Chapter 11

Transformations in three dimensions

**Goal:** Learn about translation, rotation, and shearing as linear transformations on the $w = 1$ plane of 4-space, and their matrix representations. Extend the ideas of the last chapter to 3D. Understand the rotation group for 3-space; quaternions and Rodrigues’ formula. The SVD as a tool for computation and understanding of transformations.

**After this chapter, you will know:**

- How to specify basic transformations of 3D space, including translation, rotation, and shearing transformations, as matrix multiplications.
- How to specify basic affine transformations by specifying their values on a few points (or on a few vectors).
- How to build certain important transformations like a rotation about an axis in 3-space, and reflection through a plane in 3-space.
- How to interpolate between matrix transformations when the matrices are close enough.

11.1 **Introduction: analogy with 2D, differences**

Transformations in 3-space are in many ways analogous to those in 2-space:

- Translations can be incorporated by treating 3-dimensional space as
the subset $E^3$ defined by $w = 1$ in the 4-dimensional space of points $(x, y, z, w)$. A linear transformation whose matrix has the form
\[
\begin{bmatrix}
1 & 0 & 0 & a \\
0 & 1 & 0 & b \\
0 & 0 & 1 & c \\
0 & 0 & 0 & 1
\end{bmatrix},
\]
when restricted to $E^3$, acts as a translation by $[a \ b \ c]^T$ on $E^3$.

- If $T$ is any continuous transformation that takes lines to lines, and $O$ denotes the origin of 3-space, then we can define
\[
\hat{T}(x) = T(x) - T(O)
\]
and the result is a line-preserving transformation $\hat{T}$ that takes the origin to the origin. Such a transformation is represented by multiplication by a $3 \times 3$ matrix $M$. Thus to understand line-preserving transformations on 3-space, we can decompose each into a translation (possibly the identity) and a linear transformation of 3-space.

- Projective transformations are similar to those in 2-space; instead of being undefined on a line, they are undefined on a whole plane. Otherwise, they are completely analogous.

- Scale transformations can again be uniform or nonuniform; those that are nonuniform are characterized by three orthogonal invariant directions and three scale-factors rather than just two, but nothing else is significantly different. The matrix for an axis-aligned scale by amounts $a, b,$ and $c$ along the $x, y,$ and $z$ axes respectively, is
\[
\begin{bmatrix}
a & 0 & 0 & 0 \\
b & 0 & 0 & 0 \\
c & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.
\]
Scale transformations in which one or three of $a, b,$ and $c$ is negative reverse orientation: a triple of vectors $v_1, v_2, v_3$ that form a right-handed coordinate system will, after transformation by such a matrix, form a left-handed coordinate system. A uniform scale by a negative number has all three diagonal entries negative, and hence reverses orientation.

- Similarly, shearing transformations continue to leave a line fixed. Points not on this line are moved by an amount that depends on their position relative to the line, but this position is now measured in two dimensions instead of just one.

- Reflections in 2D were either reflections through a point (the transformation $x \mapsto -x$), which turns out to be the same as rotation by an angle $\pi$, or reflections through a line. In 3D, there are reflections
through a point, a line, or a plane. Reflection through a line corresponds to rotation about the line by $\pi$. Reflection through a point is still given by the map $x \mapsto -x$; in contrast to the 2-dimensional case, this map is orientation-reversing. Finally, reflection through a plane is given by the map

$$x \mapsto x - 2(x \cdot n)n$$

where $n$ is the unit normal vector to the plane. This is is algebraically analogous to reflection through a line in two dimensions, but in three dimensions it is orientation-preserving. The matrix for this map is

$$I - 2nn^T$$

but it should come as no surprise at this point that we recommend that you use the expression $I - 2nn^T$ to create a reflection matrix rather than explicitly typing in the matrix entries, which is prone to error.

The most important difference between two and three dimensions arises when we consider rotations. In two dimensions, the set of rotations about the origin corresponds nicely with the unit circle: if $R$ is a rotation, we look at $R(e_1)$, which is a point on the unit circle. This gives a mapping from rotations to the circle; the inverse mapping is given by taking each point $[x, y]^T$ on the unit circle and associating to it the rotation whose matrix is

$$\begin{bmatrix} x & -y \\ y & x \end{bmatrix}$$

for which it’s easy to verify that $e_1$ is sent to $[x, y]^T$. Thus we can say that the set of rotations in two dimensions is a one-dimensional shape: knowing a single number (the angle of rotation) completely determines the rotation for $\text{SO}(2)$ is a 1-dimensional manifold, which is informally, a smooth shape with the property that at every point, there is essentially only one direction in which to move; in the case of the circle, this “direction” is that of increasing or decreasing angle. By contrast, the surface of the earth is a 2-manifold, because at each point there are two independent directions for motion; at any point except the poles, one can take these to be north-south and east-west; any other direction is a combination of these two.

$$1$$

1. Formally, we should say that $\text{SO}(2)$ is a 1-dimensional manifold, which is informally, a smooth shape with the property that at every point, there is essentially only one direction in which to move; in the case of the circle, this “direction” is that of increasing or decreasing angle. By contrast, the surface of the earth is a 2-manifold, because at each point there are two independent directions for motion; at any point except the poles, one can take these to be north-south and east-west; any other direction is a combination of these two.

## 11.2 Special Transformations

In this section, we’ll write down various transformations, including reflections, scaling operations, and shearing operations explicitly; the next sev-
eral sections will discuss rotations, which are considerably more complicated in 3-space than in 2-space. We’ll conclude the chapter with a discussion of interpolating transformations, and some user-interface applications of our understanding of 3-space transformations.

**Translation.** We’ve already mentioned that translations are completely analogous to those in 2-space: we attach a homogeneous coordinate and used $4 \times 4$ matrices; the rightmost column determines the amount by which the origin is translated. For the remainder of this section, we’ll ignore translation and concentrate on linear transformations on $\mathbb{R}^3$, for which the matrix representation is a $3 \times 3$ matrix (which can be thought of as the upper left block of a $4 \times 4$ matrix).

**Shearing.** We saw that in $\mathbb{R}^2$, the matrix $M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ represented a shearing operation. The vector $v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ remained fixed, i.e. $Mv = v$, while the vector $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ got tilted to become $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. This can be generalized to 3-space by simply doing nothing to the $z$-axis; the $3 \times 3$ matrix is then

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$ 

In this case we have a shearing operation that leaves the $xz$-plane ($y = 0$) fixed, but shears points in parallel planes ($y = c$) along $x$ by an amount proportional to $c$. One can read this directly from the matrix: the first and third columns are the same as those of the identity, but the second has $1$ in the first entry, indicating that the $y$-axis gets tilted in the $x$-direction.

**Inline Exercise 11.1:** Write down a shearing matrix that leaves the $xy$-plane fixed, but translates the plane $z = c$ by the vector $\begin{bmatrix} 3c \\ 2c \\ 0 \end{bmatrix}$.

**Reflection.** As we mentioned, reflection through the plane with unit normal vector $n$ is given by the map $x \mapsto x - 2(x \cdot n)$. Factoring out $x$, we get $x \mapsto (I - 2nn^T)x$.

What about reflection in a line? If the direction vector of the line is the unit vector $u$, we want to decompose $x$ into the part that points in the $u$ direction and the part that points perpendicular to $u$. By negating this latter part, we’ll reflect through the line. The decomposition is

$$x = (x \cdot u)u + (x - (x \cdot u)u).$$

We thus want to have

$$T(x) = (x \cdot u)u - (x - (x \cdot u)u).$$
Rewriting the expression on the right using the fact that $v \cdot w = v^T w$, and factoring out $x$, we get

$$
T(x) = (x \cdot u)u - (x - x \cdot u)u
= u(u^T x) - (Ix - uu^T x)
= (uu^T) x - Ix + uu^T x
= (2uu^T - I)x
$$

so that the matrix representing the transformation is just

$$
2u^Tu - I.
$$

Note that this is the negative of the matrix for plane-reflection, so that reflection of a vector $x$ through a line determined by the unit vector $u$ and reflection through the plane perpendicular to $u$ always give opposite results.

**Scaling.** Just as in $\mathbb{R}^2$, diagonal matrices represent scaling operations. If some diagonal entry is zero, that axis is “squashed” to nothing. If all three diagonal entries are nonzero, then the scaling operation is either orientation preserving or orientation reversing; if the number of negative entries is even, it’s orientation-preserving; otherwise it’s orientation-reversing. In $\mathbb{R}^2$, if the two diagonal entries were equal, the scale was a “uniform scale,” i.e., simply a multiple of the identity. In $\mathbb{R}^3$, the same is true if all three diagonal entries are equal; if just two are equal, then one of the coordinate planes undergoes a uniform scaling, while the other axis undergoes a different scaling.

### 11.3 Rotations

Rotations in 3-space are much more complicated than those in the plane; this is therefore a long section. We'll begin with some easily-derived formulas that you’re likely to use often. Then we'll discuss how to use notions like pitch, roll, and yaw (which are called Euler angles) to describe rotations, and how to describe a rotation by giving an axis of rotation and an angle through which to rotate (Rodrigues' formula), and how to find the axis and angle for a rotation (a computation that's also due to Euler). Both of these descriptions of rotations have limitations that make them unsuitable for doing interpolation between rotations, so we'll consider a third way to describe rotations: to each point $q$ of the sphere $S^3$ in four-dimensional space $\mathbb{R}^4$, we can associate a rotation $K(q)$ in a very natural way. There's a small problem, however: the points $q$ and $-q$ of $S^3$ correspond to the same rotation, so our correspondence is 2-to-1. It turns out to still be easy to use this description of rotations to perform interpolation. Following this practical description, we explain, for the mathematically-inclined, the derivation
of the map $K$. Finally, we'll discuss the exponential map, as we did for two-dimensional rotations in Chapter 10, because it's often used in physically-based animation methods.

11.3.1 Analogies between two and three dimensions

Rotations in two dimensions given by matrices of the form

$$\begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}$$

generalize nicely to three dimensions and higher. For instance, we can take the matrix for rotation through the angle $\theta$ in two dimensions and expand it to get

$$R_{xy}(\theta) = \begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix},$$

which is the rotation by angle $\theta$ in the $xy$-plane of 3-space, as we mentioned in Chapter 10. This is also sometimes called rotation about $z$ by the angle $\theta$. The advantage of calling it rotation in the $xy$-plane is that there is an easy mnemonic associated to that: for small values of $\theta$, the unit vector in the $x$-direction is rotated towards the unit vector in the $y$-direction. Corresponding statements are true of $R_{yz}$ and $R_{zx}$, which are written below. There's another advantage: while rotations in 3-space always have an axis (see Section ??) those in 2-space do not (there's no vector in $\mathbb{R}^2$ left invariant by rotation through 30 degrees, for instance), and neither do those in 4-space. But in all cases rotations can be described in terms of planes of rotation. (In 4-space, a rotation can have two orthogonal planes of rotation—see the exercises).

The analogous rotations in the $yz$- and $zx$-planes are given by

$$R_{yz}(\theta) = \begin{bmatrix}
1 & 0 & 0 \\
0 & \cos \theta & -\sin \theta \\
0 & \sin \theta & \cos \theta
\end{bmatrix}$$

and

$$R_{zx}(\theta) = \begin{bmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{bmatrix},$$

which can also be called “rotation about $x$” and “rotation about $y$”, respectively.

In contrast to the two-dimensional situation, where we found that the set of rotations was one-dimensional, in three dimensions the set $SO(3)$ of rotations is three dimensional. It is not, however, just a three-dimensional Euclidean space. One way to see it's three-dimensional is to find a mapping from an easy-to-understand three-dimensional object to $SO(3)$. We'll actually describe three such mappings, each with its own advantages and disadvantages. The first of these mappings is through Euler angles. The mapping will be “mostly one-to-one,” in much the way that the mapping of latitude and longitude to points on the globe is mostly one-to-one: each
Figure 11.1: An airplane that flies along the $z$-axis can change direction by turning to the left or right (yaw), pointing up or down (pitch) or simply spinning about its axis (roll).

point on the international dateline has two longitudes (180E and 180W), and each of the poles has infinitely many longitudes, but each other point corresponds to a single latitude-longitude pair.

11.3.2 Euler angles

Euler angles are a mechanism for creating a rotation through a sequence of three simpler rotations (called roll, pitch, and yaw). This decomposition into three simpler rotations can be done in several ways (yaw-first, roll-first, etc.); unfortunately, just about every possible way is used in some discipline. You’ll need to get used to the idea that there’s no single correct definition of Euler angles.

The most commonly-used definition in graphics describes a rotation $R$ by Euler angles $(\phi, \theta, \psi)$ as a product of three rotations. Representing the rotation $R$ by a matrix $M$, this means that the matrix is a product of three others:

$$M = R_{yz}(\psi)R_{zx}(\theta)R_{xy}(\phi)$$

Thus objects are first rotated by angle $\phi$ in the $xy$-plane, then by angle $\theta$ in the $zx$-plane, and third by angle $\psi$ in the $yz$-plane. The number $\phi$ is called pitch, $\theta$ is called yaw, and $\psi$ is called roll. If you imagine yourself flying in an airplane (see Figure 1.1) along the $x$-axis (with the $y$-axis pointing upwards) there are three direction-changes you can make: turning left or right is called yawing, pointing more up or down is called pitching, and rotating about the direction of travel is called rolling. These three are independent, in the sense that you can apply any one without the others. You can, of course, also apply them in sequence.
Writing this out in matrices, we have
\[
M = \begin{bmatrix}
1 & 0 & 0 \\
0 & \cos \psi & -\sin \psi \\
0 & \sin \psi & \cos \psi
\end{bmatrix}
\begin{bmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{bmatrix}
\begin{bmatrix}
\cos \phi & -\sin \phi & 0 \\
\sin \phi & \cos \phi & 0 \\
0 & 0 & 1
\end{bmatrix}
= \begin{bmatrix}
\cos \theta \cos \phi & -\cos \theta \sin \phi & \sin \phi \\
* & * & -\sin \psi \cos \phi \\
* & * & -\sin \psi \cos \phi
\end{bmatrix}
\] (11.1)

With the proper choice of $\phi, \theta, \text{and } \psi$, such products represent all possible rotations. To see this, we'll show how to find $\phi, \theta, \text{and } \psi$ from a rotation matrix $M$. The $(1,3)$ entry of $M$, according to Equation 1.2, must be $\sin \theta$, so $\theta$ is just the arcsine of this entry; the number thus computed will have a positive cosine. When $\cos \theta \neq 0$, the $(1,1)$ and $(1,2)$ entries of $M$ are positive multiples of $\cos \phi$ and $-\sin \phi$ by the same multiplier; that means that $\phi = \text{atan2}(m_{21}, m_{11})$. We can similarly compute $\psi$ from the last entries in the second and third rows. In the case where $\cos \theta = 0$, the angles $\phi$ and $\psi$ are not unique (much as the longitude of the north pole is not unique). But if we pick $\phi = 0$, we can use the lower-left corner and atan2 to compute a value for $\psi$. The pseudocode is given in listing 1.5, where we are assuming the existence of a $3 \times 3$ matrix class, Mat33, which uses zero-based indexing.

The angles returned are in radians, not degrees.

```csharp
void EulerFromRot(Mat33 m, out double psi, 
                     out double theta, 
                     out double phi)
{
    double costheta = Math.cos(th);
    if (Math.abs(costheta) == 0){
        phi = 0;
        psi = Math.atan2(m[1,0], m[1,1]);
    }
    else{
        phi = atan2(-m[0,1], m[0,0]);
        psi = atan2(-m[1,2], m[2,2]);
    }
}
```

Listing 11.1: Code to convert a rotation matrix to a set of Euler angles

It remains to verify that the values of $\theta, \phi, \text{and } \psi$ determined above produce matrices which, when multiplied together, really do produce the given rotation matrix $M$, but this is a straightforward computation.
Inline Exercise 11.2: Write a short program that creates a rotation matrix from Rodrigues' formula (Equation 1.8 below) and computes from it the three Euler angles. Then use Equation 1.2 to build a matrix from these three angles, and confirm that it is, in fact, your original matrix. Use a random unit direction vector and rotation amount in Rodrigues' formula.

Aside from the special case where \( \cos \theta = 0 \) in the code above, we have a one-to-one mapping from rotations to \((\theta, \phi, \psi)\) triples with \(-\pi/2 < \theta \leq \pi/2\) and \(-\pi < \phi, \psi \leq \pi\). Thus the set of rotations in 3-space is three-dimensional.

In general, you can imagine controlling the attitude of an object by specifying a rotation using \(\theta, \phi\) and \(\psi\). If you change any one of them, the rotation matrix changes a little, so you have a way of maneuvering around on \(SO(3)\). The \(\cos \theta = 0\) situation is tricky, though. If \(\theta = \pi/2\), for instance, we find that multiple \((\phi, \psi)\) pairs give the same result; varying \(\phi\) and \(\psi\) turn out to not produce independent changes in the attitude of the object. This phenomenon, in various forms, is called gimbal lock, and is one reason that Euler angles are not considered an ideal way to characterize rotations.

11.3.3 Axis-angle description of a rotation

One way to rotate 3-space is to pick a particular axis (i.e., a unit vector) and rotate about that direction by some amount. The matrix \(R_{xy}\) does this when the axis is the \(z\)-axis, for instance. We'll show, in Section ??, that every rotation in 3-space is rotation about some axis by some angle. Rodrigues [?] discovered a formula to build a rotation for any axis and angle of rotation. Letting \(\omega\) denote the unit-vector axis of rotation, and \(\theta\) the amount of rotation about \(\omega\), if we write the vector \(\omega\) in coordinates:

\[
\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix},
\]

then the matrix is

\[
M = \begin{bmatrix}
\omega_x^2 + \cos \theta (\omega_y^2 + \omega_z^2) & \omega_x \omega_y (1 - \cos \theta) + \omega_z \sin \theta & \omega_x \omega_z (1 - \cos \theta) + \omega_y \sin \theta \\
\omega_x \omega_y (1 - \cos \theta) + \omega_z \sin \theta & \omega_y^2 + \cos \theta (\omega_x^2 + \omega_z^2) & \omega_y \omega_z (1 - \cos \theta) - \omega_x \sin \theta \\
\omega_x \omega_z (1 - \cos \theta) - \omega_y \sin \theta & \omega_y \omega_z (1 - \cos \theta) + \omega_x \sin \theta & \omega_z^2 + \cos \theta (\omega_x^2 + \omega_y^2)
\end{bmatrix}.
\]

This is called the coordinate form of Rodrigues' formula. We'll see shortly how this arises. In Equation 1.3 one can see certain factors — \(\sin \theta\) and \(1 - \cos \theta\) in particular — appearing multiple times. If we factor these out,
we get
\[ M = \sin \theta \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} + (1 - \cos \theta) \begin{bmatrix} -\omega_y^2 - \omega_z^2 & \omega_x \omega_y & \omega_x \omega_z \\ \omega_x \omega_y & -\omega_x^2 - \omega_z^2 & \omega_y \omega_z \\ \omega_x \omega_z & \omega_y \omega_z & -\omega_x^2 - \omega_y^2 \end{bmatrix} + I. \] (11.4)

where we’ve used the fact that \( \omega \) is a unit vector to simplify. We’ll use the notation \( J_\omega \) to denote the skew-symmetric (i.e., \( J_\omega^T = -J_\omega \)) matrix
\[ J_\omega = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \]
throughout this chapter. With this notation, we can rewrite
\[ M = I + \sin(\theta)J_\omega + (1 - \cos \theta)J_\omega^2. \]

**Inline Exercise 11.3:** (a) Show that for any vector \( \mathbf{v} \), \( J_\omega \mathbf{v} = \omega \times \mathbf{v} \), i.e., \( J_\omega \) is the matrix for the linear transformation \( \mathbf{v} \mapsto \omega \times \mathbf{v} \). (b) Show that \( J_\omega^2 = \begin{bmatrix} -\omega_y^2 - \omega_z^2 & \omega_x \omega_y & \omega_x \omega_z \\ \omega_x \omega_y & -\omega_x^2 - \omega_z^2 & \omega_y \omega_z \\ \omega_x \omega_z & \omega_y \omega_z & -\omega_x^2 - \omega_y^2 \end{bmatrix} \), justifying the third term of Equation 1.3.3. (c) Since \( \omega \) is a unit vector, show that \( J_\omega^2 = \omega \omega^T - I \), hence \( \omega \omega^T = J_\omega^2 + I \).

**Finding an axis and angle from a rotation matrix**

Because we’ll show that every rotation has an axis, we can use Rodrigues’ formula to recover an axis from the matrix, following the approach of Palais and Palais [22]. We know that a rotation matrix must have the form
\[ M = I + \sin(\theta)J_\omega + (1 - \cos \theta)J_\omega^2. \]

for some unit vector \( \omega \) and some angle \( \theta \). The trace of this matrix (the sum of the diagonal entries) is
\[ tr(M) = tr(I + \sin(\theta)J_\omega + (1 - \cos \theta)J_\omega^2) \]
\[ = tr(I) + \sin(\theta)tr(J_\omega) + (1 - \cos \theta)tr(J_\omega^2) \]
\[ = 3 + (1 - \cos \theta)(-2(\omega_x^2 + \omega_y^2 + \omega_z^2)) \]
\[ = 3 + (1 - \cos \theta)(-2) \]
\[ = 1 + 2 \cos \theta, \]
so we can recover the angle of rotation by computing
\[ \theta = \cos^{-1} \left( \frac{tr(M) - 1}{2} \right). \]

Two special cases arise at this point:
• if $\theta = 0$, then any unit vector serves as an “axis” for the rotation, because the rotation is the identity matrix;

• if $\theta = \pi$, then $\sin \theta = 0$, and the general approach for finding the axis that we describe below will yield a zero vector.

In this second case, however, our rotation $M$ must satisfy $M^2 = I$, because $M$ is a rotation by $\pi$, so $M^2$ is a rotation by $2\pi$, hence the identity. From this we find that

$$M(M + I) = M^2 + M = I + M = M + I.$$  

That means that when we multiply $M$ by $M + I$, every column of $M + I$ remains unchanged. So any nonzero column, when normalized, can serve as the axis of rotation. We know that at least one column of $M+I$ is nonzero; otherwise $M = -I$. But this is impossible, because the determinant of $-I$ is $-1$, while that of $M$ is $+1$.

In the general case, when $\sin \theta \neq 0$, we can compute $M - M^T$, to get

$$M - M^T = I + \sin(\theta)J_\omega + (1 - \cos \theta)J_\omega^2 - I + \sin(\theta)J_\omega^T + (1 - \cos \theta)(J_\omega^2)^T.$$  

Because $J_\omega^T = -J_\omega$ and $(J_\omega^2)^T = J_\omega^2$, this simplifies to

$$M - M^T = 2 \sin \theta J_\omega.$$  

Dividing by $2 \sin \theta$ gives the matrix $J_\omega$, from which we can recover $\omega$. The code is given in Listing 1.2.

```c
void RotationToAxisAngle(Mat33 m, out Vector3D omega, out double theta)
{
    // convert a 3x3 rotation matrix m to an axis-angle representation

    theta = Math.acos( (m.trace()-1)/2);
    if (theta is near zero)
    {
        omega = new Vector3D(1,0,0); // any vector works
        return;
    }
    if (theta is near pi)
    {
        int col = column with largest entry of m in absolute value;
        omega = new Vector3D(m[0, col], m[1, col], m[2, col]);
        return;
    }
    else
    {
        mat 33 s = m - m.transpose();
        double x = -s[1,2], y = s[0,2]; z = -s[1,1];
        double t = Math.Sin(theta);
        omega = new Vector3D(x/t, y/t, z/t);
        return;
    }
}
```

Listing 11.2: Code to find the axis and angle from a rotation matrix.
Derivation of Rodrigues' formula

To rotate a vector $v$ about $\omega$ by an amount $\theta$, we'll start with two observations (see Figure 1.2):

- If $v$ is parallel to $\omega$, it should not get rotated at all.
- If $v$ is perpendicular to $\omega$, then $b = \omega \times v$ is also perpendicular to $\omega$, and $v$ should be rotated by $\theta$ towards $b$, i.e., it should be sent to $\cos(\theta)v + \sin(\theta)b$.

For an arbitrary vector $v$, we can decompose $v$ into a sum $v = w + u$ where $w$ is parallel to $\omega$, and $u$ is perpendicular to it, and apply the previous results. The decomposition is simple:

$$w = (v \cdot \omega)\omega \quad \text{and} \quad u = v - (v \cdot \omega)\omega. \tag{11.6}$$

The rotation matrix $M$ must send $v$ to

$$Mv = Mw + Mu$$

$$= w + Mu$$

$$= w + \cos(\theta)u + \sin(\theta)\omega \times u$$

$$= w + \cos(\theta)(v - (v \cdot \omega)\omega) + \sin(\theta)\omega \times (v - (v \cdot \omega)\omega)$$

$$= w + \cos(\theta)(v - (v \cdot \omega)\omega) + \sin(\theta)\omega \times v - \sin(\theta)\omega \times (v \cdot \omega)$$

$$= w + \cos(\theta)(v - (v \cdot \omega)\omega) + \sin(\theta)\omega \times v \quad \text{because} \quad \omega \times \omega = 0$$

$$= (v \cdot \omega)\omega + \cos(\theta)(v - (v \cdot \omega)\omega) + \sin(\theta)\omega \times v$$

We are almost done. We can rewrite $\omega \times v$ as $J_\omega v$ and $(v \cdot \omega)\omega$ as $(\omega T \omega)v$, and from the previous inline exercise, we have $\omega T \omega = J_\omega^2 + I$, so

$$M = (v \cdot \omega)\omega + \cos(\theta)(v - (v \cdot \omega)\omega) + \sin(\theta)\omega \times v$$

$$= (\omega T \omega)v + \cos(\theta)v - \cos(\theta)(\omega T \omega)v + \sin(\theta)J_\omega v$$

$$= (J_\omega^2 + I)v + \cos(\theta)v - \cos(\theta)(J_\omega^2 + I)v + \sin(\theta)J_\omega v$$

$$= J_\omega^2 v + v + \cos(\theta)v - \cos(\theta)J_\omega^2 v - \cos(\theta)v + \sin(\theta)J_\omega v$$

$$= J_\omega^2 v + v - \cos(\theta)J_\omega^2 v + \sin(\theta)J_\omega v$$

$$= v + (1 - \cos \theta)J_\omega^2 v + \sin(\theta)J_\omega v$$

$$= (I + (1 - \cos \theta)J_\omega^2 + \sin(\theta)J_\omega)v.$$
Figure 11.2: Rotation about $\omega$ by an angle $\theta$. (a) If $v$ is parallel to $\omega$, then rotation about $\omega$ has no effect. (b) If $v$ is perpendicular to $\omega$, then it gets rotated to $\cos(\theta)v + \sin(\theta)(\omega \times v)$. 
which is the non-coordinate form of Rodrigues’ formula.

In short, the matrix for our rotation is just

\[
M = I + (1 - \cos \theta)J^2_\omega + \sin(\theta)J_\omega
\]  

(11.8)

A few observations are in order:

• For small \( \theta \), \( M \) is nearly the identity.

• For small \( \theta \), the coefficient of the last term is approximately \( \theta \), while the coefficient of the middle term is \( 1 - \cos(\theta) \approx -\frac{\theta^2}{2} \), so the middle term is far smaller than the last. So to first order, \( M \approx I + \sin \theta J_\omega \).

11.3.4 Body-centered Euler Angles

Suppose we have a model of an airplane whose vertices are stored in an \( n \times 3 \) array \( V \). We’ve rotated the model to some position that we like, by multiplying all the vertices by some rotation matrix \( M \), i.e., we’ve computed \( W = MV \).

We now decide that we’d like to have the airplane model pitch up a little more (as if the pilot had pulled on the joystick). We could apply some Euler-angle rotations to the rotated vertices, i.e., we could compute

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & \cos \psi & -\sin \psi \\
0 & \sin \psi & \cos \psi
\end{bmatrix}
\begin{bmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{bmatrix}
\begin{bmatrix}
\cos \phi & -\sin \phi & 0 \\
\sin \phi & \cos \phi & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\( W \).

The problem is that this would take the already rotated vertices and rotate them first about the world \( z \)-axis, which might point diagonally through the airplane, and then about the world \( y \)-axis, and then about the world \( x \)-axis. It would be very hard to choose \( \psi \), \( \theta \) and \( \phi \) to have the effect we are seeking. Such a transformation would be called a “world-centered rotation”, because the description of the rotation is in terms of the axes of the world coordinate system. We could, instead, compute

\[
M
\begin{bmatrix}
1 & 0 & 0 \\
0 & \cos \psi & -\sin \psi \\
0 & \sin \psi & \cos \psi
\end{bmatrix}
\begin{bmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{bmatrix}
\begin{bmatrix}
\cos \phi & -\sin \phi & 0 \\
\sin \phi & \cos \phi & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

i.e., apply some rotations to the object’s vertices before applying the rotation \( M \) to the object. Such an operation is called a body-centered rotation or object-centered rotation. In this case, performing the rotation we seek is easy: we simply adjust the pitch angle \( \psi \). Of course, if we then want to adjust it further, we have to apply another body-centered rotation, and it appears that we’re destined to accumulate a huge sequence of matrices. One solution is to explicitly compute the product, so that we always have at
most one matrix, plus three others temporarily being adjusted until they, too, can be folded into the matrix. Another approach is to represent matrices with quaternions, as we’ll see below. In general, if $M$ is the current transformation applied to the vertex set $V$, and we alter is to $M_1 = MA$, then $A$ is called a body-centered operation, while if we alter if to $M_2 = CM$, then $C$ is called world-centered.

11.4 Advanced topic: Rotations and quaternions

We now delve into the analog to our discussion of rotations the exponential map for transformations of the plane in Section 10.10. The three-dimensional case is more complicated, but the essential ideas are similar to those of the two-dimensional case.

11.4.1 What’s a rotation?

We’ve spoken of rotations intuitively in the previous sections; now it’s time to define them formally. A rotation is a linear transformation that’s orientation-preserving and length preserving; equivalently, it’s a linear transformation whose matrix $A$ satisfies $A^T A = I$ and has positive determinant. The set of matrices satisfying $A^T A = I$ in dimension $n$ is called $O(n)$; the ones that have determinant $+1$ are called $SO(n)$. The $\det A = +1$ condition is the equivalent, in $n$-dimensions, of requiring that the columns of $A$ form a “right handed” coordinate system in $\mathbb{R}^3$. A unit vector $v$ satisfying $Av = 1v$, i.e., a unit vector $v$ that is left fixed by the rotation, is called an axis of the rotation.

To make these notions concrete: $SO(1)$ is the set of $1 \times 1$ matrices $A = [x]$ with $A^T A = [x][x] = I = [1]$ (i.e., $x^2 = 1$) and determinant one (i.e., $x = 1$). So $SO(1)$ — the set of rotations of the real line — has a single element, namely the identity. This rotation has two “axes”: the unit vectors $[1]$ and $[-1]$ are both left fixed by $A$.

The next example is $SO(2)$, which we’ve already seen corresponds to a circle — for every point $\theta$ of a circle, the matrix

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

is a rotation of the plane, and all rotations have this form. Rotations of $\mathbb{R}^2$ (other than the identity rotation) do not have axes: they leave no unit vector in the plane fixed.

In this section, we’ll discuss the set $SO(3)$ in detail. We’ll discuss a parameterization of $SO(3)$ by an easier-to-understand set (the axis-angle parameterization), and a mapping from $S^3$, the three-dimensional sphere in four dimensions, to $SO(3)$, which has very nice properties. Because $S^3$
has lots of properties analogous to those of the usual sphere in three dimensions, it’s comparatively easy to understand; because the mapping from \( S^3 \) to \( \text{SO}(3) \) is so well-behaved, we can do things like finding a shortest path in \( \text{SO}(3) \) between two elements of \( \text{SO}(3) \) (i.e., finding a sequence of rotations that interpolate nicely between two given rotations) by solving the corresponding problem in \( S^3 \).

### 11.4.2 Every rotation in 3-space has an axis

To prove that every rotation in 3-space has an axis, one approach is to analyze the eigenvalues and eigenvectors of \( M \); this is carried out in the exercises. An alternative is to essentially analyze the code given for determining the axis and angle of a rotation (which was written by assuming that these existed) and verifying that it in fact produces a valid axis of rotation. While one can do this through extensive algebraic manipulations, they’re uninformative.

We therefore include here, for the mathematically curious, a compact proof, due to Palais and Palais \[?,\] based on a clever observation about cross products and rotations, namely that if \( R \) is a rotation, then

\[ R(v \times w) = R(v) \times R(w), \]

i.e., “rotations preserve cross products.” The first step of the proof is to consider the special cases of rotation by 0 or \( \pi \) as we did in the code for axis-and-angle finding; since we already did that analysis, for the remainder of the proof, we’ll assume \( M \) is a rotation matrix whose trace is neither one nor three, so that the skew-symmetric matrix \( A = M - M^T \) is nonzero.

We claim that if \( \nu = [a_{23}, a_{31}, a_{12}]^T \), then

- \( \nu \neq 0 \), and
- \( M\nu = \nu \), so \( \omega = \nu / \|\nu\| \) is an axis for \( M \).

The first claim is self-evident: if all three entries of \( \nu \) were zero, then \( A = 0 \), which is a contradiction. We’ll now show that

- \( MAM^T = M \), and because of this
- \( M\nu = \nu \),

which will complete the proof.

Step 1: Since \( M \) is a rotation, we have \( MM^T = M^T M = I \). So

\[
MAM^T = M(M - M^T)M^T = \begin{align}
&= (MM - MM^T)M^T \\
&= MMM^T - MM^T M^T \\
&= M(MM^T) - (MM^T)M^T \\
&= MI - IM^T \\
&= A.
\]
Step 2: By the definition of $\nu$, we have $A = J_\nu$. So step one tells us that
\[ MJ_\nu M^T = J_\nu. \] (11.15)

The fact that rotations preserve cross products tells us that
\[ M(\nu \times v) = M(\nu) \times M(v) \quad \text{for all } v \in \mathbb{R}^3 \]
Using the rule that $a \times b = J_a b$ twice, we get
\[ M(J_\nu v) = J_{M(\nu)} M(v) \quad \text{for all } v \in \mathbb{R}^3 \]
which in turn implies that
\[ M J_\nu = J_{M(\nu)} M. \]
Multiplying on the right by $M^T = M^{-1}$, we get
\[ MJ_\nu M^T = M_{M(\nu)}. \]
On the other hand, Equation 1.15 tells us that
\[ MJ_\nu M^T = J_\nu. \]
Therefore $M_{M(\nu)} = J_\nu$, which implies that $M(\nu) = \nu$, as promised.

### 11.4.3 Axis-angle formulation

Rodrigues’ formula gives us a mapping
\[ H : S^2 \times (-\pi, \pi] \rightarrow \text{SO}(3) \]
which is neither one-to-one, nor invertible. In particular, $H(\nu, 0) = H(w, 0)$ for any two unit vectors $\nu$ and $w$. And $H(\nu, \theta) = H(-\nu, -\theta)$. By restricting the domain to be $S^2 \times [0, \pi]$, we reduce the latter problem slightly: we only have that $H(\nu, \pi) = H(-\nu, \pi)$. This restricted Rodrigues’ formula, like the Euler angle parameterization of $\text{SO}(3)$, provides something very like latitude-and-longitude coordinates for $\text{SO}(3)$: most rotations have only a single axis-angle description, just as most points of the globe have only a single latitude-longitude description. Certain elements of $\text{SO}(3)$ — rotations through $\pi$ about some axis — have two descriptions, just as points on the international dateline have two latitudes. And the identity rotation has an infinite family of axis-angle descriptions, just as the north and south poles have infinitely many longitudes.

But just as the page of an atlas does not capture the round shape of a globe, the axis-angle description of $\text{SO}(3)$ does not capture its shape. And just as a straight line on a Mercator projection does not correspond to a great circle (write a short program to see what it does look like), going from one axis-angle pair to another by interpolating the axes (via a great circle arc on $S^2$) and the angles independently does not correspond to a “shortest path” in $\text{SO}(3)$; it’s easy to build examples where the interpolated results look very bad. So while the Rodrigues parameterization of $\text{SO}(3)$ is often useful, we’ll now consider a different way to parameterize $\text{SO}(3)$.
11.4.4 Describing rotations via $S^3$

There's a very natural mapping from $S^3$ to $\text{SO}(3)$ (we'll describe how it arises in Section ??), given by

$$K : S^3 \rightarrow \text{SO}(3) : \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \mapsto \begin{bmatrix} a^2 + b^2 - c^2 - d^2 & 2(bc - ad) & 2(ac + bd) \\ 2(ad + bc) & a^2 - b^2 + c^2 - d^2 & 2(cd - ab) \\ 2(bd - ac) & 2(ab + cd) & a^2 - b^2 - c^2 + d^2 \end{bmatrix}.$$  

(11.16)

The map $K$ has several nice properties.

- Great circles on $S^3$ are sent, by $K$, to geodesics (paths of shortest length) in $\text{SO}(3)$.

- There's a way (which we'll describe below) to multiply together elements of $S^3$, just as we can multiply together matrices. If $q_1$ and $q_2$ are two elements of $S^3$, it turns out that

$$K(q_1q_2) = K(q_1)K(q_2),$$

where the multiplication on the left is multiplication of elements of $S^3$ (which we'll define shortly), while the multiplication on the right is the product of matrices in $\text{SO}(3)$.

- The ordinary rules of calculus work for this multiplication on $S^3$ and for the product of matrices in $\text{SO}(3)$, and we can relate these through the derivative of $K$.

When we speak of multiplying elements of $S^3$ (or $\mathbb{R}^4$), we typically refer to the elements as quaternions, just as when we treat points in the plane as complex numbers, we call them complex numbers. WPF has a Quaternion class to represent points of $\mathbb{R}^4$ and the associated multiplication. A Quaternion is simply a structure with $X,Y,Z,$ and $W$ fields; we'll discuss operations on these structures presently. For now, they're just a particular way to represent a point in four-space.

At this point, the mapping $K$ sounds great: we could just do all the computations we need to do in $S^3$, and then project the results to $\text{SO}(3)$ when we're done. This almost works, but there is a small problem: the mapping $K$ is two-to-one: every element of $\text{SO}(3)$ corresponds to two different elements of $S^3$. In fact, you can check from the definition that $K(q) = K(-q)$ for any point $q \in S^3$. It's very tempting to say “Can't we just work with half of $S^3$ then? If we did that, everything would be one-to-one!” Alas, this idea has problems as well, because “half of $S^3$” has a boundary, just as the upper hemisphere of the earth has an edge at the equator; that boundary creates problems of its own. So removing the two-to-one problem merely causes an even worse problem to arise. On the other hand, the mapping $K$ is much nicer than the axis-angle or Euler-angle parameterizations, in the sense
that it’s far more homogeneous: equal volumes in $S^3$ are mapped to equal volumes in $SO(3)$, so there’s no distortion as in the other two mappings.

By the way, although $K$ is not invertible, it’s easy to build a kind of inverse: given a $M \in SO(3)$, we can find an element $q \in S^3$ with $K(q) = M$; we just cannot claim that it is the element with this property. Recall that Rodrigues’ formula tells us that every rotation matrix has the form

$$M = I + \sin(\theta) J_\omega + (1 - \cos \theta) J_\omega^2,$$

where $\omega$ is a unit vector that’s the axis of rotation of the matrix, and $\theta$ is the angle of rotation. And we discussed how to recover the axis, $\omega$ and the angle $\theta$ from an arbitrary rotation matrix (except that when the matrix was $I$, the axis could be any unit vector, and the angle was 0). The associated element $q$ of $S^3$ has $\cos(\theta/2)$ as its first coordinate, and $\sin(\theta/2)\omega$ as its last three coordinates. The case where $\theta = 0$ and $\omega$ is indeterminate presents no problem, because $\sin(\theta/2) = 0$, so the last three entries are all zeroes. There is an ambiguity, however: when we found the axis $\omega$ and the angle $\theta$ we could instead have found $-\omega$ and $-\theta$; those two would have produced $-q$ instead of $q$. So our “inverse” to $K$ really can produce one of two opposite values, depending on choices made in the axis-and-angle computation. To make all this concrete, we’ll give pseudocode for a function $L$ whose domain is $SO(3)$, and whose codomain is pairs of antipodal points in $S^3$; $L$ will act as an inverse to $K$, in the sense that if $M \in SO(3)$ is a rotation matrix, and $L(M) = \{q_1, -q_1\}$ are two elements of $S^3$, then $K(q_1) = K(-q_1) = M$. In the following pseudocode, neither $q_1$ nor $q_2$ is guaranteed to be a continuous function of the entries of the matrix $m$.

```csharp
void RotationToQuaternion(Mat33 m, out Quaternion q1, out Quaternion q2)
{
    // convert a 3x3 rotation matrix m to the two quaternions
    // q1 and q2 that project to m under the map K.
    if (m is the identity)
    {
        q1 = new Quaternion(1, 0, 0, 0);
        q2 = -q1;
        return;
    }

    Vector3D omega = new Vector3D();
    double theta;
    RotationToAxisAngle(m, omega, theta);
    q1 = new Quaternion(Math.cos(theta/2), Math.sin(theta/2)*omega);
    q2 = -q1;
}
```

Listing 11.3: Code to convert a rotation matrix to the two corresponding quaternions

We now have a method for going from $S^3$ to $SO(3)$, and for going from $SO(3)$ back to pairs of elements of $S^3$. If we want to interpolate between
rotations in $\SO(3)$, we’ll need to be able to interpolate between points of $S^3$. We’ll begin by discussing shortest paths on the three-sphere, and then move on to interpolation, and finally apply those ideas, along with the mappings between $S^3$ and $\SO(3)$, to interpolate between rotations.

**Great circles and interpolation in $S^3$**

A great circle — a circle of maximal diameter — in $S^3$ consists of all points of the form

\[ \sin(\theta)v + \cos(\theta)u, \tag{11.17} \]

where $v$ and $u$ are points of $S^3$ with $u \cdot v = 0$ (exactly in analogy with $S^2$). Continuing the analogy with $S^2$, these great circles are “paths of shortest distance” (or geodesics), meaning that they’re the analogue of straight lines in Euclidean space: for any two points (except antipodal points) on a great circle, the shortest path between them is an arc of the great circle (indeed, since there are two arcs of the great circle between the points, the shortest path is the shorter arc).

**Spherical linear interpolation**

Suppose we have two points $q_1$ and $q_2$ of the unit sphere, and that $q_1 \neq -q_2$, i.e., they’re not antipodal. Then there’s a unique shortest path between them, just as on the earth there’s a unique shortest path from the north pole to any point except the south pole. (There is a still a shortest path; the problem is that it’s no longer unique — any line of longitude is a shortest path.)

We’ll now construct a path $\gamma$ that starts at $q_1$ (i.e., $\gamma(0) = q_1$) ends at $q_2$ (i.e., $\gamma(1) = q_2$) and goes at constant speed along the shorter great arc between them. This is called **spherical linear interpolation**, and was first described for use in computer graphics by Shoemake, who called it *slerp*. We’ll use Equation 1.17 to define our curve, with $v$ being $q_1$; for $u$, we’ll find a vector in the plane defined by $q_1$ and $q_2$ that’s orthogonal to $q_1$, which is fairly easy:

```c
{
    assert(dot(q1, q1) == 1);
    assert(dot(q2, q2) == 1);
    // build a vector in q1-q2 plane that’s perp. to q1
    double[4] u = q2 - dot(q1, q2) * q2;
    u = u / length(u); // ...and make it a unit vector.
    double angle = acos(dot(q1, q2));
    return cos(t * angle) * q1 + sin(t * angle) * u;
}
```

Listing 11.4: Code for spherical linear interpolation between two quaternions.
Figure 11.3: We have two rotation matrices, $M_1$ and $M_2$; the first corresponds to a pair of antipodal quaternions, $\pm q_1$, and the second to a different pair of antipodal quaternions $\pm q_2$. Starting at $q_1$, we choose whichever of $q_2$ and $-q_2$ is closer (in this case, $-q_2$) and interpolate between them (as indicated by the thick arc); we then can project the interpolated points to $SO(3)$ to interpolate between $M_1$ and $M_2$.

As the argument to the cosine ranges from $0$ to $\text{angle}$, the result varies from $q_1$ to $q_2$.

**Interpolating between rotations**

We now have all the tools we need to interpolate between rotations. Suppose that $M_1$ and $M_2$ are rotation matrices, with $M_1$ corresponding to the quaternions $\pm q_1$ and $M_2$ corresponding to the quaternions $\pm q_2$, as indicated schematically in Figure 11.3. Starting with $q_1$, we determine which of $q_2$ and $-q_2$ is closer, and then interpolate along an arc between $q_1$ and this point; to find interpolating rotations, we project to $SO(3)$ via $K$.

In pseudocode, we have:

```plaintext
Mat33 RotInterp(Mat33 m1, Mat33 m2, double t)
    // find a rotation that's t of the way from m1 to m2 in SO(3).
    // m1 and m2 must be rotation matrices.
{
    if (m1 * m2.T == -I) {
        Report error; can't interpolate between opposite rotations.
    }
    Quaternion q1, q1p, q2, q2p;
    RotationToQuaternion(m1, q1, q1p);
    RotationToQuaternion(m2, q2, q2p);
    if (Dot(q1, q2) < 0) q2 = q2p;
    Quaternion qi = Quaternion.slerp(q1, q2, t);
```
Listing 11.5: Code to interpolate between two rotations expressed as matrices

With this notion of “interpolate between rotations” or “interpolate between quaternions” in hand, other operations, like blending together three or four rotations, also become possible. Indeed, even operations like the curve-subdivision we did in Chapter 4 become possible in SO(3) instead of $\mathbb{R}^2$. One must be careful, however: while it’s nice to be able to interpolate between quaternions in the same way we construct a segment between points in the plane, the analogy has some weaknesses: in the plane, we can construct points $(1-t)A + tB$ for $t$ between 0 and 1, and they lie between $A$ and $B$. If we use $t > 1$, we get a point “beyond $B$”. On the other hand, if we do spherical linear interpolation between quaternions $q_1$ and $q_2$, as we increase $t$ further and further, the result eventually wraps around the sphere and returns to $q_1$.

Other “obvious” things fail as well. In the plane, we can find the center of a quadrilateral by bisecting opposite sides, and finding the midpoint of the edge between these points. It doesn’t matter which pair of opposite sides we choose — the result is the same, as shown in Figure 1.4. But with quaternions, it’s generally not the same.

**Inline Exercise 11.4:** On the two-sphere, let $A = B = (0,1,0)$, $C = (1,0,0)$, and $D = (0,1,0)$. Compute (by drawing — you should not need to perform any algebra) the center of the quadrilateral $ABCD$ twice, once using each pair of opposite sides, to verify that the results are not the same.

Buss [?] discusses thoroughly the challenges of working with such “affine combinations” of quaternions.

### 11.4.5 The multiplicative structure on the quaternions, and the derivation of the map $K$

Recall from our discussion of rotations of the plane in Chapter 10 that we could regard complex numbers either as entities in an of themselves, or as represented by points in $\mathbb{R}^2$ (the complex number $a + ib$ corresponding to the point $(a,b)$), or as corresponding to $2 \times 2$ matrices of a certain form: the complex number $a + ib$ corresponded to the matrix $\begin{bmatrix} a & -b \\ b & a \end{bmatrix}$. In this form, splitting the complex number into its real an imaginary parts corresponded to writing the matrix in the form

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} = a \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + b \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.$$
Figure 11.4: We can compute the midpoint of the quadrilateral $ABCD$ by finding the midpoints of $AB$ and $CD$ (marked by circles), and then the midpoint of the segment between them, or by doing to same process to the edges $AD$ and $BC$ (whose midpoints are marked by squares); the resulting quadrilateral midpoint (indicated by the diamond) is the same in both cases. This does not happen when we work with quaternions.

Figure 1.5 summarizes these correspondences visually, and the decomposition of a complex number into its real and imaginary parts\(^2\).

There's also an operation, *conjugation*, defined on complex numbers. The conjugate $\bar{z}$ of $z = a + ib$ is $\bar{z} = a - ib$ (i.e., we negate the imaginary part). We can also (thinking of the “point in $\mathbb{R}^2$” version of complex numbers, define the “length” of the number $z = a + ib$ as $|z| = \sqrt{a^2 + b^2}$. This can also be expressed by saying that $|z|^2 = z\bar{z}$.

**Inline Exercise 11.5:** Explicitly multiply out $z\bar{z}$ to verify that it is $a^2 + b^2$.

In the matrix form, this squared-length is exactly the determinant of the matrix. For a unit-length complex number, the inverse, $1/z$ is simple

\[^2\text{Many students, when they first are told about the complex number } i, \text{ object that there is no number whose square is } -1; \text{ in a way, the } 2 \times 2 \text{ matrix form addresses this problem: the } 2 \times 2 \text{ matrices of the form } \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \text{ are a set with operations of multiplication, addition, and multiplicative inverses (except when } a = b = 0 \text{) already defined. And the matrices with } b = 0 \text{ correspond perfectly with the familiar real numbers. One can therefore see these matrices as a new kind of field, on that "contains" the reals, but has an element whose square is } -1 \text{ (i.e., the negative of the multiplicative identity). This set of matrices has all the properties that the complex numbers are supposed to have. From a practical point of view, we could simply define them to be the complex numbers, removing the student objection. A similar argument will apply to the quaternions described below.} \]
Figure 11.5: The various ways of representing complex numbers: in the form \( x + iy \), as points of the plane, as \( 2 \times 2 \) matrices. Each has a decomposition into real and imaginary parts.

\[
\begin{bmatrix}
  x & y \\
  y & x
\end{bmatrix} = x \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + y \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}
\]

\( z = x + iy \)

\( \bar{z} \), because \( z\bar{z} = |z|^2 = 1^2 = 1 \). That means that the inverse in the matrix form is just \( \begin{bmatrix} a & b \\ -b & a \end{bmatrix} \), the transpose. So in the three forms, we have a correspondence: conjugation corresponds to reflection in the \( y \)-axis, which corresponds to matrix transpose.

The critical difference between the complex numbers and simple points of \( \mathbb{R}^2 \) is that we have a way of multiplying complex numbers; in a sense, complex multiplication defines a way of multiplying points of \( \mathbb{R}^2 \).

We'll now see a way to create a similar multiplication on \( \mathbb{R}^4 \), with a great many analogies to the complex numbers.

*Quaternions*, a kind of number first described by Hamilton [1], have the form

\[
a1 + bi + cj + dk,
\]

where \( i, j, \) and \( k \) are symbols, much as \( i \) is a new symbol when we study the complex numbers, and where we've included the symbol 1 for the sake of symmetry. Addition of quaternions is defined term-by-term, so that \( (31 + 4i + 2k) + (21 - 4i + 3j) = 51 + 3j + 2k \), for example. Subtraction is similar. The quaternion \( 0 = 1 + 0i + 0j + 0k \) is the identity element for addition. The set of all quaternions is denoted \( H \), in honor of Hamilton.

There's a natural correspondence between quaternions and elements of \( \mathbb{R}^4 \), namely,

\[
a1 + bi + cj + dk \iff \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}.
\]

Just as complex numbers have a real and imaginary part, we speak of the
real and \textit{vector part} of a quaternion. The real part of \(a_1 + b_1 + c_1 + d_1\) is \(a_1\), and the vector part is \(b_1 + c_1 + d_1\). These correspond, in 4-space, to projection on the \(x\)-axis and on the \(yz\)-subspace, respectively. A \textit{pure vector} quaternion is one of the form \(b_1 + c_1 + d_1\), i.e., one whose real part is zero. The set of pure vector quaternions is denoted \(H_0\).

There’s also a matrix version of quaternions, with a structure that’s analogous to the one for complex numbers, with the correspondence being

\[
a1 + b_1 + c_1 + d_1 \iff \begin{bmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ -c & b & a \end{bmatrix}.
\]

Just as in the complex numbers, we can decompose this into four matrices:

\[
\begin{bmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ -c & b & a \end{bmatrix} = a \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} + b \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix} + c \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} + d \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.
\]

The multiplication on quaternions is defined to correspond to matrix multiplication. It’s thus distributive, and 1 is the multiplicative identity. For \(i, j,\) and \(k\) the rules are

- \(ii = -1\)
- \(ij = k\)
- \(ik = -j\)
- \(ji = -k\)
- \(jj = -1\)
- \(jk = i\)
- \(ki = j\)
- \(kj = -i\)
- \(kk = -1.\)

Note that unlike multiplication of complex numbers, this multiplication is \textit{not} commutative.

**Inline Exercise 11.6:** Multiply the matrix corresponding to \(i\) and the matrix corresponding to \(j\) to verify that their product is the matrix corresponding to \(k\). Then multiply them in the reverse order to verify that you get the matrix for \(-k\), thus verifying two of the entries in the multiplication table for quaternions.

**Inline Exercise 11.7:** Multiply \((3i + 3j + k)(1 - 2j + 2k)\) using the rules for quaternion multiplication. You should get \(1 + 5i - 12j + k\).

**Inline Exercise 11.8:** (a) Compute the square of \(3i + j - k\). (b) Make a generalization about the square of any pure-vector quaternion.

Although multiplication is not commutative, it is associative and distributive, and hence the map

\[
L_q : H \rightarrow H : r \mapsto qr
\]
that left-multiplies a quaternion \( r \) by a fixed quaternion \( q \) is actually linear. The matrix for this transformation is the one given in Equation 1.18.

There's a similar matrix for the transformation “multiply on the right by \( q \),” i.e.

\[
R_q : H \to H : r \mapsto rq.
\]

It's

\[
\begin{bmatrix}
a & -b & -c & -d \\
b & a & d & -c \\
c & -d & a & b \\
d & c & -b & a
\end{bmatrix}.
\]

We'll return to these matrices shortly.

Continuing the analogy with complex numbers, there's also a notion of the length of a quaternion, namely

\[
|a1 + bi + cj + dk| = \sqrt{a^2 + b^2 + c^2 + d^2},
\]

and of the conjugate: the conjugate of \( q = a1 + bi + cj + dk \) is \( \bar{q} = a1 - bi - cj - dk \). The multiplicative inverse of the quaternion \( q \) is \( q^{-1} = (1/|q|)\bar{q} \).

Since a quaternion can be broken into a real part and a vector part, one sometimes writes \((r; v)\) for the quaternion \( r1 + v_xi + v_yj + v_zk \). When written this way, quaternion multiplication has an interesting form:

\[
(rs - v \cdot w; rw + sv + v \times w);
\]

the vector dot-product and cross-product appear as terms in the quaternion product. Because of this, we'll reserve the use of the dot for the vector dot-product, and always write \( q_1q_2 \) for the quaternion product.

The WPF implementation of quaternions lacks a constructor that takes a \textit{double} and a \texttt{Vector3D}, but it would have been natural to include one.

### Quaternions and rotations

Using the real-and-vector form of a unit quaternion, \( q = (r; v) \), consider what happens when you multiply on the left by \( q \) and on the right by \( \bar{q} \); we'll do this twice, once for a real-number quaternion \((1; 0)\) and once for a pure-vector quaternion \((0; w)\). Since \( q \) is a unit quaternion, we know that \( r^2 + v \cdot v = 1 \).

\[
(r; v)(1; 0)(r; -v) = (r - v \cdot 0; r0 + v + v \times 0)(r; -v)
\]

\[
= (r; v)(r; -v)
\]

\[
= (r^2 + v \cdot v; -rv + rv)
\]

\[
= (1; 0)
\]
where the second-to-last step in the simplification follows from Exercise ??.

In short, this operation sends pure-vector quaternions to pure vector quaternions. And because the operations “left-multiply by \( q \) and “right-multiply by \( q \)” are both rotations (you can easily check that their matrices have determinant one and satisfy \( MM^T = I \)), this composition is also a rotation. In fact, it’s exactly the rotation we called \( K \). In matrix form, we’re claiming that the matrix for \( I_q \), multiplied by the matrix for \( R_q \), ends up sending pure-vector quaternions to pure vector quaternions. In fact, the product is

\[
\begin{pmatrix}
a & -b & -c & -d \\
b & a & -d & c \\
c & d & a & -b \\
d & -c & b & a
\end{pmatrix}
\begin{pmatrix}
a & b & c & d \\
b & a & -d & c \\
c & d & a & -b \\
d & -c & b & a
\end{pmatrix} =
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & a^2 + b^2 - c^2 - d^2 & 2(bc - ad) & 2(ac + bd) \\
0 & 2(ad + bc) & a^2 - b^2 + c^2 - d^2 & 2(cd - ab) \\
0 & 2(bd - ac) & 2(ab + cd) & a^2 - b^2 - c^2 + d^2
\end{pmatrix}.
\]

When we consider the way this matrix operates on pure-vector quaternions (the last three coordinates of \( \mathbb{R}^4 \)), we get the lower-right \( 3 \times 3 \) part of the matrix above, i.e.,

\[
\begin{pmatrix}
a^2 + b^2 - c^2 - d^2 & 2(bc - ad) & 2(ac + bd) \\
2(ad + bc) & a^2 - b^2 + c^2 - d^2 & 2(cd - ab) \\
2(bd - ac) & 2(ab + cd) & a^2 - b^2 - c^2 + d^2
\end{pmatrix},
\]

which is the definition of \( K(q) \). It turns out that the axis for the rotation \( K(q) \) is exactly the vector part of \( q \) (normalized), and the angle of rotation \( \theta \) is related to \( a \) by \( a = \cos(\theta/2) \).

We’ll finish this section with a discussion of the exponential map for unit quaternions and for \( SO(3) \), which are closely related.

For the unit quaternions, \( S^3 \), we’ll consider the point 1; the tangent space to \( S^3 \) at 1 is the set of all quaternions perpendicular to 1, i.e., the pure-vector quaternions \( H_0 \) (see Figure 1.6). A typical pure-vector quaternion is \( w = bi + cj + dk \). The exponential map wraps the ray through \( w \) onto a great circle on \( S^3 \), in a way completely analogous to the two- and three-dimensional cases. In particular,

\[
\exp(w) = \cos(|w|) 1 + \frac{\sin |w|}{|w|} w. \tag{11.19}
\]
Figure 11.6: A schematic drawing of $S^3$, in which 1 is at the right, and i, j, and k all lie in a vertical plane. (They really lie in a 3-dimensional plane, but we’re constrained by having only three dimensions for our drawing). The tangent space at 1 consists of vectors pointing in directions perpendicular to 1, i.e., in the pure-vector directions. A ray through a typical pure-vector quaternion exponentiates to a great circle on $S^3$.

\[ \text{fig:quat-exp} \]
Figure 11.7: The exponential map $\exp_1$ from $H_0$ to $S^3$ and the exponential map $\exp_2$ from $\mathfrak{so}(3)$ to $SO(3)$ are defined very similarly, and their domains and codomains are related by the maps $J$ and $K$, as we’ll see.

This exponential map can also be computed by summing $1 + w + \frac{1}{2}w^2 + \ldots$, but the simple form above is far more convenient. And just as in the two-dimensional case with complex numbers, we can take the matrix form of $w$ (Equation 1.18) and compute the matrix exponential — the result will be the matrix form of $\cos(|w|)1 + \frac{\sin(|w|)}{|w|}w$.

By analogy, for $SO(3)$, we’ll consider the point $I$; the tangent space to $SO(3)$ at $I$ is the set of all $3 \times 3$ skew-symmetric matrices, denoted $\mathfrak{so}(3)$ (Exercise ?? shows why this is the tangent space). We thus have two parallel situations (see Figure 1.7): two different exponential maps (which we’ll call $\exp_1 : H_0 \to S^3$ and $\exp_2 : \mathfrak{so}(3) \to SO(3)$) arising from quite similar definitions, although one operates on quaternions and the other on matrices. The domains of the two maps are related (to every pure vector quaternion $w$, there’s a skew-symmetric matrix $J_w$), and the codomains are related by the map $K$; we’ll now investigate how these all fit together.

Let’s start with an example in which all the maps are easy to compute. We’ll take $w = \frac{\pi}{4}i \in H_0$. According to equation 1.19,

$$\exp_1(w) = \cos\left(\frac{\pi}{4}\right)1 + \sin\left(\frac{\pi}{4}\right)i$$

and by direct computation, we can determine that

$$K(\exp_1(w)) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Similarly,

$$J_w = \frac{\pi}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

and

$$\exp_2(J_w) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ 0 & \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix}.$$

It’s now easy to see the relationship: the square of this last matrix is \( K(\exp_1(w)) \); this pattern holds in general, not just for this particular instance, i.e., for any pure vector quaternion \( w \),

\[
\exp_2(J_w)^2 = K(\exp_1(w)).
\]

The following diagram summarizes the result visually: travelling from the upper right to lower right by either of the two paths produces the same result:

\[
\begin{array}{c}
\text{so}(3) \xleftarrow{J} H_0 \\
\exp_2 \downarrow \quad \downarrow \exp_1 \\
S^3 \quad S^3 \\
\downarrow \quad \downarrow K \\
\text{SO}(3) \xrightarrow{M \mapsto M^2} \text{SO}(3)
\end{array}
\]

A summary of conversion among quaternions, rotations, and skew-symmetric matrices

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure vector quaternions, ( H_0 )</td>
<td>Unit quaternions, ( S^3 )</td>
<td>( \exp_1 : w \mapsto \cos |w| I + \frac{\sin |w|}{|w|} w )</td>
</tr>
<tr>
<td>Pure vector quaternions, ( H_0 )</td>
<td>( \text{SO}(3) )</td>
<td>( J : w = w_x i + w_y j + w_z k \mapsto J_w = \begin{bmatrix} 0 &amp; -w_z &amp; w_y \ w_z &amp; 0 &amp; -w_x \ -w_y &amp; w_x &amp; 0 \end{bmatrix} )</td>
</tr>
<tr>
<td>Skew-symmetric matrices, ( \text{so}(3) )</td>
<td>( \text{SO}(3) )</td>
<td>( \exp_2 : J_w \mapsto I + \sin(\theta)J_\omega + (1 - \cos \theta)J_\omega^2 )</td>
</tr>
<tr>
<td>Unit quaternions, ( S^3 )</td>
<td>( \text{SO}(3) )</td>
<td>( K : (r; v) \mapsto I + 2rJ_v + 2J_v^2 )</td>
</tr>
<tr>
<td>( \text{SO}(3) )</td>
<td>( S^3 )</td>
<td>rotation about unit vector ( v ) by ( \theta \mapsto \pm(\cos(\theta/2); \sin(\theta/2)v) ).</td>
</tr>
<tr>
<td>( \text{SO}(3) )</td>
<td>Axis-and-angle</td>
<td>Listing 1.2</td>
</tr>
</tbody>
</table>

11.5 Advanced topic: rotations versus rotation-specifications

We’ve defined a rotation as a transformation having certain properties; in particular, it’s a linear function from \( \mathbb{R}^3 \) to \( \mathbb{R}^3 \) represented by multiplica-
tion by some matrix. Thus, for instance, the transformation represented by multiplication by
\[
\begin{bmatrix}
0 & -1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
can be called “rotation about the \(z\)-axis (or “in the \(xy\)-plane”) by 90 degrees.” But it’s important to understand that multiplying a vector \(v\) by this matrix doesn’t rotate \(v\) about the \(z\)-plane in the sense that this phrase is commonly used. The vector \(v\) is at no point rotated by 10, or 20, or 30 degrees. The function simply takes the coordinates of \(v\) and returns the coordinates of the rotated-by-90-degrees version of \(v\). Indeed, looking at the coordinates returned by the rotation, there’s no way to tell whether they arose as the result of rotation by 90 degrees, -270 degrees, or 450 degrees. This might seem irrelevant, in the sense that we got the rotation we wanted, but when we consider the problem of interpolating rotations, it’s really quite significant. If we attempt to build an interpolation procedure \(\text{interp}(M_1, M_2, t)\) that takes as input two matrices \(M_1\) and \(M_2\) and a fraction \(t\) by which to interpolate them, what should happen when \(M_1\) arises from the operation “rotate not at all” and so does \(M_2\)? Obviously, the interpolated rotation will rotate not at all (i.e., it’ll be the identity rotation). But what if \(M_1\) comes from “rotate not at all” and \(M_2\) comes from “rotate 360 degrees about the \(z\)-axis”? We know that we want the half-way interpolation \((t = 0.5)\) to be “rotate 180 degrees about the \(z\)-axis,” but there’s no way for our procedure to compute this: in both our cases, the matrices \(M_1\) and \(M_2\) are both the identity!

In many situations, the underlying desire is not to interpolate rotation matrices or rotation transformations, but to interpolate the rotation specifications and then compute the transformation associated to the interpolated specification. Unfortunately, interpolating specifications is not so easy. In the case of axis-angle specifications, it’s easy when the axis doesn’t change: one simply interpolates the angle. But interpolating the axis is a trickier matter. And suppose we consider two instances of the problem:

1. Rotation 1: rotate by 0 about the \(x\)-axis; rotation 2: rotate by 90 degrees about the \(x\)-axis.

2. Rotation 1: rotate by 0 about the \(y\)-axis; rotation 2: rotate by 90 degrees about the \(x\)-axis.

Should the halfway interpolated rotation in these two cases be the same or different? The initial and final rotations are identical in the two cases. The only difference is in the specification of the irrelevant direction in the no-angle rotation. Should that make a difference?

Yahia [7] has described a method for axis-angle interpolation in which the axis always has an effect, even when the angle is a multiple of \(2\pi\); aside from this artifact, the method is quite reasonable.
11.6 Interpolating matrix transformations

Despite the claim of the previous section that in general we want to inter-
polate rotation specifications rather than the transformations themselves,
there are several circumstances in which directly acting on the transform-
ations makes sense. For instance, if we’re written a physics simulator
that computes the orientations and positions of bodies at certain times,
but we’d like to “fill in” orientations and positions at intermediate times,
we can ensure that the values provided at the “key times” by the simula-
tor will be fairly close to the neighboring values (i.e., an object isn’t going
to spin 720 degrees between “key times”) by working with small enough
time-steps. Given two such “nearby” transformations, can we interpola-
te between them?

Alexa et al. [?] describe a method for interpolating transformations and
more generally for forming linear combinations of transformations. Their
method works provided that the transformations being combined are “near
enough,” in a sense we’ll describe below.

They begin by observing that since we combine transformations by com-
position, it makes sense to say that if $T$ is a transformation, then “half of
$T$ would be a transformation $S$ with $S \circ S = T$. If $T$ is the “uniformly scale by
3” transformation, then $S$ would be “uniformly scale by $\sqrt{3}$”, for instance.
(It could also be “uniformly scale by $-\sqrt{3}$”; choosing a single answer in a
way that depends continuously on $T$ requires some care; we’ll discuss this
further below). If $T$ were “translate by $(1, 2, 6)$, then $S$ would be “translate
by $(\frac{1}{2}, 1, 3)$, and so on. Generalizing one can define other scalar multiples
of $T$ similarly, and use $\circ$ to denote this operation. For example, we define
$U = \frac{p}{q} \circ T$, where $p$ and $q \neq 0$ are integers, by saying that $U$ composed with
itself $q$ times must be the same as $T$ composed with itself $p$ times. To define
$\alpha \circ T$, where $\alpha$ is not rational, we can approximate $\alpha$ as a limit of rational
numbers $r_1, r_2, \ldots$ and define $\alpha T$ to be $\lim_{i \to \infty} r_i T$. The most general defini-
tion is based on these ideas, but somewhat indirectly, just as the definition
of real powers of real numbers is slightly indirect. The matrix exponential
of Equation ?? has an inverse, the matrix logarithm, once we restrict it to
a suitable domain discussed below. Representing the transformation $T$ by
a matrix $T$, we define $\alpha \circ T$ to be the transformation represented by the
matrix $\exp(\alpha \log T)$, in exact analogy with the definition of powers for real
numbers. Just as with real numbers, where there are problems defining
“the” $n$th root of a negative real number or real-number powers of negative
reals, there are certain matrices for which the logarithm is not well-defined
— roughly those that contain a reflection — and for these the definition
fails.

Because our definition depends on matrices, we’ll now shift from dis-
cussing linear transformations to discussing the matrices for these trans-
formations; the matrix for $R$ is $R$, the matrix for $S$ is $S$, and so on.

The next idea is to define a kind of “sum” of two transformations $R$ and
S, and denoted \( R \oplus S \); the idea is to make the “sum” operation **commutative**, so that \( R \oplus S = S \oplus R \), and to have inverses and an identity for the operation, and to make \( R \oplus S = RS \) if \( R \) and \( S \) commute, i.e., if \( RS = SR \). The insight is that \( \oplus \) can be defined by “interleaving” \( R \) and \( S \), i.e., we “do a little bit of \( R \), then a little bit of \( S \), then a little more \( R \), then a little more \( S \)” and so on.

Formally, we define this in terms of matrices:

\[
R \oplus S = \lim_{n \to \infty} S^\frac{1}{n} R^\frac{1}{n} S^\frac{1}{n} R^\frac{1}{n} S^\frac{1}{n} \ldots R^\frac{1}{n} S^\frac{1}{n}
\]

where each factor appears \( n \) times in the product.

Assuming for the moment that this limit exists, we can derive various properties. First, the identity matrix is the identity element for \( \oplus \); second, because \( \log A^{-1} = -\log A \), we see that \( A^{-1} \oplus A = I \); third, if \( R \) and \( S \) commute, and so do \( R^\frac{1}{n} \) and \( S^\frac{1}{n} \) for integers \( n > 0 \), then \( R \oplus S = RS \). So \( R \oplus S \) is rather similar to ordinary matrix multiplication, in the sense that it has the same identity and inverse operations, and commutes on the same pairs of matrices.

### 11.6.1 Computing \( \alpha \odot T \) and \( R \oplus S \)

Alexa *et al.* recommend computing the matrix exponential by

- computing \( \exp(A/2^j) \), for some large enough value of \( j \), and then
- raising the result to the \( 2^j \) power to get \( \exp(A) \).

This initial scaling-down of \( A \) lets them compute the exponential using just a few terms in a power series for \( \exp \), because when \( A \) is small, large powers of \( A \) rapidly converge to zero. The particular series is one recommended in Golub and van Loan [?], the standard reference for matrix computations.

```plaintext
public Mat MatrixExponential(Mat A)
{
    // find how much to scale down A:
    int j = Math.Max(0, 1 + Math.floor(Math.Log2( ||A|| )));
    A = 2^{-j} * A;

    int q = 6; // truncation of series used for approximation
    Mat D = Mat.Identity();
    Mat N = D;
    Mat X = D;
    int c = 1;
    for(int k = 1; k <= q; k++)
    {
        c = c * (q - k + 1) / (k * (2q - k + 1));
        X = A*X;
        N = N + (-1)^k * c*X;
        D = D + (-1)^k * c*X;
    }
    X = D^{-1} * N;
}```
Listing 11.6: Matrix exponential; we assume the existence of matrix class `Mat` with matrix operations predefined.

They suggest that \( q = 6 \) provides sufficient accuracy for their applications.

For matrix logarithm, they suggest using a Taylor series near the identity; once again, if the matrix \( A \) is far from the identity, one must first adjust it. The idea is to use the fact that

\[
\log A = 2^k \log A^{1/2}
\]

for some sufficiently large value of \( k \). Doing so requires computing repeated square roots, which we describe in a moment. The number \( \epsilon \) in the code determines a threshold below which to stop computations. The make the code robust, the while-loop should be limited to a fixed number of operations to avoid numerical errors associated with small values.

To compute the square root, Alexa et al. recommend a procedure that’s a clever matrix generalization of a standard trick for finding the square root of a positive number \( x \): you guess a root \( r > 0 \) and then average \( r \) with \( x/r \). If \( r > \sqrt{x} \), then \( x/r < \sqrt{x} \), and their average is closer to \( \sqrt{x} \) than the farther one, so you repeat the process using this average as \( r \). Convergence is rapid.
public Mat MatrixSqrt(Mat A)
{
    Mat X = A;
    Mat Y = Mat.Identity();
    while( ∥XX−A∥ > ϵ ) {
        Mat iX = X.Inverse();
        Mat iY = Y.Inverse();
        X = 0.5 * ( X + iY);
        Y = 0.5 * ( Y + iX);
    }
}

Listing 11.8: Matrix square-root

Hawkins and Grimm [?] describe techniques for making the $\oplus$ and $\odot$ operations more numerically stable, and for applying them to the problem of interpolating camera transformations. They make one observation that's of general utility, namely that if the matrices $M$ and $N$ have negative determinants, then the entire logarithm-and-exponent process fails, and one cannot interpolate between them by computing

$$H(t) = ((1 - t) \odot M) \oplus (t \odot N)$$

as one might hope. But in this case, we can pick a matrix $F$ with determinant $-1$ (for example, the identity with its $(1,1)$-entry changed to a $-1$), and interpolate between $FM$ and $FN$; one then multiplies the resulting interpolated matrix by $F^{-1}$ to get the interpolant for $M$ and $N$.

11.7 Virtual track-ball and arc-ball

As an application of our study of the space of rotations, we'll now examine two methods for controlling the attitude of a 3D object that we're viewing. These two user-interface techniques are really a part of the general topic of 3D interaction (see chapter ??), but we discuss them here because they are so closely related to the study of $SO(3)$.

Our standard 3D test program can display geometric objects modeled as meshes. Imagine that we have a fixed mesh $K$ with vertexes $P_0, P_1, \ldots, P_k$. We can, before displaying the mesh $K$, apply a transformation to each of the points $P_i$ to create a new mesh. By displaying this new mesh, we see a transformed version of $K$. If we repeatedly vary the transformation applied to the mesh $K$, we'll see a sequence of new meshes that appears to move over time.

An alternative view of this is that we leave the mesh $K$ fixed, but repeatedly change our virtual camera's position and orientation. We'll take the first approach, here, however. Not surprisingly, the two are closely related: moving the object in one direction is equivalent to moving the virtual camera in the opposite direction, for instance (as long as there's only one object in the world).
Suppose we want to be able to view the object from all sides. We'll assume that it's positioned at the origin, so that applying rotations to it keeps it in the same location, but with a varying attitude. How can we control the viewing direction?

One easy-to-understand metaphor is to imagine the object as being ensnared in a large spherical block of glass (see Figure 1.8). This glass ball is so large that if it were drawn on the display, it would fill up as much of the display as possible. (For a square display, it would touch all four sides of the display). We can now imagine interacting with this virtual sphere by clicking on some point of the sphere, dragging some distance, and releasing. If we clicked first at a point \( P \) and released at the point \( Q \), this is supposed to rotate the sphere so as to make the point \( P \) move to the point \( Q \) along a great-circle arc (i.e., to rotate in the plane defined by \( P \), \( Q \), and the origin).

Of course, when we click on a point of the display with the mouse-cursor, we're not actually clicking on the sphere — what we get is the coordinates of the point on the display surface. This in turn must be used to determine a point of the sphere itself. Suppose, for now, that we know the position \( C \) of the virtual camera, and that in response to a mouse-click, we are given the position of a corresponding point \( S \) on a plane in 3-space that corresponds to our display (see Figure 1.9).

To determine the point \( P \) corresponding to this click, we ask where the ray parameterized by

\[
R(t) = C + t(S - C)
\]

meets the virtual sphere, which we'll assume, for simplicity, is the unit sphere defined by \( x^2 + y^2 + z^2 = 1 \), i.e., the unit sphere, if displayed, would just touch two sides of our display rectangle. For the point \( R(t) \) to lie on the sphere, its coordinates (which we'll call \( r_x, r_y, \) and \( r_z \)) must satisfy the defining equation of the sphere, i.e.

\[
r_x^2 + r_y^2 + r_z^2 = 1.
\]

Alternatively, we can consider the vector from the origin \( O \) to \( R(t) \), i.e., \( C + t(S - C) - O \); this vector must have unit length, i.e., it must satisfy \( (R(t) - O) \cdot (R(t) - O) = 1 \). Letting \( u \) denote \( S - C \), this becomes

\[
(C - O + tu) \cdot (C - O + tu) = 1
\]

which we can simplify and expand; letting \( c = C - O \), we get

\[
(u \cdot u)t^2 + (2c \cdot u)t + c \cdot c = 1
\]

which is a quadratic in \( t \); we solve to get

\[
t = \frac{-c \cdot u \pm \sqrt{(c \cdot u)^2 - (u \cdot u)(c \cdot c)}}{u \cdot u}.
\]

The smaller \( t \)-value — call it \( t_1 \) — corresponds to the first intersection of the ray with the sphere; using this, we can compute the sphere point

\[
P = C + t_1(S - C).
\]
Figure 11.8: The object being viewed is imagined as lying in a large glass sphere. Moving a point on the surface of the sphere moves the object inside.
Figure 11.9: When the user clicks near the upper right corner of the display, we can recover the 3-space coordinates of a corresponding point of the “imaging plane” in 3-space; we’ll use this to determine where a ray from the eye through this point hits the “virtual sphere.”
(It’s possible that both solutions for \( t \) are not real numbers, in which case the ray does not intersect the sphere: the user did not click on the image of the virtual sphere on the display.)

As the user drags the mouse, we can, at each instant, compute the corresponding sphere-point \( Q \) in the same way. From \( P \) and \( Q \), we compute a rotation of the sphere that takes \( P \) to \( Q \) along a great circle; this rotation must be about a unit vector orthogonal to \( P \) and \( Q \), and have magnitude \( \cos^{-1}(P \cdot Q) \); Rodrigues’ formula provides the matrix.

We use this matrix to multiply all vertex coordinates of the original mesh to get a new mesh for display; the resulting operation feels completely natural to many people.

Two problems remain: what happens when the user drags to a point outside the virtual sphere? And what happens when the user’s initial click is outside the virtual sphere?

Various solutions have been implemented. When the user drags outside the virtual sphere, one good solution is to treat the point \( Q \) as being the nearest point on the sphere to the ray that the user is describing; this corresponds to using \( t = \frac{-c \cdot u}{u \cdot u} \) in the quadratic-solving.

When the user clicks outside the virtual sphere, one can treat subsequent mouse-drags as instructions to rotate about the view-direction, like the “rotate object” interaction in most 2D drawing programs.

One problem with the virtual sphere controller described so far is that the action of the controller depends on the first point the user clicked; in a long interaction sequence, this may be gradually forgotten. An improved approach is to treat each mouse-drag event as defining a new motion of the sphere, taking the start point to the end point. Thus a click-and-drag becomes a sequence of mouse positions \( P_0 = P, P_1, P_2, \ldots, P_n = Q \), and the object is rotated by a sequence of virtual sphere rotations from \( P_0 \) to \( P_1 \), followed by the rotation defined by \( P_1 \) and \( P_2 \), and so on.

With this modified version of the virtual sphere, it can be difficult to return to one’s starting position; in trade for this, one gets the advantage that a click-and-drag-in-small-circles motion causes the object to spin about the view direction, which users seem to learn instinctively.

There’s a different approach to virtual-sphere rotation developed by Perlin [?], in which a click-and-drag from \( P \) to \( Q \) rotates the sphere from \( P \) towards \( Q \), but by double the angle used in the virtual sphere. A click at the center of the virtual arcball followed by a drag to the edge of the arc-ball produces not a 90-degree rotation, but a 180-degree rotation. This has the advantage that one can achieve any desired rotation of the object by a single click-and-drag (spins about the view direction, for instance, are generated by dragging from one point near the boundary of the ball to another).
11.8 Further Reading

For the mathematically inclined, the study of $\text{SO}(n)$ is covered in several books [?, ?, ?], and some of the basic properties of $\mathbb{S}^n$, $\text{SO}(n)$, and $\text{O}(n)$ as manifolds are discussed in many introductory books on manifolds [?, ?].

The classic work on quaternions is by Hamilton [?], but more modern expositions [?, ?] are much easier to read.

Quaternions are an instance of a more general phenomenon developed by Grassman [?] in which non-commutative forms of multiplication played a central role. Unfortunately Grassman’s ideas were so confusingly expressed that they were largely ignored by his contemporaries. There has been some renewed interest in them in physics (and some related interest in graphics), with recent developments being given the name geometric algebra [?, ?, ?, ?, ?, ?, ?, ?, ?, ?].

Exercises

**Exercise 11.1:** We computed the matrix for reflection through the line determined by the unit vector $u$ by reasoning about dot products. But this reflection is also identical to rotation about $u$ by 180 degrees. Use the axis-angle formula for rotation to derive the reflection matrix directly.

**Exercise 11.2:** In $\mathbb{R}^n$, suppose that $S$ is a $k$-dimensional subspace with an orthonormal basis $b_1, \ldots, b_k$. Write down a formula for reflection through this subspace.

**Exercise 11.3:** In $\mathbb{R}^n$, what is the matrix for reflection through the subspace spanned by $e_1, \ldots, e_k$, the first $k$ standard basis vectors? In terms of $k$, tell whether this reflection is orientation-preserving or reversing.

**Exercise 11.4:** Let

\[
M = \begin{bmatrix}
0 & 1 & 0 & 0 \\
-1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & -1 & 0
\end{bmatrix}.
\]

Show that there’s no “axis of rotation,” i.e., no unit vector $v$ with $Mv = v$. There are, however, two orthogonal planes of rotation: the $xy$-plane and the $zw$-plane.
Exercise 11.5: Write down the matrices for rotation by 90 degrees in the $xy$-plane and rotation by 90 degrees in the $yz$-plane. Calling these $M$ and $K$, verify that $MK \neq KM$, thus establishing that in general, if $R_1$ and $R_2$ are elements of $SO(3)$, it’s not true that $R_1R_2 = R_2R_1$; this is in sharp contrast to the set $SO(2)$ of $2 \times 2$ rotation matrices, in which any two rotations commute.

Exercise 11.6: In Listing 1.2, we have a condition “if $\theta$ is near $\pi$” which handles the special case of very-large-angle rotations by picking a nonzero column $v$ of $M + I$ as the axis. If $\theta$ is not exactly $\pi$, then $v$ will not quite be an axis. Explain why $v + Mv$ will be a much closer estimate of the axis, using a justification like the one given for the matrix square-root algorithm. Adjust the code in Listing 1.2 to apply this idea repeatedly to produce a very good approximation of the axis.

Exercise 11.7: Consider the parameterization of rotations by Euler angles, with $\theta = \pi/2$. Show that simultaneously increasing $\phi$ and decreasing $\psi$ by the same amount results in no change in the rotation matrix at all.

Exercise 11.8: The second displayed matrix in Equation 1.4 is the square of the first ($J_\omega$); it’s also symmetric. This is not a coincidence. Show that the square of a skew-symmetric matrix is always symmetric.

Exercise 11.9: Find the eigenvalues and all real eigenvectors for $J_\omega$. Do the same for $J_\omega^2$.

Exercise 11.10: Suppose that $A$ is a rotation matrix in $\mathbb{R}^3$. (a) How many eigenvalues does a $3 \times 3$ matrix have? (b) Show that the only real eigenvalue that a rotation matrix can have are $\pm 1$. Hint: a rotation preserves length. (c) Recall that for a real matrix, non-real eigenvalues comes in pairs: if $z$ is an eigenvalue, so is $\bar{z}$. Use this to conclude that $A$ must have either one or three real eigenvalues. (d) Use the fact that if $z$ is a nonzero complex number, then $z(\bar{z}) > 0$, and the fact that the determinant is the product of the eigenvalues to show that if $A$ has a non-real eigenvalue, it also has a real eigenvalue which much be one, and that if $A$ has only real eigenvalues, at least one of them must be one. (e) Conclude that since 1 is always an eigenvalue of $A$, there’s always a nonzero vector $v$ with $Av = v$, i.e., the rotation $A$ has an axis.
Exercise 11.11: Let \( q = a1 + bi + cj + dk \) be a quaternion. By computing \( q_1, q_i, q_j \) and \( q_k \), verify that the matrix for left-multiplication by \( q \) really is the one given in Equation 1.18.

Exercise 11.12: Suppose that \( q = (r; v) \) is a unit quaternion. Show that \( v^T v = J_v^2 + ||v||^2 I = J_v^2 + (1 - r^2)I \).

Exercise 11.13: If \( q = (r; v) \) is a unit quaternion, show that \( L_q R_q \) leaves the pure-vector quaternion \( (0; v) \) fixed, thus showing that it lies along the axis of \( K(q) \) (unless \( v \) is zero, of course).

Exercise 11.14: The skew-symmetric matrix \( J_\omega \) associated to a vector \( \omega \) is the matrix for the linear transformation \( v \mapsto \omega \times v \). Show that every skew-symmetric matrix \( S \) represents the cross-product with some vector \( \omega \), i.e., describe an inverse to the mapping \( \omega \mapsto J_\omega \). Now use this to explain why every skew-symmetric matrix has zero as an eigenvalue, hence \( \det S = 0 \).

Exercise 11.15: In the description of the virtual sphere controller, we used the angle \( \theta = \cos^{-1}(P \cdot Q) \), which involves a dot-product of points rather than vectors. This worked only because we assumed that the center of our virtual sphere was the origin. (a) Suppose it was some other point \( B \); how would we have computed \( \theta \)? (b) Now suppose that the virtual sphere was no longer assumed to be a unit sphere. How would we compute \( \theta \)?

Exercise 11.16: Modify the standard 3D test program to implement virtual-sphere control for viewing.

Exercise 11.17: Modify the standard 3D test program to implement the “incremental virtual sphere” control, and experiment with the spin-by-dragging-small-circles operation. As you multiply together more and more rotations, you may find that the product matrix is no longer a rotation because of roundoff error. Apply the Gram-Schmidt process you learned in linear algebra to the columns of the matrix to make sure it’s a rotation at each stage.

Exercise 11.18: Implement the arc-ball control in the standard 3D test program, and compare it to the virtual sphere controller.
Exercise 11.19: For both the virtual sphere and the arcball controller, using quaternions to represent the rotations is particularly useful, in the sense that accumulating multiple rotations generates sequences of unit quaternions, each of which always corresponds to a rotation matrix. The only problem is that as one accumulates sequences of unit quaternions, one may arrive at a non-unit quaternion because of roundoff; but fixing this is simply a matter of normalizing the vector representing the quaternion, which is easier than applying the Gram-Schmidt process to a matrix. Describe how to implement either algorithm in quaternions.

Exercise 11.20: Rotation about an axis in 3D: given a point $P$ and a direction $v$, describe how to build a transformation on $\mathbb{R}^3$ that rotates by $\theta$ about the line determined by $P$ and $v$ so that the plane through $P$, orthogonal to $v$, rotates counterclockwise by $\theta$ when viewed from the point $P + v$.

Exercise 11.21: Compute the square root of 10 by hand using the code for matrix square root. How fast does it converge to within 0.01 of the correct answer?

Exercises on paths in rotation space

In physically-based animation, we frequently describe the orientation of a body by a rotation matrix or quaternion that changes over time. The derivatives of such things are naturally important. In the following exercises, we show that these derivatives can be expressed in a body-centered form, which makes them much easier to work with.

Exercise 11.22: Suppose that

$$Q : [a, b] \to S^3,$$

where $a < 0 < b$, has $Q(0) = 1$. Then $Q(t) \cdot Q(t) = 1$ for $t \in [a, b]$ (where the dot is ordinary dot product of vectors, not quaternion multiplication). Use this to show that $Q'(t) \cdot Q(t) = 0$; conclude that $Q'(0)$ is some pure vector quaternion $\Omega$. In other words, the tangent vector to any path through $1 \in H$ lies in $H_0$, so the tangent space to $H$ at 1 is $H_0$. 
Exercise 11.23: Suppose that
\[ Q : [a, b] \rightarrow S^3, \]
and \( a < c < b \). Let \( Q_0 = Q(c) \), and define
\[ R : [a - c, b - c] \rightarrow S^3 : t \mapsto Q_0^{-1}Q(t + c). \]

(a) Explain why \( R \) satisfies the hypothesis of the previous exercise, and conclude that \( (Q_0^{-1}Q(c)) \cdot (Q_0^{-1}Q'(c)) = 0 \). (b) Now use \( Q_0 = Q(c) \) to show that \( Q_0^{-1}Q'(c) \in H_0 \), i.e., that \( Q'(c) = Q_0 \Omega \) for some pure vector quaternion \( \Omega \). (c) Applying the same reasoning for any \( c \in [a, b] \), conclude that there’s a function \( \Omega : [a, b] \rightarrow H_0 \) with
\[ Q'(t) = Q(t)\Omega(t) \]
for all \( t \in [a, b] \).

Exercise 11.24: Suppose that
\[ r : [a, b] \rightarrow SO(3), \]
where \( a < 0 < b \), has \( r(0) = I \). (a) Since \( r(s)^T r(s) = I \) (why?), use the product rule for derivatives to show that
\[ (r(s)^T r'(s))^T = -r(s)^T r'(s). \]
(b) Apply this to \( s = 0 \) to conclude that \( r'(0) \in so(3) \). In other words, the tangent vector to any path through \( I \in SO(3) \) lies in \( so(3) \), so the tangent space to \( SO(3) \) at \( I \) is \( so(3) \), the set of skew-symmetric matrices.
Exercise 11.25: Suppose that
\[ r : [a, b] \rightarrow SO(3), \]
and \( a < c < b \). Let \( r_0 = r(c) \), and define
\[ r : [a - c, b - c] \rightarrow S^3 : t \mapsto r_0^{-1}r(t + c). \]

(a) Explain why \( r \) satisfies the hypothesis of the previous exercise, and
conclude that \((r_0^{-1}r(c)) \cdot (r_0^{-1}r'(c)) = 0\). (b) Now use \( r_0 = r(c) \) to show
that \( r_0^{-1}r'(c) \in H_0 \), i.e., that \( r'(c) = r_0w \) for some pure vector quaternion \( w \). (c) Applying the same reasoning for any \( c \in [a, b] \), conclude that
there's a function \( w : [a, b] \rightarrow H_0 \) with
\[ r'(t) = r(t)w(t) \]
for all \( t \in [a, b] \).

Thus the body-centered derivative of a path in \( S^3 \) is a pure vector quaternion, and the body-centered derivative of a path in \( SO(3) \) is a skew-symmetric matrix. The next exercise shows how these are related.

Exercise 11.26: Once again suppose that
\[ Q : [a, b] \rightarrow S^3, \]
is a path in the unit quaternions, and let
\[ r = K \circ Q, \]
so that
\[ r : [a, b] \rightarrow SO(3). \]
Using the previous exercises, we know we can write body-centered forms of the derivatives of \( Q \) and \( r \), i.e., we can write
\[ Q'(t) = Q(t)\Omega(t) \quad \text{and} \quad r'(t) = r(t)w(t) \]
for functions \( \Omega : [a, b] \rightarrow H_0 \) and \( w : [a, b] \rightarrow so(3) \). Show that for all \( a < t < b \),
\[ w(t) = \frac{1}{2}J_\Omega(t). \]