$$\dot{\phi} \approx \frac{I_3}{I_s}\frac{w_3}{\cos\theta_0}, \tag{17.49a}$$

and

$$\dot{\phi} \approx \frac{mgl}{I_3\omega_3}, \tag{17.49b}$$

where $\theta_0$ is the angle between the vertical and the axis of gyroscope, and $mgl$ is the weight times the distance from the pivot to the center of mass. Demonstrate both precession rates. ■

## 17.9 ■ PROJECTS

**Project 17.19 Rotating reference frames**
Model the projectile motion of a ball thrown into the air as seen from a rotating platform. Do so by solving the equations of motion in an inertial reference frame and transforming the trajectory to the noninertial rotating frame. ■

**Project 17.20 Falling box**
Do a two-dimensional simulation of a rotating and falling box hitting and rebounding from a floor. If you are bold, do the simulation in three dimensions by adding translational center of mass coordinates to the quaternion state vector. Does the average kinetic energy of translation equal the average energy of rotation over many bounces? ■

## APPENDIX 17A: MATRIX TRANSFORMATIONS

Transformations, such as rotations, can be implemented using matrices, Euler angles, or analytic functions. All that is required is that a transformation provide a rule for mapping points in a domain to points in a range. Some, but not all, transformations have an inverse that reverses this operation. These abstract concepts are used to define the `Transformation` interface in the numerics package. This interface contains the `direct` and `inverse` methods for transforming points. The `clone` method creates a new transformation that is an exact duplicate of the existing transformation. Objects that can make copies of themselves implement the `clone` interface and are said to be *cloneable*.

```
package org.opensourcephysics.numerics;

public interface Transformation extends Cloneable {
   public void direct (double[] point);
   public void inverse (double[] point) throws
        UnsupportedOperationException;
   public Object clone();
}
```

The `Affine3DMatrix` class shown in Listing 17.16 implements the `Transformation` interface. Because rotations are very common, the class implements the `Rotation` convenience method to create a matrix using (17.16). The `direct` and `inverse` methods use the matrix and the inverse matrix to transform a point, respectively. Because a program might not need the inverse (which might not exist) and because an inverse is expensive to compute, we do not compute this matrix until it is needed, in which case the inverse is



computed only once. Note that the inverse is calculated using a numerical method known as *lower upper (LU) matrix decomposition* (see Press et al.). The name LU decomposition comes from the realization that a nonsingular square matrix $\mathcal{A}$ can be decomposed into a product of two matrices $\mathcal{L}$ and $\mathcal{U}$ whose components below and above the diagonal are zero, respectively:

$$\mathcal{A} = \mathcal{L}\mathcal{U} \tag{17.50}$$

We will not describe this technique, but you are encouraged to test that the method in the numerics package works.

**Listing 17.16** The `Affine3DMatrix` class implements the `Transformation` interface using a matrix representation of the affine transformations.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.numerics.*;

public class Affine3DMatrix implements Transformation {
   // transformation matrix
   private double[][] matrix = new double[4][4];
   // inverse transformation matrix if it exists
   private double[][] inverse = null;
   // true if inverse has been computed
   private boolean inverted = false;

   public Affine3DMatrix(double[][] matrix) {
      if(matrix==null) { // identity matrix
         this.matrix[0][0] = this.matrix[1][1] = this.matrix[2][2] =
              this.matrix[3][3] = 1;
         return;
      }
      for(int i = 0;i<matrix.length;i++) { // loop over the rows
         System.arraycopy(matrix[i], 0, this.matrix, 0,
              matrix[i].length);
      }
   }

   public static Affine3DMatrix Rotation(double theta, double[] axis) {
      Affine3DMatrix at = new Affine3DMatrix(null);
      double[][] atMatrix = at.matrix;
      double norm = Math.sqrt(axis[0]*axis[0]+axis[1]*axis[1]+
           axis[2]*axis[2]);
      double x = axis[0]/norm, y = axis[1]/norm, z = axis[2]/norm;
      double c = Math.cos(theta), s = Math.sin(theta);
      double t = 1-c;
      // matrix elements not listed are zero
      atMatrix[0][0] = t*x*x+c;
      atMatrix[0][1] = t*x*y-s*z;
      atMatrix[0][2] = t*x*y+s*y;
      atMatrix[1][0] = t*x*y+s*z;
      atMatrix[1][1] = t*y*y+c;
      atMatrix[1][2] = t*y*z-s*x;
      atMatrix[2][0] = t*x*z-s*y;
      atMatrix[2][1] = t*y*z+s*x;
      atMatrix[2][2] = t*z*z+c;
      atMatrix[3][3] = 1;
```