14

Quaternions and Rotation

Why Study Quaternions?

- **They avoid gimbal lock**. Euler angles lose a degree of freedom when two rotation axes align. Quaternions never suffer from this because they represent rotation as a single object in 4D space rather than a sequence of three rotations.
- **They interpolate rotations smoothly**. Smooth, constant-speed rotation between two orientations is hard with matrices or Euler angles. Quaternions make spherical linear interpolation (slerp) simple and numerically robust.
- **They compose rotations efficiently**. Combining two 3D rotations is just quaternion multiplication, which is both faster and more numerically stable than multiplying matrices, and less ambiguous than combining Euler angles.

14.1 Introduction

Quaternions, an extension of the complex numbers, were developed by the Irish mathematician William Rowan Hamilton in 1843 as a notation for modeling 3D mechanics. After Hamilton's death, they gradually fell from favor, displaced by vectors, which proved to be both simpler to use and more widely applicable. However, quaternions had a minor resurgence in the 1920s as a way to describe the spin of electrons and other particles in quantum mechanics, and have been a major element of 3D graphics since the 1980s as a alternative way to implement rotations instead of utilizing Euler angles or matrices [**Sho85**]. In particular, Showmake developed *Slerp* (Spherical Linear intERPolation), which allows

an object to be smoothly interpolated between two orientations. A good introduction to this use of quaternions can be found in [**Len11**] (`https://www.mathfor3dgameprogramming.com/`), and is also the subject of this chapter.

14.2 3D Rotation

Probably the most common way to express 3D rotations is through the use of Euler angles, which define an object's orientation as a sequence of rotations about the x, y, and z axes, called its roll, pitch and yaw, and depicted in Fig. 14.1.

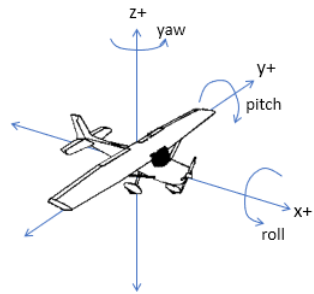


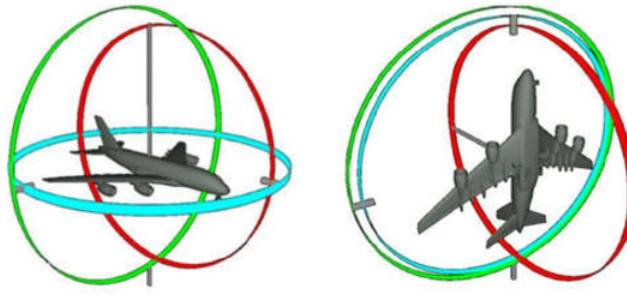
Figure 14.1. The Roll, Pitch, and Yaw of a Plane

A drawback of this approach is that a given orientation can often be defined by different sets of Euler angles. This makes it problematic to reliably undo a series of rotations and return an object to its original position. Also, applying the angle rotations in a different order usually results in different outcomes. Some of this ambiguity can be eliminated by adopting a fixed order for carrying out the rotations, typically around the x-axis, then y, then z, but it doesn't eliminate it altogether.

It's also possible to create a series of rotations with Euler angles that produce a *gimbal lock*, where a degree of freedom is (momentarily) lost. The problem was first observed with gyroscopes in the air and space industry which utilize three concentric rings corresponding to the rotation axes (see Fig. 14.2).

The rings are linked together in a way that reflects the ordering of their axial rotations: the innermost x-ring (red) is linked to the middle y-ring (blue), which connects to the outer z-ring (green). Crucially, turning a ring affects all the rings nested inside it.

When the plane is rotated upwards in Fig. 14.2(b), the y and the z- rings (blue and green) become aligned and a degree of freedom is lost because the x-ring (red) can no longer make the plane roll. Turning the x-ring will cause a yaw in a similar way to a turn to the z-ring (green).



(a) Original Orientation. (b) Pitch 90 Degrees up.

Figure 14.2. The Gimbal Lock Problem

Euler angles are difficult to interpolate: consider the case where you want to gradually turn between 359 and 0 degrees. Linear interpolation may trigger a large rotation, even though the two orientations are almost the same. Ensuring that the shortest path is chosen is easy for one axis, but non-trivial when considering all three angles (for instance, the shortest route between (240, 57, 145) and (35, -233, -270) isn't immediately clear).

These drawbacks of Euler angles are avoided by employing quaternions. As we'll see, a quaternion has a natural geometric interpretation as a rotation about a single axis, there's no possibility of gimbal lock, and interpolation and inverse operations are straight forward. Nevertheless, they do raise some difficulties; one is that quaternions are only really suited to modeling rotations. Other 3D transformations, such as translation, don't have natural quaternion representations. Another issue is the belief that quaternion maths is complicated, but if we restrict ourselves to quaternion-based rotations then they're no harder than complex numbers.

14.3 Vectors Again

We'll explain the maths behind quaternions by utilizing the vector dot and cross product, which we'll briefly describe here once again (you can also find them in section 10.3 of the Computational Geometry topic). These building blocks allow me to define Rodrigues' formula for rotation around an axis, and then it's a short step to quaternion axis-angle rotation.

14.3.1 The Dot Product. The dot product of two n-dimensional vectors $\mathbf{P}$ and $\mathbf{Q}$, written as $\mathbf{P} \cdot \mathbf{Q}$, is the scalar:

$$\mathbf{P} \cdot \mathbf{Q} = \sum_{i=1}^{n} \mathbf{P}_i \mathbf{Q}_i$$

It also satisfies the equation:

$$\mathbf{P} \cdot \mathbf{Q} = \|\mathbf{P}\| \, \|\mathbf{Q}\| \cos \alpha$$

where α is the planar angle between the lines connecting the origin to the points represented by $\mathbf{P}$ and $\mathbf{Q}$ (see Fig. 14.3). A key fact following on from this definition is that two vectors are perpendicular if and only if their dot product is zero.

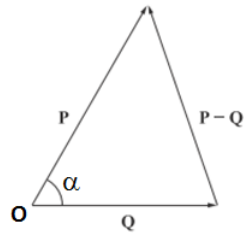


Figure 14.3. The Dot Product Angle

The situation often arises where we need to split a vector $\mathbf{P}$ into its parallel and perpendicular components relative to another vector $\mathbf{Q}$. As shown in Fig. 14.4, if we think of $\mathbf{P}$ as the hypotenuse of a right triangle, then its perpendicular projection onto $\mathbf{Q}$ is the adjacent side for the angle α.

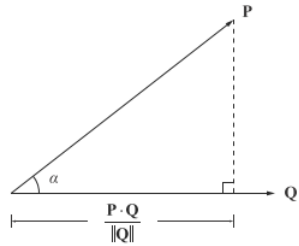


Figure 14.4. The Projection of P onto Q

Trigonometry tells us that the length of the side is $\|\mathbf{P}\| \cos \alpha$, but the dot product gives us a way to calculate this without knowing α:

$$\|\mathbf{P}\| \cos \alpha = \frac{\mathbf{P} \cdot \mathbf{Q}}{\|\mathbf{Q}\|}.$$

To obtain a vector that has this length and is parallel to $\mathbf{Q}$, we multiply by the unit vector $\mathbf{Q}/\|\mathbf{Q}\|$:

$$\mathbf{P}_{\parallel} = \frac{\mathbf{P}\cdot\mathbf{Q}}{\|\mathbf{Q}\|^2}\mathbf{Q}$$

The perpendicular component of $\mathbf{P}$ with respect to $\mathbf{Q}$, $\mathbf{P}_{\perp}$, is obtained by subtracting the parallel component from $\mathbf{P}$:

$$\mathbf{P}_{\perp} = \mathbf{P} - \mathbf{P}_{\parallel} = \mathbf{P} - \frac{\mathbf{P}\cdot\mathbf{Q}}{\|\mathbf{Q}\|^2}\mathbf{Q}$$

14.3.2 The Cross Product. The cross product of $\mathbf{P}$ and $\mathbf{Q}$ returns a vector perpendicular to both of them:

$$\mathbf{P}\times\mathbf{Q} = \langle\, \mathbf{P}_y\mathbf{Q}_z - \mathbf{P}_z\mathbf{Q}_y,\ \mathbf{P}_z\mathbf{Q}_x - \mathbf{P}_x\mathbf{Q}_z,\ \mathbf{P}_x\mathbf{Q}_y - \mathbf{P}_y\mathbf{Q}_x \,\rangle$$

This has many uses in computer graphics, such as the calculation of a surface normal given two tangents to a point.

A commonly used way to generate the product is by evaluating the *pseudo-determinant*:

$$\mathbf{P}\times\mathbf{Q} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \mathbf{P}_x & \mathbf{P}_y & \mathbf{P}_z \\ \mathbf{Q}_x & \mathbf{Q}_y & \mathbf{Q}_z \end{vmatrix}$$

where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are unit vectors parallel to the x, y, and z axes:

$$\mathbf{i} = \langle 1,0,0\rangle, \quad \mathbf{j} = \langle 0,1,0\rangle, \quad \mathbf{k} = \langle 0,0,1\rangle.$$

The top row of the determinant consists of vectors while the remaining entries are scalars. Nevertheless, the usual evaluation produces the cross product:

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \mathbf{P}_x & \mathbf{P}_y & \mathbf{P}_z \\ \mathbf{Q}_x & \mathbf{Q}_y & \mathbf{Q}_z \end{vmatrix} = \mathbf{i}(\mathbf{P}_y\mathbf{Q}_z - \mathbf{P}_z\mathbf{Q}_y) - \mathbf{j}(\mathbf{P}_x\mathbf{Q}_z - \mathbf{P}_z\mathbf{Q}_x) + \mathbf{k}(\mathbf{P}_x\mathbf{Q}_y - \mathbf{P}_y\mathbf{Q}_x).$$

The cross product's length satisfies the equation:

$$\|\mathbf{P}\times\mathbf{Q}\| = \|\mathbf{P}\|\|\mathbf{Q}\|\sin\alpha$$

where α is the angle between the lines connecting the origin to the points represented by $\mathbf{P}$ and $\mathbf{Q}$.

Any nonzero cross product result must be perpendicular to the two vectors multiplied together, but there are two directions that satisfy this requirement. The correct direction is specified by the 'right-hand rule' shown in Fig. 14.5.

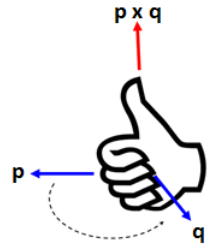


Figure 14.5. The Right-hand Rule for the Cross Product

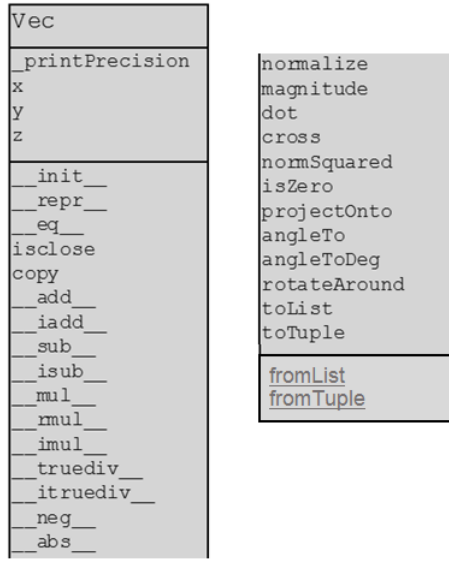


Figure 14.6. Vec Class Diagram

14.4 `Vec`: A vector class in Python

Implementing a 3D vector class in Python is fairly straight forward, so we won't describe much of it here. Its class diagram is given in Fig. 14.6. and the source is online at `Vec.py`.

A Vec object contains x, y, and z values, and supports a standard range of arithmetic operations, as well as dot and cross product. A sample of Vec's capabilities can be seen by calling Vec.py:

```
> python Vec.py
a = Vec(3.00, 4.00, 0.00)
-a = Vec(-3.00, -4.00, 0.00)
|a| = 5.0
b = Vec(0.00, 5.00, 0.00)
a + b = Vec(3.00, 9.00, 0.00)
a . b = 20
a × b = Vec(0.00, 0.00, 15.00)
Normalized 'a' = Vec(0.60, 0.80, 0.00)
Projection of a onto b = Vec(0.00, 4.00, 0.00)
Angle between a and b (deg) = 36.870
Rotate 'a' by 90° around Z axis = Vec(-4.00, 3.00, 0.00)
```

14.4.1 Rotation in 2D. Fig. 14.7 illustrates how a 90-degree counterclockwise rotation of a 2D vector $\mathbf{P}$ in the x-y plane can be accomplished by exchanging its x and y coordinates and negating the new x value. The rotated vector $\mathbf{Q} = \langle -\mathbf{P}_y, \mathbf{P}_x \rangle$ and $\mathbf{P}$ form an orthogonal *basis* for the plane, which means that any other vector can be expressed as a linear combination of these two.

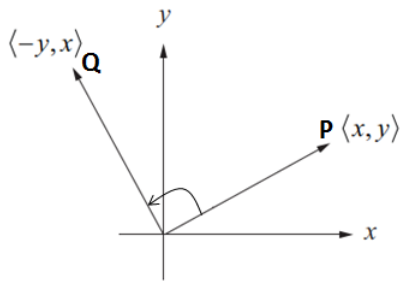


Figure 14.7. Rotation by 90 degrees in the x-y plane

Fig. 14.8 utilizes this basis to show how a vector $\mathbf{P}'$ can be expressed in terms of its components parallel to $\mathbf{P}$ and $\mathbf{Q}$.

Trigonometry lets us write:

$$\mathbf{P}' = \mathbf{P}\cos\theta + \mathbf{Q}\sin\theta$$

which allows us to define $\mathbf{P}'$s x and y components solely in terms of $\mathbf{P}$:

$$\begin{aligned} \mathbf{P}'_x &= \mathbf{P}_x\cos\theta - \mathbf{P}_y\sin\theta \\ \mathbf{P}'_y &= \mathbf{P}_y\cos\theta + \mathbf{P}_x\sin\theta \end{aligned}$$

Or in matrix form:

$$\mathbf{P}' = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \mathbf{P}$$

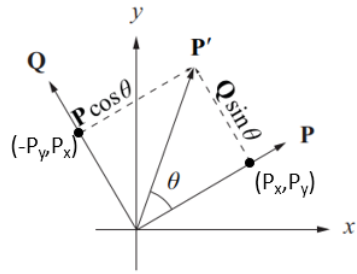


Figure 14.8. $\mathbf{P}'$ in terms of $\mathbf{P}$ and $\mathbf{Q}$

14.5 Rodrigues' Rotation Formula (RRF)

Rodrigues' rotation formula, named after Olinde Rodrigues, is an efficient way to rotate a vector given an axis and angle of rotation. Suppose we want to rotate $\mathbf{v}$ through an angle θ about an axis represented by the unit vector $\mathbf{k}$. We begin by separating $\mathbf{v}$ into its components parallel and perpendicular to $\mathbf{k}$, as in Fig. 14.9.

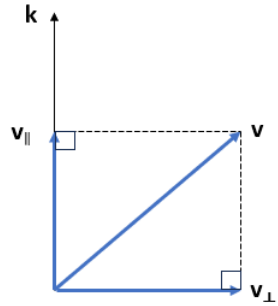


Figure 14.9. $\mathbf{v}$'s Projection ($\mathbf{v}_{\parallel}$) and Perpendicular ($\mathbf{v}_{\perp}$) relative to $\mathbf{k}$

Since $\mathbf{k}$ is a unit vector, the projection of $\mathbf{v}$ onto $\mathbf{k}$ simplifies to:

$$\mathbf{v}_{\parallel} = (\mathbf{k} \cdot \mathbf{v})\mathbf{k}$$

Then the component of $\mathbf{v}$ perpendicular to $\mathbf{k}$ is:

$$\mathbf{v}_{\perp} = \mathbf{v} - (\mathbf{k} \cdot \mathbf{v})\mathbf{k}$$

Since $\mathbf{v}_{\parallel}$ is unchanged during $\mathbf{v}$'s rotation around $\mathbf{k}$, the task reduces to $\mathbf{v}_{\perp}$'s reorientation. This is depicted in Fig. 14.10, which is a view of $\mathbf{v}_{\perp}$ from above, looking down the $\mathbf{k}$ axis.

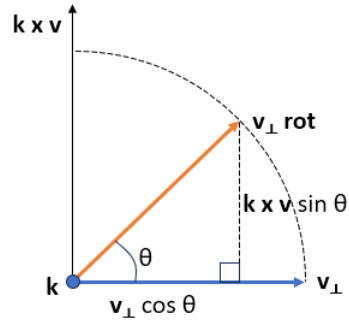


Figure 14.10. Rotating $\mathbf{v}_\perp$, seen from above

The rotation takes place in the plane perpendicular to $\mathbf{k}$. One axis of this plane is $\mathbf{v}_\perp$, and the other can be obtained with $\mathbf{k} \times \mathbf{v}$. Using these two vectors as a basis, the rotation through θ can be represented by:

$$\begin{aligned} \mathbf{v}_\perp \text{ rot} &= \mathbf{v}_\perp \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta \\ &= [\mathbf{v} - (\mathbf{k} \cdot \mathbf{v})\mathbf{k}] \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta \end{aligned}$$

After the rotation of $\mathbf{v}_\perp$ about $\mathbf{k}$, we add back $\mathbf{v}_\parallel$ to get the rotation for $\mathbf{v}$:

$$\begin{aligned} \mathbf{v}_{\text{rot}} &= \mathbf{v}_\parallel + \mathbf{v}_\perp \text{rot} \\ &= \mathbf{v}_\parallel + [\mathbf{v} - (\mathbf{k} \cdot \mathbf{v})\mathbf{k}] \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta \\ &= (\mathbf{k} \cdot \mathbf{v})\mathbf{k} + [\mathbf{v} - (\mathbf{k} \cdot \mathbf{v})\mathbf{k}] \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta \\ &= \mathbf{v} \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta + \mathbf{k}(\mathbf{k} \cdot \mathbf{v})(1 - \cos\theta) \end{aligned}$$

Fig. 14.11 gives a 3D view of the rotation of $\mathbf{v}$, resulting in $\mathbf{v}_{\text{rot}}$.

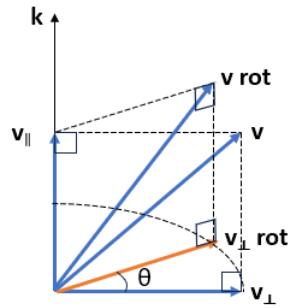


Figure 14.11. Rotating $\mathbf{v}$ by θ around $\mathbf{k}$

This rotation is implemented by rotateAround() in `Vec.py`:

```
def rotateAround(self, axis, theta):
  k = axis.normalize()
  cosA = math.cos(theta)
  sinA = math.sin(theta)
  vterm1 = self*cosA
  vterm2 = k.cross(self)*sinA
  vterm3 = k*(k.dot(self)*(1 - cosA))
  return vterm1 + vterm2 + vterm3  # rotated vector
```

There's a simple example of the effect of rotateAround() included in the output of the call to vec.py given above:

```
a = Vec(3.00, 4.00, 0.00)
   #  many more lines
Rotate 'a' by 90° around Z axis = Vec(-4.00, 3.00, 0.00)
```

The corresponding code is:

```
  a = Vec(3, 4, 0)
  zAxis = Vec(0, 0, 1)
  print("Rotate 'a' by 90° around Z axis =",
                a.rotateAround(zAxis, math.pi/2))
```

Apart from very basic examples, it can be difficult to decide if the vector calculated by rotateAround() is correct, which prompted me to implement VecPlot.py. It utilizes MatPlotlib to plot normalized vectors inside a 3D unit sphere. A visualization of the preceding example is achieved with the code:

```
a = Vec(3, 4, 5).normalize()
zAxis = Vec(0, 0, 1)
b = a.rotateAround(zAxis, math.pi/2)

plotter = VecPlot("Rotate 'a' by 90 degs around Z axis")
plotter.addVector(a, label='a', color='blue')
plotter.addVector(b, label='b', color='red')
plotter.show()
```

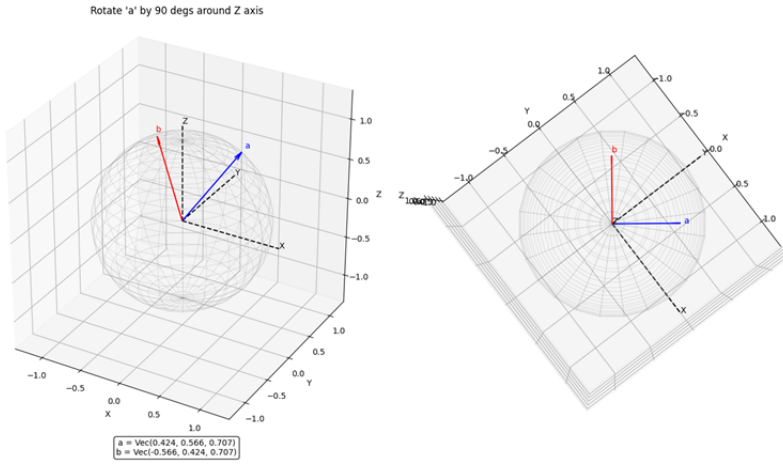
The resulting plot is shown twice in Fig. 14.12.

14.6 Quaternions

A quaternion is defined as

$$\mathbf{q} = (w, x, y, z) = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

but is often written as $\mathbf{q} = s + \mathbf{v}$, where s is a scalar corresponding to the w component, and $\mathbf{v}$ is a vector holding the x, y, z elements.



(a) Original Plot (b) View Looking Down the Z Axis

Figure 14.12. Rotate 'a' by 90° around the z-axis

Quaternion multiplication employs the following rules for $\mathbf{i}, \mathbf{j}$ and $\mathbf{k}$:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1,$$

$$\mathbf{ij} = -\mathbf{ji} = \mathbf{k}, \quad \mathbf{jk} = -\mathbf{kj} = \mathbf{i}, \quad \mathbf{ki} = -\mathbf{ik} = \mathbf{j}.$$

The presence of $\mathbf{i}, \mathbf{j}$ and $\mathbf{k}$ means that we can use operations like cross products and dot products which simplify quaternion multiplication quite a bit. For instance, for $\mathbf{q_1} = s_1 + \mathbf{v_1}$ and $\mathbf{q_2} = s_2 + \mathbf{v_2}$, we have:

$$\begin{aligned}
\mathbf{q_1 q_2} &= (s_1 s_2 - \mathbf{v_1} \cdot \mathbf{v_2}) + (s_1 \mathbf{v_2} + s_2 \mathbf{v_1} + \mathbf{v_1} \times \mathbf{v_2}) \\
\mathbf{q_2 q_1} &= (s_1 s_2 - \mathbf{v_1} \cdot \mathbf{v_2}) + (s_2 \mathbf{v_1} + s_1 \mathbf{v_2} + \mathbf{v_2} \times \mathbf{v_1}) \\
&= (s_1 s_2 - \mathbf{v_1} \cdot \mathbf{v_2}) + (s_1 \mathbf{v_2} + s_2 \mathbf{v_1} - \mathbf{v_1} \times \mathbf{v_2})
\end{aligned}$$

The second multiplication differs due to the non-commutativity of the cross product.

The crucial step in quaternion multiplication is how

$$\begin{aligned}
\mathbf{q_1 q_2} &= (s_1 + \mathbf{v_1})(s_2 + \mathbf{v_2}) \\
&= s_1 s_2 + s_1 \mathbf{v_2} + s_2 \mathbf{v_1} + \mathbf{v_1 v_2}
\end{aligned}$$

becomes

$$\mathbf{q_1 q_2} = s_1 s_2 + s_1 \mathbf{v_2} + s_2 \mathbf{v_1} - \mathbf{v_1} \cdot \mathbf{v_2} + \mathbf{v_1} \times \mathbf{v_2}$$

In other words, how can the quaternion multiplication of $\mathbf{v_1}$ and $\mathbf{v_2}$ be expressed in terms of the dot and cross products:

$$\mathbf{v_1 v_2} = -\mathbf{v_1} \cdot \mathbf{v_2} + \mathbf{v_1} \times \mathbf{v_2}$$

$\mathbf{v_1}$ and $\mathbf{v_2}$ can be written in full as:

$$\begin{aligned}\mathbf{v_1} &= a_1\mathbf{i} + b_1\mathbf{j} + c_1\mathbf{k}\\ \mathbf{v_2} &= a_2\mathbf{i} + b_2\mathbf{j} + c_2\mathbf{k}\end{aligned}$$

Their product is:

$$\begin{aligned}\mathbf{v_1}\mathbf{v_2} &= (a_1\,\mathbf{i} + b_1\,\mathbf{j} + c_1\,\mathbf{k})(a_2\,\mathbf{i} + b_2\,\mathbf{j} + c_2\,\mathbf{k})\\ &= a_1a_2\mathbf{ii} + a_1b_2\mathbf{ij} + a_1c_2\mathbf{ik} +\\ &\quad b_1a_2\mathbf{ji} + b_1b_2\mathbf{jj} + b_1c_2\mathbf{jk} +\\ &\quad c_1a_2\mathbf{ki} + c_1b_2\mathbf{kj} + c_1c_2\mathbf{kk}\end{aligned}$$

The quaternion multiplication rules simplify this to:

$$\begin{aligned}&= a_1a_2(-1) + a_1b_2\mathbf{k} + a_1c_2(-\mathbf{j}) +\\ &\quad b_1a_2(-\mathbf{k}) + b_1b_2(-1) + b_1c_2\mathbf{i} +\\ &\quad c_1a_2\mathbf{j} + c_1b_2(-\mathbf{i}) + c_1c_2(-1)\end{aligned}$$

The scalar and vector elements can be grouped together. The scalar part:

$$-a_1a_2 - b_1b_2 - c_1c_2 = -(a_1a_2 + b_1b_2 + c_1c_2) = -\mathbf{v_1}\cdot\mathbf{v_2}$$

The vector part can be organized into its $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ components:

$$(b_1c_2 - c_1b_2)\mathbf{i} + (c_1a_2 - a_1c_2)\mathbf{j} + (a_1b_2 - b_1a_2)\mathbf{k} = \mathbf{v_1}\times\mathbf{v_2}$$

The left side of this equality can be written as the pseudo-determinant:

$$\begin{vmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k}\\ a_1 & a_2 & a_3\\ b_1 & b_2 & b_3\\ c_1 & c_2 & c_3\end{vmatrix}$$

which highlights its equivalence to the cross product of $\mathbf{v_1}$ and $\mathbf{v_2}$.

Combining the two parts:

$$\mathbf{v_1}\mathbf{v_2} = -\mathbf{v_1}\cdot\mathbf{v_2} + \mathbf{v_1}\times\mathbf{v_2}$$

A quaternion without a scalar is *pure*, and multiplication of pure quaternions can be simplified. When the pure quaternions $\mathbf{v}$ and $\mathbf{u}$ are multiplied together they become:

$$\begin{aligned}\mathbf{vu} &= -\mathbf{v}\cdot\mathbf{u} + \mathbf{v}\times\mathbf{u}\\ \mathbf{uv} &= -\mathbf{u}\cdot\mathbf{v} + \mathbf{u}\times\mathbf{v}\\ &= -\mathbf{v}\cdot\mathbf{u} - \mathbf{v}\times\mathbf{u}\end{aligned}$$

These expressions can be utilized to define the dot product and cross product in terms of the pure quaternion product:

$$\mathbf{v} \times \mathbf{u} = \frac{1}{2}(\mathbf{vu} - \mathbf{uv})$$
$$\mathbf{v} \cdot \mathbf{u} = -\frac{1}{2}(\mathbf{vu} + \mathbf{uv})$$

The *conjugate* of a quaternion $\mathbf{q} = s + \mathbf{v}$, denoted $\bar{\mathbf{q}}$, is $s - \mathbf{v}$. The product of a quaternion $\mathbf{q}$ with its conjugate equals the dot product of $\mathbf{q}$ with itself, and also the square of its magnitude:

$$\mathbf{q}\bar{\mathbf{q}} = \bar{\mathbf{q}}\mathbf{q} = \mathbf{q} \cdot \mathbf{q} = \|\mathbf{q}\|^2 = q^2.$$

This square, usually called the *norm*, can also be expressed as:

$$N\mathbf{q} = \mathbf{q}\bar{\mathbf{q}} = w^2 + x^2 + y^2 + z^2$$

The conjugate can be utilized to define the inverse of a nonzero quaternion:

$$\mathbf{q}^{-1} = \frac{\bar{\mathbf{q}}}{\|\mathbf{q}\|^2}.$$

This can be utilized with the magnitude to obtain:

$$\mathbf{q}\mathbf{q}^{-1} = \frac{\mathbf{q}\bar{\mathbf{q}}}{\|\mathbf{q}\|^2} = 1,$$

and similarly $\mathbf{q}^{-1}\mathbf{q} = 1$.

A *unit quaternion* $\hat{\mathbf{q}}$ is one where $N\mathbf{q} = 1$. If we substitute this into the formula for the inverse, we obtain $\mathbf{q}^{-1} = \hat{\mathbf{q}}$.

$\hat{\mathbf{q}}$ can also be represented by

$$\hat{\mathbf{q}} = \cos\theta + \hat{\mathbf{u}}\sin\theta$$

where $\hat{\mathbf{u}}$ is a unit vector. Also, Euler's formula for complex numbers

$$e^{ix} = \cos x + i \sin x$$

generalizes to unit quaternions:

$$e^{\hat{\mathbf{u}}\theta} = \cos\theta + \hat{\mathbf{u}}\sin\theta = \hat{\mathbf{q}}.$$

From this we can define the power of a unit quaternion:

$$\hat{\mathbf{q}}^t = (\cos\theta + \hat{\mathbf{u}}\sin\theta)^t = \exp(t\hat{\mathbf{u}}\theta) = \cos(t\theta) + \hat{\mathbf{u}}\sin(t\theta).$$

and the logarithm:

$$\log(\hat{\mathbf{q}}) = \log(\cos\theta + \hat{\mathbf{u}}\sin\theta) = \log(\exp(\hat{\mathbf{u}}\theta)) = \hat{\mathbf{u}}\theta.$$

14.7 Rotations with Quaternions

Suppose $\mathbf{k}$ is a unit vector and $\mathbf{v}$ some other vector, then rotating $\mathbf{v}$ counterclockwise around $\mathbf{k}$ by θ radians employs the unit quaternion $\hat{\mathbf{p}}$ (which we'll write as p in this section):

$$p = \cos(\theta/2) + \sin(\theta/2)\,\mathbf{k}$$

p is a unit quaternion because

$$p = \cos\left(\frac{\theta}{2}\right) + 0 \cdot \mathbf{i} + 0 \cdot \mathbf{j} + \sin\left(\frac{\theta}{2}\right)\mathbf{k}$$

So:

$$\|p\|^2 == \cos^2\left(\frac{\theta}{2}\right) + \sin^2\left(\frac{\theta}{2}\right) = 1$$

The rotation of $\mathbf{v}$ utilizes quaternion multiplication with p and its conjugate:

$$\mathbf{v}_{\text{Rot}} = p\mathbf{v}p^{-1} = p\mathbf{v}\bar{p}$$

Proving $\mathbf{v}_{\text{rot}} = p\mathbf{v}\bar{p}$ involves rewriting the right-hand side to produce Rodrigues' rotation formula, with the help of some trigonometry identities:

(a) $\cos(2x) = \cos^2 x - \sin^2 x$

(b) $\sin(2x) = 2\sin x \cos x$

(c) $2\sin^2 x = 1 - \cos(2x)$

(d) $\mathbf{u} \times \mathbf{v} = \frac{1}{2}(\mathbf{uv} - \mathbf{vu})$. (This expression was described in the previous section.)

(e) $\hat{\mathbf{u}}\mathbf{v}\hat{\mathbf{u}} = -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} + \mathbf{v}$. Since $\hat{\mathbf{u}} \cdot \mathbf{v} = -\frac{1}{2}(\hat{\mathbf{u}}\mathbf{v} + \mathbf{v}\hat{\mathbf{u}})$, then $\hat{\mathbf{u}}\mathbf{v} = -2\hat{\mathbf{u}} \cdot \mathbf{v} - \mathbf{v}\hat{\mathbf{u}}$, and so:

$$\begin{aligned}
\hat{\mathbf{u}}\mathbf{v}\hat{\mathbf{u}} &= [-2\hat{\mathbf{u}} \cdot \mathbf{v} - \mathbf{v}\hat{\mathbf{u}}]\hat{\mathbf{u}} \\
&= -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} - \mathbf{v}\hat{\mathbf{u}}\hat{\mathbf{u}} \\
&= -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} - \mathbf{v}(\hat{\mathbf{u}} \times \hat{\mathbf{u}} - \hat{\mathbf{u}} \cdot \hat{\mathbf{u}}) \\
&= -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} - \mathbf{v}(-1) \\
&= -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} + \mathbf{v}
\end{aligned}$$

With these identities, we can expand and rewrite $p\mathbf{v}\bar{p}$ relatively easily. The labeled brackets refer to particular identities listed above.

$$
\begin{aligned}
p\mathbf{v}\bar{p} &= (\cos(\theta/2) + \sin(\theta/2)\mathbf{k})\,\mathbf{v}\,(\cos(\theta/2) - \sin(\theta/2)\mathbf{k}) \\
&= \cos^2(\theta/2)\mathbf{v} + \sin(\theta/2)\cos(\theta/2)\,(\mathbf{kv} - \mathbf{vk}) - \sin^2(\theta/2)\,\mathbf{kvk} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (d) \qquad\qquad (e) \\
&= \cos^2(\theta/2)\mathbf{v} + \sin(\theta/2)\cos(\theta/2)2(\mathbf{k}\times\mathbf{v}) - \sin^2(\theta/2)(-2(\mathbf{k}\cdot\mathbf{v})\mathbf{k} + \mathbf{v}) \\
&= (\cos^2(\theta/2) - \sin^2(\theta/2))\,\mathbf{v} + 2\sin(\theta/2)\cos(\theta/2)(\mathbf{k}\times\mathbf{v}) + 2\sin^2(\theta/2)(\mathbf{k}\cdot\mathbf{v})\mathbf{k} \\
&\qquad\quad (a) \qquad\qquad\quad (b) \qquad\qquad\quad (c) \\
&= (\cos\theta)\,\mathbf{v} + (\sin\theta)(\mathbf{k}\times\mathbf{v}) + (1 - \cos\theta)(\mathbf{k}\cdot\mathbf{v})\mathbf{k} \\
&= \mathbf{v}_{\text{Rot}}
\end{aligned}
$$

In other words, quaternion rotation is equivalent to Rodrigues' rotation. This may make you wonder why we should bother with quaternions at all. Although the two techniques are the same in the math sense, quaternions offer several practical advantages:

(1) **Numerical stability**. When the rotation angle is very small, numerical errors are more likely when using axis-angles. For instance, the direction of the rotation may suddenly change.

(2) **Simple composition**. The composition of multiple axis-angles requires them to be converted into matrices so they can be multiplied, which is both slower and more complex than quaternion composition.

(3) **Interpolation**. The interpolation of axis-angles between two orientations can generate incorrect results due to numerical issues, resulting in extraneous rotations or interpolations that don't follow the shortest path between the start and end points. Quaternions are very well suited to interpolation, as following sections will illustrate.

14.8 A Quaternion class in Python

Our Python quaternion class, `Quat.py`, utilizes `Vec.py` to represent a quaternion as a scalar plus a vector. Quat.py offers a range of basic arithmetic and other operations for quaternions, as well as an extensive range of interpolation methods that we'll describe in later sections. Quat's UML class diagram is given in Fig. 14.13.

Quat.py contains a simple test-rig which rotates an x-axis vector to point along the y-axis by employing a quaternion rotation of 90° around the z-axis. The code:

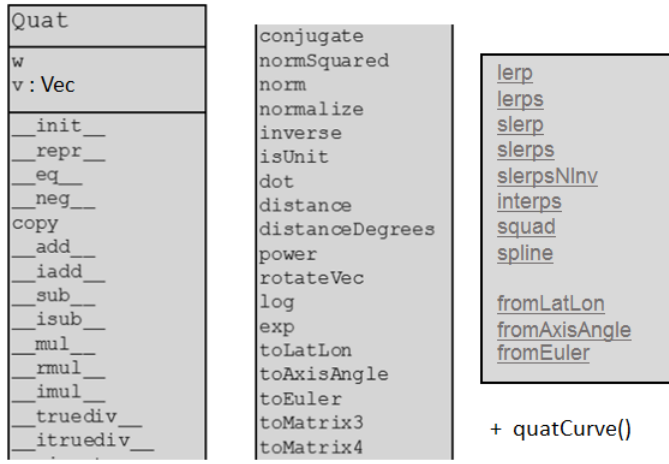


Figure 14.13. Quat Class Diagram

```
  axis = Vec(0, 0, 1)  # z-axis
  print("Create quat from Axis, Angle:\n", axis, ",", 90)
  p = Quat.fromAxisAngle(axis, math.pi/2)
  print("Result:", p)
  v1 = Vec(1, 0, 0) # x-axis
  v2 = p.rotateVec(v1)
  print(f"Quat rotates {v1} --> {v2}")
```

The output:

```
Create quat from Axis, Angle:
 Vec(0.000, 0.000, 1.000) , 90
Result: Quat(0.707, Vec(0.000, 0.000, 0.707))
Quat rotates Vec(1.000, 0.000, 0.000) --> Vec(0.000, 1.000, 0.000)
```

The p unit quaternion is created from the axis-angle by fromAxisAngle():

```
  @staticmethod
  def fromAxisAngle(axis, angleRad):
    half = angleRad/2
    return Quat(math.cos(half),
                axis.normalize()*math.sin(half))
```

This is a translation of:

$$p = \cos(\theta/2) + \sin(\theta/2)\,\mathbf{k}$$

Note that we ensure that the vector (called `axis` in the code) is normalized.

The rotation of a vector (`vec`) is achieved by passing it to p's rotateVec():

```
  def rotateVec(self, vec):
    if not isinstance(vec, Vec):
      raise TypeError("Input must be a Vec instance")
    if not self.isUnit(1e-6):
      print("Warning: Quaternion is not unit length.")

    v = Quat(0, vec)
    return (self * v * self.conjugate()).v
```

Aside from the error checking, the last line implements:

$$\mathbf{v}_{\text{Rot}} = p\mathbf{v}\bar{p}$$

The VecPlot class can be employed to show rotations like the one above. QuatPlot.py contains several examples separated into functions. quats1() replicates the example above:

```
def quats1(plotter):
  v1 = Vec(1, 0, 0)
  q = Quat.fromAxisAngle(z, math.pi/2)
  v2 = q.rotateVec(v1)
  print(f"Rotate {v1} around {q} to {v2}")

  plotter.addVector(v1, label='v1', color='blue')
  plotter.addVector(v2, label='v2', color='red')
  axis, angle =  Quat.toAxisAngle(q)
  ang = math.degrees(angle)
  plotter.addVector(axis, label='q('+str(ang) +')', color='green')
```

The result is shown in Fig. 14.14.

The only new element here is that toAxisAngle() is used to convert the quaternion back into an {axis, angle} pair so the axis can be drawn as an arrow and the angle written as part of its label.

A more complex example is implemented in quats2(). We want to rotate $\mathbf{v} = 2\mathbf{i} + 1\mathbf{j}$ by $\theta = \pi/3$ radians about $\mathbf{u} = \frac{1}{\sqrt{2}}\mathbf{j} + \frac{1}{\sqrt{2}}\mathbf{k}$. This is implemented by:

```
def quats2(plotter):
  v1 = Vec(2, 1, 0)
  u = Vec(0, 1/math.sqrt(2), 1/math.sqrt(2))
  # q = Quat(math.cos(math.pi/6),
             # u * math.sin(math.pi/6))
  q = Quat.fromAxisAngle(u, math.pi/3)
  v2 = q.rotateVec(v1)
  print(f"Rotate {v1} around {q} to {v2}")

  axis, angle = q.toAxisAngle()
  ang = round(math.degrees(angle), 1)
  plotter.addVector(v1, label='v1', color='blue')
```

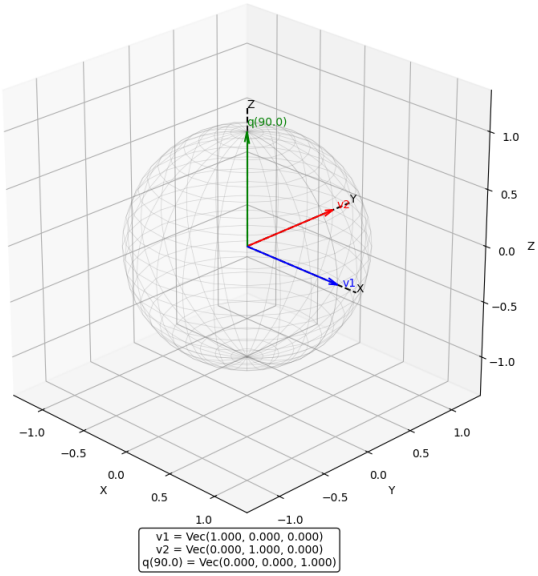


Figure 14.14. Plotting a 90° Rotation around a Quaternion

```
plotter.addVector(v2, label='v2', color='red')
plotter.addVector(axis, label='Ax:'+str(ang), color='green')
```

Note that we've commented out a slightly longer way to define the quaternion since using fromAxisAngle() is more intuitive. The plot is shown in Fig. 14.15.

We've rotated the plot in Fig. 14.15 to better highlight the 60° rotation. Also note that the vectors are all displayed in normalized form since the sphere has unit radius.

14.9 Spherical Linear Interpolation (Slerp)

Since quaternions can be represented by vectors, they're easy to animate through intermediate orientations via interpolation. The simplest approach is *linear interpolation* which utilizes two unit quaternions $\mathbf{q}_0$ and $\mathbf{q}_1$ in

$$\mathbf{q}(t) = (1-t)\mathbf{q}_0 + t\mathbf{q}_1.$$

$\mathbf{q}(t)$ changes smoothly along the line segment connecting $\mathbf{q}_0$ and $\mathbf{q}_1$ as t varies from 0 to 1 (see Fig. 14.16).

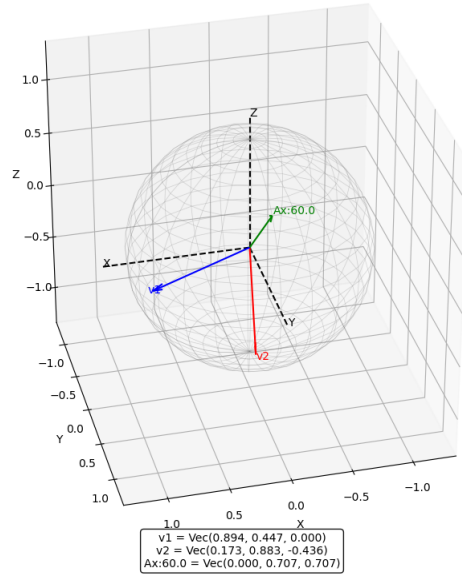


Figure 14.15. Plotting a 60° Rotation around a Quaternion

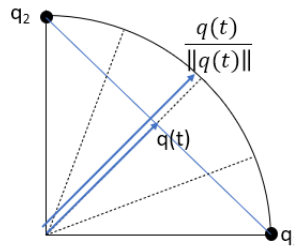


Figure 14.16. Linear Interpolation

Fig. 14.16 also illustrates that $\mathbf{q}(t)$ doesn't maintain a unit length as it moves, but this can be remedied by normalization:

$$\mathbf{q}(t) = \frac{(1-t)\mathbf{q}_0 + t\mathbf{q}_1}{\|(1-t)\mathbf{q}_0 + t\mathbf{q}_1\|}.$$

The implementation of Lerp in `Quat.py` follows this definition:

```
@staticmethod
def lerp(q0, q1, t):
  t = max(0, min(1, t))
  return ((q0 * (1 - t)) + (q1 * t)).normalize()
```

The dotted lines in Fig. 14.16 divide the straight-line route into four equal quarter lengths with a trip time for each equal to 1/4 of the total. However, after normalization the four quarter-arcs are no longer equal in length, being noticeably shorter at the two ends. Consequently, the object speeds up during the middle part of the interpolation.

What we really want is a function that interpolates between the quaternions, preserves unit length, and sweeps through the angle between them at a *constant* rate. We'll utilize Fig. 14.17 to define these requirements more mathematically.

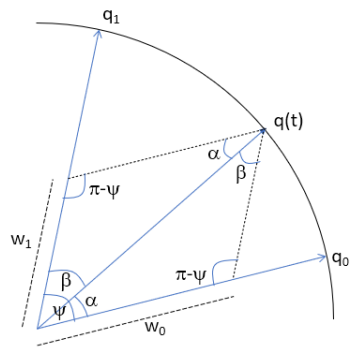


Figure 14.17. Spherical Linear Interpolation

Given the unit quaternions $\mathbf{q}_0$, $\mathbf{q}_1$, and parameter t:

$$\psi = \cos^{-1}(\mathbf{q}_0 \cdot \mathbf{q}_1)$$

This assumes that the quaternion dot product is defined in the same way as the dot product of 4D vectors:

$$\mathbf{q}_0 \cdot \mathbf{q}_1 \;=\; |\mathbf{q}_0||\mathbf{q}_1|\cos(\psi)$$

where ψ is the angle between them. However, since these are unit quaternions, the equation simplifies to:

$$\mathbf{q}_0 \cdot \mathbf{q}_1 = \cos(\psi)$$

Using ψ:

$$\alpha = t\psi; \quad \beta = (1-t)\psi$$

The function calculates a weighted sum:

$$\mathbf{q}(t) = w_0\mathbf{q}_0 + w_1\mathbf{q}_1$$

This can be rewritten by applying the Law of Sines to the triangles in Fig. 14.17:

$$\frac{\sin\alpha}{w_1} = \frac{\sin\beta}{w_0} = \frac{\sin(\pi-\psi)}{1} = \sin\psi$$

and so

$$w_0 = \sin\beta/\sin\psi \quad w_1 = \sin\alpha/\sin\psi$$

The interpolation function becomes:

$$\mathbf{q}(t) = \frac{\mathbf{q}_0\sin\beta + \mathbf{q}_1\sin\alpha}{\sin\psi}$$

If we replace α and β:

$$\mathbf{q}(t) = \frac{\mathbf{q}_0\sin((1-t)\psi) + \mathbf{q}_1\sin(t\psi)}{\sin\psi}$$

The Slerp implementation in :

```
@staticmethod
def slerp(q0, q1, t):
  t = max(0, min(1, t))
  q0 = q0.normalize()
  q1 = q1.normalize()

  # get cosine of angle between quats
  dotCos = q0.dot(q1)
  if dotCos < 0:
    # negate q1 to take shorter path
    q1 = -q1
    dotCos = -dotCos

  # Clamp dotCos to avoid numerical issues with acos
  dotCos = max(-1.0, min(1.0, dotCos))

  # If quaternions are very close, use linear interpolation
  if dotCos > 0.9995:
    return Quat.lerp(q0, q1, t)

  # Use more numerically stable formulation
  theta = math.acos(dotCos)
  sinTheta = math.sin(theta)
  t_theta = theta * t
  s0 = math.cos(t_theta) - (math.sin(t_theta) * dotCos)/sinTheta
  s1 = math.sin(t_theta) / sinTheta
  return (s0*q0 + s1*q1).normalize()
```

One complication is that as ψ approaches zero (called `theta` in the code), the estimation may lose accuracy, so the implementation switches to linear interpolation.

The interpolation at the end of the function uses a numerically more stable version of $\sin(\psi(1-t))/\sin\psi$:

$$\begin{aligned}\frac{\sin\psi(1-t)}{\sin\psi} &= \frac{\sin(\psi-\psi t)}{\sin\psi}\\ &= \frac{\sin\psi\cdot\cos\psi t - sin\psi t\cdot\cos\psi}{\sin\psi}\\ &= \cos\psi t - \frac{\sin\psi t\cdot\cos\psi}{\sin\psi}\end{aligned}$$

In summary:

$$\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,t) = (\cos\psi t - \frac{\sin\psi t\cdot\cos\psi}{\sin\psi})\mathbf{q}_0 + \frac{\sin(\psi t)}{\sin\psi}\mathbf{q}_1$$

Slerp() has the following properties:

(a) When $t=0$, $\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,0)=\mathbf{q}_0$.

(b) When $t=1$, $\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,1)=\mathbf{q}_1$.

(c) For all t, $|\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,t)|=1$

(d) Slerp()'s speed remains constant, which can be expressed as:

$$\left|\frac{d}{dt}\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,t)\right| = |\psi|$$

(a) and (b) can be shown by plugging $t=0$ and $t=1$ into the function. Proving (c) and (d) are a bit more involved, but here are two hints. The proof of (c) utilizes the fact that for unit vectors $\mathbf{v}$ and $\mathbf{w}$ we have:

$$\begin{aligned}|\alpha\mathbf{v}+\beta\mathbf{w}|^2 &= (\alpha\mathbf{v}+\beta\mathbf{w})\cdot(\alpha\mathbf{v}+\beta\mathbf{w}) = \alpha^2\mathbf{v}\cdot\mathbf{v} + 2\alpha\beta\mathbf{v}\cdot\mathbf{w} + \beta^2\mathbf{w}\cdot\mathbf{w}\\ &= \alpha^2 + 2\alpha\beta\cos\psi + \beta^2\end{aligned}$$

This expression comes in useful for rewriting the right-hand side of the Slerp() equation.

The proof of (d) starts with the observation:

$$\begin{aligned}\text{Slerp}(\mathbf{q}_0,\mathbf{q}_t,t) &= \frac{\sin(\psi(1-t))}{\sin\psi}\mathbf{q}_0 + \frac{\sin(\psi t)}{\sin\psi}\mathbf{q}_1\\ &= \frac{1}{\sin\psi}[\sin(\psi(1-t))\mathbf{q}_0 + \sin(\psi t)\mathbf{q}_1]\end{aligned}$$

The derivative is therefore:

$$\frac{d}{dt}\text{Slerp}(\mathbf{q}_0,\mathbf{q}_t,t) = \frac{1}{\sin\psi}[-\psi\cos(\psi(1-t))\mathbf{q}_0 + \psi\cos(\psi t)\mathbf{q}_1]$$

If you're still in the dark, this approach is based on material prepared by Justin Wyss-Gallifent for his course on 'Mathematics and Geometry for Computer Graphics' available at `https://www.math.umd.edu/~immortal/MATH431/`.

14.10 Visualizing Lerp and Slerp

As we've seen, normalized linear interpolation (Lerp) has the drawback that it doesn't trace out the arc between the starting and ending quaternions at a constant rate. This is remedied by switching to spherical linear interpolation (Slerp). One way to visualize this difference is to calculate a series of interpolated quaternions using both approaches, and plot their turning angles.

interps.py sets the starting and ending quaternions to be a y-axis vector with a initial rotation of 0° and a final rotation of 180°. Their Lerp and Slerp interpolations are divided into 100 steps, and the resulting angles relative to a reference direction are plotted in Fig. 14.18.

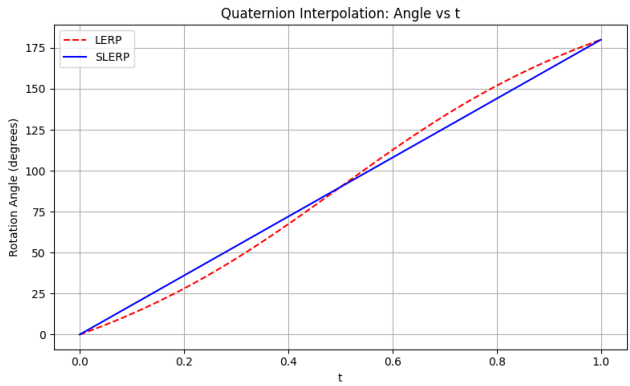


Figure 14.18. Angle Interpolations for Lerp and Slerp

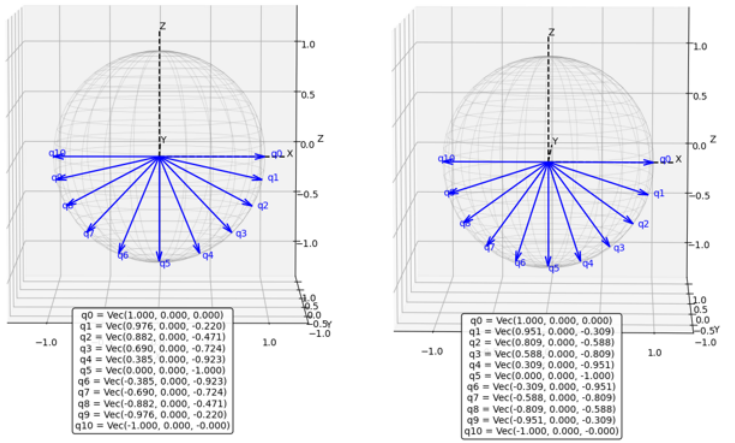
The solid blue Slerp line is linear, meaning that the rate of change of its rotation angle is constant, whereas the Lerp curve shows the angle changing relatively slow at the endpoints, and fastest in the middle (when $t = 0.5$).

The code that generates this data:

```
# two quaternions: 0° and 180° rotation about Y-axis
q1 = Quat.fromAxisAngle(Vec(0, 1, 0), 0)
q2 = Quat.fromAxisAngle(Vec(0, 1, 0), math.pi)
vRef = Vec(1, 0, 0)  # Reference dir for angle calc
tValues = []
angleLerp = [] ; angleSlerp = []
for i in range(101):
  t = i / 100
  qLerp = Quat.lerp(q1, q2, t)
  qSlerp = Quat.slerp(q1, q2, t)
  vL = qLerp.rotateVec(vRef)
  vS = qSlerp.rotateVec(vRef)
```

```
  tValues.append(t)
  angleLerp.append( vRef.angleToDeg(vL))
  angleSlerp.append( vRef.angleToDeg(vS))
```

Another visualization approach is to use the VecPlot class to display the Lerp and Slerp interpolations. This is done by the lerpShow() and slerpShow() examples in QuatPlot.py. The starting and ending quaternions are the same as before: 0° and 180° rotations about the y-axis, but only nine intermediate quaternions are generated. These are used to rotate a vector originally pointing along the x-axis, and the results are drawn in Fig. 14.19.



a) Using Lerp b) Using Slerp

Figure 14.19. Lerp vs. Slerp Interpolation

The most visible difference is around the middle of the interpolations.

lerpShow() and slerpShow() are very similar apart from their calls to Quat.lerp() and Quat.slerp(). slerpShow() is:

```
def slerpShow(plotter):
  q1 = Quat.fromAxisAngle(Vec(0, 1, 0), 0)
  q2 = Quat.fromAxisAngle(Vec(0, 1, 0), math.pi)
  n = 10
  steps = [t/n for t in range(n+1)]
  for i, t in enumerate(steps):
    qt = Quat.slerp(q1, q2, t)
    v = qt.rotateVec(Vec(1, 0, 0)) # x-axis vec
    plotter.addVector(v, label='q'+str(i), color='blue')
```

14.11 Navigating Again

Section 13 on 'Navigating the Earth' discusses spherical trigonometry using the problem of navigating over the Earth's surface (assuming that it's a feature-less sphere). The two main types of curve we encountered were Great Circles and Rhumb Lines, and these were illustrated in Python with the help of the Cartopy mapping module (`https://scitools.org.uk/cartopy/docs/latest/`).

When Slerp() interpolates between two quaternions it traces out the shortest path between them in a Great Circle arc. If these quaternions are converted into latitude and longitude coordinates, they can be used to plot a path between two locations on the Earth.

The conversion of a latitude and longitude into a quaternion assumes that the sphere (see Fig. 14.20) employs an (x, y, z) coordinate system centered around O, with polar coordinates (ρ, λ, ψ). ρ is the length of the point's vector (which we'll treat as a unit radius in the code), λ corresponds to its longitude, while ψ is its inclination, which is equivalent to $\pi/2 - \phi$ (ϕ is the point's latitude).

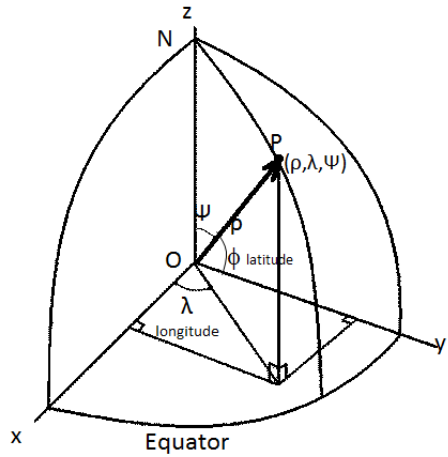


Figure 14.20. 3D Polar and Cartesian Coordinates

The Cartesian coordinates can be expressed as:

$$x = \rho \sin\psi \cos\lambda, \quad y = \rho \sin\psi \sin\lambda \quad z = \rho \cos\psi$$

ψ can be replaced by the latitude ϕ:

$$\begin{array}{rcccl} x & = & \rho \sin(\pi/2 - \phi) \cos\lambda & = & \rho \cos\phi \cos\lambda \\ y & = & \rho \sin(\pi/2 - \phi) \sin\lambda & = & \rho \cos\phi \sin\lambda \\ z & = & \rho \cos(\pi/2 - \phi) & = & \rho \sin\phi \end{array}$$

Since (x, y, z) is a unit vector, it becomes the pure quaternion $q = [0, (x, y, z)]$.

This is implemented by fromLatLon() in Quat.py:

```
@staticmethod
def fromLatLon(latDeg, lonDeg):
  latRad = math.radians(latDeg)
  lonRad = math.radians(lonDeg)
  x = math.cos(latRad) * math.cos(lonRad)
  y = math.cos(latRad) * math.sin(lonRad)
  z = math.sin(latRad)
  return Quat(0, Vec(x, y, z))
```

QuatOrtho.py draws a Great Circle arc between two cities on an Orthographic map, and adds Lerp and Slerp interpolated points along the arc. The curve of the arc is drawn in two ways – using Cartopy and with greatCircleCoords() from our nav.py. Fig. 14.21 shows the map generated for a trip between New York and Cape Town.

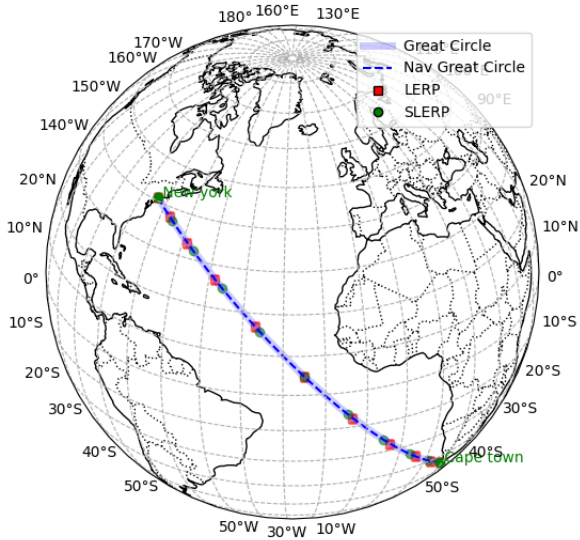


Figure 14.21. From New York to Cape Town

It's possible to call QuatOrtho.py with additional arguments to switch off the Lerp or Slerp points, which makes it easier to judge the arc distances.

Inside QuatOrtho.py the Lerp and Slerp points are generated in a similar way, so we'll focus on just the Slerp code. The interpolated quaternions are generated using:

```
quats = Quat.slerpsNInv(qStart, qEnd, markStep)
coords = [ q.toLatLon() for q in quats]
plotMarkers(ax, coords, "o", "green")
```

slerpsNInv() calls Quat.slerpNInv() multiple times with different t values to generate a list of interpolated quaternions. slerpNInv() is a simplified version of slerp() which doesn't perform quaternion inversion to switch from a long path to a short one, which isn't needed here.

toLatLon() converts a quaternion of the form $q = [0, (x, y, z)]$ back to a latitude and longitude pair. This is achieved by treating (x, y, z) as a Cartesian coordinate and converting it to polar values:

```
def toLatLon(self):
  axis = self.v
  r = axis.magnitude()
  if r == 0:
    raise ValueError("Quaternion has zero vector part")

  latDeg = math.degrees( math.asin(axis.z/r))
  lonDeg = math.degrees( math.atan2(axis.y, axis.x))
  return latDeg, lonDeg
```

14.12 Spherical Cubic Interpolation (Squad)

Slerp() produces smooth animations, but it always follows a Great Circle arc between its two quaternions. This means that a more complex path using a series of quaternions will usually be quite jagged, perhaps changing direction drastically at each quaternion. This is illustrated by TourSlerps.py which draws a series of Great Circles arcs between cities on an Orthographic map (see Fig. 14.22).

To smoothly interpolate through such a sequence, splines are needed, producing something similar to Fig. 14.23.

Consider four quaternions, **p**, **a**, **b**, and **q** as the vertices of a quadrilateral (see Fig. 14.24). **c** is interpolated along the 'edge' from **p** to **q** using Slerp(). At the same time, **d** is interpolated along the 'edge' from **a** to **b**. Finally, the edge between **c** and **d** is interpolated to get **e**, creating a *Squad* quaternion. This is defined like so:

$$\text{Squad}(t, \mathbf{p}, \mathbf{q}, \mathbf{a}, \mathbf{b}) = \text{Slerp}\big(2t(1-t),\ \text{Slerp}(t, \mathbf{p}, \mathbf{q}),\ \text{Slerp}(t, \mathbf{a}, \mathbf{b})\big)$$

The path from **p** to **q** is no longer the shortest (i.e. it's *not* a Great Curve arc), but curves toward the edge linking **a** and **b**.

Squad() is the quaternion equivalent of a cubic spline – it generates a smooth rotation interpolation through a sequence of quaternions by using tangents to

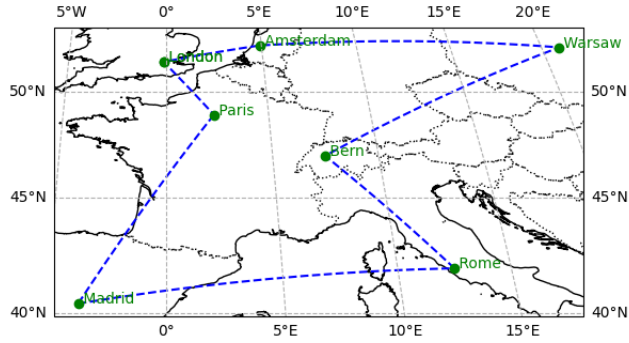


Figure 14.22. Slerps Cities Tour

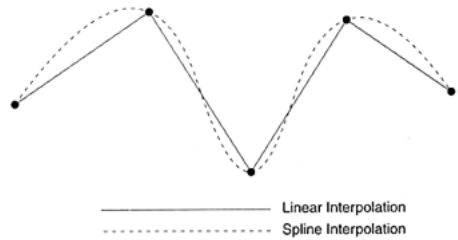


Figure 14.23. Linear vs. Spline Interpolation

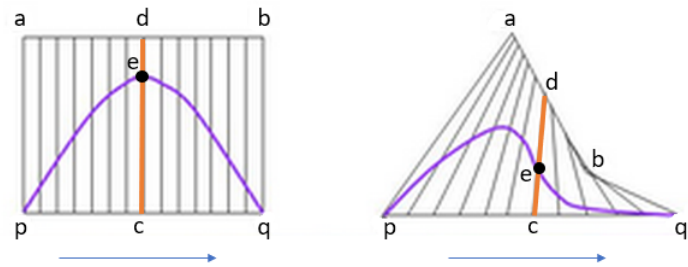


Figure 14.24. Two Examples of Spherical Cubic Interpolation

ensure that the change in direction and rotation between the quaternions is continuous.

Let $\{\mathbf{q}_n, \mathbf{a}_n, \mathbf{b}_n\}_{n=0}^{N-1}$ be a sequence of N quaternions, with

$$S_n(t) = \text{Squad}(t; \mathbf{q}_n, \mathbf{q}_{n+1}, \mathbf{a}_n, \mathbf{b}_n)$$

To ensure the smooth interpolations, $\mathbf{a}_n$ and $\mathbf{b}_n$ are calculated with:

$$\begin{aligned} \mathbf{a}_n &= \mathbf{q}_n \exp((-\log(\mathbf{q}_n^{-1}\mathbf{q}_{n-1}) + \log(\mathbf{q}_n^{-1}\mathbf{q}_{n+1}))/4) \\ \mathbf{b}_n &= \mathbf{a}_{n+1} \end{aligned}$$

$\mathbf{q}_n^{-1}\mathbf{q}_{n-1}$ is the rotation from $\mathbf{q}_n$ to $\mathbf{q}_{n-1}$, $\mathbf{q}_n^{-1}\mathbf{q}_{n+1}$ the rotation from $\mathbf{q}_n$ to $\mathbf{q}_{n+1}$, and log() maps them into vectors in the tangent space at $\mathbf{q}_n$.

A tangent space is a "flat" space that best approximates a curved surface at a given point, meaning that we can do vector math there without worrying about curvature. The unit quaternion $\mathbf{q}$ is a 3-sphere in 4D space, and Fig. 14.25 attempts to illustrate it in 3D.

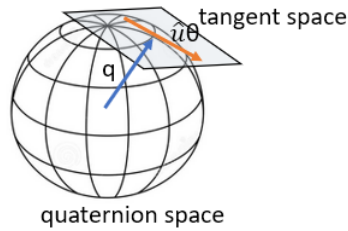


Figure 14.25. Quaternion and Tangent Spaces

After the forward and backward tangent vectors have been averaged and scaled, exp() maps the resulting tangent back to the quaternion sphere. The behavior of log() and exp() can be understood by writing the unit quaternion in Euler form:

$$\mathbf{q} = \cos\theta + \hat{\mathbf{u}}\sin\theta = \exp(\hat{\mathbf{u}}\theta)$$

where $\hat{\mathbf{u}}$ is the rotation axis and θ the rotation angle divided by 2. log($\mathbf{q}$) returns $\hat{\mathbf{u}}\theta$, an angular velocity vector that's a tangent to the quaternion.

A good mathematical description of both Slerp and Squad can be found in the online paper "Quaternion Algebra and Calculus" by David Eberly at `https://www.geometrictools.com/Documentation/Quaternions.pdf` [**Ebe10**]. An even more detailed study can be found in [**DKL98**].

The implementation of Squad in `Quat.py` is based around three functions: squad(), slerpNInv(), and spline(). squad() looks much like its math definition:

```
def squad(q0, q1, a, b, t):
  c = Quat.slerpNInv(q0, q1, t)
  d = Quat.slerpNInv(a, b, t)
  return Quat.slerpNInv(c, d, 2*t*(1 - t))
```

It's common for slerp() implementations to invert one of the input quaternions when the angle between the two exceeds 90°. Although it's true that $\mathbf{q}$ and $-\mathbf{q}$ represent the same rotation, Slerp$(t, \mathbf{p}, \mathbf{q})$ does not produce the same result as Slerp$(t, \mathbf{p}, -\mathbf{q})$. Since the control points $\mathbf{a}$ and $\mathbf{b}$ are chosen to work with positive values of $\mathbf{q}$, it's best not to invert the input quaternions in the version of slerp() used with squad().

spline() does the job of generating a control point $\mathbf{a}_n$ by using $\mathbf{q}_{n-1}$, $\mathbf{q}_n$, and $\mathbf{q}_{n+1}$:

```
  @staticmethod
  def spline(qPrev, q, qNext):
    qInv = q.inverse()
    delta = ((qInv * qPrev).log() +
             (qInv * qNext).log()) * -0.25
    return q * delta.exp()
```

Squad() works with a sequence of N quaternions, iterating through them using $\mathbf{q}_{n-1}$, $\mathbf{q}_n$, and $\mathbf{q}_{n+1}$ as n is incremented. This process is managed by quatCurve() located in Quat.py but not part of the Quat class. The code:

```
def quatCurve(quats, segSteps):
  if not quats:
      return []
  if len(quats) == 1:
      return [quats[0]]

  curve = []
  numSegs = len(quats) - 1
  for i in range(numSegs):
    q1 = quats[i]
    q2 = quats[i+1]

    # previous and next quaternions
    qPrev = quats[max(0, i-1)]  # 0 is min index
    qNext = quats[min(numSegs, i+2)] # numSegs is max index

    # control points 'a' and 'b'
    a = Quat.spline(qPrev, q1, q2)
    b = Quat.spline(q1, q2, qNext)

    # squad interpolation
    for step in range(segSteps):
      t = step / segSteps
      iquat = Quat.squad(q1, q2, a, b, t)
      curve.append(iquat)

  # Add last control point to ensure the curve ends
```

```
  # at the final quaternion.
  if numSegs > 0:
    curve.append(quats[-1])

  return curve
```

quatCurve() returns a list of interpolated quaternions using squad(). One tricky aspect is dealing with the beginning and end of the sequence of quaternions. In particular, how should the control points a_{-1} and b_N be determined? The simplest solution is to just use the nearest quaternions: q_0 and q_{N-1} respectively.

quatCurve() makes the implementation of the Squad-like version of the cities tour quite straight forward. TourSquad.py draws a series of curves (i.e. *not* Great Circles arcs) between the cities on an Orthographic map (see Fig. 14.26).

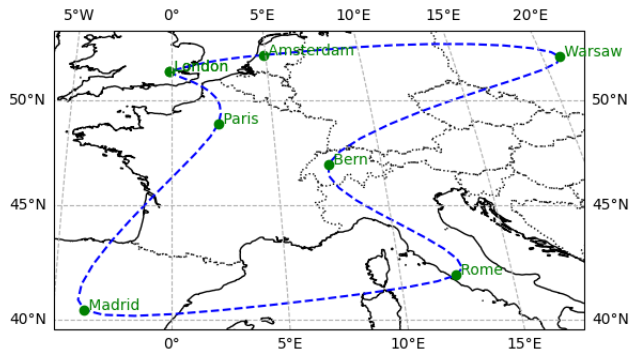


Figure 14.26. Squad Cities Tour

The relevant snippet of code from TourSquad.py is:

```
quats = [ Quat.fromLatLon(*coord) for coord in cityCoords]
iQuats = quatCurve(quats, 10)
iCoords = [ q.toLatLon() for q in iQuats]
```

`quats` holds the quaternions for the cities, which are used by quatCurve() to generate ten interpolated quaternions between each city, which are collected in the `iquats` list. This is converted back to a list of latitude and longitudes, and plotted on the map by Cartopy.

Exercises

(1) Given that $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1$, show that -1 commutes with $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$.

(2) Given that $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$, show that $\mathbf{jk} = \mathbf{i}$.

(3) Show that the real quaternions embed the complex numbers as Hamilton intended; that is

$$(a, b, 0, 0) + (c, d, 0, 0) = (a + c, b + d, 0, 0)$$

and

$$(a, b, 0, 0)(c, d, 0, 0) = (ac - bd, ad + bc, 0, 0).$$

(4) What quaternion represents a rotation through 60° about the x-axis?

(5) What rotation does the quaternion $\mathbf{q} = 0.707 + 0.189\mathbf{i} + 0.378\mathbf{j} + 0.567\mathbf{k}$ represent?

(6) Consider a rotation through 30° about $(0, 1, 0)$. Find the new position under this rotation of the point $\mathbf{v} = (3, 4, 5)$ using quaternions.

(7) Find a single rotation that is equivalent to the following sequence of rotations: a rotation through 50° about the axis $(1, 2, 4)$, followed by a rotation through 37° about the axis $(2, -1, 2)$.

(8) Prove that a quaternion $\mathbf{q}$ represents a rotation if and only if $\mathbf{q}$ is a unit quaternion.

(9) Complete the proofs of properties (c) and (d) for the Slerp() function in section 14.9 using the hints provided.

Answers

Code for questions 4-7 can be found in exs.py

(1) $\mathbf{i}^2 = -1 \Rightarrow \mathbf{i}^3 = (-1)\mathbf{i} \Rightarrow \mathbf{i}\mathbf{i}^2 = (-1)\mathbf{i} \Rightarrow \mathbf{i}(-1) = (-1)\mathbf{i}$. Similarly for $\mathbf{j}$ and $\mathbf{k}$.

(2) $\mathbf{ijk} = -1 \Rightarrow \mathbf{i}^2\mathbf{jk} = -\mathbf{i} \Rightarrow -1\mathbf{jk} = -\mathbf{i} \Rightarrow \mathbf{jk} = \mathbf{i}$.

(3) $(a + b\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) + (c + d\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) = (a + b) + (c + d)\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$;
$(a + b\mathbf{i} + 0\mathbf{j} + 0\mathbf{k})(c + d\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) = (ac - bd) + (ad + bc)\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$.

(4) The quaternion that represents the rotation is

$$\mathbf{q} = \cos(60°/2) + \sin(60°/2)(\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) = \frac{\sqrt{3}}{2} + \frac{1}{2}\mathbf{i}.$$

(5) The magnitude of the rotation is $2\cos^{-1} 0.707 = 2 \times 45° = 90°$. The axis of the rotation is $(0.189, 0.378, 0.567)$.

(6) Using quaternions, $\mathbf{q} = 0.966 + 0.259\mathbf{j}$, $\mathbf{q}^{-1} = 0.966 - 0.259\mathbf{j}$, and $\mathbf{v} = 3.0\mathbf{i} + 4.0\mathbf{j} + 5.0\mathbf{k}$, so $\mathbf{qvq}^{-1} = 5.098\mathbf{i} + 4.000\mathbf{j} + 2.830\mathbf{k}$.

(7) The quaternion $\mathbf{q}_1 = 0.906 + 0.092\mathbf{i} + 0.184\mathbf{j} + 0.369\mathbf{k}$ represents a rotation through 50° about $(1, 2, 4)$, while $\mathbf{q}_2 = 0.948 + 0.212\mathbf{i} - 0.106\mathbf{j} + 0.212\mathbf{k}$ represents a rotation through 37° about $(2, -1, 2)$. Thus, the quaternion

$$\mathbf{q}_3 = \mathbf{q}_2\mathbf{q}_1 = 0.781 + 0.201\mathbf{i} + 0.021\mathbf{j} + 0.590\mathbf{k}$$

represents the first rotation followed by the second. The magnitude of this rotation is $2\cos^{-1} 0.781 = 77.22°$, and its unit axis is

$$1/\sin(77.22°/2)(0.201, 0.021, 0.590) = (0.322, 0.033, 0.946).$$

(8) Let $\mathbf{v} = (x, y, z)$ be a unit vector and $\mathbf{q} = \sin(\theta/2) + \cos(\theta/2)\mathbf{v}$ be the quaternion that represents a rotation through θ about $\mathbf{v}$. Then $\mathbf{q} = \sin(\theta/2) + x\cos(\theta/2)\mathbf{i} + y\cos(\theta/2)\mathbf{j} + z\cos(\theta/2)\mathbf{k}$, so that

$$\begin{aligned} |\mathbf{q}| &= \sqrt{\sin^2(\theta/2) + x^2\cos^2(\theta/2) + y^2\cos^2(\theta/2) + z^2\cos^2(\theta/2)} \\ &= \sqrt{\sin^2(\theta/2) + \cos^2(\theta/2)(x^2 + y^2 + z^2)}. \end{aligned}$$

But $\mathbf{v} = (x, y, z)$ is a unit vector, so $x^2 + y^2 + z^2 = 1$, and

$$|\mathbf{q}| = \sqrt{\sin^2(\theta/2) + \cos^2(\theta/2)} = 1.$$

Thus, $\mathbf{q}$ is a unit quaternion.

Conversely, let $\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ be a unit quaternion. Then $w^2 + x^2 + y^2 + z^2 = 1$, so that the absolute value of each component of $\mathbf{q} \leq 1$. In particular, $-1 \leq w \leq 1$; there exists an angle θ with $\sin(\theta/2) = w$. Then $x^2 + y^2 + z^2 = 1 - \sin^2(\theta/2) = \cos^2(\theta/2)$, and $\mathbf{q}$ represents the rotation through θ about the axis $\mathbf{v} = (x/\cos(\theta/2), y/\cos(\theta/2), z/\cos(\theta/2))$.

Note that $\mathbf{v}$ is a unit vector:

$$(x/\cos(\theta/2))^2 + (y/\cos(\theta/2))^2 + (z/\cos(\theta/2))^2 = \cos^2(\theta/2)/\cos^2(\theta/2) = 1.$$