CHAPTER 13

Spacecraft Attitude Control

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 in this chapter, we will be concerned only with three axis pointing. We will use reaction wheels 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 because they exchange momentum with the rest of the spacecraft. You can reorient the spacecraft using reaction wheels without any external torques. Before reaction wheels were introduced, thrusters were often used for orientation control. This would consume the 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. One rotates about the x axis, one rotates about the y axis, and the third about the z axis. The shaft of the motor is attached to the rotor of the wheel which is attached to the spacecraft. The torque applied between the wheel and spacecraft causes 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.

13.1 Creating a Dynamical 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 13.1.

M. Paluszek and S. Thomas, *MATLAB Recipes*,
https://doi.org/10.1007/978-1-4842-6124-8_13

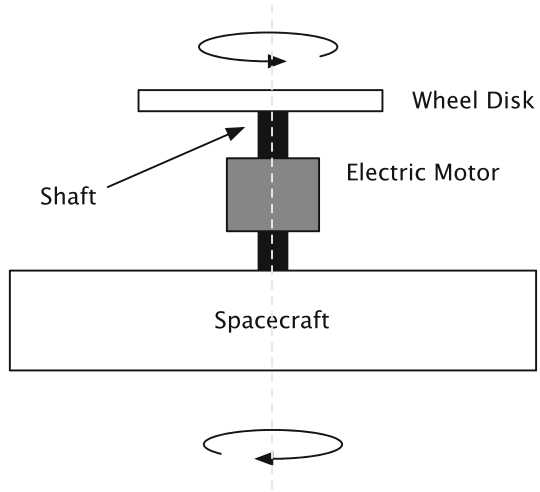

Figure 13.1: *A reaction wheel. The reaction wheel platter spins in one direction, and the spacecraft spins in the opposite direction.*

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 we assume that the wheels are perfectly symmetric, are aligned with the three body axes, and have a diagonal inertia matrix, we can model the spacecraft dynamics with the following coupled first-order differential equations:

$$I\dot{\omega} + \omega^{\times}\left(I\omega + I_w(\omega_w + \omega)\right) + I_w(\dot{\omega}_w + \dot{\omega}) = T \tag{13.1}$$

$$I_w\left(\dot{\omega}_w + \dot{\omega}\right) = T_w \tag{13.2}$$

I is the 3-by-3 inertia matrix of the spacecraft and does not include the inertia of the wheels.

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \tag{13.3}$$

The inertia 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.

$$\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{13.4}$$

ω_w is the angular rate of the reaction wheels:

$$\omega_w = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \tag{13.5}$$

for wheels 1, 2, and 3. 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 polar inertia of the wheels (we assume that they all have the same polar inertia). We can substitute the second equation into the first to simplify the equations.

$$I\dot{\omega} + \omega^{\times}\left(I\omega + I_w(\omega_w + \omega)\right) + T_w = T \tag{13.6}$$

$$I_w\left(\dot{\omega}_w + \dot{\omega}\right) = T_w \tag{13.7}$$

This term

$$T_e = \omega^{\times}\left(I\omega + I_w(\omega_w + \omega)\right) \equiv \omega \times h \tag{13.8}$$

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

$$I\dot{\omega} + T_w = T \tag{13.9}$$

$$I_w\left(\dot{\omega}_w + \dot{\omega}\right) = T_w \tag{13.10}$$

For kinematics, we will 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 (dynamical quantities) to specify the orientation. The problem with Euler angles is that they introduce singularities, that is, certain orientations where an angle is undefined, and therefore they are not suitable for simulations. The derivative of the quaternion from the inertial frame to the body frame is

$$\dot{q} = \frac{1}{2}\begin{bmatrix} 0 & \omega^T \\ -\omega & \omega^{\times} \end{bmatrix} \tag{13.11}$$

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

$$\omega^{\times} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{13.12}$$

The skew matrix always has zeros on the diagonal, and the matrix is equal to the negative of its transpose. The wheel torque is a combination of friction torque and control torque. Reaction wheels are usually driven by brushless direct current (DC) motors that have the back electromotive force canceled by current feedback within the motor electronics. The total reaction wheel torque is therefore

$$T_w = T_c + T_f \tag{13.13}$$

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

$$T_f = -k_d \omega_k \tag{13.14}$$

k_d is the damping coefficient. If k_d is large, we can compensate for it proactively by feeding the expected friction torque forward into the controller. This requires careful calibration of the wheel to determine the damping coefficient.

First, we will 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`. We 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 up of assemblies.

The default unit vectors for the wheel are along orthogonal axes, that is, 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 nonspinning parts of the wheels are lumped in with the inertia matrix.

RHSSpacecraftWithRWA.m

```
1  %% RHSSPACECRAFTWITHRWA Compute the dynamics for a spacecraft with
       reaction wheels.
33 function [xDot, hECI] = RHSSpacecraftWithRWA( ~, x, d )
34
35 % Default data structure
36 if( nargin == 0 )
37   xDot = struct('inr',eye(3), 'torque',[0;0;0],'inrRWA',
         0.001*[1;1;1],...
38                 'torqueRWA',[0;0;0],'uRWA',eye(3), 'damping',[0;0;0]);
39   if( nargout == 0 )
40     disp('RHSSpacecraftWithRWA struct:')
41   end
42   return
43 end
```

The dynamical equations for the spacecraft are given in the following lines of code. We 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 of this inverse. 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.

We 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. We also input unit vectors for each

wheel. The unit vector is the axis or rotation for each. As a consequence, the wheels do not have to be aligned with x, y, and z, that is, do not have to be orthogonal.

```
% 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;
```

Note that `uRWA` is an array of the reaction wheel unit vectors, that is, the spin vectors. In computing `h`, we have to transform ω into the wheel frame using the transpose of `uRWA` and then transform back before adding the wheel component to the core component, $I\omega$. The wheel dynamics are given next, note the use of the backslash operator to solve the set of linear equations for $\dot{\omega}$, `omegaDotCore`:

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

The total state derivative is in these lines:

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

% 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 anger rates for all the states and then rerun it with a smaller time step. The change in angular momentum should decrease as the time step is decreased.

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

13.2 Computing Angle Errors from Quaternions

Problem

We 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

We will make three proportional-derivative (PD) controllers, one for each axis. We need a function to take two quaternions and compute the small angles between them as input to these controllers.

How It Works

If we are pointing at an inertial target and wish to control about that orientation, we 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.

$$\dot{\theta} = \omega \tag{13.15}$$

We will also multiply both sides of the Euler equation, Equation 13.9, 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 the acceleration. Our system equations now become

$$\ddot{\theta} + a_w = a \tag{13.16}$$

$$I_w\left(\dot{\omega}_w + \dot{\omega}\right) = -T_w \tag{13.17}$$

The first equation is now three decoupled second-order equations, just as in our Chapter 7. We can stabilize this system with our standard PD controller.

We need attitude angles as input to the PD controllers to compute our control torques. Our examples will only be for small angular displacements from the nominal attitude. We will pass the control code a target quaternion, and it will compute Δ angles, or we will 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

$$q_\Delta \approx \begin{bmatrix} 1 \\ -\theta_1/2 \\ -\theta_2/2 \\ -\theta_3/2 \end{bmatrix} \tag{13.18}$$

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

$$q_\Delta = q^T q_T \tag{13.19}$$

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 we put them in the same file.

ErrorFromQuaternion.m

```
%% ERRORFROMQUATERNION Compute small angle error between two
    quaternions.
function deltaAngle = ErrorFromQuaternion( q, qTarget )

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

%% ErrorFromQuaternion>QMult Multiply two quaternions
% Q2 transforms from A to B and Q1 transforms from B to C
% so Q3 transforms from A to C.
%
%   Q3 = QMult( Q2 ,Q1 )
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,:)];

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

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

The control system is implemented in the simulation loop (in the next recipe) with the following code:

SpacecraftSim.m

```
  % 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
```

```
65      u    = [0;0;0];
66    end
67
68    % Wheel torque is on the left hand side
69    d.torqueRWA = d.inr*u;
```

13.3 Simulating the Controlled Spacecraft

Problem

We want to test our attitude controller and see how it performs.

Solution

The solution is to build a MATLAB script in which we 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 canceled.

How It Works

We build a simulation script for the controller, `SpacecraftSim`. The first thing we do with the script is to set parameters at the top of the file to check the angular momentum conservation by running the simulation for 300 seconds at time steps 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. We will however see the 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 time step makes the integration more exact. Remember that for fourth-order Runge-Kutta, the error goes as the fourth power of the time step. 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.

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

Figure 13.2 shows the results of the tests using the above initialization code. The momentum growth is four orders of magnitude lower in the test with a 0.1 second time step 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.

We initialize the script by using our data structure feature of the RHS function. This is shown in the following with parameters for a run with the control system on. The rates are

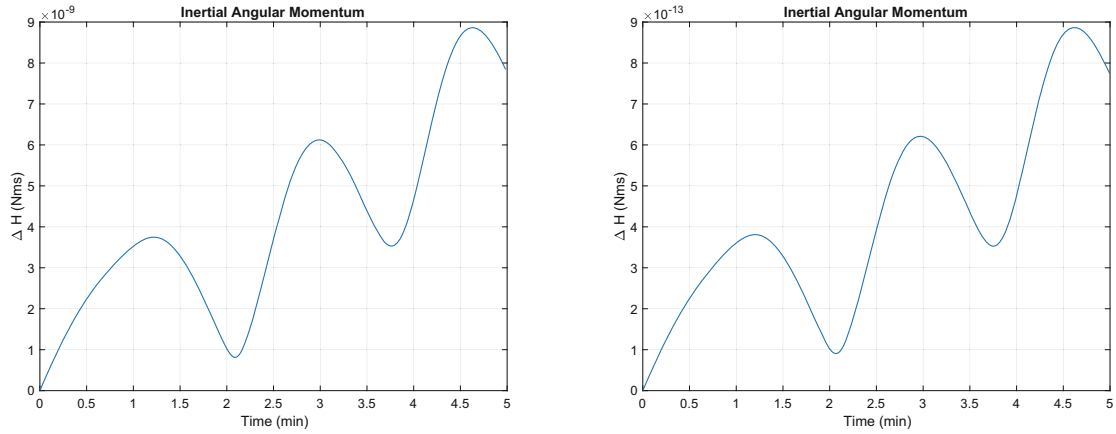

Figure 13.2: *Angular momentum conservation for 1 second and 0.1 second time steps. The growth is four orders of magnitude lower in the 0.1 second test, to $1e^{-13}$ from $1e^{-9}$.*

now initialized to zero, and we use the time step of 1 second, which showed sufficiently small momentum growth in our previous test.

```
%% Spacecraft reaction wheel simulation script
% An attitude control simulation using reaction wheels.
%% Data structure for the right hand side
d       = RHSSpacecraftWithRWA;
```

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 is quite low, and the wheels have torque limits, leading to a `wN` much smaller than the robotics recipe. All three controllers are identical.

```
%% Control design
% Design a PD controller. The same controller is used for all 3 axes.
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, we initialize the plotting array with zeros. By allocating memory for the array, we speed up the code as memory allocation is usually slow. 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. We compute the momentum for plotting purposes and, finally, integrate one time step.

```
%% Simulation
% Initialize the plotting arrays and perform a fixed timestep loop
    using
% Runge-Kutta integration.

% 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
```

Our 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
% Generate plots of the attitude, body and wheel rates, control torque, angular
% momentum, and anglular error. If there is no external disturbance torque than
% angular momentum should be conserved.
[t,tL] = TimeLabel((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,...
```

Note how `PlotSet` makes plotting much easier to set up and read code than if we use MATLAB's built-in `plot` and supporting functions. You do lose some flexibility. The y axis labels use LaTeX notation. LaTeX is a technical publications package. This provides limited LaTeX syntax, such as Greek letters, subscripts, and superscripts. You can set the plotting to full LaTeX mode to get access to all LaTeX commands and formatting.

Note that we compute the angle error directly from the target and true quaternion. This represents our attitude sensor. In 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.

We will 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.

```
% 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 13.3 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 PD control method will have steady-state error for constant (or more generally, nonzero-mean) disturbances. The integral control would be required to compensate for such disturbances.

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 13.4 shows the output plots.

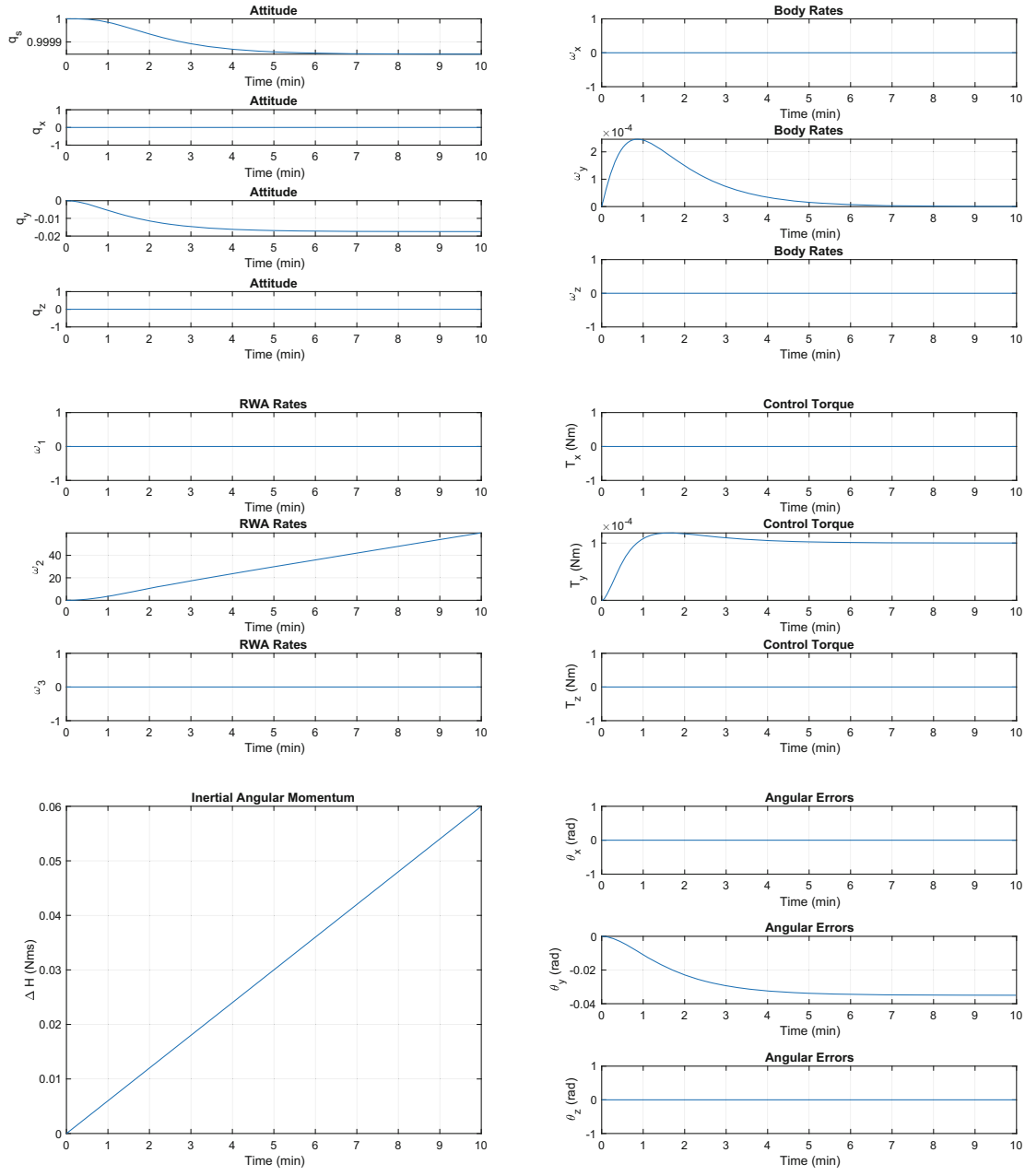

Figure 13.3: *Controlling a suddenly applied external torque.*

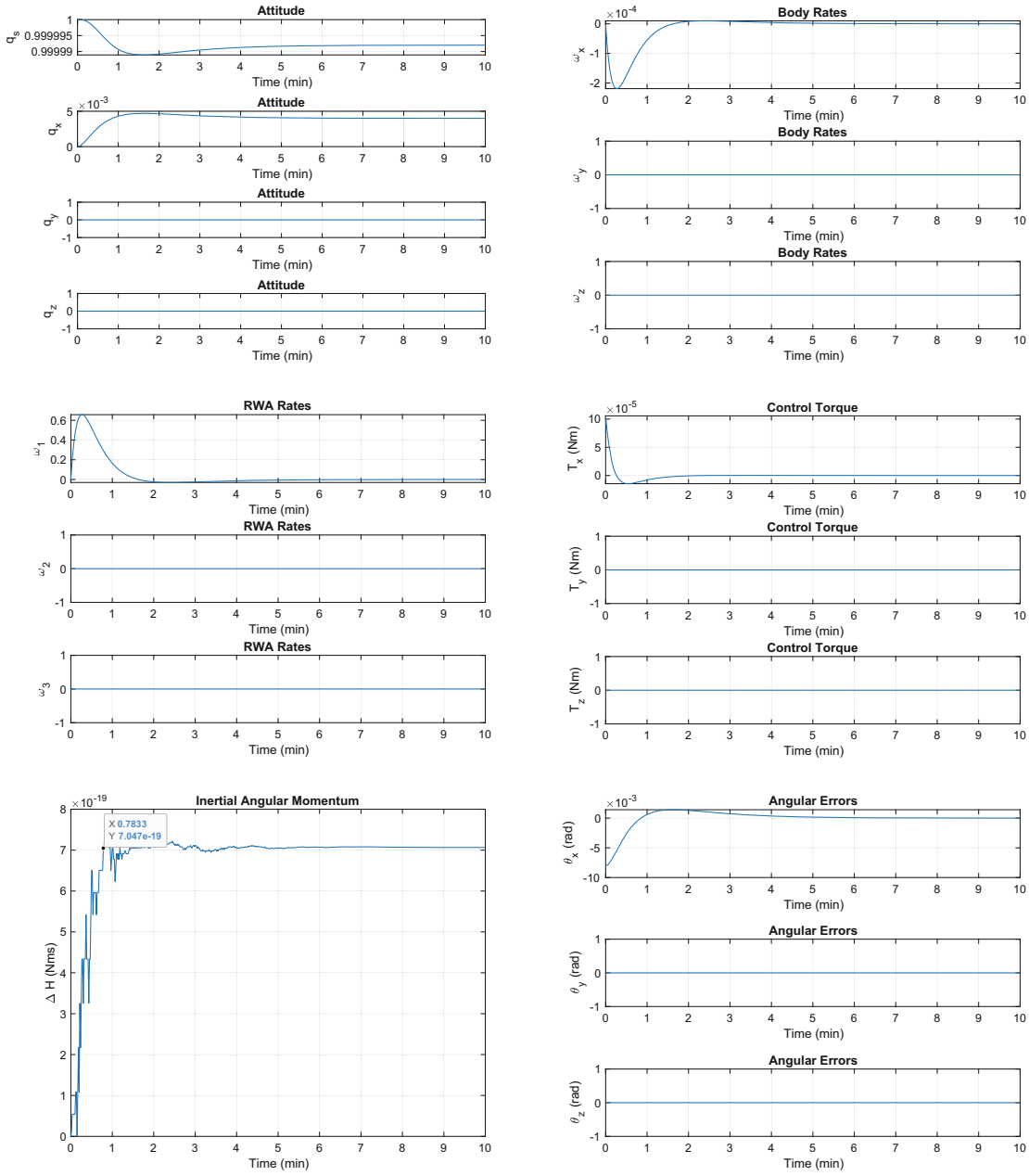

Figure 13.4: *Response to a small change in attitude.*

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, it means that the simulation probably has an error in the dynamical equations. 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 don't change the total inertial angular momentum.

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.

13.4 Performing Batch Runs of a Simulation

Problem

We've used our simulation script to verify momentum conservation and test our controller, but note how we 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; we don't know the initial conditions that generated any particular plot. We may want to run our simulation for a whole set of inputs and do a Monte Carlo analysis.

Solution

We'll create a new function based on our script with inputs for our critical parameters. A new data structure will store both our inputs and the outputs so we can save individual runs to mat-files. This will make 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, then save the input structure along with your generated output. The resulting function header is shown in the following listing. The `input` structure includes our RHS data, controller data, and simulation timing data.

SpacecraftSimFunction.m

```
1  %% SPACECRAFTSIMFUNCTION Spacecraft reaction wheel simulation function
2  % Perform a simulation of a spacecraft with reaction wheels given a
3  % particular initial state. If there are no inputs it will perform a
       demo for an
4  % open loop system. If there are no outputs it will create plots via
5  % PlotSpacecraftSim.
6  %% Form
7  %  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    (.)   Input structure
%                  .x0     (7+n,1) Initial state
%                  .qTarget (4,1) Target quaternion
%                  .xPlot (7+n,:) State data
%                  .dPlot (4+n,:) Torque and angle error data
%                  .tPlot   (1,:)  Time data
%                  .yLabel  {}     State labels
%                  .dLabel  {}     Data labels
%                  .tLabel  ''     Time label string
%% See also
% RHSSpacecraftWithRWA, ErrorFromQuaternion, PDControl, RungeKutta,
    TimeLabel,
% PlotSpacecraftSim
```

Now, we can write a script that calls our simulation function in a loop. The possibilities are endless – you can test different targets, vary the initial conditions for a Monte Carlo simulation, and 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, we will find the maximum control torque applied in each run.

BatchSimRuns.m

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

%% Initialization
sim = struct;
% 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];
```

```
23
24 % Target quaternions
25 qTarget = QUnit([1;0.004;0.0;0]); % Normalize
26
27 %% Control design
28 % Design a PD controller
29 dC             = PDControl( 'struct' );
30 dC(1).zeta     = 1;
31 dC(1).wN       = 0.02;
32 dC(1).wD       = 5*dC(1).wN;
33 dC(1).tSamp    = sim.dT;
34 dC(1)          = PDControl( 'initialize', dC(1) );
35
36 % Make all 3 axis controllers identical
37 dC(2)          = dC(1);
38 dC(3)          = dC(1);
39
40 sim.pd = dC;
41
42 %% Spacecraft model
43 % Make the spacecraft nonspherical; no disturbances
44 rhs         = RHSSpacecraftWithRWA;
45 rhs.inr     = [3 0 0;0 10 0;0 0 5];  % kg-m^2
46 rhs.torque = [0;0;0]; % Disturbance torque
47 sim.rhs     = rhs;
49
50 %% Simulation loop
51 clear d;
52 for k = 1:10;
53   % change something in your initial conditions and simulate
54   x0(5) = 1e-3*k;
55   thisD = SpacecraftSimFunction( x0, qTarget, sim );
56
57   % save the run results as a mat-file
58   thisDir = fileparts(mfilename('fullpath'));
59   fileName = fullfile(thisDir,'Output',sprintf('Run%d',k));
60   save(fileName,'-struct','thisD');
61
62   % store the run output
63   d(k) = thisD;
64 end
65
66 %% Perform statistical analysis on results
67 % ... as you wish
68 for k = 1:length(d)
69   tMax(k) = max(max(d(k).dPlot(2:4,:)));
70 end
71 figure;
72 plot(1:length(d),tMax);
73 xlabel('Run')
74 ylabel('Torque (Nm)')
75 title('Maximum Control Torque');
```

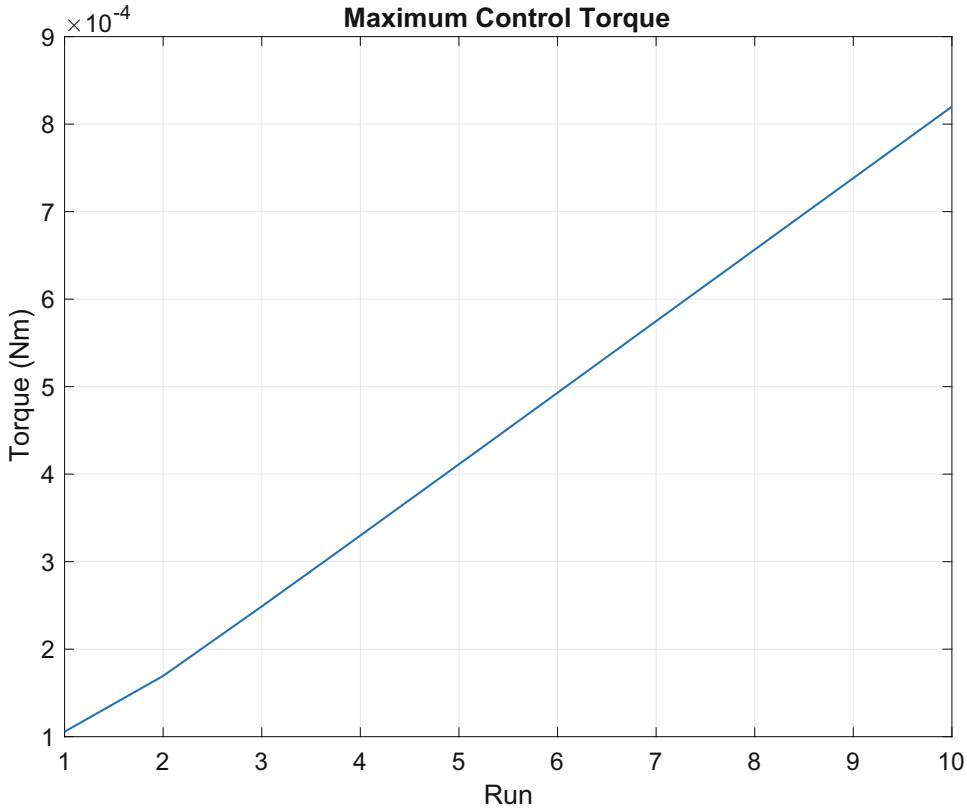

Figure 13.5: *Maximum control torque over ten simulation runs.*

```
76
77  % Plot a single case
78  kPlot = 4;
79  PlotSpacecraftSim( d(4) );
```

Figure 13.5 shows the maximum torque results. Each run has a larger initial angular velocity. We expect to see this trend, because the torque control is proportional to the angular rate.

An individual run's output is shown as follows:

```
>> d(1)

ans =

      input: [1x1 struct]
         x0: [10x1 double]
    qTarget: [4x1 double]
      xPlot: [10x600 double]
      dPlot: [7x600 double]
      tPlot: [1x600 double]
```

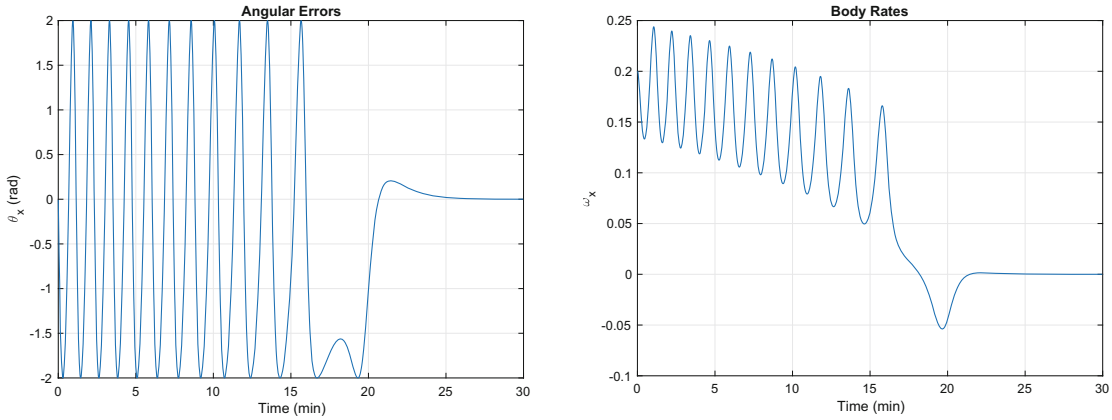

Figure 13.6: *Control response to a large rate in x. The rate does damp out, eventually!*

```
    tLabel: 'Time (min)'
    yLabel: {1x10 cell}
    dLabel: {1x7 cell}
```

As another interesting example, we can give the spacecraft a higher initial rate and see how the controller responds. From the command line, we change the initial rate around the x axis to be 0.2 rad/sec and call the simulation function with no outputs, so that it will generate the full suite of plots. We see that the response takes a long time, over 20 minutes, but the rate does eventually damp out. Figure 13.6 shows the damping response.

The full simulation function is shown in the following. 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 13.2).

```
36  function d = SpacecraftSimFunction( x0, qTarget, input )
37
38  % Handle inputs
39  if nargin == 0
40    % perform an open loop simulation
41    disp('SpacecraftSimFunction: Open loop simulation for 10 minutes.
          Initial rates are random.')
42    input = struct;
43    input.rhs   = RHSSpacecraftWithRWA;
44    input.pd    = [];
45    input.dT    = 1; % sec
46    input.tEnd = 600; % sec
47    input.controlIsOn = false;
48    x0 = [1;0;0;0;1e-3*randn(6,1)];
49    SpacecraftSimFunction( x0, [], input );
50    return;
51  end
52
```

```
53  if isempty(x0)
54    qECIToBody  = [1;0;0;0];
55    omega       = [0;0;0]; % rad/sec
56    omegaRWA    = [0;0;0];  % rad/sec
57    x0 = [qECIToBody;omega;omegaRWA];
58  end
59
60  if isempty(qTarget)
61    qTarget = x0(1:4);
62  end
63
64  % State vector
65  x = x0;
66  nWheels = length(x0)-7;
67
68  % Plotting and number of steps
69  n   = ceil(input.tEnd/input.dT);
70  xP  = zeros(length(x),n);
71  dP  = zeros(7,n);
72
73  % Find the initial angular momentum
74  d = input.rhs;
75  [~,hECI0] = RHSSpacecraftWithRWA(0,x,d);
76
77  % Run the simulation
78  for k = 1:n
79    % Control
80    u = [0;0;0];
81    angleError = [0;0;0];
82    if( input.controlIsOn )
83      % Find the angle error
84      angleError = ErrorFromQuaternion( x(1:4), qTarget );
85      % Update the controllers individually
86      for j = 1:nWheels
87        [u(j), input.pd(j)] = PDControl('update',angleError(j),input.pd(j
              ));
88      end
89    end
90
91    % Wheel torque
92    d.torqueRWA = d.inr*u;
93
94    % Get the delta angular momentum
95    [~,hECI]   = RHSSpacecraftWithRWA(0,x,d);
96    dHECI      = hECI - hECI0;
97    hMag       = sqrt(dHECI'*dHECI);
98
99    % Plot storage
100   xP(:,k)        = x;
101   dP(:,k)        = [hMag;d.torqueRWA;angleError];
102
103   % Right hand side
```

```
104   x       = RungeKutta(@RHSSpacecraftWithRWA,0,x,input.dT,d);
105 end
106
107 [t,tL] = TimeLabel((0:(n-1))*input.dT);
108
109 % Record initial conditions and results
110 d = struct;
111 d.input   = input;
112 d.x0      = x0;
113 d.qTarget = qTarget;
114 d.xPlot = xP;
115 d.dPlot = dP;
116 d.tPlot = t;
117 d.tLabel = tL;
118
119 y = cell(1,nWheels);
120 for k = 1:nWheels
121   y{k} = sprintf('\\omega_%d',k);
122 end
123 d.yLabel = [{'q_s','q_x','q_y','q_z','\omega_x','\omega_y','\omega_z'}
      y];
124 d.dLabel = {'\Delta H (Nms)','T_x (Nm)', 'T_y (Nm)', 'T_z (Nm)', ...
125   '\theta_x (rad)', '\theta_y (rad)', '\theta_z (rad)'};
126
127 if nargout == 0
128   PlotSpacecraftSim( d );
129 end
```

The plotting code is put into a separate function that accepts the output data structure. We create and save the plot labels in the simulation function. This allows us to replot any saved output. We add a statement to check for nonzero angle errors before creating the control and angle error 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.m

```
1  %% PLOTSPACECRAFTSIM Plot the spacecraft simulation output
2  %% Form
3  %  PlotSpacecraftSim( d )
4  %% Inputs
5  %  d  (.)   Simulation data structure
6  %% Outputs
7  % None.
12
```

```
13 function PlotSpacecraftSim( d )
14
15 t  = d.tPlot;
16
17 yL = d.yLabel(1:4);
18 PlotSet( d.tPlot, d.xPlot(1:4,:), 'x label', d.tLabel, 'y label', yL
       ,...
19   'plot title', 'Attitude', 'figure title', 'Attitude');
20
21 yL = d.yLabel(5:7);
22 PlotSet(d.tPlot, d.xPlot(5:7,:), 'x label', d.tLabel, 'y label', yL,...
23   'plot title', 'Body Rates', 'figure title', 'Body Rates');
24
25 yL = d.yLabel(8:end);
26 PlotSet( t, d.xPlot(8:end,:), 'x label', d.tLabel, 'y label', yL,...
27   'plot title', 'RWA Rates', 'figure title', 'RWA Rates');
28
29 yL = d.dLabel(1);
30 PlotSet( d.tPlot, d.dPlot(1,:), 'x label', d.tLabel, 'y label', yL,...
31   'plot title', 'Inertial Angular Momentum',...
32   'figure title', 'Inertial Angular Momentum');
33
34 if any(d.dPlot(5:end,:)~=0)
35   yL = d.dLabel(2:4);
36   PlotSet( d.tPlot, d.dPlot(2:4,:), 'x label', d.tLabel, 'y label', yL
         ,...
37     'plot title', 'Control Torque', 'figure title', 'Control Torque');
38   yL = d.dLabel(5:end);
39   PlotSet( d.tPlot, d.dPlot(5:end,:), 'x label', d.tLabel, 'y label',
         yL,...
40     'plot title', 'Angular Errors', 'figure title', 'Angular Errors');
41 end
```

An interesting exercise for the reader 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 has 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. We also learned about quaternions and how to implement kinematics of rigid body with quaternions. We showed how to get angle errors from two quaternions. Table 13.1 lists the code developed in the chapter.

Table 13.1: *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