CHAPTER 7

■ ■ ■

Classifying a Pirouette

7.1 Introduction

A pirouette is a familiar step in ballet. There are many types of pirouettes. We will focus on an en dehors (outside) pirouette from fourth position. The dancer pliés (does a deep knee bend) then straightens her legs producing both an upward force to get on the tip of her pointe shoe and a torque to turn about her axis of revolution.

In this chapter, we will classify pirouettes. Four dancers will each do ten double pirouettes, and we will use them to train the deep learning network. The network can then be used to classify pirouettes.

This chapter will involve real-time data acquisition and deep learning. We will spend a considerable amount of time in this chapter creating software to interface with the hardware. While it is not deep learning, it is important to know how to get data from sensors for use in deep learning work. We give code snippets in this chapter. Only a few can be cut and pasted into the MATLAB command window. You'll need to run the software in the downloadable library. Also remember, you will need the Instrument Control Toolbox for this project.

Our subject dancers showing a pirouette are shown in Figure 7.1. We have three female dancers and one male dancer. Two of the women are wearing pointe shoes. The measurements will be accelerations, angular rates, and orientation. There really isn't any limit to the movements the dancers could do. We asked them all to do double pirouettes starting from fourth position and returning to fourth position. Fourth position is with one foot behind the other and separated by a quarter meter or so. This is in contrast to fifth position where the feet are right against each other. Each is shown at the beginning, middle, and end of the turn. All have slightly different positions, though all are doing very good pirouettes. There is no one ‘‘right” pirouette. If you were to watch the turns, you would not be able to see that they are that different. The goal is to develop a neural network that can classify their pirouettes.

This kind of tool would be useful in any physical activity. An athlete could train a neural network to learn any important movement. For example, a baseball pitcher's pitch could be learned. The trained network could be used to compare the same movement at any other time to see if it has changed. A more sophisticated version, possibly including vision, might suggest

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_7

Figure 7.1: Dancers doing pirouettes. The stages are from left to right.

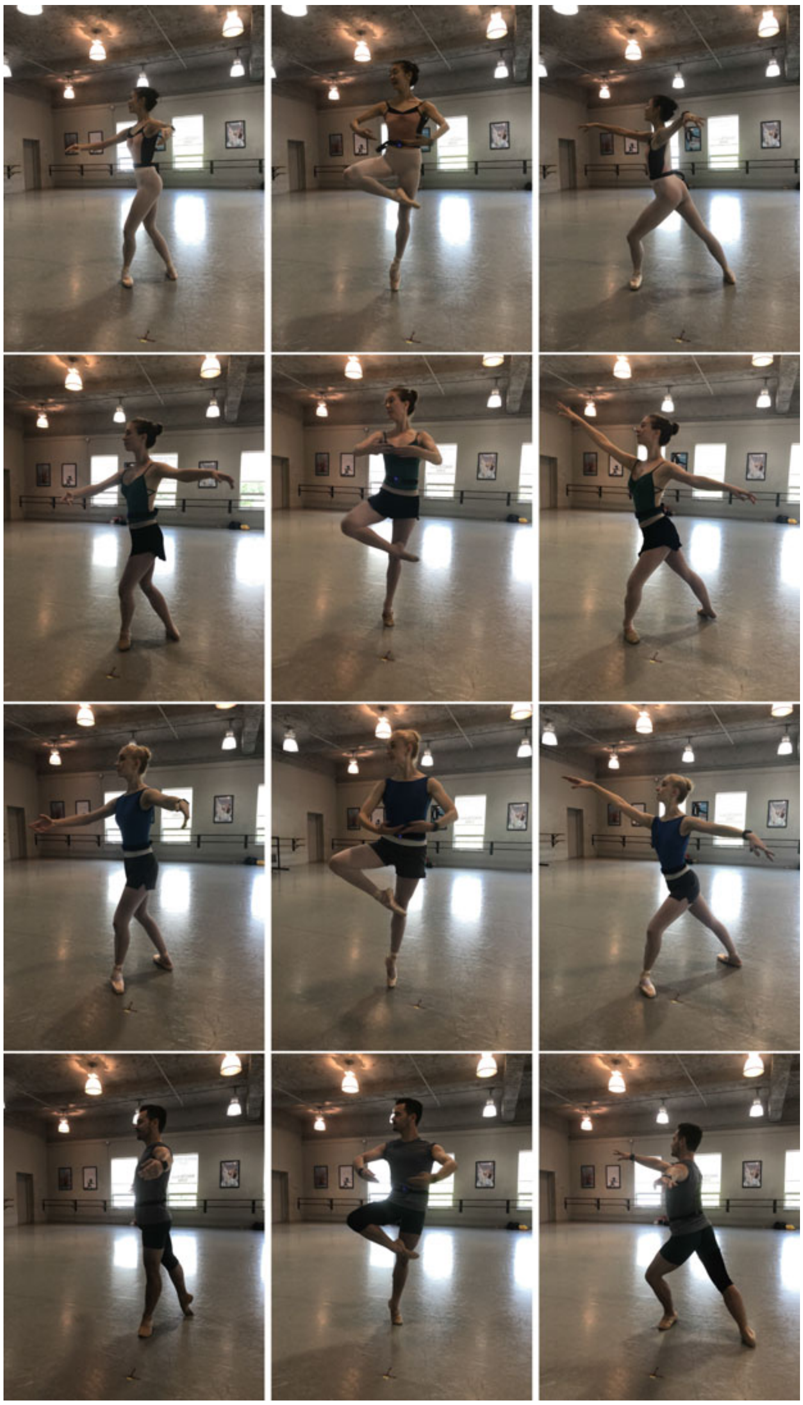
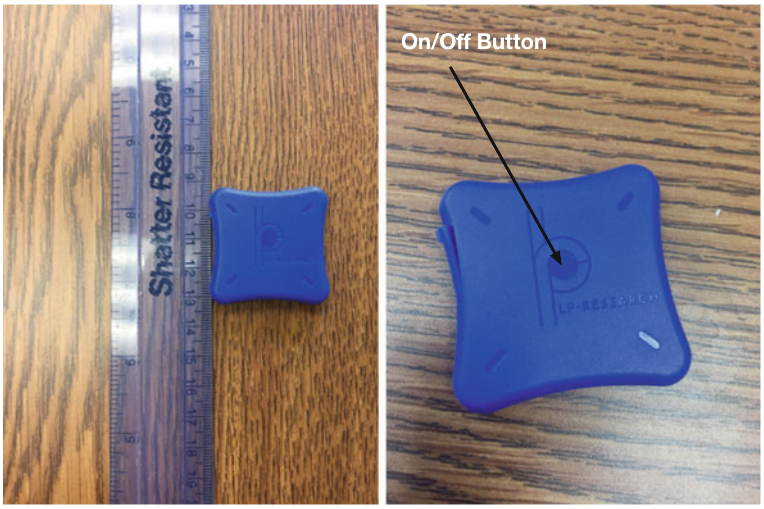


Table 7.1: LPMS-B2: 9-Axis Inertial Measurement Unit (IMU).

Parameter	**Description**
Bluetooth	2.1 + EDR / Low Energy (LE) 4.1
Communication distance	< 20 m
Orientation range	Roll: $\pm180°$; Pitch: $\pm90°$; Yaw: $\pm180°$
Resolution	$< 0.01°$
Accuracy	$< 0.5°$(static), $< 2°$ RMS (dynamic)
Accelerometer	3 axes, $\pm2/\pm4/\pm8/\pm16$ g, 16 bits
Gyroscope 3 axes	$\pm125/\pm245/\pm500/\pm1000/\pm2000°$/s, 16 bits
Data output format	2 Raw data / Euler angle / Quaternion
Data transmission rate up to	2400Hz

Figure 7.2: LPMS-B2: 9-Axis Inertial Measurement Unit (IMU). The on/off button is highlighted on the right.



how to fix problems or identify what has changed. This would be particularly valuable for rehabilitation.

7.1.1 Inertial Measurement Unit

Our sensing means will be the LPMS-B2 IMU with its parameters shown in Table 7.1 that has Bluetooth. The range is sufficient to work in a ballet studio.

The IMU has many other outputs that we will not use. A close up of the IMU is shown in Figure 7.2.

We will first work out the details of the data acquisition. We will then build a deep learning algorithm to train the system and later to take data and classify the pirouette as being a pirouette done by a particular dancer. We'll build up the data acquisition by first writing the MATLAB code to acquire the data. We will then create functions to display the data. We will then integrate it all into a GUI. Finally, we will create the deep learning classification system.

7.1.2 Physics

A pirouette is a complex multiflexible body problem. The pirouette is initiated by the dancer doing a pliè and then using his or her muscles to generate a torque about the spin axis and forces to get onto her pointe shoe and over her center of mass. Her muscles quickly stop the translational motion so that she can focus on balancing as she is turning. The equation of rotational motion is known as Euler's equation and is

$$T = I\dot{\omega} + \omega^{\times} I\omega \tag{7.1}$$

where ω is the angular rate and I is our inertia. T is our external torque. The external torque is due to a push off the floor and gravity. This is vector equation. The vectors are T and ω. I is a 3×3 matrix.

$$T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \tag{7.2}$$

$$\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{7.3}$$

Each component is a value about a particular axis. For example, T_x is the torque about the x-axis attached to the dancer. Figure 7.3 shows the system. We will only be concerned with rotation assuming all translational motion is damped. If the dancer's center of mass is not above the box of her pointe shoe, she will experience an overturning torque.

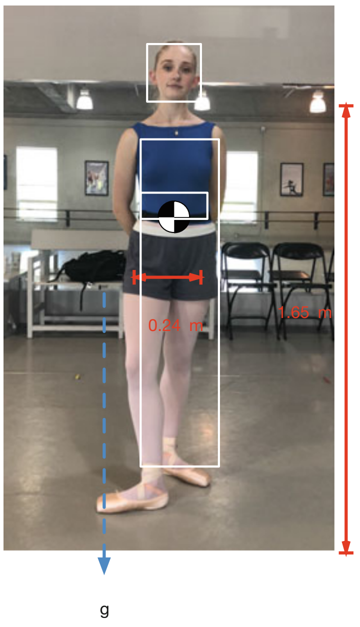
The dynamical model is three first-order couple differential equations. Angular rate ω is the state, that is, the quantity being differentiated. The equation says that the external torque (due to pushing off the floor or due to pointe shoe drag) is equal to the angular acceleration plus the Euler coupling term. This equation assumes that the body is rigid. For a dancer, it means she is rotating and no part is moving with respect to any other part. Now in a proper pirouette, this is never true if you are spotting! But let's suppose you are one of those dancers who don't spot. Let's forget about the angular rate coupling term, which only matters if the angular rate is large. Let's just look at the first two terms which are $T = I\dot{\omega}$. Expanded

$$\begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} \tag{7.4}$$

Let's look at the equation for T_z. We just multiply the first row of the inertia matrix times the angular rate vector.

$$T_z = I_{xz}\dot{\omega}_x + I_{yz}\dot{\omega}_y + I_{zz}\dot{\omega}_z \tag{7.5}$$

Figure 7.3: The center of mass of the dancer. External forces act at the center of mass. Rotations are about the center of mass.



This means a torque around the z-axis influences the angular rates about all 3 axes.

We can write out the required torque.

$$\begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} I_{xz} \\ I_{yz} \\ I_{zz} \end{bmatrix} \dot{\omega}_z \tag{7.6}$$

This is the perfect pirouette push off because it only creates rotation about the vertical axis, which is what we want in a pirouette. To this you need to add the forces needed to get on pointe with your center of mass over your pointe shoe tip.

While turning, the only significant external torque is due to friction between the pointe shoe tip and the floor. Friction resists both turning motion and translational motion. You don't want a slight side force, perhaps due to a less than great partner, to cause you to slide.

Our IMU measures angular rates and linear accelerations. Angular rates are the quantities in Euler equations. However, since the IMU is not at the dancer's center of mass, it will also measure angular accelerations along with the acceleration of the center of mass. We locate it at the dancer's waist so it is not too far from the spin axis but it still sees a component.

$$a_\omega = r_{\text{IMU}}^{\times} \dot{\omega} \tag{7.7}$$

where r_{IMU} is the vector from the dancer's center of mass to the IMU.

For a dancer doing a pirouette, Euler's equation is not sufficient. A dancer can transfer momentum internally to stop a pirouette and needs a little jump to get on demi-pointe or pointe. To model this, we add additional terms.

$$\begin{aligned}
T &= I\dot{\omega} + \omega^{\times}\left[I\omega + uI_i(\Omega_i + \omega_z) + uI_h(\Omega_h + \omega_z)\right] + u\left(T_i + T_h\right) && (7.8)\\
T_i &= I_i\left(\dot{\Omega}_i + \dot{\omega}_z\right) && (7.9)\\
T_h &= I_h\left(\dot{\Omega}_h + \dot{\omega}_z\right) && (7.10)\\
F &= m\ddot{z} && (7.11)
\end{aligned}$$

where m is the mass, F is the vertical force, and z is the vertical direction. I_i is the internal inertia for control, and I_h is the head inertia. I includes both of these already. That is, I is the total body inertia that includes the internal ''wheel,'' body, and head. T_i is the internal torque. T_h is the head torque (for spotting). The internal torques, T_i, and T_h are between the body and the internal ''wheel'' or head. For example, T_h causes the head to move one way and the body the other. If you are standing, the torque you produce from your feet against the floor prevents your body from rotating. T is the external torque due to friction and the initial push off by the feet. The unit vector is

$$u = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad (7.12)$$

There are six equations in total. The first is a vector equation with three components, the second two are scalar equations. The vector equation is three equations, and each scalar equation is just one equation. We can use these to create a simulation of a dancer. The second component models all z-axis internal rotation, including spotting.

7.2 Data Acquisition

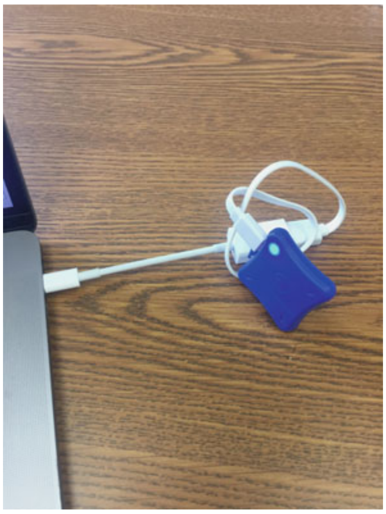
7.2.1 Problem

We want to get data from the Bluetooth IMU.

7.2.2 Solution

We will use the MATLAB `bluetooth` function. We'll create a function to read data from the IMU.

Figure 7.4: The IMU is connected to the bluetooth device from a MacBook Pro via USB C and a ubiquitous Mac dongle. This is only for charging purposes. Once it is charged, you can use without the dongle.



7.2.3 How It Works

We will write an interface to the bluetooth device. First make sure the IMU is charged. Connect it to your computer as shown in Figure 7.4. Push the button on the back. This turns it on and off. The status is indicated by the LED. The IMU comes with support software from the vendor, but you will not need any of their software as MATLAB does all the hard work for you.

Let's try commanding the IMU. Type `btInfo = instrhwinfo('Bluetooth')` and you should get the following:

```
>> btInfo = instrhwinfo('Bluetooth')

btInfo =

  HardwareInfo with properties:

        RemoteNames: {'LPMSB2-4B31D6'}
          RemoteIDs: {'btspp://00043E4B31D6'}
    BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
     JarFileVersion: 'Version 4.0'

Access to your hardware may be provided by a support package. Go to the
    Support Package Installer to learn more.
```

This shows that your IMU is discoverable. There is no support package available from the MathWorks. Now type `b = Bluetooth(btInfo.RemoteIDs1,1)` (this can be slow). The number is the channel. The `Bluetooth` function requires the Instrument Control Toolbox for MATLAB.

```
>> b = Bluetooth(btInfo.RemoteIDs{1},1)

   Bluetooth Object : Bluetooth-btspp://00043E4B31D6:1

   Communication Settings
      RemoteName:         LPMSB2-4B31D6
      RemoteID:           btspp://00043e4b31d6
      Channel:            1
      Terminator:         'LF'

   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
      BytesAvailable:     0
      ValuesReceived:     0
      ValuesSent:         0
```

Note that the Communication State Status shows closed. We need to open the device by typing `fopen(b)`. If you don't have this device, just type

```
>> btInfo = instrhwinfo('Bluetooth')
btInfo =
  HardwareInfo with properties:

        RemoteNames: []
          RemoteIDs: []
    BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
     JarFileVersion: 'Version 4.0'
Access to your hardware may be provided by a support package. Go to the
    Support Package Installer to learn more.
```

This says it cannot recognize remote names or ids. You may need a support package for your device in this case.

Click connect and the device will open. Now type `a= fscanf(b)` and you will get a bunch of unprintable characters. We now have to write code to command the device. We will leave the device in streaming mode. The data unit format is shown in Table 7.2. Each packet is really 91 bytes long even though the table only shows 67 bytes. The 67 bytes are all the useful data.

We read the binary and put it into a data structure using `DataFromIMU`. `typecast` converts from bytes to float.

Table 7.2: Reply data.

Byte	**Content (hex)**	**Meaning**
0	3A	Packet Start
1	01	OpenMAT ID LSB (ID=1)
2	00	OpenMAT MSB
3	09	Command No. LSB (9d = GET_SENSOR_DATA)
4	00	Command No. MSB
5	00	Data Length LSB
6	00	Data Length MSB
7--10	xxxxxxxx	Timestamp
11--14	xxxxxxxx	Gyroscope data x-axis
15--18	xxxxxxxx	Gyroscope data y-axis
19--22	xxxxxxxx	Gyroscope data z-axis
23--26	xxxxxxxx	Accelerometer x-axis
27--30	xxxxxxxx	Accelerometer y-axis
31--34	xxxxxxxx	Accelerometer z-axis
35--38	xxxxxxxx	Magnetometer x-axis
39--42	xxxxxxxx	Magnetometer y-axis
43--46	xxxxxxxx	Magnetometer z-axis
47--50	xxxxxxxx	Orientation quaternion q0
51--54	xxxxxxxx	Orientation quaternion q1
55--58	xxxxxxxx	Orientation quaternion q2
59--62	xxxxxxxx	Orientation quaternion q3
63	xx	Check sum LSB
64	xx	Check sum MSB
65	0D	Message end byte 1
66	0A	Message end byte 2

DataFromIMU.m

```
function d = DataFromIMU( a )

d.packetStart    = dec2hex(a(1));
d.openMATIDLSB   = dec2hex(a(2));
d.openMATIDMSB   = dec2hex(a(3));
d.cmdNoLSB       = dec2hex(a(4));
d.cmdNoMSB       = dec2hex(a(5));
d.dataLenLSB     = dec2hex(a(6));
d.dataLenMSB     = dec2hex(a(7));
d.timeStamp      = BytesToFloat( a(8:11) );
d.gyro           = [ BytesToFloat( a(12:15) );...
                     BytesToFloat( a(16:19) );...
                     BytesToFloat( a(20:23) )];
d.accel          = [ BytesToFloat( a(24:27) );...
                     BytesToFloat( a(28:31) );...
                     BytesToFloat( a(32:35) )];
d.quat           = [ BytesToFloat( a(48:51) );...
                     BytesToFloat( a(52:55) );...
```

```
                       BytesToFloat( a(56:59) );...
                       BytesToFloat( a(60:63) )];
d.msgEnd1        = dec2hex(a(66));
d.msgEnd2        = dec2hex(a(67));

%% DataFromIMU>BytesToFloat
function r = BytesToFloat( x )

r = typecast(uint8(x),'single');
```

We've wrapped all of this into the script `BluetoothTest.m`. We print out a few samples of the data to make sure our bytes are aligned correctly.

BluetoothTest.m

```
%% Script to read binary from the IMU

% Find available Bluetooth devices
btInfo = instrhwinfo('Bluetooth')

 % Display the information about the first device discovered
btInfo.RemoteNames(1)
btInfo.RemoteIDs(1)

% Construct a Bluetooth Channel object to the first Bluetooth device
b = Bluetooth(btInfo.RemoteIDs{1}, 1);

% Connect the Bluetooth Channel object to the specified remote device
fopen(b);

% Get a data structure
tic
t = 0;
for k = 1:100
  a   = fread(b,91);
  d   = DataFromIMU( a );
  fprintf('%12.2f [%8.1e %8.1e %8.1e] [%8.1e %8.1e %8.1e] [%8.1f %8.1f
      %8.1f %8.1f]\n',t,d.gyro,d.accel,d.quat);
  t = t + toc;
  tic
end
```

```
When we run the script we get the following output.

>> BluetoothTest

btInfo =

  HardwareInfo with properties:

        RemoteNames: {'LPMSB2-4B31D6'}
```

```
           RemoteIDs: {'btspp://00043E4B31D6'}
     BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
      JarFileVersion: 'Version 4.0'

Access to your hardware may be provided by a support package. Go to the
    Support Package Installer to learn more.

ans =

  1x1 cell array

    {'LPMSB2-4B31D6'}

ans =

  1x1 cell array

    {'btspp://00043E4B31D6'}

ans =

  1x11 single row vector

    1.0000    0.0014    0.0023   -0.0022    0.0019   -0.0105   -0.9896
          0.9200   -0.0037    0.0144    0.3915

ans =

  1x11 single row vector

    2.0000   -0.0008    0.0023   -0.0016    0.0029   -0.0115   -0.9897
          0.9200   -0.0037    0.0144    0.3915

ans =

  1x11 single row vector

    3.0000    0.0004    0.0023   -0.0025    0.0028   -0.0125   -0.9900
          0.9200   -0.0037    0.0144    0.3915
```

The first number in each row vector is the sample, the next three are the angular rates from the gyro, the next three the accelerations, and the last four the quaternion. The acceleration is mostly in the -z direction which means that +z is in the button direction. Bluetooth, like all wireless connections, can be problematic. If you get this error

```
Index exceeds the number of array elements (0).

Error in BluetoothTest (line 7)
btInfo.RemoteNames(1)
```

turn the IMU on and off. You might also have to restart MATLAB at times. This is because RemoteNames is empty, and this test is assuming it will not be. MATLAB then gets confused.

7.3 Orientation

7.3.1 Problem

We want to use quaternions to represent the orientation of our dancer in our deep learning system.

7.3.2 Solution

Implement basic quaternion operations. We need quaternion operations to process the quaternions from the IMU.

7.3.3 How It Works

Quaternions are the preferred mathematical representation of orientation. Propagating a quaternion requires fewer operations than propagating a transformation matrix and avoids singularities that occur with Euler angles. A quaternion has four elements, which corresponds to a unit vector a and angle of rotation ϕ about that vector. The first element is termed the ''scalar component'' s, and the next three elements are the ''vector'' components v. This notation is shown as follows [21]:

$$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} s \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \cos\frac{\phi}{2} \\ a_1 \sin\frac{\phi}{2} \\ a_2 \sin\frac{\phi}{2} \\ a_3 \sin\frac{\phi}{2} \end{bmatrix} \tag{7.13}$$

The ''unit'' quaternion which represents zero rotation from the initial coordinate frame has a unit scalar component and zero vector components. This is the same convention used on the Space Shuttle, although other conventions are possible.

$$q_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{7.14}$$

In order to transform a vector from one coordinate frame a to another b using a quaternion q_{ab}, the operation is

$$u_b = q_{ab}^T u_a q_{ab} \tag{7.15}$$

using quaternion multiplication with the vectors defined as quaternions with a scalar part equal to zero, or

$$x_a = \begin{bmatrix} 0 \\ x_a(1) \\ x_a(2) \\ x_a(3) \end{bmatrix} \tag{7.16}$$

For example, the quaternion

$$\begin{bmatrix} 0.7071 \\ 0.7071 \\ 0.0 \\ 0.0 \end{bmatrix} \tag{7.17}$$

represents a pure rotation about the x-axis. The first element is 0.7071 and equals the $\cos(90°/2)$. We cannot tell the direction of rotation from the first element. The second element is the 1 component of the unit vector, which in this case is

$$\begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix} \tag{7.18}$$

times the argument $\sin(90°/2)$. Since the sign is positive, the rotation must be a positive 90° rotation.

We only need one routine that converts the quaternion, which comes from the IMU, into a transformation matrix for visualization. We do this because multiplying a 3× n array of vectors for the vertices of our 3D model by a matrix is much faster than transforming each vector with a quaternion.

QuaternionToMatrix.m

```
        2*(q(2)*q(4)+q(1)*q(3));...
        2*(q(2)*q(3)+q(1)*q(4)),...
        q(1)^2-q(2)^2+q(3)^2-q(4)^2,...
        2*(q(3)*q(4)-q(1)*q(2));...
        2*(q(2)*q(4)-q(1)*q(3)),...
        2*(q(3)*q(4)+q(1)*q(2)),...
        q(1)^2-q(2)^2-q(3)^2+q(4)^2];
```

Note that the diagonal terms have the same form. The off-diagonal terms also all have the same form.

7.4 Dancer Simulation

7.4.1 Problem

We want to simulate a dancer for readers who don't have access to the hardware.

7.4.2 Solution

We will write a right-hand side for the dancer based on the preceding equations and write a simulation with a control system.

7.4.3 How It Works

The right-hand side implements the dancer model. It includes an internal control ‘‘wheel'' and a degree of freedom for the head movement. The default data structure is returned if you call it without arguments.

RHSDancer.m

```
%% RHSDANCER Implements dancer dynamics
% This is a model of dancer with one degree of translational freedom
% and 5 degrees of rotational freedom including the head and an
    internal
% rotational degree of freedom.
%% Form:
%   xDot = RHSDancer( x, ~, d )
%% Inputs
%   x       (11,1)      State vector [r;v;q;w;wHDot;wIDot]
%   t       (1,1)    Time (unused) (s)
%   d       (1,1)    Data structure for the simulation
%                    .torque   (3,1) External torque (Nm)
%                    .force    (1,1) External force (N)
%                    .inertia  (3,3) Body inertia (kg-m^2)
%                    .inertiaH (1,1) Head inertia (kg-m^2)
%                    .inertiaI (1,1) Inner inertia (kg-m^2)
%                    .mass     (1,1) Dancer mass (kg)
%
%% Outputs
%   xDot    (11,1)      d[r;v;q;w;wHDot;wIDot]/dt

function xDot = RHSDancer( ~, x, d )

% Default data structure
if( nargin < 1 )
   % Based on a 0.15 m radius, 1.4 m long cylinders
  inertia = diag([8.4479    8.4479    0.5625]);
  xDot    = struct('torque',[0;0;0],'force',0,'inertia',inertia,...
  'mass',50,'inertiaI',0.0033,'inertiaH',0.0292,'torqueH',0,'torqueI'
      ,0);
  return
end
```

The remainder mechanizes the equations given earlier. We add an additional equation for the integral of the z-axis rate. This makes the control system easier to write. We also include the gravitational acceleration in the force equation.

```
% Use local variables
v      = x(2);
q      = x(3:6);
w      = x(7:9);
wI     = x(10);
wH     = x(11);

% Unit vector
u      = [0;0;1];

% Gravity
g      = 9.806;

% Attitude kinematics (not mentioned in the text)
qDot     = QIToBDot( q, w );

% Rotational dynamics Equation 7.6
wDot     = d.inertia\(d.torque - Skew(w)*(d.inertia*w + d.inertiaI*(wI +
     w(3))...
        + d.inertiaH*(wH + w(3))) - u*(d.torqueI + d.torqueH));
wHDot = d.torqueH/d.inertiaH - wDot(3);
wIDot = d.torqueI/d.inertiaI - wDot(3);

% Translational dynamics
vDot   = d.force/d.mass - g;

% Assemble the state vector
xDot     = [v; vDot; qDot; wDot; wHDot; wIDot; w(3)];
```

The simulation setup gets default parameters from `RHSDancer`.

```
d         = RHSDancer;
n         = 800;
dT        = 0.01;
xP        = zeros(16,n);
x         = zeros(12,1);
x(3)      = 1;
g         = 9.806;
dancer    = 'Robot_1';
```

It then sets up the control system. We use a proportional derivative controller for z position and a rate damper to stop the pirouette. The position control is done by the foot muscles. The rate damping is our internal damper wheel.

```
% Control system for 2 pirouettes in 6 seconds
tPirouette  = 6;
```

```
zPointe     = 6*0.0254;
tPointe     = 0.1;
kP          = tPointe/dT;
omega       = 4*pi/tPirouette;
torquePulse = d.inertia(3,3)*omega/tPointe;
tFriction   = 0.1;
a           = 2*zPointe/tPointe^2 + g;
kForce      = 1000;
tau         = 0.5;
thetaStop   = 4*pi - pi/4;
kTorque     = 200;
state       = zeros(10,n);
```

The simulation loop calls the right-hand side and the control system. We call RHSDancer.m to get the linear acceleration.

```
%% Simulate
for k = 1:n
  d.torqueH = 0;
  d.torqueI = 0;

  % Get the data for use in the neural network
  xDot = RHSDancer(0,x,d);

  state(:,k)  = [x(7:9);0;0;xDot(2);x(3:6)];

  % Control
  if( k < kP )
    d.force  = d.mass*a;
    d.torque = [0;0;torquePulse];
  else
    d.force  = kForce*(zPointe-x(1) -x(2)/tau)+ d.mass*g;
    d.torque = [0;0;-tFriction];
  end

  if( x(12) > thetaStop )
    d.torqueI = kTorque*x(9);
  end

  xP(:,k)    = [x;d.force;d.torque(3);d.torqueH;d.torqueI];
  x          = RungeKutta(@RHSDancer,0,x,dT,d);
end
```

The control system includes a torque and force pulse to get the pirouette going.

```
  % Control
  if( k < kP )
    d.force  = d.mass*a;
    d.torque = [0;0;torquePulse];
  else
    d.force  = kForce*(zPointe-x(1) -x(2)/tau)+ d.mass*g;
    d.torque = [0;0;-tFriction];
  end
```

The remainder of the script plots the results and outputs the data, which would have come from the IMU, into a file.

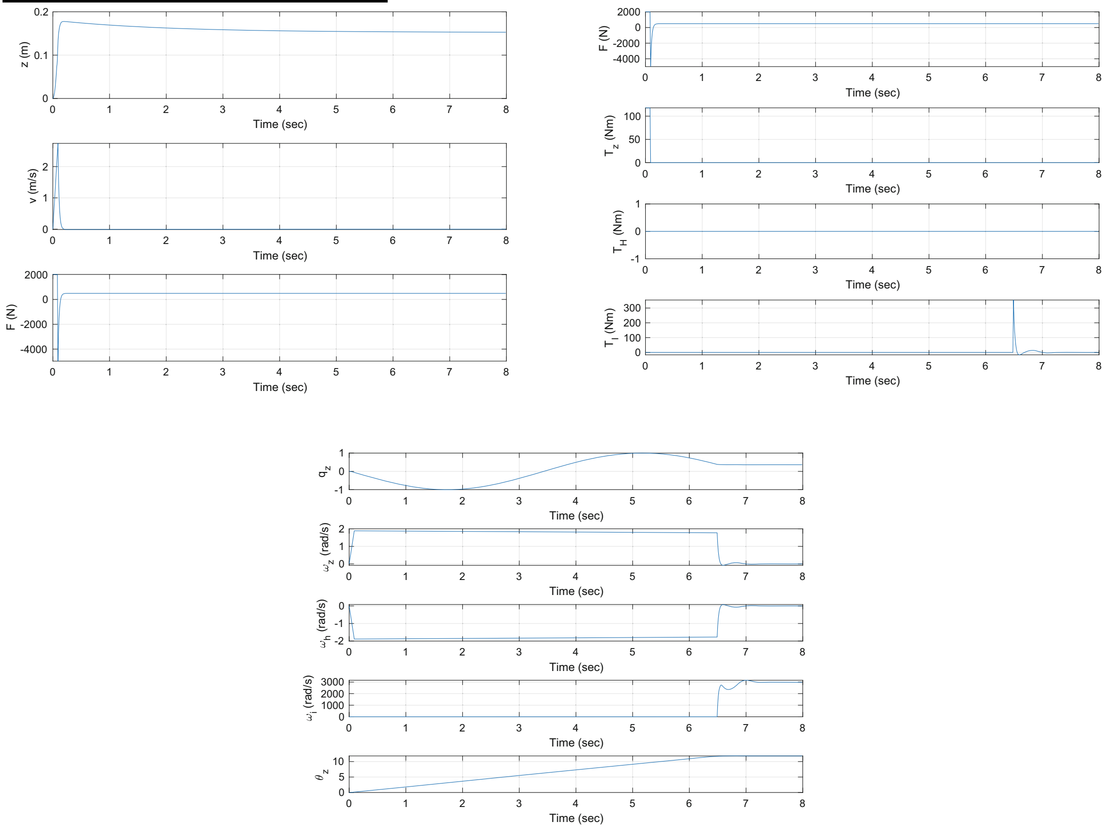
Simulation results for a double pirouette are shown in Figure 7.5. We stop the turn at 6.5 seconds, hence the pulse.

You can create different dancers by varying the mass properties and the control parameters.

```
zPointe      = 6*0.0254;
tPointe      = 0.1;
tFriction    = 0.1;
kForce       = 1000;
tau          = 0.5;
kTorque      = 200;
```

We didn't implement spotting control (looking at the audience as much as possible during the turn). It would rotate the head so that it faces forward whenever the head was within 90

Figure 7.5: Simulation of a double pirouette.



degrees or so of front. We'd need to add a head angle for that purpose to the right-hand side, much like we added the z-axis angle.

7.5 Real-Time Plotting

7.5.1 Problem

We want to display data from the IMU in real time. This will allow us to monitor the pirouettes.

7.5.2 Solution

Use `plot` with `drawnow` to implement multiple figures of plots.

7.5.3 How It Works

The main function is a switch statement with two cases. The function also has a built-in demo. The first case, `initialize`, initializes the plot figures. It stores everything in a data structure that is returned on each function call. This is one way for a function to have a memory. We return the data structure from each subfunction.

GUIPlots.m

```
switch( lower(action) )
  case 'initialize'
    g = Initialize( g );

  case 'update'
    g = Update( g, y, t );

end
```

The first case, `initialize`, initializes the figure window.

```
%% GUIPlots>Initialize
function g = Initialize( g )

lY = length(g.yLabel);

% Create tLim if it does not exist
if( ~isfield(g, 'tLim' ) )
  g.tLim = [0 1];
end

g.tWidth = g.tLim(2) - g.tLim(1);

% Create yLim if it does not exist
if( ~isfield( g, 'yLim' ) )
  g.yLim = [-ones(lY,1), ones(lY,1)];
end

% Create the plots
```

```
lP = length(g.yLabel);
y  = g.pos(2); % The starting y position
for k = 1:lP
  g.h(k) = subplot(lP,1,k);
  set(g.h(k),'position',[g.pos(1) y g.pos(3) g.pos(4)]);
  y = y - 1.4*g.pos(4);
  g.hPlot(k) = plot(0,0);
  g.hAxes(k) = gca;
  g.yWidth(k) = (g.yLim(k,2) - g.yLim(k,1))/2;
  set(g.hAxes(k),'nextplot','add','xlim',g.tLim);
  ylabel( char(g.yLabel{k}) )
  grid on
end
xlabel( g.tLabel );
```

The second case, update, updates the data displayed in the plot. It leaves the existing figures, subplots, and labels in place and just updates the plots of the line segments with new data. It can change the size of the axes as needed. The function adds a line segment for each new data point. This way no storage is needed external to the plot. It reads `xdata` and `ydata` and appends the new data to those arrays.

```
function g = Update( g, y, t )

% See if the time limits have been exceeded
if( t > g.tLim(2) )
  g.tLim(2)  = g.tWidth + g.tLim(2);
  updateAxes = true;
else
  updateAxes = false;
end

lP = length(g.yLabel);
for k = 1:lP
  subplot(g.h(k));
  yD = get(g.hPlot(k),'ydata');
  xD = get(g.hPlot(k),'xdata');
  if( updateAxes )
    set( gca, 'xLim', g.tLim );
    set( g.hPlot(k), 'xdata',[xD t],'ydata',[yD y(k)]);
  else
    set( g.hPlot(k), 'xdata',[xD t],'ydata',[yD y(k)] );
  end

end
```

The built-in demo plots six numbers. It updates the axes in time once. It sets up a figure window with six plots. You need to create the figure and save the figure handle before calling `GUIPlots`.

```
g.hFig  = NewFig('State');
```

The pause in the demo just slows down plotting so that you can see the updates. The height (the last number in `g.pos`) is the height of each plot. If you happen to set the locations of the plots out of the figure window, you will get a MATLAB error. `g.tLim` gives the initial time limits in second. The upper limit will expand as data is entered.

```
function Demo

g.yLabel = {'x' 'y' 'z' 'x_1' 'y_1' 'z_1'};
g.tLabel = 'Time (sec)';
g.tLim   = [0 100];
g.pos    = [0.100    0.88    0.8    0.10];
g.width  = 1;
g.color  = 'b';

g.hFig  = NewFig('State');
set(g.hFig, 'NumberTitle','off' );

g        = GUIPlots( 'initialize', [], [], g );

for k = 1:200
  y = 0.1*[cos((k/100))-0.05;sin(k/100)];
  g = GUIPlots( 'update', [y;y.^2;2*y], k, g );
  pause(0.1)
end

g        = GUIPlots( 'initialize', [], [], g );

for k = 1:200
  y = 0.1*[cos((k/100))-0.05;sin(k/100)];
  g = GUIPlots( 'update', [y;y.^2;2*y], k, g );
  pause(0.1)
end
```

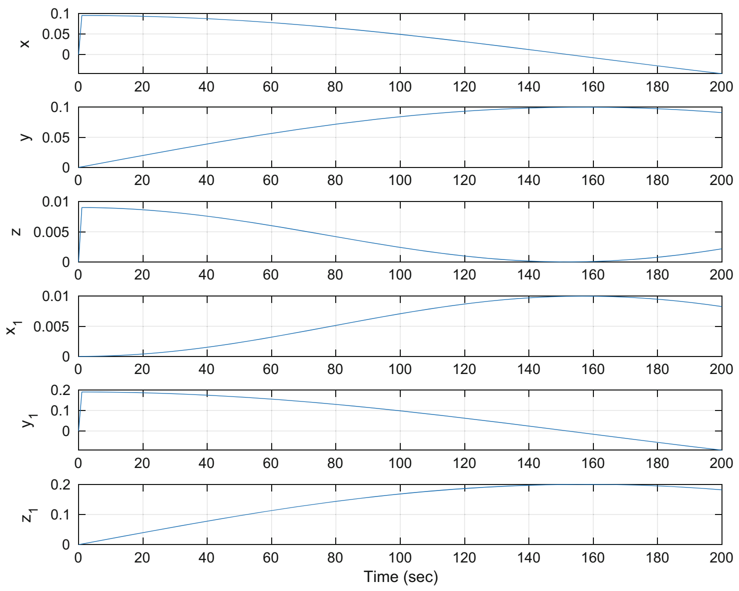
Figure 7.6 shows the real-time plots at the end of the demo.

7.6 Quaternion Display

7.6.1 Problem

We want to display the dancer's orientation in real time.

Figure 7.6: Real-time plots.



7.6.2 Solution

Use `patch` to draw an OBJ model in a three-dimensional plot. The figure is easier to understand than the four quaternion elements. Our solution can handle 3-axis rotation although typically we will only see single-axis rotation.

7.6.3 How It Works

We start with our `Ballerina.obj` file. It only has vertices and faces. A 3D drawing consists of a set of vertices. Each vertex is a point in space. The vertices are organized into faces. Each face is a triangle. Triangles are used for 3D drawings because they always form a plane. 3D processing hardware is designed to work with triangles, so this also gives the fastest results. The obj files for our software can only contain triangles. Each face can have only three vertices. Generally, obj files can have any size polynomials, that is, faces with more than three points. Most sources of obj files can provide tessellation services to convert polygons with more than three vertices into triangles. `LoadOBJ.m` will not draw models with anything other than triangular faces.

The main part of the function uses a case statement to handle the three actions. The first action just returns the defaults, which is the name of the default obj file. The second reads in the file and initializes the patches. The third updates the patches. `patch` is the MATLAB name for a set of triangles. The function can be passed a figure handle. A figure handle tells it into which figure the 3D model should be drawn. This allows it to be used as part of a GUI, as will be shown in the next section.

QuaternionVisualization.m

```
function m = QuaternionVisualization( action, x, f )

persistent p

% Demo
if( nargin < 1 )
  Demo
  return
end

switch( lower(action) )
    case 'defaults'
        m = Defaults;

    case 'initialize'
        if( nargin < 2 )
            d   = Defaults;
        else
            d   = x;
        end

        if( nargin < 3 )
          f = [];
        end

        p = Initialize( d, f );

    case 'update'
        if( nargout == 1 )
            m = Update( p, x );
        else
            Update( p, x );
        end
end
```

Initialize loads the obj file. It creates a figure and saves the object data structure. It sets shading to interpolated and lighting to Gouraud. Gouraud is a type of lighting model named after its inventor. It then creates the patches and sets up the axis system. We save handles to all the patches for updating later. We also place a light.

```
function p = Initialize( file, f )

if( isempty(f) )
  p.fig = NewFigure( 'Quaternion' );
else
  p.fig = f;
end

g     = LoadOBJ( file );
p.g   = g;
```

```

shading interp
lighting gouraud

c = [0.3 0.3 0.3];

for k = 1:length(g.component)
  p.model(k)    = patch('vertices', g.component(k).v, 'faces', g.
      component(k).f, 'facecolor',c,'edgecolor',c,'ambient',1,'
      edgealpha',0 );
end

xlabel('x');
ylabel('y');
zlabel('z');

grid
rotate3d on
set(gca,'DataAspectRatio',[1 1 1],'DataAspectRatioMode','manual')

light('position',10*[1 1 1])

view([1 1 1])
```

In `Update` we convert the quaternion to a matrix, because it is faster to matrix multiply all the vertices with one matrix multiplication. The vertices are n by 3 so we transpose before the matrix multiplication. We use the patch handles to update the vertices. The two options at the end are to create movie frames or just update the drawing.

```
function m = Update( p, q )

s = QuaternionToMatrix( q );

for k = 1:length(p.model)
  v = (s*p.g.component(k).v')';
  set(p.model(k),'vertices',v);
end

if( nargout > 0 )
        m = getframe;
else
        drawnow;
end
```

This is the built-in demo. We vary the 1 and 4 elements of the quaternion to get rotation about the z-axis.

```
function Demo

QuaternionVisualization( 'initialize', 'Ballerina.obj' );
```

Figure 7.7: Dancer orientation. The obj file is by the artist loft_22 and is available from TurboSquid.

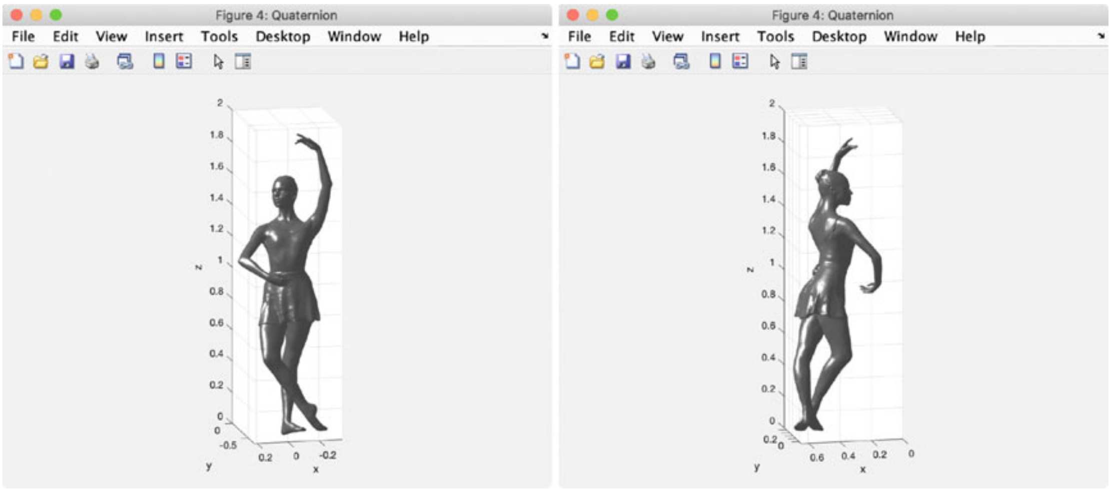


Figure 4: Quaternion
File Edit View Insert Tools Desktop Window Help
z
y
x
Figure 4: Quaternion
File Edit View Insert Tools Desktop Window Help
z
y
x

Figure 7.7 shows two orientations of the dancer during the demo. The demo produces an animation of the dancer rotating about the z-axis. The rotation is slow because of the number of vertices. The figure is not articulated so the entire figure is rotated as a rigid body. MATLAB doesn't make it easy to texture map so we don't bother. In any case, the purpose of this function is just to show orientation so it doesn't matter.

7.7 Data Acquisition GUI

7.7.1 Problem

Build a data acquisition GUI to display the real-time data and output it into training sets.

7.7.2 Solution

Integrate all the preceding recipes into a GUI.

7.7.3 How It Works

We aren't going to use MATLAB's Guide to build our GUI. We will hand code it, which will give you a better idea of how a GUI really works.

We will use nested functions for the GUI. The inner functions have access to all variables in the outer functions. This also makes using callbacks easy as shown in the following code snippet.

```
function DancerGUI( file )
function DrawGUI(h)
 uicontrol( h.fig,'callback',@SetValue);
   function SetValue(hObject, ~, ~ )
```

```
    % do something
   end
 end
 end
```

A callback is a function called by a `uicontrol` when the user interacts with the control. When you first open the GUI, it will look for the bluetooth device. This can take a while.

Everything in `DrawGUI` has access to variables in `DancerGUI`. The GUI is shown in Figure 7.8. The 3D orientation display is in the upper left corner. Real-time plots are on the right. Buttons are on the lower left, and the movie window is on the right.

The upper left picture shows the dancer's orientation. The plots on the right show angular rates and accelerations from the IMU. From top to bottom of the buttons

1. Turn the 3D on/off. The default model is big, so unless you add your own model with fewer vertices, it should be set to off.
2. The text box to its right is the name of the file. The GUI will add a number to the right of the name for each run.
3. Save saves the current data to a file.
4. Calibrate sets the default orientation and sets the gyro rates and accelerations to whatever it is reading when you hit the button. The dancer should be still when you hit calibrate. It will automatically compute the gravitational acceleration and subtract it during the test.
5. Quit closes the GUI.
6. Clear data clears out all the internal data storage.
7. Start/Stop starts and stops the GUI.

The remaining three lines display the time, the angular rate vector, and the acceleration vector as numbers. This is the same data that is plotted.

The first part creates the figure and draws the GUI. It initializes all the fields for `GUIPlots`. It reads in a default picture for the movie window as a placeholder.

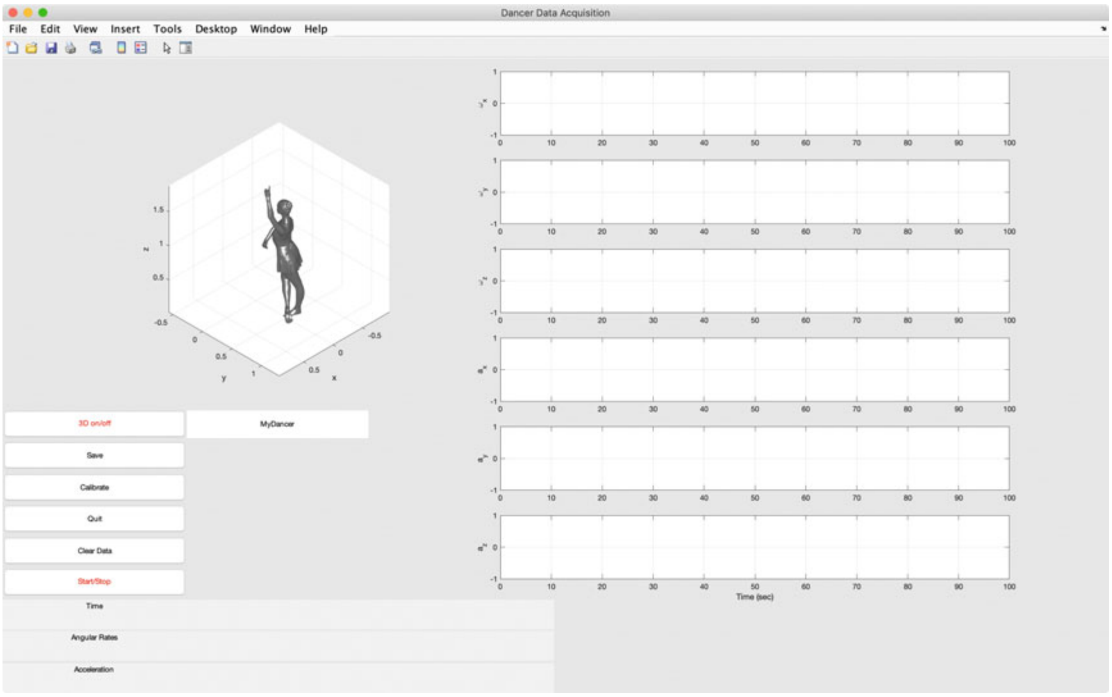
DancerGUI.m

```
function DancerGUI( file )

% Demo
if( nargin < 1 )
  DancerGUI('Ballerina.obj');
  return
end

% Storage of data need by the deep learning system
kStore      = 1;
accelStore  = zeros(3,1000);
gyroStore   = zeros(3,1000);
```

Figure 7.8: Data acquisition GUI.



```matlab
quatStore    = zeros(4,1000);
timeStore    = zeros(1,1000);
time         = 0;
on3D         = false;
quitNow      = false;

sZ = get(0,'ScreenSize') + [99 99 -200 -200];

h.fig = figure('name','Dancer Data Acquisition','position',sZ,'units','pixels',...
  'NumberTitle','off','tag','DancerGUI','color',[0.9 0.9 0.9]);

% Plot display
gPlot.yLabel   = {'\omega_x' '\omega_y' '\omega_z' 'a_x' 'a_y' 'a_z'};
gPlot.tLabel   = 'Time (sec)';
gPlot.tLim     = [0 100];
gPlot.pos      = [0.45    0.88    0.46    0.1];
gPlot.color    = 'b';
gPlot.width    = 1;

% Calibration
q0             = [1;0;0;0];
a0             = [0;0;0];

```

```
dIMU.accel    = a0;
dIMU.quat     = q0;

% Initialize the GUI
DrawGUI;
```

The notation

```
'\omega_x'
```

is latex format. This will generate ω_x.

The next part tries to find Bluetooth. It first sees if Bluetooth is available at all. It then enumerates all Bluetooth devices. It looks through the list to find our IMU.

```
if( ~isempty(btInfo.RemoteIDs) )
   % Display the information about the first device discovered
  btInfo.RemoteNames(1)
  btInfo.RemoteIDs(1)
  for iB = length(btInfo.RemoteIDs)
    if( strcmp(btInfo.RemoteNames(iB),'LPMSB2-4B31D6') )
      break;
    end
  end
  b       = Bluetooth(btInfo.RemoteIDs{iB}, 1);
  fopen(b); % No output allowed for some reason
  noIMU   = false;
  a       = fread(b,91);
  dIMU    = DataFromIMU( a );
else
  warndlg('The IMU is not available.', 'Hardware Configuration')
  noIMU   = true;
end
```

The following is the run loop. If no IMU is present, it synthesizes data. If the IMU is found, the GUI reads data from the IMU in 91 byte chunks. The `uiwait` is to wait until the user hits the start button. When used for testing, the IMU should be on the dancer. The dancer should remain still when the start button is pushed. It will then calibrate the IMU. Calibration fixes the quaternion reference and removes the gravitational acceleration. You can also hit the calibration button at any time.

```
% Wait for user input
uiwait;
% The run loop
time  = 0;
tic
while(1)
  if( noIMU )
    omegaZ      = 2*pi;
    dT          = toc;
    time         = time + dT;
    tic
    a            = omegaZ*time;
    q            = [cos(a);0;0;sin(a)];
    accel        = [0;0;sin(a)];
    omega        = [0;0;omegaZ];
  else
    % Query the bluetooth device
    a       = fread(b,91);
    pause(0.1); % needed so not to overload the bluetooth device

    dT      = toc;
    time    = time + dT;
    tic

    % Get a data structure
    if( length(a) > 1 )
      dIMU    = DataFromIMU( a );
    end
    accel   = dIMU.accel - a0;
    omega   = dIMU.gyro;
    q       = QuaternionMultiplication(q0,dIMU.quat);

    timeStore(1,kStore)   = time;
    accelStore(:,kStore)  = accel;
    gyroStore(:,kStore)   = omega;
    quatStore(:,kStore)   = q;
    kStore = kStore + 1;
  end
```

```
      dIMU    = DataFromIMU( a );
    end
    accel   = dIMU.accel - a0;
    omega   = dIMU.gyro;
```

This code closes the GUI and displays the IMU data.

```
  if( quitNow )
    close( h.fig )
    return
  else
    if( on3D )
      QuaternionVisualization( 'update', q );
    end
    set(h.text(1),'string',sprintf('[%5.2f;%5.2f;%5.2f] m/s^2',accel));
    set(h.text(2),'string',sprintf('[%5.2f;%5.2f;%5.2f] rad/s',omega));
    set(h.text(3),'string',datestr(now));
    gPlot = GUIPlots( 'update', [omega;accel], time, gPlot );
  end
end
```

The drawing code uses `uicontrol` to create all the buttons. `GUIPlots` and `QuaternionVisualization` are also initialized. The `uicontrol` that require an action have callbacks.

```
  if( quitNow )
    close( h.fig )
    return
  else
    if( on3D )
      QuaternionVisualization( 'update', q );
    end
    set(h.text(1),'string',sprintf('[%5.2f;%5.2f;%5.2f] m/s^2',accel));
    set(h.text(2),'string',sprintf('[%5.2f;%5.2f;%5.2f] rad/s',omega));
    set(h.text(3),'string',datestr(now));
    gPlot = GUIPlots( 'update', [omega;accel], time, gPlot );
  end
end

%% DancerGUI>DrawButtons
function DrawGUI

  % Plots
  gPlot = GUIPlots( 'initialize', [], [], gPlot );

  % Quaternion display
  subplot('position',[0.05 0.5 0.4 0.4],'DataAspectRatio',[1 1 1],'
      PlotBoxAspectRatio',[1 1 1] );
  QuaternionVisualization( 'initialize', file, h.fig );

  % Buttons
  f    = {'Acceleration', 'Angular Rates' 'Time'};
  n    = length(f);
  p    = get(h.fig,'position');
  dY   = p(4)/20;
  yH   = p(4)/21;
  y    = 0.5;
  x    = 0.15;
  wX   = p(3)/6;

  % Create pushbuttons and defaults
  for k = 1:n
    h.pushbutton(k) = uicontrol( h.fig,'style','text','string',f{k},'
        position',[x    y wX yH]);
    h.text(k)       = uicontrol( h.fig,'style','text','string','',  '
        position',[x+wX y 2*wX yH]);
    y               = y + dY;
  end
```

`uicontrol` takes parameter pairs, except for the first argument that can be a figure handle. There are lot of parameter pairs. The easiest way to explore them is to type

```
h = uicontrol;
get(h)
```

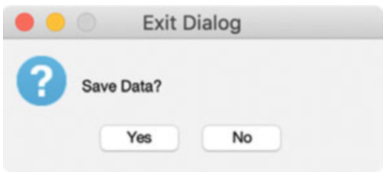
All types of `uicontrol` that handle user interaction have ‘‘callbacks” that are functions that do something when the button is pushed or menu item is selected. We have five `uicontrol` with callbacks. The first uses `uiwait` and `uiresume` to start and stop data collection.

```
  % Start/Stop button callback
  function StartStop(hObject, ~, ~ )
    if( hObject.Value )
      uiresume;
    else
      SaveFile;
      uiwait
    end
  end
```

The second uses `questdlg` to ask if you want to save the data that has been stored in the GUI. This produces the modal dialog shown in Figure 7.9.

```
  % Quit button callback
  function Quit(~, ~, ~ )
    button = questdlg('Save Data?','Exit Dialog','Yes','No','No');
    switch button
      case 'Yes'
        % Save data
      case 'No'
    end
    quitNow = true;
    uiresume
  end
```

Figure 7.9: Modal dialog.



The third, `Clear`, clears the data storage arrays. It resets the quaternion to a unit quaternion.

```
  % Clear button callback
  function Clear(~, ~, ~ )
    kStore      = 1;
    accelStore  = zeros(3,1000);
    gyroStore   = zeros(3,1000);
    quatStore   = zeros(4,1000);
    timeStore   = zeros(1,1000);
    time        = 0;
  end
```

The fourth, `calibrate`, runs the calibration procedure.

```
  % Calibrate button callback
  function Calibrate(~, ~, ~ )
    a       = fread(b,91);
    dIMU    = DataFromIMU( a );
    a0      = dIMU.accel;
    q0      = dIMU.quat;
    QuaternionVisualization( 'update', q0 )
  end
```

The fifth, `SaveFile`, saves the recorded data into a mat file for use by the deep learning algorithm.

```
  % Save button call back
  function SaveFile(~,~,~)
    cd TestData
    fileName = get(h.matFile,'string');
    s = dir;
    n = length(s);
    fNames = cell(1,n-2);
    for kF = 3:n
      fNames{kF-2} = s(kF).name(1:end-4);
    end
    j = contains(fNames,fileName);
    m = 0;
    if( ~isempty(j) )
      for kF = 1:length(j)
        if( j(kF))
          f = fNames{kF};
          i = strfind(f,'_');
          m = str2double(f(i+1:end));
        end
      end
    end
```

We make it easier for the user to save files by reading the directory and adding a number to the end of the dancer filename that is one greater than the last filename number.

7.8 Making the IMU Belt

7.8.1 Problem

We need to attach the IMU to our dancer.

7.8.2 Solution

We use the arm strap that is available from the manufacturer. We buy an elastic belt and make one that fits around the dancer's waist.

7.8.3 How It Works

Yes, software engineers need to sew. Figure 7.10 shows the process. The two products used to make the data acquisition belt are

1. LPMS-B2 Holder (available from Life Performance Research)
2. Men's No Show Elastic Stretch Belt Invisible Casual Web Belt Quick Release Flat Plastic Buckle (available from Amazon)

Remove the holder from the LPMS-B2 Holder. Cut the belt at the buckle and slide the holder onto the belt. Sew the belt at the buckle.

The sensor on a dancer is shown in Figure 7.11. We had the dancer stand near the laptop during startup. We didn't have any range problems during the experiments. We didn't try it with across the floor movement as one would have during grande allegro.

Figure 7.10: Elastic belt manufacturing. We use the two items on the left to make the one on the right.

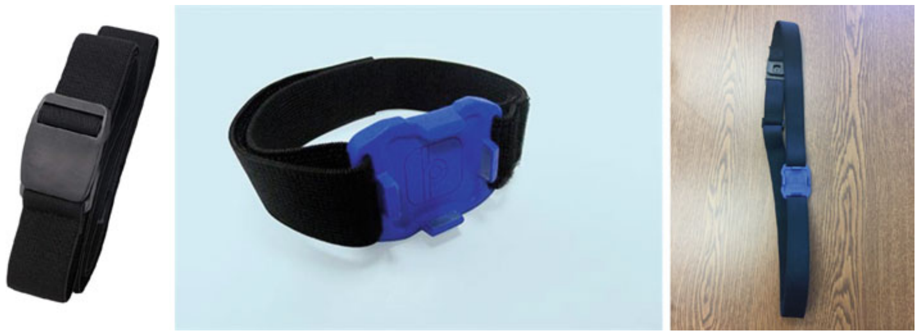
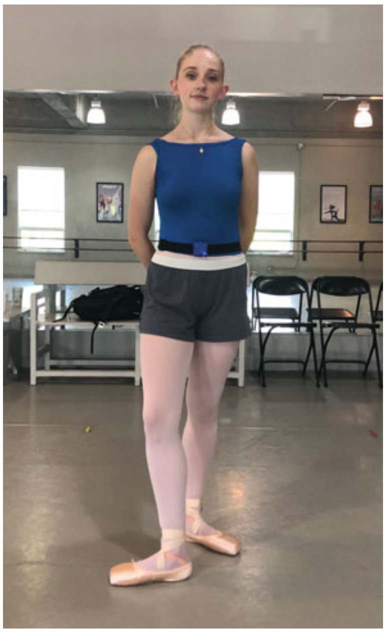


Figure 7.11: Dancer with the sensor belt. The blue light means it is collecting data.



7.9 Testing the System

7.9.1 Problem

We want to test the data acquisition system. This will find any problems with the data acquisition process.

7.9.2 Solution

Have a dancer do changements, which are small jumps changing the foot position on landing.

7.9.3 How It Works

The dancer puts on the sensor belt, we push the calibrate button, then she does a series of changements. She stands about 2 m from your computer to make acquisition easier. The dancer will do small jumps, known as changements. A changement is a small jump where the feet change positions starting from fifth position. If the right foot is in fifth position front at the start, it is in the back at the finish. Photos are shown in Figure 7.12.

The time scale is a bit long. You can see that the calibration does not lead us to a natural orientation in the axis system in the GUI. It doesn't matter from a data collection point of view but is an improvement we should make in the future. The changement is shown in Figure 7.13. The dancer is still at the beginning and end.

The interface to the bluetooth device doesn't do any checking or stream control. Some bluetooth data collection errors occur from time to time. Typically, they happen after 40 seconds of data collection.

Figure 7.12: Dancer doing a changement. Notice the feet when she is preparing to jump. The second image shows her feet halfway through the jump.

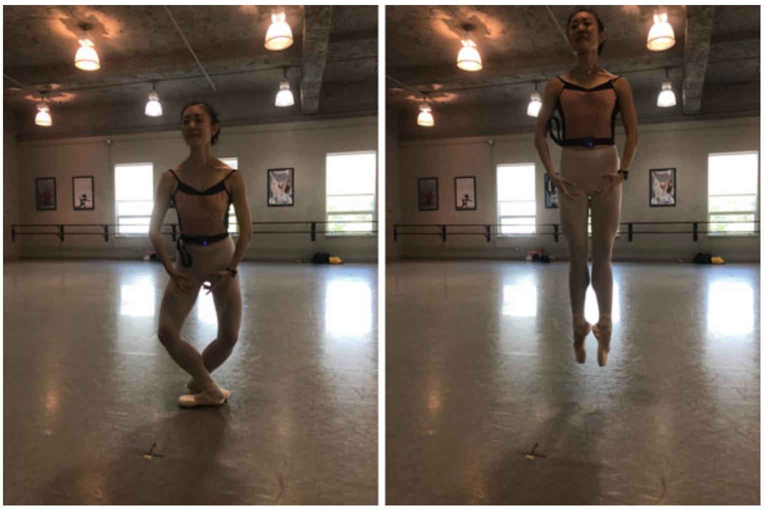
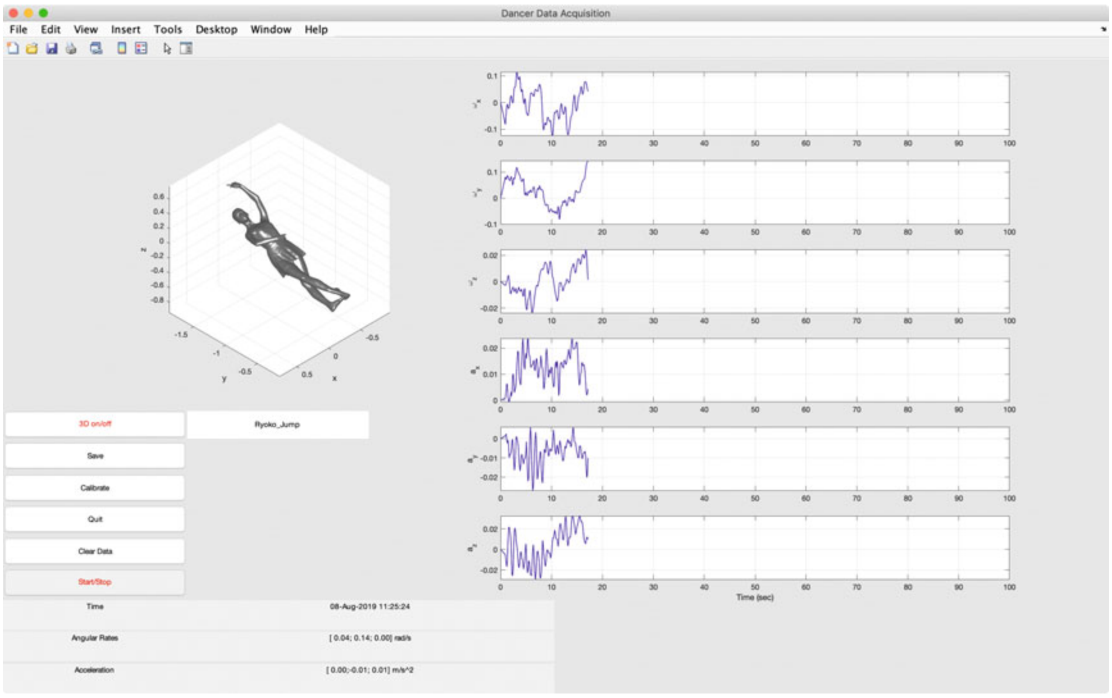


Figure 7.13: Data collected during the changement.



Time	08-Aug-2019 11:25:24
Angular Rates	[0.04; 0.14; 0.00] rad/s
Acceleration	[0.00;-0.01; 0.01] m/s^2

```
index exceeds the number of array elements (0).
Error in instrhwinfo>bluetoothCombinedDevices (line 976)
        uniqueBTName = allBTName(uniqueRowOrder);
Error in instrhwinfo (line 206)
                    tempOut = bluetoothCombinedDevices(tempOut);
Error in DataAcquisition (line 13)
btInfo  = instrhwinfo('Bluetooth');
```

If this happens, turn the device on and off. Restart MATLAB if that doesn't work. Another bluetooth error is

```
ans =
  1x1 cell array
    {'LPMSB2-4B31D6'}
ans =
  1x1 cell array
    {'btspp://00043E4B31D6'}
Error using Bluetooth (line 104)
Cannot Create: Java exception occurred:
java.lang.NullPointerException
        at com.mathworks.toolbox.instrument.BluetoothDiscovery.
            searchDevice(BluetoothDiscovery.java:395)
        at com.mathworks.toolbox.instrument.BluetoothDiscovery.
            discoverServices(BluetoothDiscovery.java:425)
        at com.mathworks.toolbox.instrument.BluetoothDiscovery.
            hardwareInfo(BluetoothDiscovery.java:343)
        at com.mathworks.toolbox.instrument.Bluetooth.<init>(Bluetooth.
            java:205).
```

This is a MATLAB error and requires restarting MATLAB. It doesn't happen very often. We ran the entire data collection with four dancers doing ten pirouettes each without ever experiencing the problem.

7.10 Classifying the Pirouette

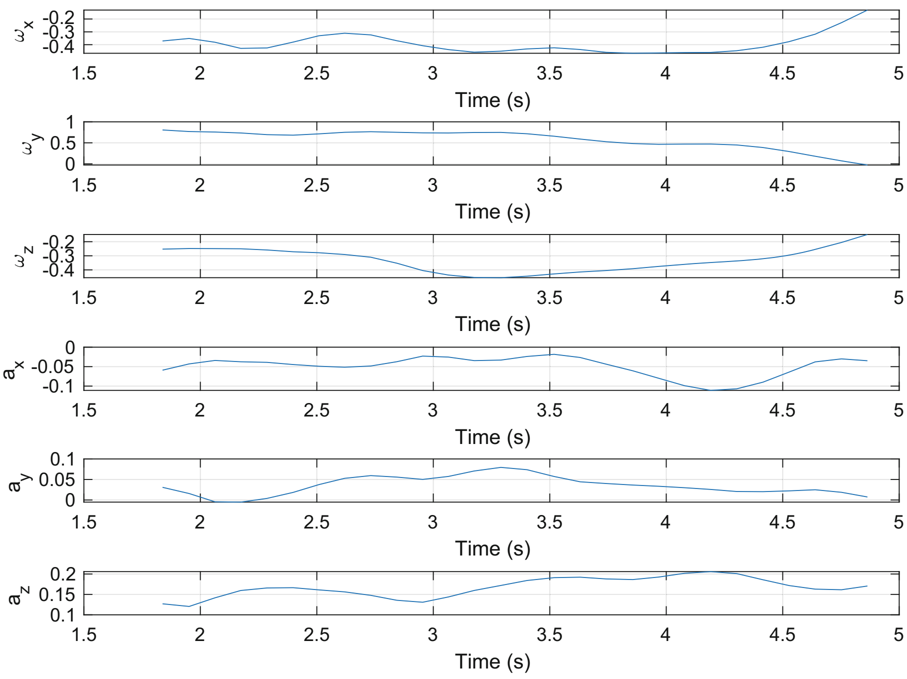
7.10.1 Problem

We want to classify the pirouettes of our four dancers.

7.10.2 Solution

Create an LSTM that classifies pirouettes according to dancer. The four labels are the dancer names.

Figure 7.14: A pirouette. Angular rate and linear acceleration are shown.



7.10.3 How It Works

The script takes one file and displays it. Figure 7.14 shows a double pirouette. A turn takes only a few seconds.

```
dancer = {'Ryoko' 'Shaye', 'Emily', 'Matanya'};

%% Show one dancer's data
cd TestData
s    = load('Ryoko_10.mat');
yL   = {'\omega_x' '\omega_y' '\omega_z' 'a_x' 'a_y' 'a_z'};
PlotSet(s.time,s.state(1:6,:),'x label','Time (s)','y label',yL,'figure
      title',dancer{1});
```

We load in the data and limit the range to 6 seconds. Sometimes the IMU would run longer due to human error. We also remove sets that are bad.

DancerNN.m

```
%% Load in and process the data
n = 40;
% Get the data and remove bad data sets
i = 0;
for k = 1:length(dancer)
  for j = 1:10
    s    = load(sprintf('%s_%d.mat',dancer{k},j));
    cS   = size(s.state,2);
    if( cS > 7 )
      i       = i + 1;
      d{i,1}  = s.state; %#ok<*SAGROW>
      t{i,1}  = s.time;
      c(i,1)  = k;
    end
  end
end

fprintf('%d remaining data sets out of %d total.\n',i,n)

for k = 1:4
  j = length(find(c==k));
  fprintf('%7s data sets %d\n',dancer{k},j)
end

n = i;

cd ..

% Limit the range to 6 seconds
tRange = 6;
for i = 1:n
  j = find(t{i} - t{i,1} > tRange );
  if( ~isempty(j) )
    d{i}(:,j(1)+1:end)= [];
  end
end
```

We then train the neural network. We use a bidirectional LSTM to classify the sequences. There are ten features, four quaternion measurements, three rate gyro, and three accelerometer. The four quaternion numbers are coupled through the relationship

$$1 = q_1^2 + q_2^2 + q_3^2 + q_4^2 \tag{7.19}$$

However, this should not impact the learning accuracy aside from slowing down the learning.

We then load in and process the data. Some data sets didn't have any data and need to be removed. We also limit the range to 6 seconds since sometimes the data collection did not stop after the pirouette ended.

```
%% Load in and process the data
n = 40;
% Get the data and remove bad data sets
i = 0;
for k = 1:length(dancer)
  for j = 1:10
    s   = load(sprintf('%s_%d.mat',dancer{k},j));
    cS  = size(s.state,2);
    if( cS > 7 )
      i       = i + 1;
      d{i,1}  = s.state; %#ok<*SAGROW>
      t{i,1}  = s.time;
      c(i,1)  = k;
    end
  end
end

fprintf('%d remaining data sets out of %d total.\n',i,n)

for k = 1:4
  j = length(find(c==k));
  fprintf('%7s data sets %d\n',dancer{k},j)
end

n = i;

cd ..

% Limit the range to 6 seconds
tRange = 6;
for i = 1:n
  j = find(t{i} - t{i,1} > tRange );
  if( ~isempty(j) )
    d{i}(:,j(1)+1:end)= [];
  end
end

%% Set up the network
numFeatures = 10; % 4 quaternion, 3 rate gyros, 3 accelerometers
numHiddenUnits = 400;
numClasses = 4; % Four dancers

layers = [ ...
    sequenceInputLayer(numFeatures)
    bilstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
disp(layers)

options = trainingOptions('adam', ...
    'MaxEpochs',60, ...
    'GradientThreshold',1, ...
    'Verbose',0, ...
    'Plots','training-progress');
```

We then train the neural network. We use a bidirectional LSTM to classify the sequences. This is a good choice because we have access to the full sequence. For a classifier using `bilstmLayer`, we must set the `'outputMode'` to `'last'`. This is followed by a fully connected layer, a Softmax for producing normalized maximums, and finally the classification layer.

```
%% Train the network
nTrain   = 30;
kTrain   = randperm(n,nTrain);
xTrain   = d(kTrain);
yTrain   = categorical(c(kTrain));
net      = trainNetwork(xTrain,yTrain,layers,options);

%% Test the network
kTest    = setdiff(1:n,kTrain);
xTest    = d(kTest);
yTest    = categorical(c(kTest));
yPred    = classify(net,xTest);

% Calculate the classification accuracy of the predictions.
acc          = sum(yPred == yTest)./numel(yTest);
disp('Accuracy')
disp(acc);
```

```
>> DancerNN
36 remaining data sets out of 40 total.
   Ryoko data sets 6
  Shaye data sets 10
  Emily data sets 10
Matanya data sets 10
 5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 10 dimensions
     2   ''   BiLSTM                  BiLSTM with 400 hidden units
     3   ''   Fully Connected         4 fully connected layer
     4   ''   Softmax                 softmax
     5   ''   Classification Output   crossentropyex
```

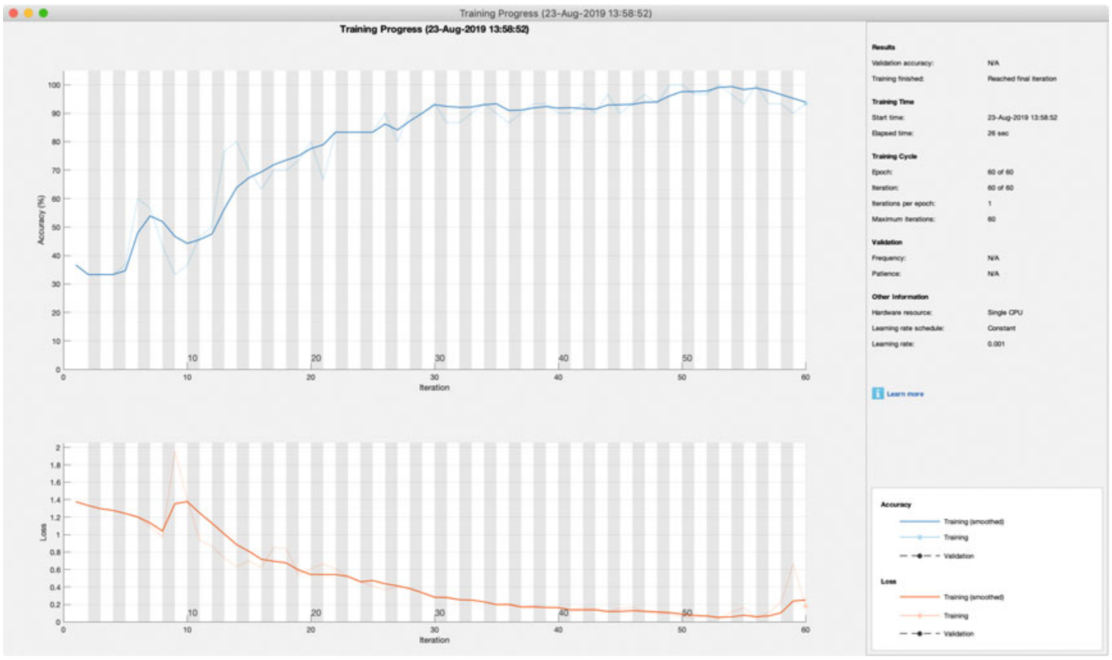
The training GUI is shown in Figure 7.15. It converges fairly well.
We test neural network against the unused data.

```
kTrain   = randperm(n,nTrain);
xTrain   = d(kTrain);
yTrain   = categorical(c(kTrain));
net      = trainNetwork(xTrain,yTrain,layers,options);

%% Test the network
kTest    = setdiff(1:n,kTrain);
xTest    = d(kTest);
yTest    = categorical(c(kTest));
yPred    = classify(net,xTest);
```

```
Accuracy
    0.8333
```

Figure 7.15: Neural net training.



Results

Validation accuracy:	N/A
Training finished:	Reached final iteration

Training Time

Start time:	23-Aug-2019 13:58:52
Elapsed time:	26 sec

Training Cycle

Epoch:	60 of 60
Iteration:	60 of 60
Iterations per epoch:	1
Maximum iterations:	60

Validation

Frequency:	N/A
Patience:	N/A

Other Information

Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.001

Table 7.3: Hardware.

Component	**Supplier**	**Part Number**	**Price**
IMU	LP-Research Inc.	LPMS-B2: 9-Axis Inertial Measurement Unit	\$299.00
IMU Holder	LP-Research Inc.	LPMS-B2: Holder	\$30.00
Belt	Amazon	Men's Elastic Stretch Belt Invisible Casual Trousers Webbing Belt Plastic Buckle Black Fits 24" to 42"	\$10.99

The result, $> 80\%$, is pretty good considering the limited amount of data. Four Ryoko sets were lost due to errors in data collection. It is interesting that the deep learning network could distinguish the dancers' pirouettes. The data itself did not show any easy-to-spot differences. Calibration could have been done better to make the data more consistent between dancers. It would have been interesting to collect data on multiple days. Other experiments would be to classify pirouettes done in pointe shoes and without. We might also have had the dancers do different types of turns to see if the network could still identify the dancer.

7.11 Hardware Sources

Table 7.3 gives the hardware used in this chapter along with the prices (in US dollars) at the time of publication.