```
Transformation getTransformation() {
   rotation.setCoordinates(state[0], state[2], state[4], state[6]);
   return rotation;
}

void setBodyFrameOmega(double[] omega) {
   // use components for clarity
   double q0 = state[0], q1 = state[2], q2 = state[4],
        q3 = state[6];
   double wx = omega[0];
   double wy = omega[1];
   double wz = omega[2];
   state[1] = 0.5*(-q1*wx-q2*wy-q3*wz); // dq0/dt
   state[3] = 0.5*(q0*wx-q3*wy+q2*wz);  // dq1/dt
   state[5] = 0.5*(q3*wx+q0*wy-q1*wz);  // dq2/dt
   state[7] = 0.5*(-q2*wx+q1*wy+q0*wz); // dq3/dt
   updateVectors();
}

public double[] getBodyFrameOmega() {
   return omegaBody;
}

public double[] getBodyFrameAngularMomentum() {
   return angularMomentumBody;
}

void updateVectors() {
   double q0 = state[0], q1 = state[2], q2 = state[4],
        q3 = state[6];
   omegaBody[0] = 2*(-q1*state[1]+q0*state[3]+q3*state[5]-
        q2*state[7]);
   omegaBody[1] = 2*(-q2*state[1]-q3*state[3]+q0*state[5]+
        q1*state[7]);
   omegaBody[2] = 2*(-q3*state[1]+q2*state[3]-q1*state[5]+
        q0*state[7]);
   angularMomentumBody[0] = I1*omegaBody[0];
   angularMomentumBody[1] = I2*omegaBody[1];
   angularMomentumBody[2] = I3*omegaBody[2];
}

public void advanceTime() {
   solver.step();
   double norm = 1/Math.sqrt(state[0]*state[0]+state[2]*state[2]+
        state[4]*state[4]+state[6]*state[6]);
   state[0] *= norm;
   state[2] *= norm;
   state[4] *= norm;
   state[6] *= norm;
   updateVectors();
}

public double[] getState() {
   return state;
}
```



```
   public void getRate(double[] state, double[] rate) {
      computeBodyFrameAcceleration(state);
      double sum = 0;
      for(int i = 1;i<9;i += 2) { // sum the q dot values
         sum += state[i]*state[i];
      }
      sum = -2.0*sum;
      // use q components for clarity
      double q0 = state[0], q1 = state[2], q2 = state[4],
           q3 = state[6];
      rate[0] = state[1];
      rate[1] = 0.5*(-q1*wxdot-q2*wydot-q3*wzdot+q0*sum);
      rate[2] = state[3];
      rate[3] = 0.5*(q0*wxdot-q3*wydot+q2*wzdot+q1*sum);
      rate[4] = state[5];
      rate[5] = 0.5*(q3*wxdot+q0*wydot-q1*wzdot+q2*sum);
      rate[6] = state[7];
      rate[7] = 0.5*(-q2*wxdot+q1*wydot+q0*wzdot+q3*sum);
      rate[8] = 1.0; // time rate
   }

   void computeBodyFrameTorque(double[] state) {
      t1 = t2 = t3 = 0;
   }

   void computeBodyFrameAcceleration(double[] state) {
      // use components for clarity
      double q0 = state[0], q1 = state[2], q2 = state[4],
            q3 = state[6];
      double wx = 2*(-q1*state[1]+q0*state[3]+q3*state[5]-q2*state[7]);
      double wy = 2*(-q2*state[1]-q3*state[3]+q0*state[5]+q1*state[7]);
      double wz = 2*(-q3*state[1]+q2*state[3]-q1*state[5]+q0*state[7]);
      computeBodyFrameTorque(state);
      wxdot = (t1-(I3-I2)*wz*wy)/I1; // Euler's equations of motion
      wydot = (t2-(I1-I3)*wx*wz)/I2;
      wzdot = (t3-(I2-I1)*wy*wx)/I3;
   }
}
```

The `RigidBody` class makes use of the `RigidBodyUtil` utility class in the ch17 package to handle quaternion normalization and transformations between space and body frames. The `RigidBodyUtil` class is not listed. The static `spaceToBody` method multiplies a given vector by (17.34) and the static `bodyToSpace` method multiplies the given vector by the inverse.

`FreeRotationApp` and `FreeRotationView` use a subclass of `RigidBody` to display the dynamics of free rotation. The `FreeRotation` subclass does a number of simple housekeeping chores such as computing the principal moments using the formulas for an ellipsoid in Table 17.1. `FreeRotationApp` animates torque-free rotation by extending `AbstractSimulation` and implementing the `doStep` method. These classes are not listed because they are similar to other classes we have studied.

`FreeRotationSpaceView` shown in Listing 17.13 displays the body, the angular momentum vector, and the angular velocity vector in a `Display3DFrame`. The body is represented using an ellipsoid whose principal axes are $(2a, 2b, 2c)$, and the orientation of