

```
function d = SpacecraftSimFunction( x0, qTarget, input )

%% Handle inputs
if nargin == 0
   % perform an open loop simulation
   input = struct;
   input.rhs = RHSSpacecraftWithRWA;
   input.pd = [];
   input.dT = 1; % sec
   input.tEnd = 600; % sec
   input.controlIsOn = false;
   x0 = [1;0;0;0;1e-3*randn(6,1)];
   SpacecraftSimFunction( x0, [], input );
   return;
end

if isempty(x0)
    qECIToBody = [1;0;0;0];
    omega = [0;0;0]; % rad/sec
    omegaRWA = [0;0;0]; % rad/sec
    x0 = [qECIToBody;omega;omegaRWA];
end

if isempty(qTarget)
   qTarget = x0(1:4);
end

%% Simulation
% State vector
x = x0;
nWheels = length(x0)-7;
```

```
% Plotting and number of steps
n = ceil(input.tEnd/input.dT);
xP = zeros(length(x),n);
dP = zeros(7,n);

% Find the initial angular momentum
d = input.rhs;
[~,hECI0] = RHSSpacecraftWithRWA(0,x,d);

% Run the simulation
for k = 1:n
   % Control
   u = [0;0;0];
   angleError = [0;0;0];
   if( input.controlIsOn )
      % Find the angle error
      angleError = ErrorFromQuaternion( x(1:4), qTarget );
      % Update the controllers individually
      for j = 1:nWheels
        [u(j), input.pd(j)] = PDControl('update',angleError(j),input.pd(j));
      end
   end

   % Wheel torque
   d.torqueRWA = d.inr*u;

   % Get the delta angular momentum
   [~,hECI] = RHSSpacecraftWithRWA(0,x,d);
   dHECI = hECI - hECI0;
   hMag = sqrt(dHECI'*dHECI);

   % Plot storage
   xP(:,k) = x;
   dP(:,k) = [hMag;d.torqueRWA;angleError];

   % Right hand side
   x = RungeKutta(@RHSSpacecraftWithRWA,0,x,input.dT,d);
end

[t,tL] = TimeLabl((0:(n-1))*input.dT);

%% Store data
% Record initial conditions
d = struct;
d.input = input;
d.x0 = x0;
d.qTarget = qTarget;
d.xPlot = xP;
d.dPlot = dP;
d.tPlot = t;
d.tLabel = tL;
```

```
y = cell(1,nWheels);
for k = 1:nWheels
   y{k} = sprintf('\\omega_%d',k);
end
d.yLabel = {'q_s','q_x','q_y','q_z','\omega_x','\omega_y','\omega_z',y{:}};
d.dLabel = {'\Delta␣H␣(Nms)','T_x␣(Nm)', 'T_y␣(Nm)', 'T_z␣(Nm)', ...
'\theta_x␣(rad)', '\theta_y␣(rad)', '\theta_z␣(rad)'};

if nargout == 0
  PlotSpacecraftSim( d );
end
```

The plotting code is put into a separate function that accepts the output data structure. Create and save the plot labels in the simulation function. This allows you to replot any saved output. Add a statement to check for non-zero angle errors before creating the control and angle errors plots, since they are not needed for open-loop simulations.

■ **Tip** Use the fields in your structure for plotting without renaming the variables locally, so you can copy/paste individual plots to the command line after doing a run of your simulation.

```
%% PlotSpacecraftSim Plot the spacecraft simulation output
%% Form
% PlotSpacecraftSim( d )
%% Inputs
% d (.) Simulation data structure

%% Copyright
% Copyright (c) 2015 Princeton Satellite Systems, Inc.
% All rights reserved.

function PlotSpacecraftSim( d )

t = d.tPlot;

yL = d.yLabel(1:4);
PlotSet( d.tPlot, d.xPlot(1:4,:), 'x␣label', d.tLabel, 'y␣label', yL,...
  'plot␣title', 'Attitude', 'figure␣title', 'Attitude');

yL = d.yLabel(5:7);
PlotSet(d.tPlot, d.xPlot(5:7,:), 'x␣label', d.tLabel, 'y␣label', yL,...
  'plot␣title', 'Body␣Rates', 'figure␣title', 'Body␣Rates');

yL = d.yLabel(8:end);
PlotSet( t, d.xPlot(8:end,:), 'x␣label', d.tLabel, 'y␣label', yL,...
  'plot␣title', 'RWA␣Rates', 'figure␣title', 'RWA␣Rates');
```

```
yL = d.dLabel(1);
PlotSet( d.tPlot, d.dPlot(1,:), 'x␣label', d.tLabel, 'y␣label', yL,...
  'plot␣title', 'Inertial␣Angular␣Momentum',...
  'figure␣title', 'Inertial␣Angular␣Momentum');

if any(d.dPlot(5:end,:)~=0)
  yL = d.dLabel(2:4);
  PlotSet( d.tPlot, d.dPlot(2:4,:), 'x␣label', d.tLabel, 'y␣label', yL,...
    'plot␣title', 'Control␣Torque', 'figure␣title', 'Control␣Torque');
  yL = d.dLabel(5:end);
  PlotSet( d.tPlot, d.dPlot(5:end,:), 'x␣label', d.tLabel, 'y␣label', yL,...
    'plot␣title', 'Angular␣Errors', 'figure␣title', 'Angular␣Errors');
end
```

An interesting exercise for you would be to replace the fixed disturbance input, `d.torque`, with a function handle that calls a disturbance function. This forms the basis of spacecraft simulation in our Spacecraft Control Toolbox, where the disturbances are calculated from the spacecraft geometry and space environment as it rotates and moves along its orbit.

Summary

This chapter demonstrated how to write the dynamics and implement a simple control law for a spacecraft with reaction wheels. Our control system is only valid for small angle changes and will not work well if the angular rates on the spacecraft get large. In addition, we do not consider the torque or momentum limits on the reaction wheels. You also learned about quaternions and how to implement kinematics of rigid body with quaternions. We showed you how to get angle errors from two quaternions. Table 12-1 lists the code developed in this chapter.

Table 12-1. *Chapter Code Listing*

File	Description
`RHSSpacecraftWithRWA`	RHS for spacecraft with reaction wheels.
`ErrorFromQuaternion`	Spacecraft simulation script.
`SpacecraftSim`	Spacecraft simulation script.
`SpacecraftSimBatch`	Spacecraft simulation function.
`BatchSimRuns`	Multiple runs of the spacecraft simulation.
`PlotSpacecraftSim`	Plot the simulation results.
`QTForm`	Transform a vector opposite the direction of the quaternion.
`QUnit`	Normalize a quaternion.