CHAPTER 9

Electric Motors

We will model a three-phase permanent magnet motor driven by a direct current (DC) power source. This has three coils on the stator and permanent magnets on the rotor. This type of motor is driven by a DC power source with six semiconductor switches that are connected to the three coils, known as the A, B, and C coils. Two or more coils can be used to drive a brushless DC motor, but three coils are particularly easy to implement. This type of motor is used in many industrial applications today, including electric cars and robotics. It is sometimes called a brushless DC motor (BLDC) or a permanent magnet synchronous motor (PMSM).

Pulsewidth modulation is used for the switching because it is efficient; the switches are off when not needed. Coding the model for the motor and the pulsewidth modulation is relatively straightforward. In the simulation, we will demonstrate using two time steps, one for the simulation to handle the pulsewidths and one for the outer control loop. The simulation script will have multiple control flags to allow for debugging this complex system.

Figure 9.1 shows the big picture in this chapter. We will look at the motor model first (the AC motor block), then at the pulsewidth modulation (the SVPWM and three-phase inverter block). The controller is covered last. Each of these major functions is in a separated gray block.

9.1 Modeling a Three-Phase Brushless Permanent Magnet Motor

Problem

We want to model a three-phase permanent magnet synchronous motor in a form suitable for control system design. A conceptual drawing is shown in Figure 9.2. The motor has three stator windings and one permanent magnet on the rotor. The magnet has two poles or one pole pair. The coordinate axes are the a, b, and c on the stator, one axis at the center of each coil following the right-hand rule, and the (d,q) coordinates fixed to the magnet in the rotating frame. In motor applications, the axes represent currents or voltages, not positions like in mechanical engineering.

M. Paluszek and S. Thomas, *MATLAB Recipes*,
https://doi.org/10.1007/978-1-4842-6124-8_9

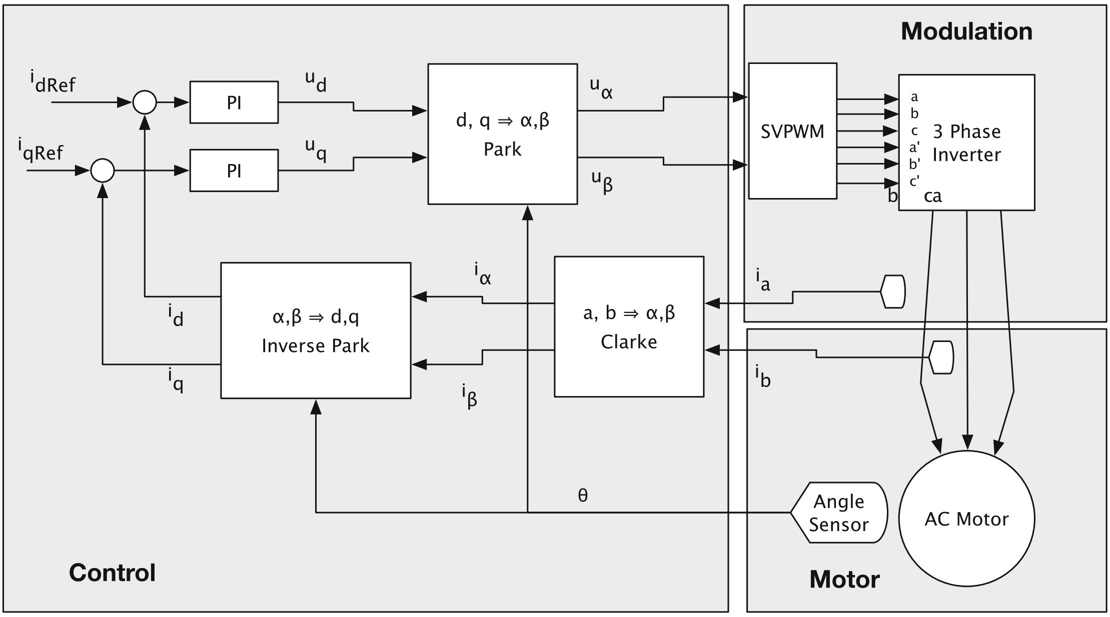

Figure 9.1: *Motor controller. PI is the proportional-integral controller. PWM the is pulsewidth modulation. There are two current sensors measuring i_a and i_b and one angle sensor measuring θ.*

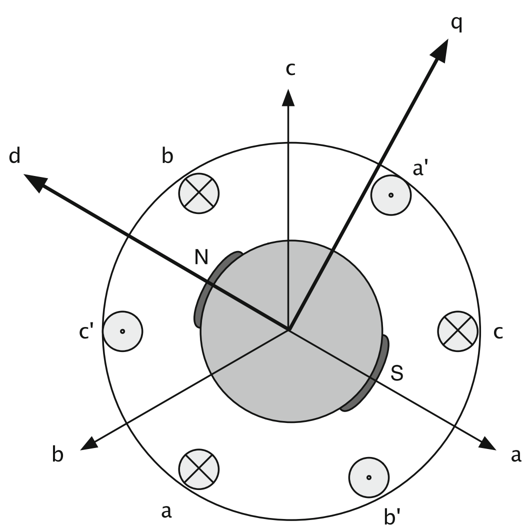

Figure 9.2: *Motor diagram showing the three-phase coils a, b, and c on the stator and the two-pole magnet (N,S) on the rotor. The $\times$ means the current is going into the paper; the dot means it is coming out of the paper.*

Solution

The solution is to model a motor with three stator coils and permanent magnets on the rotor. We have to model the coil currents and the physical state of the rotor.

How It Works

Permanent magnet synchronous motors use two or more windings in the stator and permanent magnets in the rotor. The rotor can have any even number of magnet poles. The phasing of the currents in the stator coils must be synchronized with the position of the rotor. Define the inductance matrix L, which gives the coupling between currents in different loops[1]:

$$L = \frac{1}{d}\begin{bmatrix} 2L_{ss} - L_m & L_m & L_m \\ L_m & 2L_{ss} - L_m & L_m \\ L_m & L_m & 2L_{ss} - L_m \end{bmatrix} \tag{9.1}$$

where

$$d = 2L_{ss}^2 - L_{ss}L_m - L_m^2 \tag{9.2}$$

L_m is the mutual inductance of the phase windings and L_{ss} is the self-inductance. Self-inductance is the effect of a current in a loop on itself. Mutual inductance is the effect of the current in one loop on another loop. The three-phase current array, i, is

$$i = \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} \tag{9.3}$$

where i_a is the phase A stator winding current, i_b is the phase B current, and i_c is the phase C current.

The phase voltage vector, u, is

$$u = \begin{bmatrix} u_a \\ u_b \\ u_c \end{bmatrix} \tag{9.4}$$

where u_a is the phase A stator winding voltage. The dynamical equations are a set of first-order differential equations and are

$$\begin{bmatrix} \dot{i} \\ \dot{\omega}_e \\ \dot{\theta}_e \end{bmatrix} = \begin{bmatrix} -r_sL & 0 & 0 \\ 0 & -\frac{b}{J} & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} i \\ \omega_e \\ \theta_e \end{bmatrix} + \psi \begin{bmatrix} -L\omega_e \\ \frac{p^2 i^T}{4J} \\ 0\,0\,0 \end{bmatrix}\begin{bmatrix} \cos\theta_e \\ \cos(\theta_e + \frac{2\pi}{3}) \\ \cos(\theta_e - \frac{2\pi}{3}) \end{bmatrix} + \begin{bmatrix} L \\ 0\,0\,0 \\ 0\,0\,0 \end{bmatrix} u$$
$$+ \begin{bmatrix} 0 \\ \frac{p}{2J} \\ 0 \end{bmatrix} T_L \tag{9.5}$$

[1]Lyshevski, S. E. “Electromechanical Systems, Electric Machines, and Applied Mechatronics,” CRC Press, 2000, pp. 589-627.

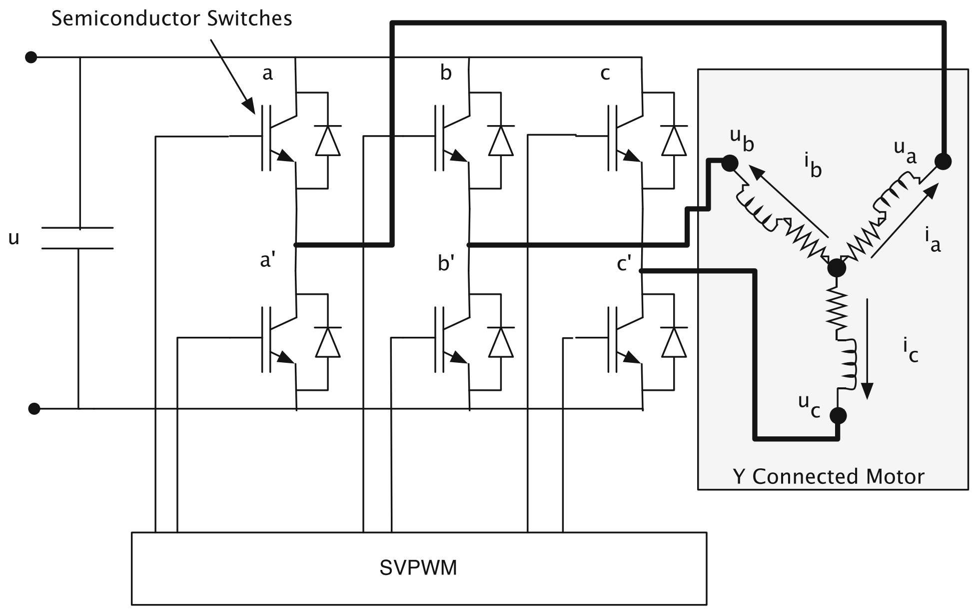

Figure 9.3: *Motor three-phase driver circuitry. The semiconductor switches shown in the diagram are IGBT (integrated gate bipolar transistors). The pulsewidth modulation block, SVPWM, is discussed in Recipe 9.3.*

where ω is the rotor angular rate, θ is the rotor angle, p is the number of rotor poles, b is the viscous damping coefficient, r_s is the stator resistance, ψ is the magnetic flux, T_L is the load torque, and J is the rotor inertia. i and u are the phase winding 3-vectors shown earlier, and L is the 3-by-3 inductance matrix also shown earlier. Equation 9.5 is actually five equations in matrix form. The first three equations, for the current array i, are the electrical dynamics. The last two for ω_e and θ_e are the mechanical dynamics represented in electrical coordinates.

The driver circuitry is shown in Figure 9.3. It has six semiconductor switches. In this model, they are considered ideal, meaning they can switch instantaneously at any frequency we desire. In practice, switches will have a maximum switching speed and will have some transient response. Note that the motor is Y connected, meaning that the ends of the three-phase windings are tied together.

The right-hand-side code is shown in the following. The first output is the state derivative, as needed for integration. The second output is the electrical torque needed for the control. The first block of code defines the motor model data structure with the parameters needed by our dynamics equation. This structure can be retrieved by calling the function with no inputs. The remaining code implements Equation 9.5. Note the suffix `M` used for ω and θ, to reinforce that these are mechanical quantities; this distinguishes them from the electrical quantities which are related by $p/2$, where p is the number of poles. The use of M and E subscripts is typical when writing software for motors.

The function returns a default data structure if no input arguments are passed to it. This is a convenient way for the designer of the code to give users a working starting point for the model.

This way, the user only has to change parameters that are different from the default. It lets the user get up and running quickly.

The electrical torque is a second output argument. It is not used during numerical integration but is helpful when debugging the function. It is useful to output quantities that a user might want to plot too. MATLAB is helpful in allowing multiple outputs for a function.

RHSPMMachine.m

```
%% RHSPMMACHINE Permanent magnet machine model in ABC coordinates.
% Assumes a 3 phase machine in a Y connection. The permanent magnet
% flux
% distribution is assumed sinusoidal.
%% Forms
%   d = RHSPMMachine
%   [xDot,tE] = RHSPMMachine( ~, x, d )
%
%% Inputs
%   t    (1,1)     Time, unused
%   x    (5,1)     The state vector [iA;iB;iC;omegaE;thetaE]
%   d     (.)      Data structure
%                      .lM    (1,1) Mutual inductance
%                      .psiM  (1,1) Permanent magnet flux
%                      .lSS   (1,1) Stator self inductance
%                      .rS    (1,1) Stator resistance
%                      .p     (1,1) Number of poles (1/2 pole pairs)
%                      .u     (3,1) [uA;uB;uC]
%                      .tL    (1,1) Load torque
%                      .bM    (1,1) Viscous damping (Nm/rad/s)
%                      .j     (1,1) Inertia
%                      .u     (3,1) Phase voltages [uA;uB;uC]
%
%% Outputs
%   x    (5,1)     The state vector derivative
%   tE   (1,1)     Electrical torque
%
%% Reference
% Lyshevski, S. E., "Electromechanical Systems, Electric Machines, and
% Applied Mechatronics," CRC Press, 2000.

function [xDot, tE] = RHSPMMachine( ~, x, d )

if( nargin == 0 )
  xDot = struct('lM',0.0009,'psiM',0.069, 'lSS',0.0011,'rS',0.5,'p'
      ,2,...
                'bM',0.000015,'j',0.000017,'tL',0,'u',[0;0;0]);
  return
end

% Pole pairs
pP = d.p/2;

```

```
% States
i       = x(1:3);
omegaE = x(4);
thetaE = x(5);

% Inductance matrix
denom = 2*d.lSS^2 - d.lSS*d.lM - d.lM^2;
l2    = d.lM;
l1    = 2*d.lSS - l2;
l     = [l1 l2 l2;l2 l1 l2;l2 l2 l1]/denom;

% Right hand side
tP3       = 2*pi/3;
c         = cos(thetaE + [0;-tP3;tP3]);
iDot      = l*(d.u - d.psiM*omegaE*c - d.rS*i);
tE        = pP^2*d.psiM*i'*c;
omegaDot = (tE - d.bM*omegaE - 0.5*pP*d.tL)/d.j;
xDot      = [iDot;omegaDot;omegaE];
```

9.2 Controlling the Motor

Problem

We want to control the motor to produce a desired torque. Specifically, we need to compute the voltages to apply to the stator coils.

Solution

We will use *field-oriented control* with a proportional-integral controller to control the motor. Field-oriented control is a control method where the stator currents are transformed into two orthogonal components. One component defines the magnetic flux of the motor and the other defines the torque. The control voltages we calculate will be implemented using pulsewidth modulation of the semiconductor switches as developed in the previous recipe. Torque control is only one type of motor control. Speed control is often the goal. Robots often have position control as the goal. One could use torque control as an inner loop for either a speed controller or position controller.

How It Works

The motor controller is shown in Figure 9.1. This implements field-oriented control (FOC). FOC effectively turns the brushless three-phase motor into a commutated DC motor.

There are three electrical frames of reference in this problem. The first is the (a,b,c) frame which is the frame of the three-phase stator as in Figure 9.2. This is a time-varying frame. We next want to transform into a two-axis time-varying frame, the (α,β) frame, and then into a two-axis time-invariant frame, the (d, q) frame, which is also known as the direct-quadrature axes and is fixed to the permanent magnet. In our frames, each axis is a current. Since with a

Y-connected motor the sum of the currents is zero:

$$0 = i_a + i_b + i_c \tag{9.6}$$

we need only work with two currents, i_a and i_b.

The (d, q) to (α, β) transformation is known as the Forward Park transformation:

$$\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix} = \begin{bmatrix} \cos\theta_e & -\sin\theta_e \\ \sin\theta_e & \cos\theta_e \end{bmatrix} \begin{bmatrix} u_d \\ u_q \end{bmatrix} \tag{9.7}$$

This transforms from the stationary d, q frame to the rotating (α, β) frame. θ_e is in electrical axes and equals $\frac{1}{2}p\,\theta_M$ where p is the number of magnet poles. The Forward Clarke transformation for a Y-connected motor is

$$\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} u_a \\ u_b \end{bmatrix} \tag{9.8}$$

These two transformations are implemented in the functions `ClarkeTransformationMatrix` and `ParkTransformationMatrix`. They allow us to go from the time-varying (a,b,c) frame to the time-invariant, but rotating, (d,q) frame.

The equations for a general permanent magnet machine in the direct-quadrature frame are

$$u_q = r_s i_q + \omega_e(L_d i_d + \psi) + \frac{dL_q i_q}{dt} \tag{9.9}$$

$$u_d = r_s i_d - \omega_e L_q i_q + \frac{d(L_d i_d + \psi)}{dt} \tag{9.10}$$

where u are the voltages, i are the currents, r_s is the stator resistance, L_q and L_d are the d and q phase inductances, ω_e is the electrical angular rate, and ψ is the flux due to the permanent magnets. The electrical torque produced is

$$T_e = \frac{3}{2}p((L_d i_d + \psi)i_q - L_q i_q i_d) \tag{9.11}$$

where p is the number of pole pairs.

The torque equation is

$$T_e = T_L + b\omega_m + J\frac{d\omega_m}{dt} \tag{9.12}$$

where b is the mechanical damping coefficient, T_L is the external load torque, and J is the inertia, and the relationship between the mechanical and the electrical angular rate is

$$\omega_e = p\omega_m \tag{9.13}$$

The more pole pairs you have, the higher the electrical frequency. In a magnet surface mount machine with coils in slots, $L_d = L_q \equiv L$, and ψ and the inductances are not functions of time. The equations simplify to

$$u_q = r_s i_q + \omega_e L i_d + \omega_e \psi + L\frac{di_q}{dt} \tag{9.14}$$

$$u_d = r_s i_d - \omega_e L i_q + L\frac{di_d}{dt} \tag{9.15}$$

We control direct current i_d to zero. If i_d is zero, control is linear in i_q. The torque is now

$$T_e = \frac{3}{2} p \psi i_q \tag{9.16}$$

Thus, the torque is a function of the quadrature current i_q only. We can therefore control the electrical torque by controlling the quadrature current. The quadrature current is in turn controlled by the direct and quadrature phase voltages. The desired current i_q^s can now be computed from the torque set point T_e^s.

$$i_q^s = \frac{2}{3} T_e^s / (p\psi) \tag{9.17}$$

We will use a proportional-integral controller to compute the (d,q) voltages. The proportional part of the control drives errors to zero. However, if there is a steady disturbance, there will be an offset. The integral part can drive an error due to such a steady disturbance to zero. Without the integral term, a steady disturbance will result in a steady error. A proportional-integral controller is of the form

$$u = K \left(1 + \frac{1}{\tau} \int \right) y \tag{9.18}$$

where u is the control, y is the measurement, τ is the integrator time constant, and K is the forward (proportional) gain. Our control u will be the phase voltages, and our measurement y is the current error in the (d,q) frame.

$$u_{(d,q)} = -k_F \left(i_{err} + \frac{1}{\tau} \int i_{err} \right) \tag{9.19}$$

where

$$\left[\begin{array}{c} i_d \\ i_q \end{array} \right]_{err} = \left[\begin{array}{c} i_d \\ i_q \end{array} \right] - \left[\begin{array}{c} 0 \\ i_q^s \end{array} \right] \tag{9.20}$$

We now write a function, `TorqueControl`, that calculates the control voltages $u_{(\alpha,\beta)}$ given the current state x. The state vector is the same as Recipe 9.1, that is, current i in the (a,b,c) frame plus the angle states θ and ω. We use the Park and Clarke transformations to compute the current in the (d,q) frame. We can then implement the proportional-integral controller with Euler integration. The function uses its data structure as memory – the updated structure `d` is passed back as an output. `TorqueControl` is shown as follows. This function will return a default data structure if no inputs are passed into the function.

TorqueControl.m

```
%% TORQUECONTROL Compute torque control of an AC machine
% Determines the quadrature current needed to produce a torque and uses
    a
% proportional integral controller to control the motor. We control the
```

```
% direct current to zero since we want to use just the magnet flux to react
% with the quadrature current. We could control the direct current to
% another value to implement field-weakening control but this would result
% in a nonlinear control system.
%% Forms
%   d = TorqueControl
%   [u, d, iAB] = TorqueControl( torqueSet, x, d )
%
%% Inputs
%   torqueSet (1,1)    Set point torque
%   x         (5,1)    State [ia;ib;ic;omega;theta]
%   d         (.)        Control data structure
%                      .kF      (1,1) Forward gain
%                      .tauI    (1,1) Integral time constant
%                      .iDQInt  (2,1) Integral of current errors
%                      .dT      (1,1) Time step
%                      .psiM    (1,1) Magnetic flux
%                      .p       (1,1) Number of magnet poles
%
%%  Outputs
%   u         (2,1)    Control voltage [alpha;beta]
%   d         (.)      Control data structure
%   iAB       (2,1)    Steady state currents [alpha;beta]
%

function [u, d, iAB] = TorqueControl( torqueSet, x, d )

% Default data structure
if( nargin == 0 )
  u = struct('kF',0.003,'tauI',0.001, 'iDQInt',[0;0], 'dT', 0.01,...
             'psiM',0.0690,'p',2);
  if( nargout == 0 )
    disp('TorqueControl struct:');
  end
  return
end

% Clarke and Park transforms
thetaE = 0.5*d.p*x(5);
park   = ParkTransformationMatrix( thetaE );
iPark  = park';
clarke = ClarkeTransformationMatrix;
iDQ    = iPark*clarke*x(1:2);

% Set point to produce the desired torque [iD;iQ]
iDQSet = [0;(2/3)*torqueSet/(d.psiM*d.p)];

% Error
iDQErr = iDQ - iDQSet;

```

```
% Integral term
d.iDQInt = d.iDQInt + d.dT*iDQErr;

% Control
uDQ = -d.kF*(iDQErr + d.iDQInt/d.tauI);
u   = park*uDQ;

% Steady state currents
if( nargout > 2 )
  iAB = park*iDQSet;
end
```

9.3 Pulsewidth Modulation of the Switches

Problem

In the previous recipe, we calculate the control voltages to apply to the stator. Now we want to take those control voltages as an input and drive the switches via pulsewidth modulation.

Solution

We will use the Space Vector Modulation to go from a rotating two-dimensional (α,β) frame to the rotating three-dimensional (a,b,c) stator frame, which is more computationally efficient than modulating in (a,b,c) directly.

How It Works

We will use Space Vector Modulation to drive the switches for pulsewidth modulation.[2] This goes from (α,β) coordinates to switch states (a,b,c). Each node of each phase is either connected to ground or to $+u$. These values are shown in Figure 9.4. The six spokes in the diagram, as well as the origin, correspond to the eight discrete switch states.

Table 9.1 delineates each of these eight discrete switch states, the corresponding vector in the (α,β) coordinates, and the resulting voltages. Note that the O vectors are at the origin of the Space Vector Modulation, while the U vectors are at 60-degree increments. The states are indexed from 0 to 7 with 0 being all open states and 7 being all closed.

In order to produce the desired torque, we must use a combination of the vectors or switch states so that we achieve the desired voltage on average. We select the two vectors O or U bracketing the desired angle in the (α,β) plane; these are designated k and $k+1$ where k refers to the number of the vector in Table 9.1. We must then calculate the amount of time to spend in each switch state, for each pulsewidth period. The durations of these two segments, T_k and T_{k+1}, are found from this equation:

$$\begin{bmatrix} T_k \\ T_{k+1} \end{bmatrix} = \frac{\sqrt{3}}{2}\frac{T_s}{u_d}\begin{bmatrix} \sin\frac{k\pi}{3} & -\cos\frac{k\pi}{3} \\ -\sin\frac{(k-1)\pi}{3} & \cos\frac{(k-1)\pi}{3} \end{bmatrix}\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix} \tag{9.21}$$

[2] Analog Devices, "Implementing Space Vector Modulation with the ADMCF32X," ANF32X-17, January 2000.

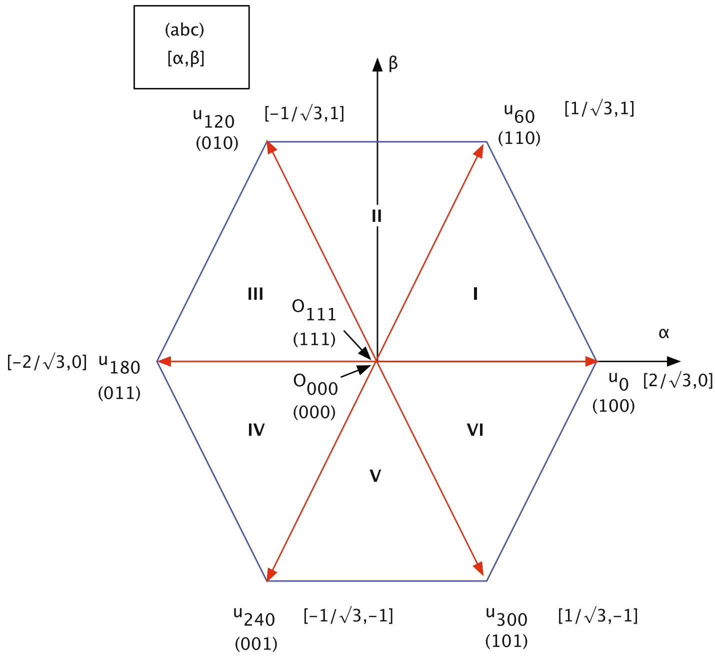

Figure 9.4: *Space Vector Modulation in (α,β) coordinates. We determine which sector (in Roman numerals) we are in and then pick the appropriate vectors to apply so that they on average attain the desired voltage. The numbers in brackets are the normalized $[\alpha, \beta]$ voltages.*

Table 9.1: *Space Vector Modulation. In the vector names, O means open and U means a voltage is applied, while the subscripts denote the angle in the α-β plane. The switch states are a, b, c as shown in Figure 9.3, where 1 means a switch is closed and 0 means it is open.*

k	abc	Vector	u_a/u	u_b/u	u_c/u	u_{ab}/u	u_{bc}/u	u_{ac}/u
0	000	O_{000}	0	0	0	0	0	0
1	110	U_{60}	2/3	1/3	-1/3	1	0	-1
2	010	U_{120}	1/3	1/3	-2/3	0	1	-1
3	011	U_{180}	-1/3	2/3	-1/3	-1	1	0
4	001	U_{240}	-2/3	1/3	1/3	-1	0	1
5	101	U_{300}	-1/3	-1/3	2/3	0	-1	1
6	100	U_{360}	1/3	-2/3	1/3	1	-1	0
7	111	O_{111}	0	0	0	0	0	0

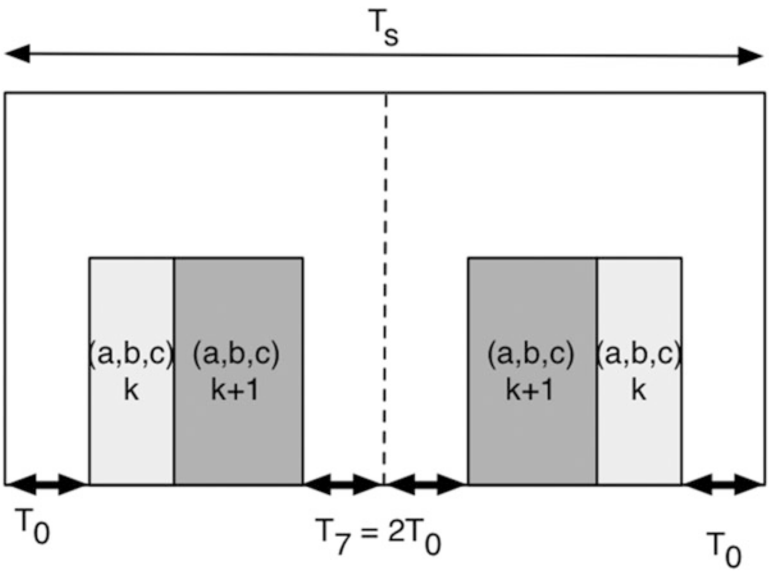

Figure 9.5: *Pulse period segments. Each pulse period T_s is divided into seven segments so that the two switching patterns k and $k+1$ are applied symmetrically.*

The corresponding (a,b,c) switch patterns are each used for the calculated time, averaging to the designated voltage.

The time spent in each pattern, T_k or T_{k+1}, is then split into two equal portions so that the total pulse pattern is symmetric. The zero time T_0, when no switching is required, is split evenly between the endpoints and the middle of the pulse T_s – so that the time in the middle pattern (O_{111}) is twice the time in each end pattern (O_{000}). This results in a total of seven segments depicted in Figure 9.5. The total middle time is designated T_7.

$$T_0 = \frac{1}{4}\left(T_s - (T_k + T_{k+1})\right) \tag{9.22}$$

The implementation of the pulse segments is slightly different for the even and odd sectors in Figure 9.4. Both are symmetric about the midpoint of the pulse as described, but we reverse the implementation of patterns k and $k+1$. This is shown for the resulting voltages u in the following equations. We use the first in even sectors and the second in odd sectors.

$$\begin{bmatrix} u_0 & u_k & u_{k+1} & u_7 & u_{k+1} & u_k & u_0 \end{bmatrix} \tag{9.23}$$

and

$$\begin{bmatrix} u_0 & u_{k+1} & u_k & u_7 & u_k & u_{k+1} & u_0 \end{bmatrix} \tag{9.24}$$

Using the different patterns for odd and even vectors minimizes the number of commutations per cycle.

We determine the sector from the angle Θ formed by the commanded voltages u_α and u_β:

$$\Theta = \operatorname{atan}\frac{u_\beta}{u_\alpha} \tag{9.25}$$

The pulsewidth modulation routine, SVPWM, does not actually perform an arctangent. Rather, it looks at the unit u_α and u_β vectors and determines first their quadrant and then their sector without any need for trigonometric operations.

The first section of SVPWM implements the timing for the pulses. Just as in the previous recipe for the controller, the function uses its data structure as memory – the updated structure is passed back as an output. This is an alternative to persistent variables.

SVPWM.m

```
function [s, d] = SVPWM( t, d )

% Default data structure
if( nargin < 1 && nargout == 1 )
  s = struct( 'dT',1e-6,'tLast',-0.0001,'tUpdate',0.001,'u',[0;0],...
              'uM',10,'tP',zeros(1,7),'sP',zeros(3,7));
  return;
end

% Run the demo
if( nargin < 1 )
  disp('Demo of SVPWM:');
  Demo;
  return;
end

% Update the pulsewidths at update time
if( t >= d.tLast + d.tUpdate || t == 0 )
  [d.sP, d.tP] = SVPW( d.u, d.tUpdate, d.uM );
  d.tLast      = t;
end

% Time since initialization of the pulse period
dT = t - d.tLast;
s  = zeros(3,1);

for k = 1:7
  if( dT < d.tP(k) )
    s = d.sP(:,k);
    break;
  end
end
```

The pulsewidth vectors are computed in the subfunction SVPW. We first compute the quadrant and then the sector without using any trigonometric functions. This is done using simple if/else statements and a switch statement. Note that the modulation index k is simply designated k and $k+1$ is designated kP1. We then compute the times for the two space vectors that bound the sector. We then assemble the seven subperiods.

```
function [sP, tP] = SVPW( u, tS, uD )

% Make u easier to interpret
alpha = 1;
beta  = 2;

% Determine the quadrant
if( u(alpha) >= 0 )
  if( u(beta) > 0 )
    q = 1;
  else
    q = 4;
  end
else
  if( u(beta) > 0 )
    q = 2;
  else
    q = 3;
  end
end

sqr3 = sqrt(3);

% Find the sector. k1 and k2 define the edge vectors
switch q
  case 1 % [+,+]
    if( u(beta) < sqr3*u(alpha) )
      k       = 1;
      kP1     = 2;
      oddS    = 1;
    else
      k       = 2;
      kP1     = 3;
      oddS    = 0;
    end
  case 2  % [-,+]
    if( u(beta) < -sqr3*u(alpha) )
      k       = 3;
      kP1     = 4;
      oddS    = 1;
   else
      k       = 2;
      kP1     = 3;
      oddS    = 0;
    end
  case 3 % [-,-]
    if( u(beta) < sqr3*u(alpha) )
      k       = 5;
      kP1     = 6;
      oddS    = 1;
   else
```

```
      k       = 4;
      kP1     = 5;
      oddS    = 0;
    end
  case 4 % [+,-]
    if( u(beta) < -sqr3*u(alpha) )
      k       = 5;
      kP1     = 6;
      oddS    = 1;
    else
      k       = 6;
      kP1     = 1;
      oddS    = 0;
    end
end

% Switching sequence
piO3     = pi/3;
kPiO3    = k*pi/3;
kM1PiO3 = kPiO3-piO3;

% Space vector pulsewidths
t  = 0.5*sqr3*(tS/uD)*[ sin(kPiO3)      -cos(kPiO3);...
                       -sin(kM1PiO3)   cos(kM1PiO3)]*u;

% Total zero vector time
t0 = tS - sum(t);

t  = t/2;

% Different order for odd and even sectors
if( oddS )
  sS  = [0 k kP1 7 kP1 k 0];
  tPW = [t0/4 t(1) t(2) t0/2 t(2) t(1) t0/4];
else
  sS  = [0 kP1 k 7 k kP1 0];
  tPW = [t0/4 t(2) t(1) t0/2 t(1) t(2) t0/4];
end
tP   = [tPW(1) zeros(1,6)];

for k = 2:7
  tP(k)  = tP(k-1) + tPW(k);
end

% The switches corresponding to each voltage vector
% From 0 to 7
%                a b c
s            = [ 0 0 0;...
                 1 0 0;...
                 1 1 0;...
                 0 1 0;...
                 0 1 1;...
```

```
                 0 0 1;...
                 1 0 1;
                 1 1 1]';

 sP = zeros(3,7);
 for k = 1:7
   sP(:,k) = s(:,sS(k)+1);
 end
```

The built-in demo is fairly complex so it is in a separate subfunction. We simply specify an example input u using trigonometric functions.

```
function Demo
%%% SVPWM>Demo Function demo
% Calls SVPWM with a sinusoidal input u.
% This demo will run through an array of times and create a plot of the
% resulting voltages.

d     = SVPWM;
tEnd  = 0.003;
n     = tEnd/d.dT;
a     = linspace(0,pi/4,n);
tP3   = 2*pi/3;
uABC  = 0.5*[cos(a);cos(a-tP3);cos(a+tP3)];
uAB   = ClarkeTransformationMatrix*uABC(1:2,:); % a-b to alpha-beta
tSamp = 0;
t     = 0;
tPP   = 1;
x     = zeros(4,n);
for k = 1:n
  if( t >= tSamp )
    tSamp = tSamp + d.tUpdate;
    tPP   = ~tPP;
  end
  d.u    = uAB(:,k);
  [s, d] = SVPWM( t, d );
  t      = t + d.dT;
  x(:,k) = [SwitchToVoltage(s,d.uM);tPP];
end
```

Figure 9.6 shows the state vector pulsewidth modulation from the built-in demo. There are three pulses in the plot, each 0.001 seconds long. Each pulse period has seven subperiods.

The function `SwitchToVoltage` converts switch states to voltages. It assumes instantaneous switching and no switch dynamics.

SwitchToVoltage.m

```

% Switch states [a;b;c]
sA     = [1  1  0  0  0  1;...
```

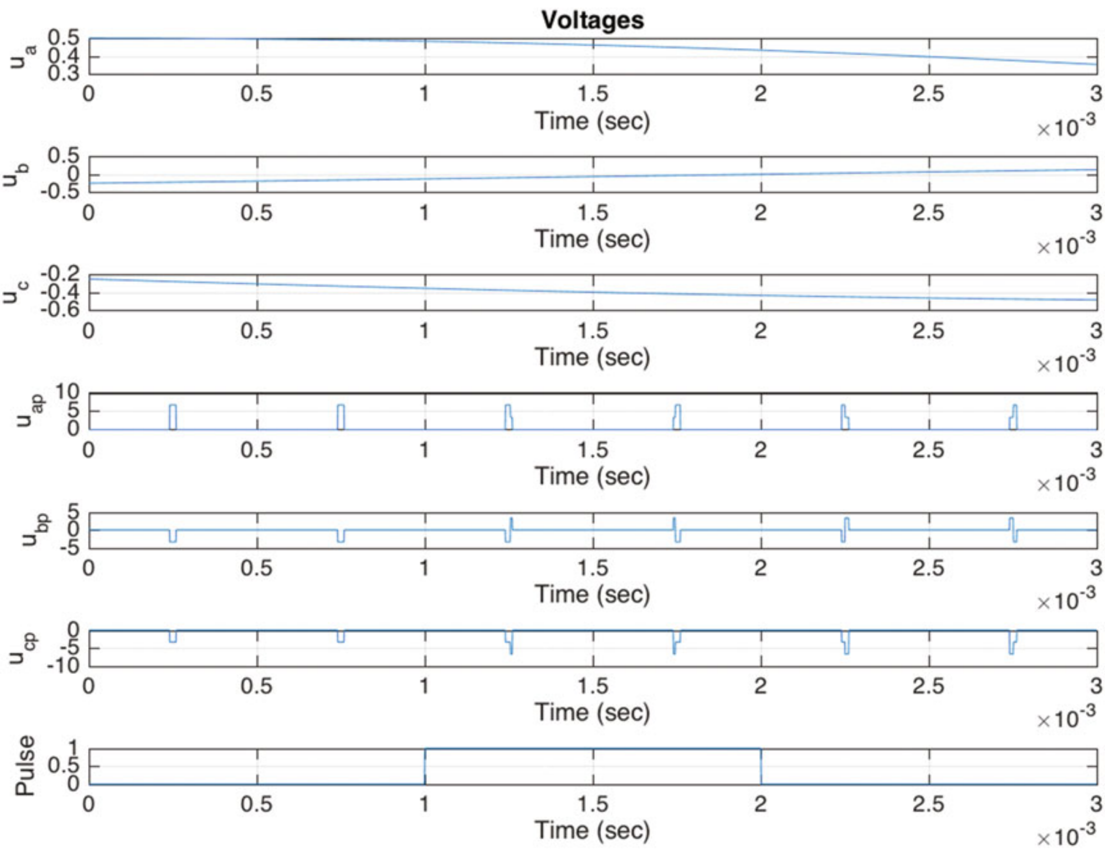

Figure 9.6: *The desired voltage vector and the Space Vector Modulation pulses and pulsewidth. The bottom plot shows the pulse periods. Note that the pulse sequences are symmetric within each pulse period.*

```
             0   1   1   1   0   0;...
             0   0   0   1   1   1];

% Array of voltages
uA    = [ 2   1  -1  -2  -1   1;...
           -1   1   2   1  -1  -2;...
           -1  -2  -1   1   2   1];

% Find the correct switch state
u     = [0;0;0];
for k = 1:6
  if( sum(sA(:,k) - s) == 0 )
    u = uA(:,k)*uDC/3;
    break;
  end
end
```

9.4 Simulating the Controlled Motor

Problem

We want to simulate the motor with torque control using Space Vector Modulation.

Solution

Write a script to simulate the motor with the controller. We include options for closed loop control and balanced three-phase voltage inputs.

How It Works

The header for the script, `PMMachineDemo`, is shown in the following listing. The control flags `bypassPWM` and `torqueControlOn` are described as well as the two periods implemented, one for the simulation and a longer period for the control.

PMMachineDemo.m

```
%% Simulation of a permanent magnet AC motor
% Simulates a permanent magnet AC motor with torque control. The simulation has
% two options. The first is torqueControlOn. This turns torque control on and
% off. If it is off the phase voltages are a balanced three phase voltage set.
%
% bypassPWM allows you to feed the phase voltages directly to the motor
% bypassing the pulsewidth modulation switching function. This is useful for
% debugging your control system and other testing.
%
% There are two time constants for this simulation. One is the control period
% and the second is the simulation period. The latter is much shorter because it
% needs to simulate the pulsewidth modulation.
%
% For control testing the load torque and setpoint torque should be the same.
```

The body of the script follows. Three different data structures are initialized from their corresponding functions as described in the previous recipes, that is, from `SVPWM`, `TorqueControl`, and `RHSPMMachine`. Note that we are only simulating the motor for a small fraction of a second, 0.05 seconds, and the time step is just 1e-6 seconds. The controller time step is set to 100 times the simulation time step.

```
%% Initialize all data structures
dS        = SVPWM;
dC        = TorqueControl;
d         = RHSPMMachine;
```

```
dC.psiM = d.psiM;
dC.p    = d.p;
d.tL    = 1.0; % Load torque (Nm)

%% User inputs
tEnd               = 0.05;      % sec
torqueControlOn = false;
bypassPWM          = false;
torqueSet          = 1.0;       % Set point (Nm)
dC.dT              = 100*dS.dT; % 100x larger than simulation dT
dS.uM              = 1.0;       % DC Voltage at the input to the switches
magUABC            = 0.1;       % Voltage for the balanced 3 phase voltages

if (torqueControlOn && bypassPWM)
  error('The control requires PWM to be on.');
end

%% Run the simulation
nSim = ceil(tEnd/dS.dT);
xP   = zeros(10,nSim);
x    = zeros(5,1);

% We require two timers as the control period is larger than the simulation
% period
t    = 0.0; % simulation timer
tC   = 0.0; % control timer

for k = 1:nSim
  % Electrical degrees
  thetaE = x(5);
  park   = ParkTransformationMatrix( thetaE );
  clarke = ClarkeTransformationMatrix;

  % Compute the voltage control
  if( torqueControlOn && t >= tC )
    tC          = tC + dC.dT;
    [dS.u, dC] = TorqueControl( torqueSet, x, dC );
  elseif( ~torqueControlOn )
    tP3  = 2*pi/3;
    uABC = magUABC*dS.uM*[cos(thetaE);cos(thetaE-tP3);cos(thetaE+tP3)];
    if( bypassPWM )
      d.u = uABC;
    elseif( t >= tC )
      tC   = tC + dC.dT;
      dS.u = park*clarke*uABC(1:2,:);
    end
  end

  % Space Vector Pulsewidth Modulation
  if( ~bypassPWM )
```

```
      dS.u    = park'*dS.u;
      [s,dS] = SVPWM( t, dS );
      d.u     = SwitchToVoltage(s,dS.uM);
    end

    % Get the torque output for plotting
    [~,tE]   = RHSPMMachine( 0, x, d );
    xP(:,k)  = [x;d.u;torqueSet;tE];

    % Propagate one simulation step
    x = RungeKutta( @RHSPMMachine, 0, x, dS.dT, d );
    t = t + dS.dT;
  end

  %% Generate the time history plots
  [t, tL]    = TimeLabel( (0:(nSim-1))*dS.dT );

  figure('name','3 Phase Currents');
  plot(t, xP(1:3,:));
  grid on;
  ylabel('Currents');
  xlabel(tL);
  legend('i_a','i_b','i_c')

  PlotSet( t, xP([4 10],:), 'x label', tL, 'y label', {'\omega_e' 'T_e (
      Nm)'}, ...
    'plot title','Electrical', 'figure title','Electrical');

  thisTitle = 'Phase Voltages';
  if ~bypassPWM
    thisTitle = [thisTitle ' - PWM'];
  end

  PlotSet( t, xP(6:8,:), 'x label', tL, 'y label', {'u_a' 'u_b' 'u_c'},
      ...
    'plot title',thisTitle, 'figure title',thisTitle);

  thisTitle = 'Torque/Speed';
  if ~bypassPWM
    thisTitle = [thisTitle ' - PWM'];
  end
```

We turn off torque control to test the motor simulation with the results shown in Figure 9.7. The two plots show the torque speed curves. The first is with direct three-phase excitation, that is, bypassing the pulsewidth modulation, by setting `bypassPWM` to `false`. Directly controlling the phase voltages this way, while creating the smoothest response, would require linear amplifiers which are less efficient than switches. This would make the motor much less efficient overall and would generate unwanted heat. The second plot is with Space Vector Pulsewidth Modulation. The plots are nearly identical, indicating that the pulsewidth modulation is working.

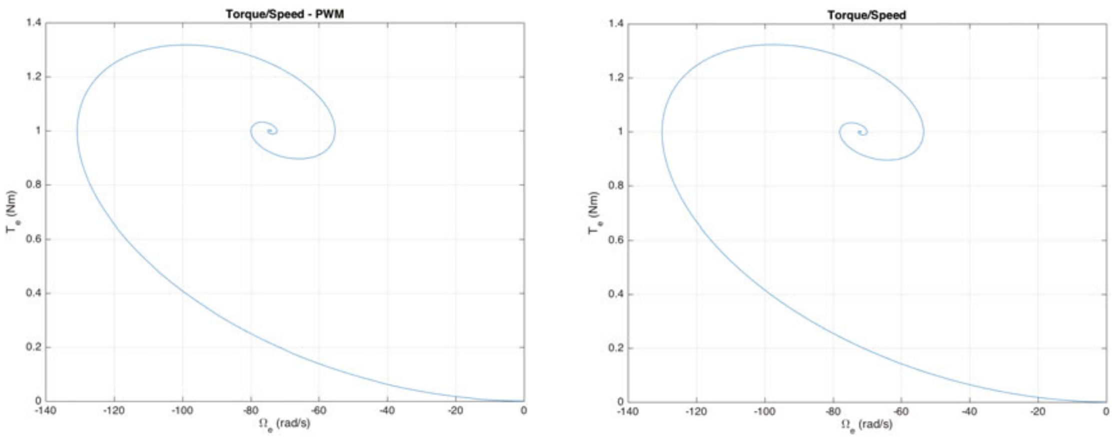

Figure 9.7: *Torque speed curves for a balanced three-phase voltage excitation and a load torque of 1.0 Nm. The left figure shows the curve for the direct three-phase input, and the right shows the curve for the Space Vector Pulsewidth Modulation input. They are nearly identical.*

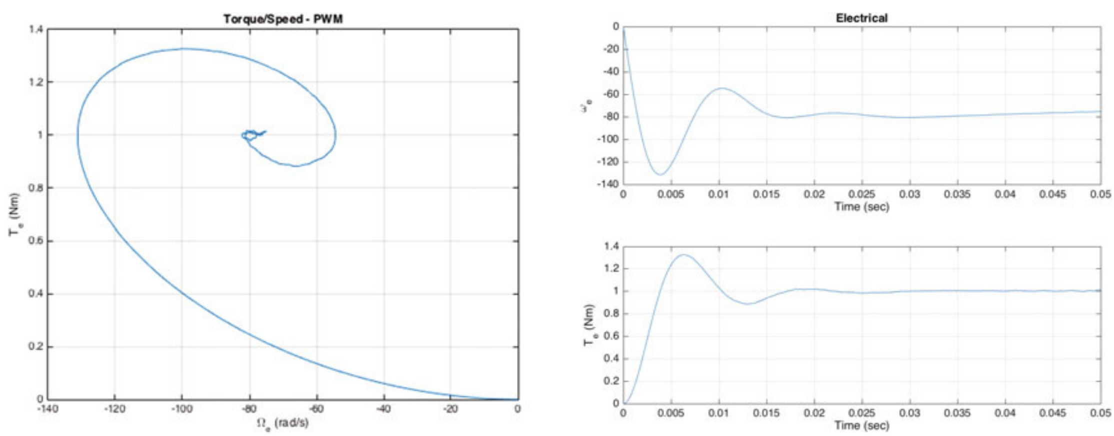

Figure 9.8: *PI torque control of the motor.*

We now turn on torque control, via the `torqueControlOn` flag, and get the results shown in Figure 9.8. The overshoot is typical for torque control. Note that the load torque is set equal to the torque set point of 1 Nm. There is limit cycling near the end point.

The pulsewidths and resulting coil currents are shown in Figure 9.9. A zoomed view of the end of the pulsewidth plot with shading added to alternate pulsewidths is in Figure 9.10. This makes it easier to see the segments of the pulsewidths and verify that they are symmetric.

The code which adds the shading uses `fill` with transparency via the `alpha` parameter. In this case, we hard-code the function to show the last five pulsewidths, but this could be generalized to a time window, or to shade the entire plot. We did take the time to add an input for the pulsewidth length, so that this could be changed in the main script and the function

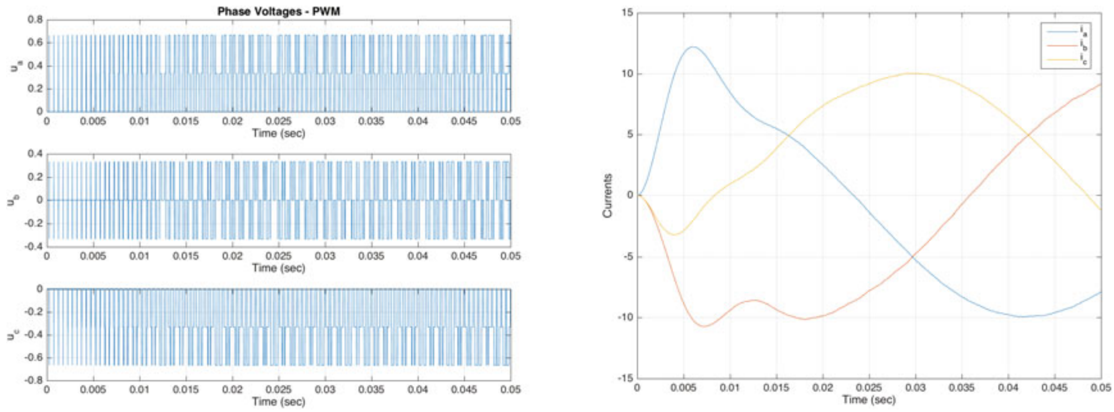

Figure 9.9: *Voltage pulsewidths and resulting currents for PI torque control.*

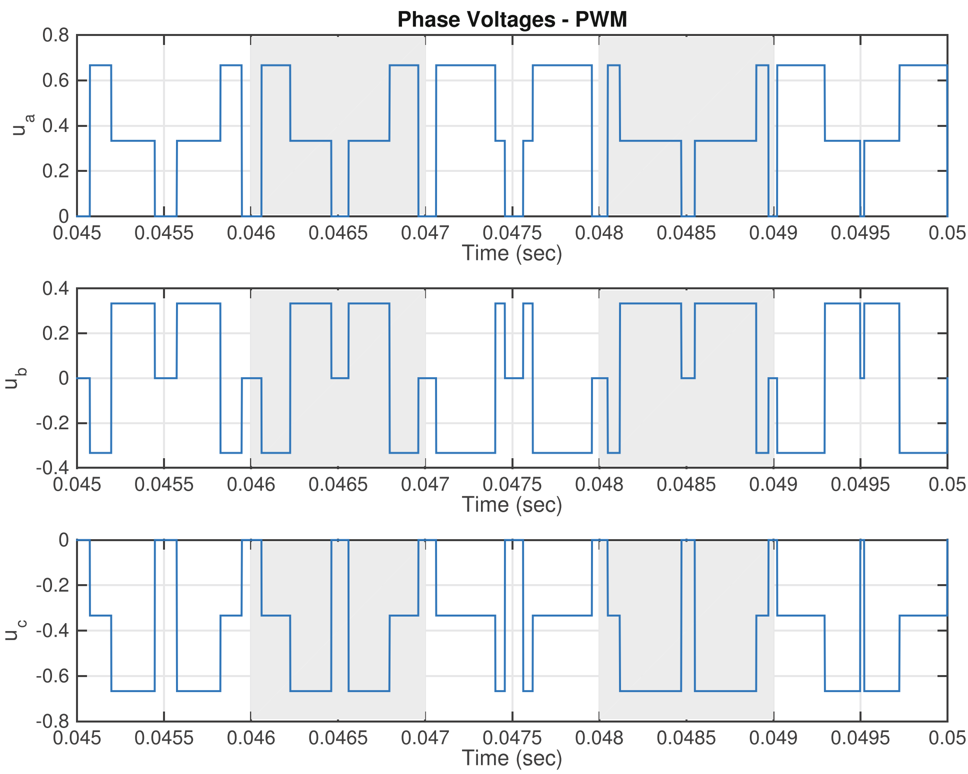

Figure 9.10: *Pulsewidths with shading.*

would still work. Note that we reorder the axes children as the last step, to keep the shading from obscuring the plot lines.

AddFillToPWM.m

```
function AddFillToPWM( dT )

if nargin == 0
  dT = 0.001;
end

hAxes = get(gcf,'children');
nAxes = length(hAxes);

for j = 1:nAxes
  if strcmp(hAxes(j).type,'axes')
    axes(hAxes(j));
    AddFillToAxes;
  end
end

  function AddFillToAxes

  hold on;
  y = axis;
  xMin = y(2) - 5*dT;
  xMax = y(2);
  axis([xMin xMax y(3:4)])
  x0    = xMin;
  yMin  = y(3) + 0.01*(y(4)-y(3));
  yMax  = y(4) - 0.01*(y(4)-y(3));
  for k = [2 4]
    xMinK = x0 + (k-1)*dT;
    xMaxK = x0 + k*dT;

    fill([xMinK xMaxK xMaxK xMinK],...
         [yMin,yMin,yMax,yMax],...
         [0.8 0.8 0.8],'edgecolor','none','facealpha',0.5);

  end
  babes = get(gca,'children');
  set(gca,'children',[babes(end); babes(1:end-1)])
  hold off;

  end

end
```

Summary

This chapter has demonstrated how to write the dynamics and implement a field-oriented control law for a three-phase motor. We use a proportional-integral controller with Space Vector Pulsewidth Modulation to drive the six switches. This produces a low-cost controller for a motor. Table 9.2 lists the code developed in the chapter.

Table 9.2: *Chapter Code Listing*

File	Description
`AddFillToPWM`	Add shading to the motor pulsewidth plot
`ClarkeTransformationMatrix`	Clarke transformation matrix
`ParkTransformationMatrix`	Park transformation matrix
`PMMachineDemo`	Permanent magnet motor demonstration
`RHSPMMachine`	Right-hand side of a permanent magnet brushless three-phase electrical machine
`SVPWM`	Implements Space Vector Pulsewidth Modulation
`SwitchToVoltage`	Converts switch states to voltages
`TorqueControl`	Proportional-integral torque controller