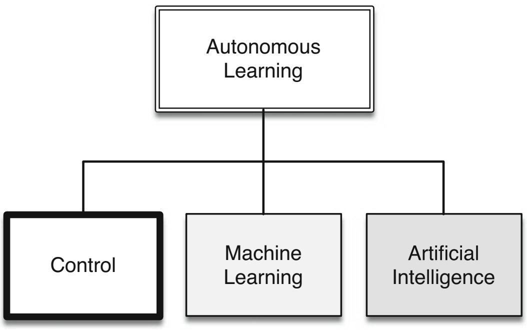
CHAPTER 5

Adaptive Control



Control systems need to react to the environment in a predictable and repeatable fashion. Control systems take measurements and use them to control the process. For example, a ship measures its heading and changes its rudder angle to attain a desired heading.

Typically, control systems are designed and implemented with all of the parameters hard-coded into the software. This works very well in most circumstances, particularly when the system is well known during the design process. When the system is not well defined or is expected to change significantly during operation, it may be necessary to implement learning control. For example, the batteries in an electric car degrade over time. This leads to less range. An autonomous driving system would need to learn that range was decreasing. This would be done by comparing the distance traveled with the battery's state of charge. More drastic, and sudden, changes can alter a system. For example, in an aircraft, the air data system might fail due to a sensor malfunction. If GPS were still operating, the plane would want to switch to a GPS-only system. In a multi-input-multi-output control system, a branch may fail, due to a failed actuator or sensor. The system might have to be modified to operate branches in that case.

Learning and adaptive control are often used interchangeably. In this chapter, you will learn a variety of techniques for adaptive control for different systems. Each technique is applied to a different system, but all are generally applicable to any control system.

Figure 5.1 provides a taxonomy of adaptive and learning control. The paths depend on the nature of the dynamical system. The rightmost branch is tuning. This is something a designer would do during testing, but it could also be done automatically as will be described in the self-tuning Recipe 5.1. The next path is for systems that will vary with time. Our first example of a system with time-varying parameters applies Model Reference Adaptive Control (MRAC) for a spinning wheel. This is discussed in Section 5.2.

M. Paluszek, S. Thomas, *MATLAB Machine Learning Recipes*,
https://doi.org/10.1007/978-1-4842-9846-6_5

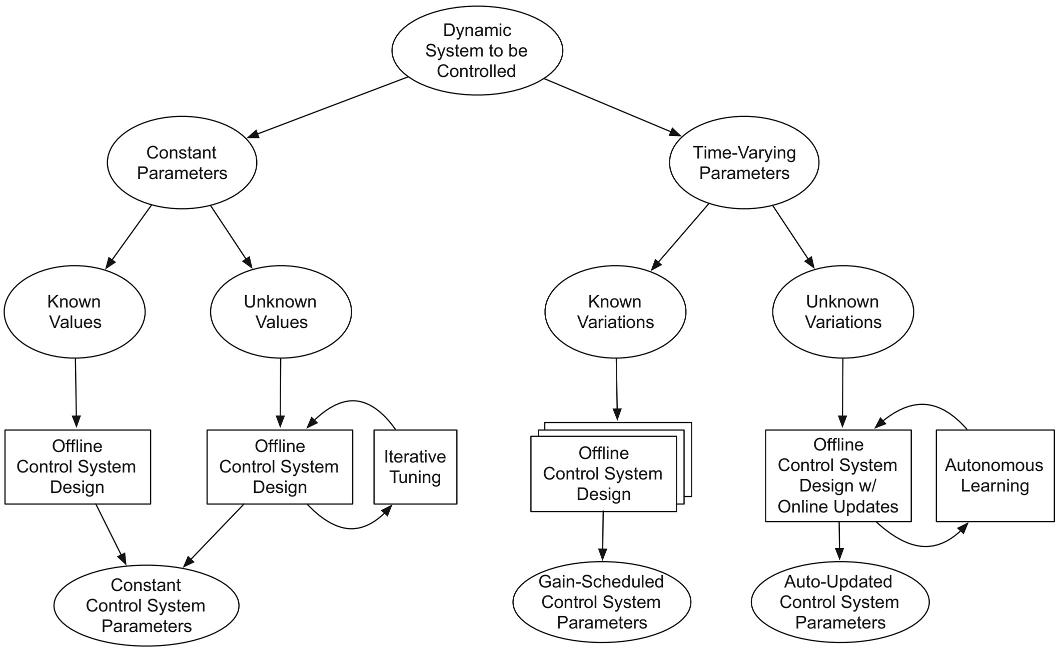

Dynamic System to be Controlled
Constant Parameters
Time-Varying Parameters
Known Values
Unknown Values
Known Variations
Unknown Variations
Offline Control System Design
Offline Control System Design
Iterative Tuning
Offline Control System Design
Offline Control System Design w/ Online Updates
Autonomous Learning
Constant Control System Parameters
Gain-Scheduled Control System Parameters
Auto-Updated Control System Parameters

Figure 5.1: *Taxonomy of adaptive or learning control*

The next example is ship control. Your goal is to control the heading angle. The dynamics of the ship are a function of the forward speed. While it isn't learning from experience, it is adapting based on information about its environment.

The last example is a spacecraft with variable inertia. This shows very simple parameter estimation.

5.1 Self-Tuning: Tuning an Oscillator

We want to tune a damper so that we critically damp a spring system for which the spring constantly changes. Our system will work by perturbing the undamped spring with a step and measuring the frequency using a Fast Fourier Transform. We then compute the damping using the frequency and add a damper to the simulation. We then measure the undamped natural frequency again to see that it is the correct value. Finally, we set the damping ratio to 1 and observe the response. The frequency is measured during operation, so this is an example of online learning. The system is shown in Figure 5.2.

In Chapter 4, we introduced parameter identification in the context of Kalman Filters, which is another way of finding the frequency. The approach here is to collect a large sample of data and process it in batch to find the natural frequency. The equations for the system are

$$\dot{r} = v \tag{5.1}$$

$$m\dot{v} = -cv - kr \tag{5.2}$$

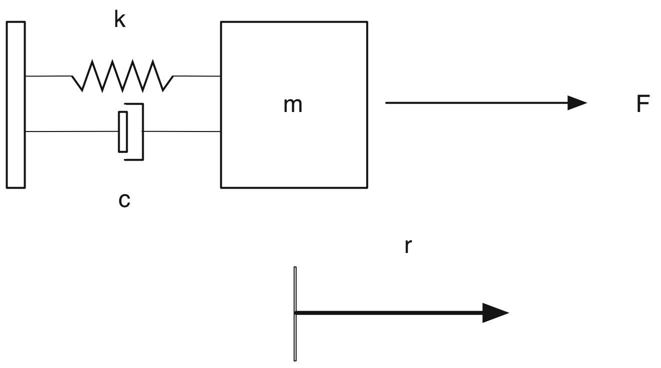


Figure 5.2: *Spring-mass-damper system. The mass is on the right. The spring is on the top to the left of the mass. The damper is below. F is the external force, m is the mass, k is the stiffness, and c is the damping*

c is the damping and k is the stiffness. The damping term causes the velocity to go to zero. The stiffness term bounds the range of motion (unless the damping is negative). The dot above the symbols means the first derivative with respect to time. That is

$$\dot{r} = \frac{dr}{dt} \tag{5.3}$$

The equations state that the change in position with respect to time is the velocity, and the mass times the change in velocity with respect to time is equal to a force proportional to its velocity and position. The second equation is Newton's law:

$$F = ma \tag{5.4}$$

where F is force, m is mass, and a is acceleration.

■ **TIP** Weight is the mass times the acceleration of gravity.

$$F = -cv - kr \tag{5.5}$$

$$a = \frac{dv}{dt} \tag{5.6}$$

5.1.1 Problem

We want to identify the frequency of an oscillator and tune a control system to that frequency.

5.1.2 Solution

The solution is to have the control system measure the frequency of the spring. We will use an FFT to identify the frequency of the oscillation.

5.1.3 How It Works

The following script shows how an FFT identifies the oscillation frequency for a damped oscillator.

The function is shown in the following code. We use the `RHSOscillator` dynamical model for the system. We start with a small initial position to get it to oscillate. We also have a small damping ratio so it will damp out. The resolution of the spectrum is dependent on the number of samples:

$$r = \frac{2\pi}{nT} \tag{5.7}$$

where n is the number of samples and T is the sampling period. The maximum frequency is

$$\omega = \frac{nr}{2} \tag{5.8}$$

The following shows the simulation loop and `FFTEnergy` call.

FFTSim.m

```matlab
nSim            = 2^16;           % Number of time steps
dT              = 0.1;            % Time step (sec)
dRHS            = RHSOscillator;  % Get the default data structure
dRHS.omega        = 0.1;            % Oscillator frequency
dRHS.zeta       = 0.1;            % Damping ratio
x               = [1;0];          % Initial state [position;velocity]
y1Sigma         = 0.001;          % 1 sigma position measurement noise

%% Simulation
xPlot = zeros(3,nSim);

for k = 1:nSim
  % Measurements
  y             = x(1) + y1Sigma*randn;
  % Plot storage
  xPlot(:,k)    = [x;y];
  % Propagate (numerically integrate) the state equations
  x             = RungeKutta( @RHSOscillator, 0, x, dT, dRHS );
end
```

FFTEnergy is shown as follows.

FFTEnergy.m

```matlab
function [e, w, wP] = FFTEnergy( y, tSamp, aPeak )
n = size( y, 2 );

% If the input vector is odd drop one sample
if( 2*floor(n/2) ~= n )
  n = n - 1;
  y = y(1:n,:);
end

x  = fft(y);
e  = real(x.*conj(x))/n;

hN = n/2;
e  = e(1,1:hN);
r  = 2*pi/(n*tSamp);
w  = r*(0:(hN-1));

if( nargout > 2 )
  k  = e > aPeak*max(e) ;
  wP = w(k);
end
```

The Fast Fourier Transform takes the sampled time sequence and computes the frequency spectrum. We compute the FFT using MATLAB's fft function. We take the result and multiply it by its conjugate to get the energy. The first half of the result has the frequency information. aPeak is to indicate peaks for the output. It is just looking for values greater than a certain threshold.

Figure 5.3 shows the damped oscillation. Figure 5.4 shows the spectrum. We find the peak by searching for the maximum value. The noise in the signal is seen at the higher frequencies. A noise-free simulation is shown in Figure 5.5.

The tuning approach is to

1. Excite the oscillator with a pulse
2. Run it for 2^n steps
3. Do an FFT
4. If there is only one peak, compute the damping gain

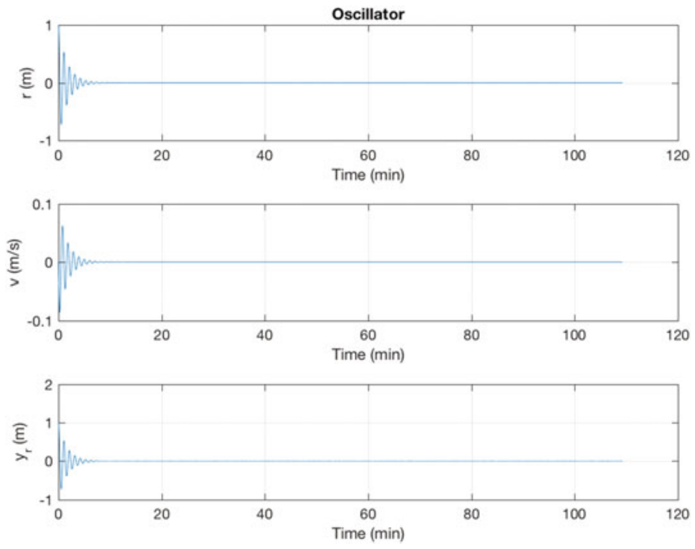


Figure 5.3: *Simulation of the damped oscillator. The damping ratio ζ is 0.5, and the undamped natural frequency ω is 0.1 rad/s*

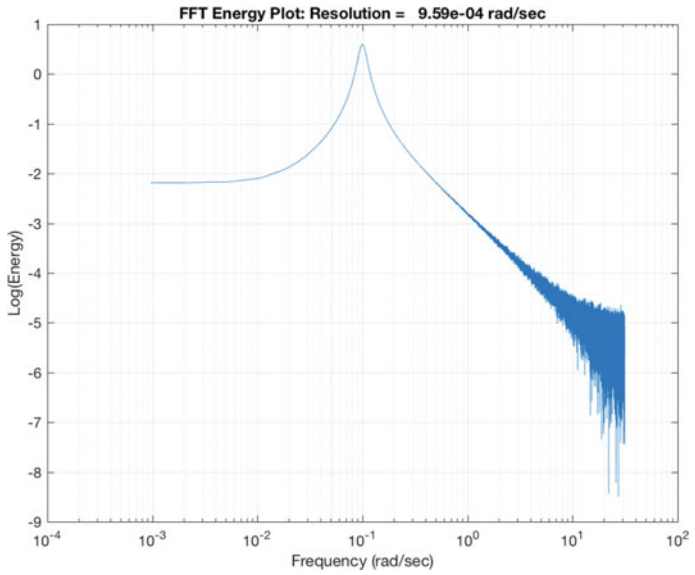


Figure 5.4: *The frequency spectrum. The peak is at the oscillation frequency of 0.1 rad/sec*

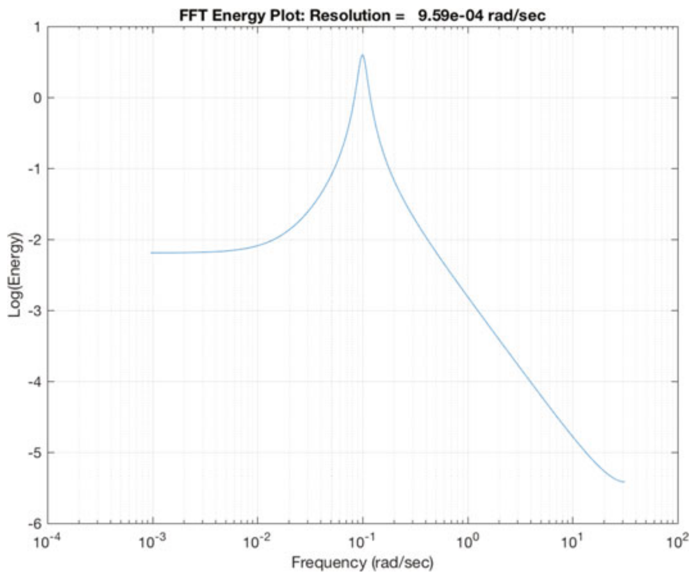


Figure 5.5: *The frequency spectrum without noise. The peak of the spectrum is at 0.1 rad/s in agreement with the simulation*

The script `TuningSim` calls `FFTEnergy.m` with `aPeak` set to 0.7. The value for `aPeak` is found by looking at a plot and picking a suitable number. The disturbances are Gaussian-distributed accelerations, and there is noise in the measurement. Note that this simulation uses a different right-hand-side function `RHSOscillatorControl`. The measurement with noise is implemented as

TuningSim.m

```
    % Measurements
    y            = x(1) + y1Sigma*randn;
```

The disturbances are implemented with a step perturbation, which ends at a given step, and random noise:

TuningSim.m

```
    dRHS.a       = aJ + a1Sigma*randn;
    if( k == kPulseStop )
        aJ = 0;
    end
```

The tuning code using `FFTEnergy` is shown in the following snippet.

TuningSim.m

```
FFTEnergy( yFFT, dT );
[ ~, ~, wP] = FFTEnergy( yFFT, dT );
if( length(wP) == 1 )
  wOsc    = wP;
  fprintf(1,'\tEstimated oscillator frequency %12.4f rad/s\n',wP);
  dRHS.c      = 2*zeta*wOsc;
else
  fprintf(1,'\tTuned\n');
end
```

The entire loop is run four times, with the first time undamped and the second, third, and fourth times updating the tuned gain. The results in the command window are

```
>> TuningSim
1:      Estimated oscillator frequency       0.0997 rad/s
2:      Tuned
3:      Tuned
4:      Tuned
```

If the random noise is large enough, the loop may tune more than once. Running it a few times or increasing the noise will show this behavior.

As you can see from the FFT plots in Figure 5.6, the spectra are “noisy” due to the sensor noise and Gaussian disturbance. The criteria for determining that the system is underdamped it is a distinctive peak. If the noise is large enough, we have to set lower thresholds to trigger the tuning. The top-left FFT plot shows the 0.1 rad/s peak. After tuning, we damp the oscillator sufficiently so that the peak is diminished. The time plot in Figure 5.6 (the bottom plot) shows that, initially, the system is lightly damped. After tuning, it oscillates very little. There is a slight transient every time the tuning is adjusted at 1.9, 3.6, and 5.5 seconds. The FFT plots (the top right and middle two) show the data used in the tuning.

An important point is that we must stimulate the system to identify the peak. All system identification, parameter estimation, and tuning algorithms have this requirement. An alternative to a pulse (which has a broad frequency spectrum) would be to use a sinusoidal sweep. That would excite any resonances and make it easier to identify the peak. However, care must be taken when exciting a physical system at different frequencies to ensure it does not have an unsafe or unstable response at natural frequencies.

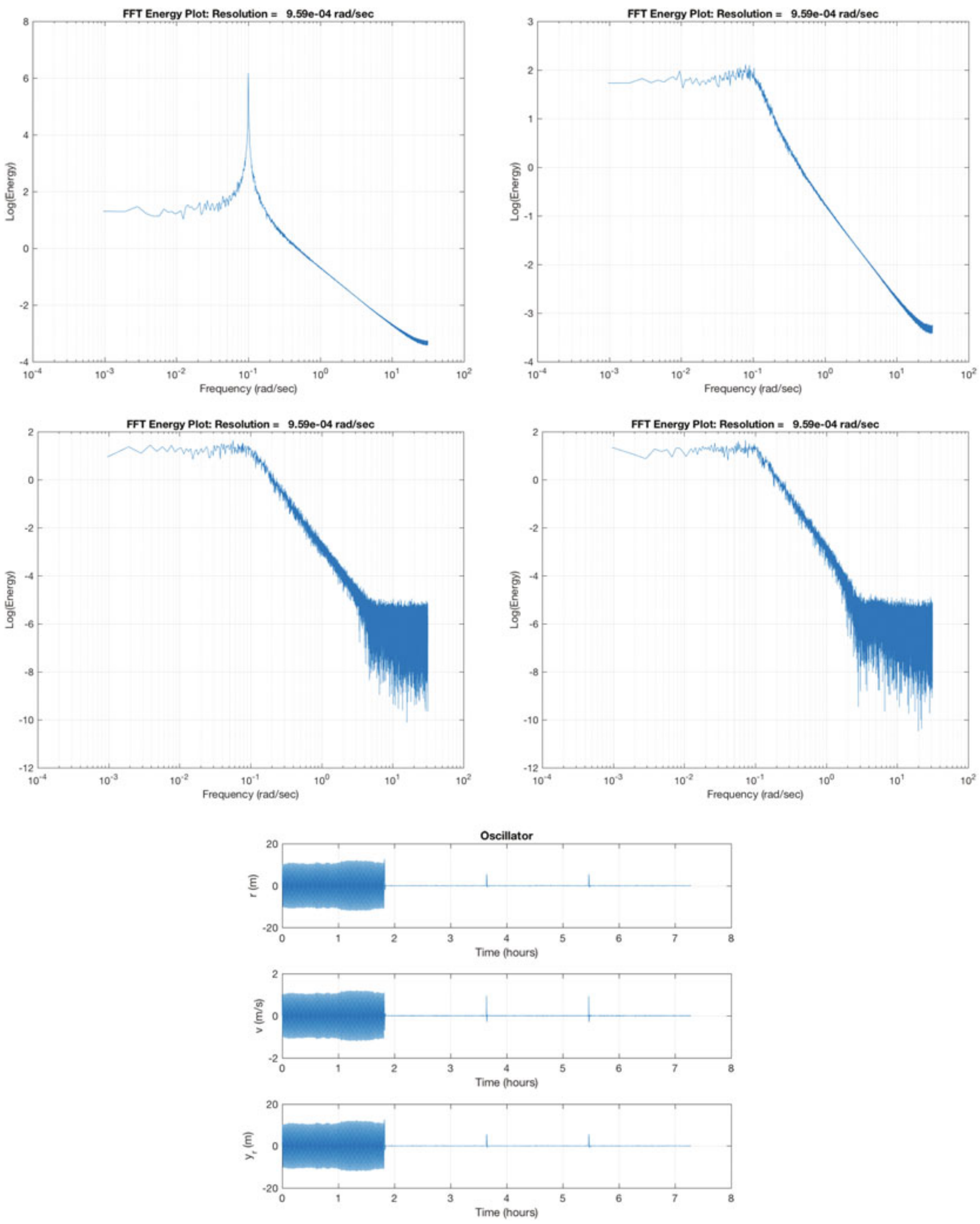


Figure 5.6: *Tuning simulation results. The first four plots are the frequency spectra taken at the end of each sampling interval; the last shows the results over time. Upper left, before tuning, the peak is seen*

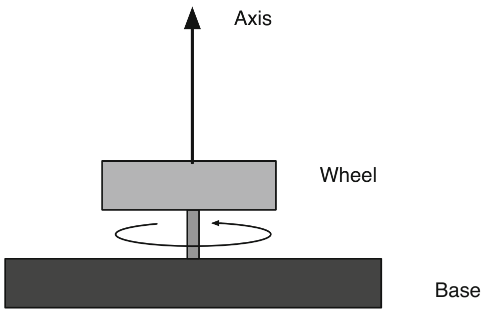


Figure 5.7: *Speed control of a rotor for the Model Reference Adaptive Control demo*

5.2 Implement MRAC

Our next example is to control a rotor with an unknown load so that it behaves in a desired manner. We will use Model Reference Adaptive Control (MRAC). The dynamical model of the rotary joint is [3] and is shown in Figure 5.7.

$$\frac{d\omega}{dt} = -a\omega + bu_c + u_d \tag{5.9}$$

where the damping a and/or input constants b are unknown. ω is the angular rate. u_c is the input voltage, and u_d is a disturbance angular acceleration. This is a first-order system that is modeled by one first-order differential equation. We would like the system to behave like the reference model:

$$\frac{d\omega}{dt} = -a_m\omega + b_m u_c + u_d \tag{5.10}$$

5.2.1 Problem

We want to control a system to behave like a particular model. Our example is a simple rotor.

5.2.2 Solution

The solution is to implement a Model Reference Adaptive Control (MRAC) function.

5.2.3 How It Works

The idea is to have a dynamic model that defines the behavior of your system. You want your system to have the same dynamics. This desired model is the reference, hence the name Model Reference Adaptive Control (MRAC). We will use the MIT rule [3] to design the adaptation system. The MIT rule was first developed at the MIT Instrumentation Laboratory (now Draper Laboratory), which developed the NASA Apollo and Space Shuttle guidance and control systems.

Consider a closed-loop system with one adjustable parameter, θ. θ is a parameter, not an angle. The desired output is y_m. The error is

$$e = y - y_m \tag{5.11}$$

Define a loss function (or cost) as

$$J(\theta) = \frac{1}{2}e^2 \tag{5.12}$$

The square removes the sign. If the error is zero, the cost is zero. We would like to minimize $J(\theta)$. To make J small, we change the parameters in the direction of the negative gradient of J or

$$\frac{d\theta}{dt} = -\gamma\frac{\partial J}{\partial\theta} = -\gamma e\frac{\partial e}{\partial\theta} \tag{5.13}$$

This is the MIT rule. If the system is changing slowly, then we can assume that θ is constant as the system adapts. γ is the adaptation gain. Our dynamic model is

$$\frac{d\omega}{dt} = a\omega + bu_c \tag{5.14}$$

We would like it to be the model:

$$\frac{d\omega_m}{dt} = a_m\omega_m + b_m u_c \tag{5.15}$$

a and b are the actual unknown parameters. a_m and b_m are the model parameters. We would like a and b to be a_m and b_m. Let the controller for our rotor be

$$u = \theta_1 u_c - \theta_2\omega \tag{5.16}$$

The second term provides the damping. The controller has two adaptation parameters. If they are chosen to be

$$\theta_1 = \frac{b_m}{b} \tag{5.17}$$

$$\theta_2 = \frac{a_m - a}{b} \tag{5.18}$$

the input-output relations of the system and model are the same. This is called perfect model following. This is not required. To apply the MIT rule, write the error as

$$e = \omega - \omega_m \tag{5.19}$$

With the parameters θ_1 and θ_2, the system is

$$\frac{d\omega}{dt} = -(a + b\theta_2)\omega + b\theta_1 u_c \tag{5.20}$$

where γ is the adaptation gain. To continue with the implementation, we introduce the operator $p = \frac{d}{dt}$. We then write

$$p\omega = -(a + b\theta_2)\omega + b\theta_1 u_c \tag{5.21}$$

or

$$\omega = \frac{b\theta_1}{p + a + b\theta_2} u_c \tag{5.22}$$

We need to get the partial derivatives of the error with respect to θ_1 and θ_2. These are

$$\frac{\partial e}{\partial \theta_1} = \frac{b}{p + a + b\theta_2} u_c \tag{5.23}$$

$$\frac{\partial e}{\partial \theta_2} = -\frac{b^2\theta_1}{(p + a + b\theta_2)^2} u_c \tag{5.24}$$

from the chain rule for differentiation. Noting that

$$u_c = \frac{p + a + b\theta_2}{b\theta_1}\omega \tag{5.25}$$

the second equation becomes

$$\frac{\partial e}{\partial \theta_2} = \frac{b}{p + a + b\theta_2} y \tag{5.26}$$

Since we don't know a, let's assume that we are pretty close to it. Then let

$$p + a_m \approx p + a + b\theta_2 \tag{5.27}$$

Our adaptation laws are now

$$\frac{d\theta_1}{dt} = -\gamma \left(\frac{a_m}{p + a_m} u_c \right) e \tag{5.28}$$

$$\frac{d\theta_2}{dt} = \gamma \left(\frac{a_m}{p + a_m} \omega \right) e \tag{5.29}$$

Let

$$x_1 = \frac{a_m}{p + a_m} u_c \tag{5.30}$$

$$x_2 = \frac{a_m}{p + a_m} \omega \tag{5.31}$$

which are differential equations that must be integrated. The complete set is

$$\frac{dx_1}{dt} = -a_m x_1 + a_m u_c \tag{5.32}$$

$$\frac{dx_2}{dt} = -a_m x_2 + a_m \omega \tag{5.33}$$

$$\frac{d\theta_1}{dt} = -\gamma x_1 e \tag{5.34}$$

$$\frac{d\theta_2}{dt} = \gamma x_2 e \tag{5.35}$$

Our only measurement would be ω which would be measured with a tachometer. As noted before, the controller is

$$u = \theta_1 u_c - \theta_2 \omega \tag{5.36}$$

$$e = \omega - \omega_m \tag{5.37}$$

$$\frac{d\omega_m}{dt} = -a_m \omega_m + b_m u_c \tag{5.38}$$

The MRAC is implemented in the function `MRAC` shown in its entirety in the following listing. The controller has five differential equations that are propagated. The states are $[x_1, x_2, \theta_1, \theta_2, \omega_m]$. `RungeKutta` is used for the propagation, but a less computationally intensive lower-order integrator, such as Euler, could be used instead. The function returns the default data structure if no inputs and one output is specified. The default data structure has reasonable values. That makes it easier for a user to implement the function. It only propagates one step.

MRAC.m

```
function d = MRAC( omega, d )

if( nargin < 1 )
  d = DataStructure;
  return
end

d.x = RungeKutta( @RHS, 0, d.x, d.dT, d, omega );
d.u = d.x(3)*d.uC - d.x(4)*omega;

%% MRAC>DataStructure
function d = DataStructure
% Default data structure

d = struct('aM',2.0,'bM',2.0,'x',[0;0;0;0;0],'uC',0,'u',0,'gamma',1,'dT
    ',0.1);

%% MRAC>RHS
function xDot = RHS( ~, x, d, omega )
% RHS for MRAC
```

```

e    = omega - x(5);
xDot = [-d.aM*x(1) + d.aM*d.uC;...
        -d.aM*x(2) + d.aM*omega;...
        -d.gamma*x(1)*e;...
         d.gamma*x(2)*e;...
        -d.aM*x(5) + d.bM*d.uC];
```

Now that we have the MRAC controller done, we'll write some supporting functions and then test it all out in `RotorSim`.

5.3 Generating a Square Wave Input

5.3.1 Problem

We need to generate a square wave to stimulate the rotor in the previous recipe.

5.3.2 Solution

For simulation and testing our controller, we will generate a square wave with a function.

5.3.3 How It Works

`SquareWave` generates a square wave. The first few lines are our standard code for running a demo or returning the data structure.

SquareWave.m

```
function [v,d] = SquareWave( t, d )

if( nargin < 1 )
  if( nargout == 0 )
    Demo;
  else
    v = DataStructure;
  end
  return
end

if( d.state == 0 )
  if( t - d.tSwitch >= d.tLow )
    v         = 1;
    d.tSwitch = t;
    d.state   = 1;
  else
    v         = 0;
  end
else
  if( t - d.tSwitch >= d.tHigh )
    v         = 0;
    d.tSwitch = t;
```

```
      d.state    = 0;
    else
      v          = 1;
    end
  end
```

This function uses `d.state` to determine if it is in the high or low part of a square wave. The width of the low part of the wave is set in `d.tLow`. The width of the high part of the square wave is set in `d.tHigh`. It stores the time of the last switch in `d.tSwitch`.

A square wave is shown in Figure 5.8. There are many ways to specify a square wave. This function produces a square wave with a minimum of zero and a maximum of one. You specify the time at zero and the time at one to create the square wave.

We adjusted the y-axis limit and line width using the following code.

SquareWave.m

```
PlotSet(t,v,'x label', 't (sec)', 'y label', 'v', 'plot title','Square
    Wave',...
         'figure title', 'Square Wave');
set(gca,'ylim',[0 1.2])
h = get(gca,'children');
set(h,'linewidth',1);
```

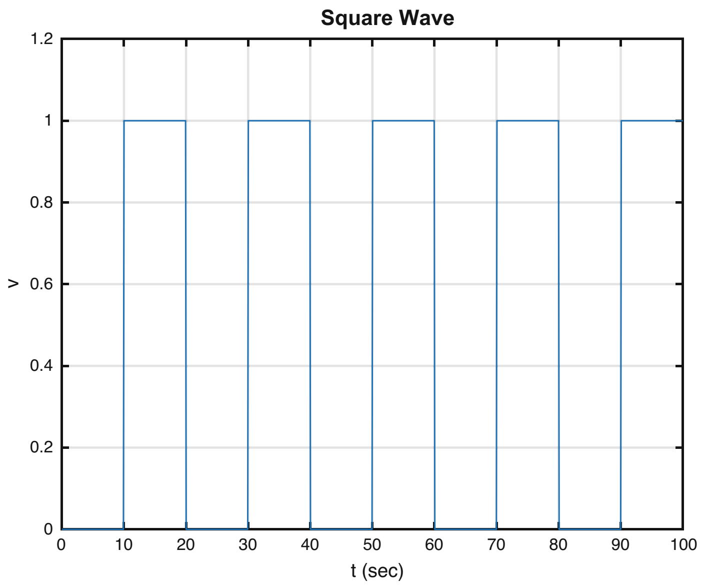


Figure 5.8: *Square wave*

■ **TIP** `h = get(gca,'children')` gives you access to the line data structure in a plot for the most recent axes.

5.4 Demonstrate MRAC for a Rotor

5.4.1 Problem

We want to create a recipe to control our rotor using MRAC.

5.4.2 Solution

The solution is to implement our Model Reference Adaptive Control (MRAC) function in a MATLAB script from Recipe 5.2.

5.4.3 How It Works

MRAC is implemented in the script `RotorSim`. It calls `MRAC` to control the rotor. As in our other scripts, we use `PlotSet` for our 2D plots. Notice that we use two new options. One `'plot set'` allows you to put more than one line on a subplot. The other `'legend'` adds legends to each plot. The cell array argument to `'legend'` has a cell array for each plot. In this case, we have two plots each with two lines, so the cell array is

```
{{'true', 'estimated'} ,{'Control' ,'Command'}}
```

Each plot legend is a cell entry within the overall cell array.

The rotor simulation script with MRAC is shown in the following listing. The square wave functions generate the command to the system that ω should track. `RHSRotor`, `SquareWave`, and `MRAC` all return default data structures. `MRAC` and `SquareWave` are called once per pass through the loop. The simulation right-hand-side, that is the dynamics of the rotor, in `RHSRotor`, are then propagated using `RungeKutta`. Note that we pass to pointer for `RHSRotor` to `RungeKutta`.

RotorSim.m

```
%% Initialize
nSim     = 4000;      % Number of time steps
dT     = 0.1;       % Time step (sec)
dRHS     = RHSRotor;      % Get the default data structure
dC     = MRAC;
dS     = SquareWave;
x        = 0.1;       % Initial state vector

%% Simulation
xPlot  = zeros(4,nSim);
theta  = zeros(2,nSim);
t      = 0;
for k = 1:nSim
```

```

  % Plot storage
  xPlot(:,k)   = [x;dC.x(5);dC.u;dC.uC];
  theta(:,k)   = dC.x(3:4);
  [uC, dS]     = SquareWave( t, dS );
  dC.uC        = 2*(uC - 0.5);
  dC           = MRAC( x, dC );
  dRHS.u       = dC.u;

  % Propagate (numerically integrate) the state equations
  x            = RungeKutta( @RHSRotor, t, x, dT, dRHS );
  t            = t + dT;
end
```

■ **TIP** Pass pointers `@fun` instead of strings `'fun'` to functions whenever possible.

`RHSRotor` is shown as follows.

RHSRotor.m

```
function xDot = RHSRotor( ~, x, d )

if( nargin < 1 )
  xDot = struct('a',1,'b',0.5,'u',0);
  return
end

xDot   = -d.a*x + d.b*d.u;
```

The dynamics are just one line of code. The remaining returns the default data structure.

The results are shown in Figure 5.9. We set the adaptation gain, γ, to 1. a_m and b_m are set equal to 2. a is set equal to 1 and b to $\frac{1}{2}$.

The first plot shows the rotor's estimated and true angular rates on top and the control demand and actual control sent to the wheel on the bottom. The desired control is a square wave (generated by `SquareWave`). Notice the transient in the applied control at the transitions of the square wave. The control amplitude is greater than the commanded control. Notice also that the angular rate approaches the desired commanded square wave shape.

Figure 5.10 shows the convergence of the adaptive gains, θ_1 and θ_2. They have converged by the end of the simulation.

MRAC learns the gains of the system by observing the response to the control excitation. It requires excitation to converge. This is the nature of all learning systems. If there is insufficient stimulation, it isn't possible to observe the behavior of the system, so there is not enough information for learning. It is easy to find an excitation for a first-order system. For higher-order systems or nonlinear systems, this can be more difficult.

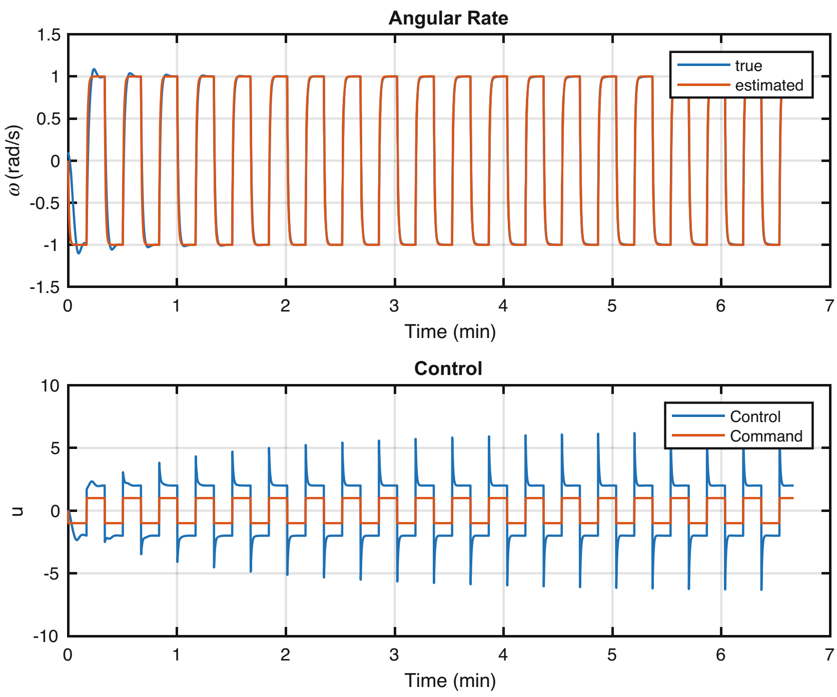


Figure 5.9: *MRAC control of a rotor*

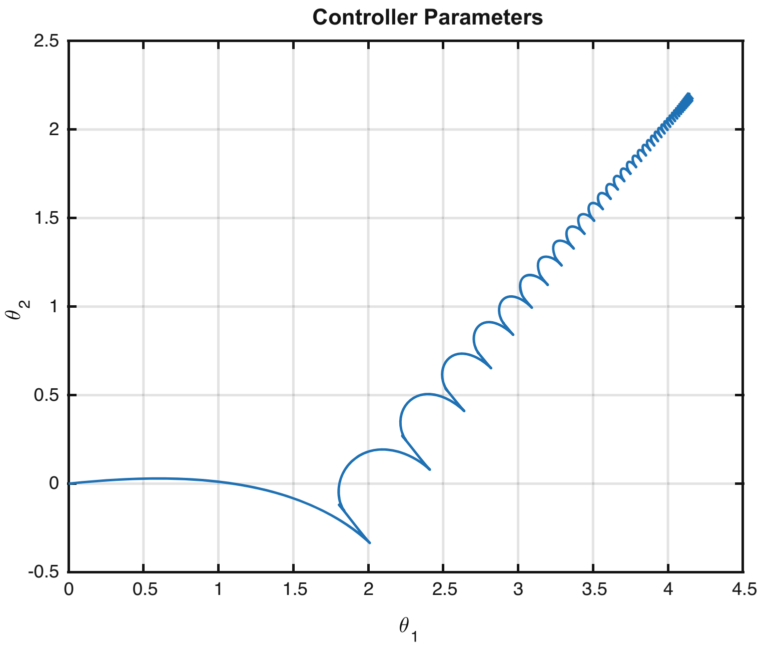


Figure 5.10: *Gain convergence in the MRAC controller*

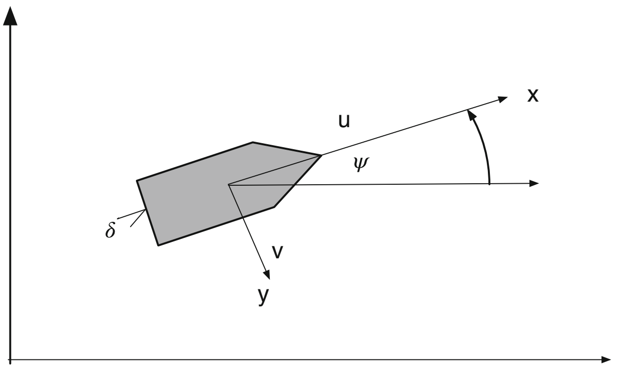


Figure 5.11: *Ship heading control for gain scheduling control*

5.5 Ship Steering: Implement Gain Scheduling for Steering Control of a Ship

5.5.1 Problem

We want to steer a ship at all speeds. The problem is that the dynamics are speed dependent, making this a nonlinear problem. The model is shown in Figure 5.11.

5.5.2 Solution

The solution is to use gain scheduling to set the gains based on speeds. The gain schedule is learned by automatically computing gains from the dynamical equations of the ship. This is similar to the self-tuning example except that we are seeking a set of gains for all speeds, not just one. In addition, we assume that we know the model of the system.

5.5.3 How It Works

The dynamical equations for the heading of a ship are in state space form [3]:

$$\begin{bmatrix} \dot{v} \\ \dot{r} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \left(\frac{u}{l}\right) a_{11} & u a_{12} & 0 \\ \left(\frac{u}{l^2}\right) a_{21} & \left(\frac{u}{l}\right) a_{22} & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} v \\ r \\ \psi \end{bmatrix} + \begin{bmatrix} \left(\frac{u^2}{l}\right) b_1 \\ \left(\frac{u^2}{l^2}\right) b_2 \\ 0 \end{bmatrix} \delta + \begin{bmatrix} \alpha_v \\ \alpha_r \\ 0 \end{bmatrix} \tag{5.39}$$

v is the transverse speed, u is the ship's speed, l is the ship length, r is the turning rate, and ψ is the heading angle. α_v and α_r are disturbances. The ship is assumed to be moving at speed u. This is achieved by the propeller that is not modeled. The control is rudder angle δ. Notice that if $u = 0$, the ship cannot be steered. All of the coefficients in the state matrix are functions of u, except for the heading angle. Our goal is to control the heading given the disturbance acceleration in the first equation and the disturbance angular rate in the second.

The disturbances only affect the dynamics states, r, and v. The last state, ψ, is a kinematic state and does not have a disturbance.

Table 5.1: *Ship parameters [3]*

Parameter	Minesweeper	Cargo	Tanker
l	55	161	350
a_{11}	-0.86	-0.77	-0.45
a_{12}	-0.48	-0.34	-0.44
a_{21}	-5.20	-3.39	-4.10
a_{22}	-2.40	-1.63	-0.81
b_1	0.18	0.17	0.10
b_2	1.40	-1.63	-0.81

The ship model is shown in the following code, `RHSShip`. The second and third outputs are for use in the controller. Notice that the differential equations are linear in the state and the control. Both matrices are a function of the forward velocity. We are not trying to control the forward velocity, it is an input to the system. The default parameters for the minesweeper are given in Table 5.1. These are the same numbers that are in the default data structure.

RHSShip.m

```
function [xDot, a, b] = RHSShip( ~, x, d )

if( nargin < 1 )
  xDot = struct('l',100,'u',10,'a',[-0.86 -0.48;-5.2 -2.4],'b'
      ,[0.18;-1.4],'alpha',[0;0;0],'delta',0);
  return
end

uOL   = d.u/d.l;
uOLSq = d.u/d.l^2;
uSqOl = d.u^2/d.l;
a     = [  uOL*d.a(1,1) d.u*d.a(1,2) 0;...
          uOLSq*d.a(2,1) uOL*d.a(2,2) 0;...
                      0            1 0];
b     = [uSqOl*d.b(1);...
          uOL^2*d.b(2);...
          0];

xDot  = a*x + b*d.delta + d.alpha;
```

In the ship simulation, `ShipSim`, we linearly increase the forward speed while commanding a series of heading `psi` changes. The controller takes the state space model at each time step and computes new gains which are used to steer the ship. The controller is a linear quadratic regulator. We can use full-state feedback because the states are easily modeled. Such controllers will work perfectly in this case but are a bit harder to implement when you need to estimate some of the states or have unmodeled dynamics.

ShipSim.m

```
for k = 1:nSim
  % Plot storage
  xPlot(:,k)   = x;
  dRHS.u       = u(k);
  % Control
  % Get the state space matrices
  [~,a,b]      = RHSShip( 0, x, dRHS );
  gain(k,:)    = QCR( a, b, qC, rC );
  dRHS.delta   = -gain(k,:)*[x(1);x(2);x(3) - psi(k)]; % Rudder angle
  delta(k)     = dRHS.delta;
  % Propagate (numerically integrate) the state equations
  x            = RungeKutta( @RHSShip, 0, x, dT, dRHS );
end
```

The quadratic regulator generator code is shown in the following listing. It generates the gain from the matrix Riccati equation. A Riccati equation is an ordinary differential equation that is quadratic in the unknown function. In steady state, this reduces to the algebraic Riccati equation that is solved in this function.

QCR.m

```
function k = QCR( a, b, q, r )

[sinf,rr] = Riccati( [a,-(b/r)*b';-q',-a'] );

if( rr == 1 )
  disp('Repeated roots. Adjust q, r or n');
end

k = r\(b'*sinf);

function [sinf, rr] = Riccati( g )
%% Ricatti
%   Solves the matrix Riccati equation in the form
%
%   g = [a   r ]
%       [q  -a']

rg = size(g);

[w, e] = eig(g);

es = sort(diag(e));

% Look for repeated roots
j = 1:length(es)-1;

if ( any(abs(es(j)-es(j+1))<eps*abs(es(j)+es(j+1))) )
  rr = 1;
```

```
else
  rr = 0;
end

% Sort the columns of w
ws    = w(:,real(diag(e)) < 0);

sinf  = real(ws(rg/2+1:rg,:)/ws(1:rg/2,:));
```

a is the state transition matrix, b is the input matrix, q is the state cost matrix, and r is the control cost matrix. The bigger the elements of q, the more cost we place on deviations of the states from zero. That leads to tight control at the expense of more control. The bigger the elements of b the more cost we place on control. Bigger b means less control. Quadratic regulators guarantee stability if all states are measured. They are a very handy controller to get something working. The results are given in Figure 5.12. Note how the gains evolve.

The gain on the angular rate r is nearly constant. Notice that the ψ range is very small! Normally, you would zoom out the plot. The other two gains increase with speed. This is an example of gain scheduling. The difference is that we autonomously compute the gains from perfect measurements of the ship's forward speed.

ShipSimDisturbance is a modified version of ShipSim that is a shorter duration, with only one-course change, and with disturbances in both angular rate and lateral velocity. The results are given in Figure 5.13.

5.6 Spacecraft Pointing

5.6.1 Problem

We want to control the orientation of a spacecraft with thrusters for control. We do not know the inertia, which has a major impact on control.

5.6.2 Solution

The solution is to use a parameter estimator to estimate the inertia and feed it into the control system.

5.6.3 How It Works

The spacecraft model is shown in Figure 5.14.

The dynamical equations are

$$I = I_0 + m_f r_f^2 \quad (5.40)$$

$$T_c + T_d = I\ddot{\theta} + \dot{m}_f r_f^2 \dot{\theta} \quad (5.41)$$

$$\dot{m}_f = -\frac{T_c}{r u_e} \quad (5.42)$$

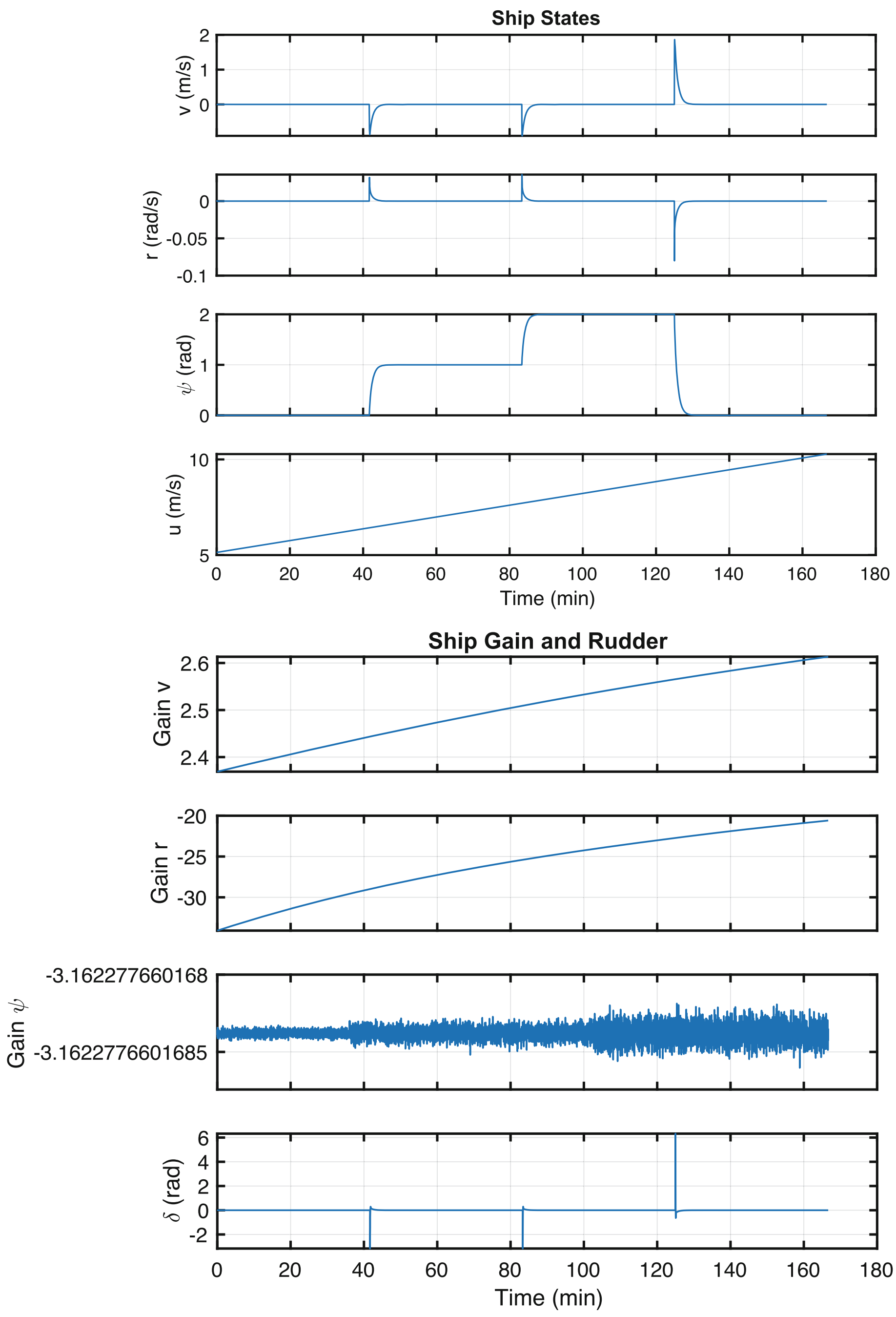


Figure 5.12: *Ship steering simulation. The states are shown on the top with the forward velocity. The gains and rudder angle are shown on the bottom. Notice the "pulses" in the rudder to make the maneuvers*

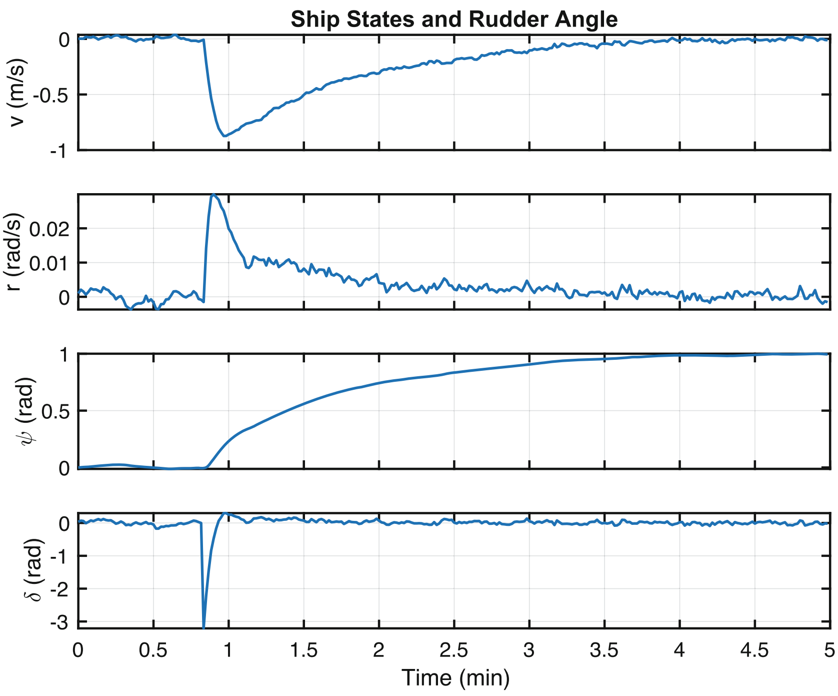


Figure 5.13: *Ship steering simulation. The states are shown on the left with the rudder angle. The disturbances are Gaussian white noise*

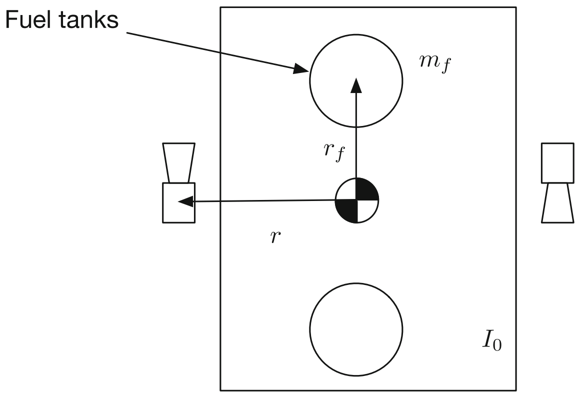


Figure 5.14: *Spacecraft model*

where I is the total inertia, I_0 is the constant inertia for everything except the fuel mass, T_c is the thruster control torque, T_d is the disturbance torque, m_f is the total fuel mass, r_f is the distance to the fuel tank center (moment arm), r is the vector to the thrusters, u_e is the thruster exhaust velocity, and θ is the angle of the spacecraft axis. Fuel consumption is balanced between the two tanks, so the center of mass remains at (0,0). The second term in the second equation is the inertia derivative term, which adds damping to the system.

Our controller is a PD (proportional derivative) controller of the form

$$T_c = Ia \tag{5.43}$$

$$a = -K(\theta + \tau\dot{\theta}) \tag{5.44}$$

K is the forward gain and τ the rate constant. We design the controller for unit inertia and then estimate the inertia so that our dynamic response is always the same. We will estimate the inertia using a very simple algorithm:

$$I_k = K_I I_{k-1} - (1 - K_I)\frac{T_{c_k}}{\ddot{\theta}_k} \tag{5.45}$$

K_I is less than or equal to one. We will do this only when the control torque is not zero and the change in rate is not zero. This is a first difference approximation and should be good if we don't have a lot of noise. The following code snippet shows the simulation loop with the control system. The dynamics are in `RHSSpacecraft.m`.

SpacecraftSim.m

```matlab
%% Controller
kForward   = 0.05;
tau        = 10;
tCThresh   = 0.00;
kI         = 0.9; % Inertia filter gain

%% Simulation
xPlot      = zeros(7,nSim);
inrEst     = 1.01*(dRHS.i0 + dRHS.rF^2*x(3)) + 0.05*randn(1)*dRHS.i0;
dRHS.tC    = 0;

for k = 1:nSim
  % Control
  dRHS.tC = -inrEst*kForward*(x(1) + tau*x(2));
  % Collect plotting information
  [xDot,inrTrue] = RHSSpacecraft(0,x,dRHS);
  omegaDot = xDot(2); % from gyro
  if( abs(dRHS.tC) > tCThresh  )
    inrEst = kI*inrEst + (1-kI)*dRHS.tC/omegaDot;
  end
  xPlot(:,k) = [x;inrEst;dRHS.tD;dRHS.tC;inrTrue];
        % Propagate (numerically integrate) the state equations
        x = RungeKutta( @RHSSpacecraft, 0, x, dT, dRHS );
end
```

We only estimate inertia when the control torque is above a threshold. This prevents us from responding to noise. We also incorporate the inertia estimator in a simple low-pass filter. The results are shown in Figure 5.15. The threshold means the algorithm only estimates inertia at the very beginning of the simulation when it is reducing the attitude error.

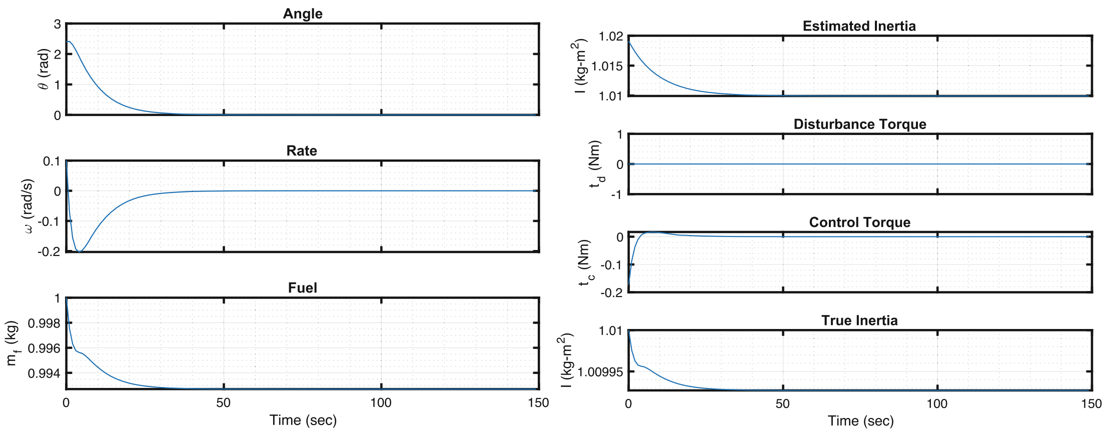


Figure 5.15: *States and control outputs from the spacecraft simulation*

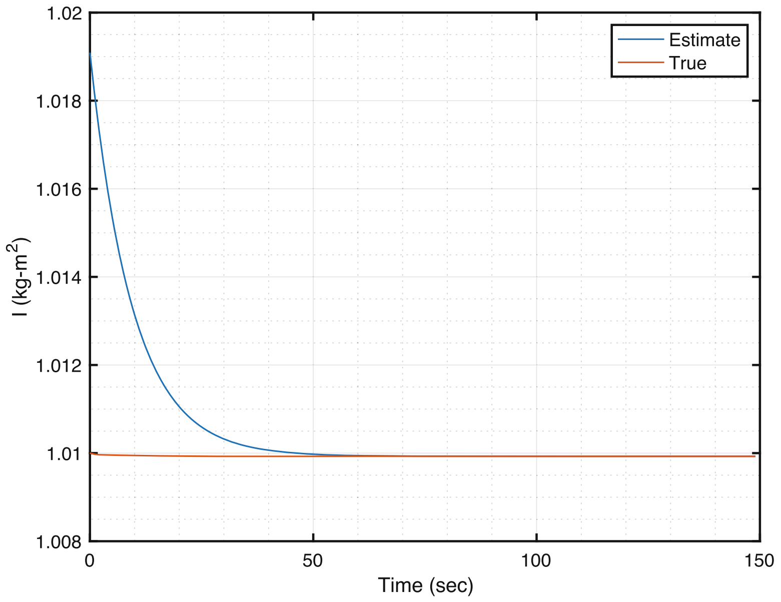


Figure 5.16: *Estimated and actual inertia from the spacecraft simulation*

The dynamics function computes the true inertia from the fuel mass state and the dry mass inertia. This allows the script to compare the estimate against the truth value in Figure 5.16.

This algorithm appears crude, but it is fundamentally all we can do in this situation given just angular rate measurements. Note that the inertia estimate happens while the control is operating, making this a nonlinear controller. More sophisticated filters or estimators could improve the performance.

5.7 Direct Adaptive Control

5.7.1 Problem

We want to control a system for which the plant is unknown. This is one in which the order and parameters for the model are unknown.

5.7.2 Solution

The solution is to use direct adaptation based on Lyapunov control.

5.7.3 How It Works

Assume the dynamics equation is

$$\dot{y} = ay + bu \tag{5.46}$$

u is the control. If a is < 0, the system will always converge. If we use feedback control of the form $u = -ky$, then

$$\dot{y} = (a - bk)y + bu_d \tag{5.47}$$

where u_d is an external disturbance. If $a-bk$ is positive, the system is unstable. If we don't know a or b, then we can't guarantee stability with a fixed gain control. We could try and estimate a and b and then design the controller in real time. A simple approach [18] is an adaptive controller. Assume that $b > 0$, then the gain is

$$\dot{k} = y^2 \tag{5.48}$$

This is known as a universal regulator. To show this is stable, pick the Lyapunov function:

$$V - \frac{y^2}{2} \tag{5.49}$$

Its derivative is

$$\dot{V} = (a - bk)y^2 = (-bk)\dot{k} \tag{5.50}$$

Integrating

$$\frac{y^2}{2} = ak - \frac{bk^2}{2} + C \tag{5.51}$$

Since $\dot{k} > 0$, k can only increase. k has to be bounded because, otherwise, the right-hand side could be negative, which is impossible because the left-hand side is always positive. The following script implements the controller with $a > 0$. Notice how the controller drives the error to zero.

DirectAdaptiveControl.m

```
%% Direct adaptive control demo
% Reference: ECE 517: Nonlinear and Adaptive Control Lecture Notes
% Daniel Liberzon November 3, 2021

n       = 1000;
dT      = 0.1;

% Plant
a       = 0.1;
b       = 1;
x       = 0.1;

% Initial gain
gain    = 0.1;

% Storage
xP      = zeros(3,n);
for k = 1:n
  gain        = gain + dT*x^2;
  u           = -gain*x;
  xP(:,k)     = [x;u;gain];
  x           = RungeKutta(@RHS,0,x,dT,a,b,u);
end

yL = {'x','u','K'};

t  = (0:n-1)*dT;

TimeHistory(t,xP,yL,'Direct Adaptive Control');

%% Right hand side
function xDot = RHS(~,x,a,b,u)

xDot = a*x + b*u;

end
```

The results are shown in Figure 5.17. Note the rapid convergence. No knowledge of a or b is required. a and b are never estimated.

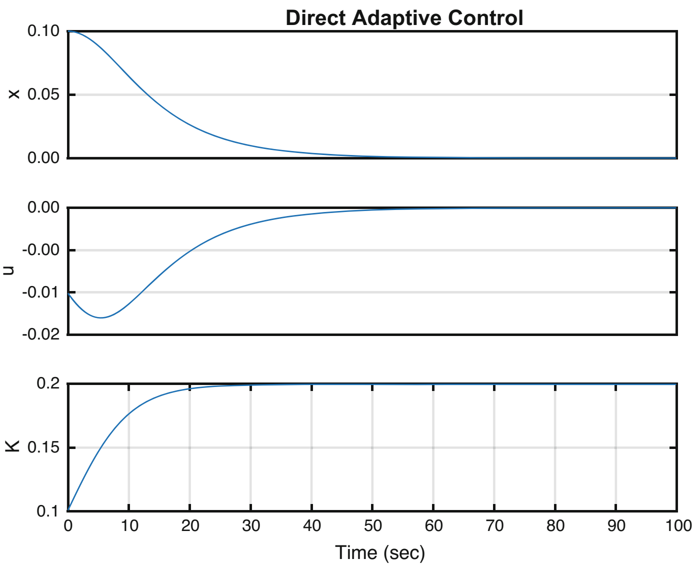


Figure 5.17: *Direct adaptive control*

Table 5.2: *Chapter Code Listing*

File	Description
`DirectAdaptiveControl`	Direct adaptive control simulations
`FFTEnergy`	Generates FFT energy
`FFTSim`	Demonstration of the Fast Fourier Transform
`MRAC`	Implements Model Reference Adaptive Control
`QCR`	Generates a full-state feedback controller
`RHSOscillatorControl`	Right-hand side of a damped oscillator with a velocity gain
`RHSRotor`	Right-hand side for a rotor
`RHSShip`	Right-hand side for a ship steering model
`RHSSpacecraft`	Right-hand side for a spacecraft model
`RotorSim`	Simulation of Model Reference Adaptive Control
`ShipSim`	Simulation of ship steering
`ShipSimDisturbance`	Simulation of ship steering with disturbances
`SpacecraftSim`	Spacecraft control with inertia estimation
`SquareWave`	Generates a square wave
`TuningSim`	Controller tuning demonstration
`WrapPhase`	Keeps angles between $-\pi$ and π

5.8 Summary

This chapter has demonstrated adaptive or learning control. You learned about model tuning, model reference adaptive control, adaptive control, and gain scheduling. Table 5.2 lists the functions and scripts included in the companion code.