Why Do the Unit Quaternions Double-Cover the Space of Rotations?

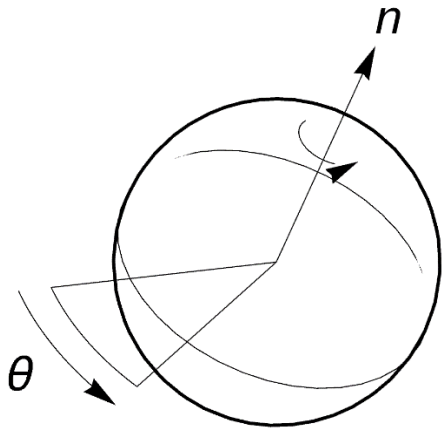
Neil Bickford

1. Computing with Quaternions

The unit quaternion

$$\boldsymbol{q} = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) \\ x\sin\left(\frac{\theta}{2}\right) \\ y\sin\left(\frac{\theta}{2}\right) \\ z\sin\left(\frac{\theta}{2}\right) \end{pmatrix}$$

represents a counterclockwise rotation by the angle θ around the normalized axis $\boldsymbol{n} = (x, y, z)^T$.



We'll sometimes use q_w, q_x, q_y, and q_z to refer to the four components of a quaternion.

We can compose quaternions in the same way we can compose rotations: the product **r** of quaternions **p** and **q**

$$\boldsymbol{r} = \boldsymbol{pq}$$

represents the rotation given by performing $\boldsymbol{q}$, then by performing $\boldsymbol{p}$. For instance, if **q** is a rotation by the angle θ around the axis $(x, y, z)^T$, then the product of **q** with itself is twice the rotation:

$$\boldsymbol{q}^2 = \left(\cos\left(\frac{2\theta}{2}\right), x\sin\left(\frac{2\theta}{2}\right), y\sin\left(\frac{2\theta}{2}\right), z\sin\left(\frac{2\theta}{2}\right)\right)^T.$$

Similarly,

$$\boldsymbol{q}^3 = \left(\cos\left(\frac{3\theta}{2}\right), x\sin\left(\frac{3\theta}{2}\right), y\sin\left(\frac{3\theta}{2}\right), z\sin\left(\frac{3\theta}{2}\right)\right)^T$$

and so on. The inverse of a unit quaternion[1] is given by reversing its rotation:

$$\boldsymbol{q}^{-1} = \left(\cos\left(-\frac{\theta}{2}\right), x\sin\left(-\frac{\theta}{2}\right), y\sin\left(-\frac{\theta}{2}\right), z\sin\left(-\frac{\theta}{2}\right)\right)^T.$$

If we ever need to, we can write out the result of performing $\boldsymbol{q}$, then $\boldsymbol{p}$ (like finding the rotation that corresponds to a product of two other rotations), as another quaternion:

$$\boldsymbol{pq} = \begin{pmatrix} p_w q_w - p_x q_x - p_y q_y - p_z q_z \\ p_x q_w + p_w q_x - p_z q_y + p_y q_z \\ p_y q_w + p_z q_x + p_w q_y - p_x q_z \\ p_z q_w - p_y q_x + p_x q_y + p_w q_z \end{pmatrix}$$

This gives us a way to express the product of two quaternions. Note that we have to be careful about the order in which we apply quaternions (and rotations); for instance, a 90° rotation around the *x* axis followed by a 90° rotation around the *y* axis produces a different result than a 90° rotation around the *y* axis followed by a 90° rotation around the x axis.

We can also express this as a matrix-vector product, which is useful for computer implementation: if $\boldsymbol{r} = \boldsymbol{pq}$, then

$$\begin{pmatrix} r_w \\ r_x \\ r_y \\ r_z \end{pmatrix} = \begin{pmatrix} p_w & -p_x & -p_y & -p_z \\ p_x & p_w & -p_z & p_y \\ p_y & p_z & p_w & -p_x \\ p_z & -p_y & p_x & p_w \end{pmatrix} \begin{pmatrix} q_w \\ q_x \\ q_y \\ q_z \end{pmatrix}$$

We can even express quaternions themselves as 4x4 matrices and have all of the normal notation carry over:

$$\begin{pmatrix} r_w & -r_x & -r_y & -r_z \\ r_x & r_w & -r_z & r_y \\ r_y & r_z & r_w & -r_x \\ r_z & -r_y & r_x & r_w \end{pmatrix} = \begin{pmatrix} p_w & -p_x & -p_y & -p_z \\ p_x & p_w & -p_z & p_y \\ p_y & p_z & p_w & -p_x \\ p_z & -p_y & p_x & p_w \end{pmatrix} \begin{pmatrix} q_w & -q_x & -q_y & -q_z \\ q_x & q_w & -q_z & q_y \\ q_y & q_z & q_w & -q_x \\ q_z & -q_y & q_x & q_w \end{pmatrix}$$

(We'll show how to derive this in section 5.)

Alternatively, here's an easy way to remember the product of two quaternions: Imagine we extend the real numbers with three symbols *i, j,* and *k* (in the same way that we can extend the real line to get the complex numbers) with the properties that

$$i^2 = -1, j^2 = -1, k^2 = -1, \text{and } ijk = -1.$$

[1] There are such things as non-unit quaternions, which we won't talk about in this article outside of this footnote. They can be thought of as combinations of a rotation and a scale by the length of the quaternion, the square root of the *norm* given by $N(\boldsymbol{q}) = |q|^2 = q_w^2 + q_x^2 + q_y^2 + q_z^2$. The unit quaternions are those quaternions with norm 1: $q_w^2 + q_x^2 + q_y^2 + q_z^2 = 1$. For the inverse of a general quaternion, divide the coefficients above by $N(\boldsymbol{q})$.

This norm happens to satisfy $N(\boldsymbol{pq}) = N(\boldsymbol{p})N(\boldsymbol{q})$ for any two quaternions $\boldsymbol{p}$ and $\boldsymbol{q}$, which gives a quick way to derive Euler's four-square identity. As it turns out, the requirement for such a norm to exist is the main reason why normed division algebras over the reals are only possible in dimensions 1, 2, 4, and 8.

From these properties, we can derive the product of any two basis elements: $ij = k, ji = -k, jk = i, kj = -i, ki = j$, and $ki = j$. We can then write the quaternion $\boldsymbol{q} = (q_w, q_x, q_y, q_z)$ as

$$q_w + q_x i + q_y j + q_z k$$

and have all the normal multiplication work:

$$\begin{aligned}
&(p_w + p_x i + p_y j + p_z k)(q_w + q_x i + q_y j + q_z k) \\
&= (p_w q_w - p_x q_x - p_y q_y - p_z q_z) + (p_w q_x + p_x q_w) i + (p_w q_y + p_y q_w) j + (p_w q_z + p_z q_w) k + p_x q_y ij \\
&\qquad + p_x q_z ik + p_y q_x ji + p_y q_z jk + p_z q_x ki + p_z q_y kj \\
&= (p_w q_w - p_x q_x - p_y q_y - p_z q_z) + (p_x q_w + p_w q_x - p_z q_y + p_y q_z) i \\
&\qquad + (p_y q_w + p_z q_x + p_w q_y - p_x q_z) j + (p_z q_w - p_y q_x + p_x q_y + p_w q_z) k.
\end{aligned}$$

With the equations above, quaternions give us a fast and efficient way to store rotations, express the composition of rotations, and, importantly, to *smoothly blend* between rotations (which we'll cover in section 5.)

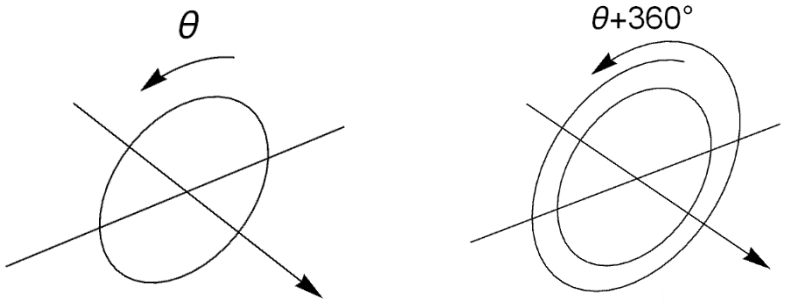
Now, here's something interesting: Consider the rotation around an axis $(n_x, n_y, n_z)^T$ by an angle θ, represented by a quaternion:

$$\boldsymbol{q} = \left(\cos\left(\frac{\theta}{2}\right), n_x \sin\left(\frac{\theta}{2}\right), n_y \sin\left(\frac{\theta}{2}\right), n_z \sin\left(\frac{\theta}{2}\right)\right)^T.$$

If we rotate around this axis by an additional 360°, we get

$$\boldsymbol{q}' = \left(-\cos\left(\frac{\theta}{2}\right), -n_x \sin\left(\frac{\theta}{2}\right), -n_y \sin\left(\frac{\theta}{2}\right), -n_z \sin\left(\frac{\theta}{2}\right)\right)^T$$

This is a different quaternion, but *it represents the same rotation*; we've just rotated an extra 360°.



Rotating an additional 360°, for a total of 720°, brings us back to the first quaternion.

In fact, we see the following: Every rotation – an angle around some axis – is represented not by one, but by *two* quaternions. In this way, we say that the unit quaternions *double-cover* the space of rotations. But why do quaternions *have* to double-cover rotations? Could we – say – just do something like replacing $\frac{\theta}{2}$ by θ?[2]

[2] Answer: No, because then all 180° rotations would be represented by the same four-vector, $(-1, 0, 0, 0)^T$.

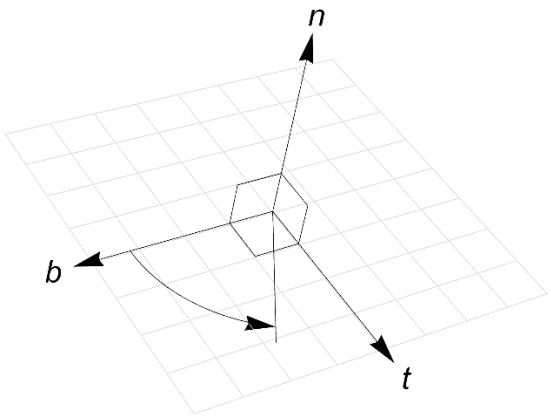
This article is about this phenomenon of double-covering. In short: When we're talking about interpolating between paths in the space of rotations, it actually matters how many times our rotation has completed a full circle.

In a wider mathematical frame, it turns out that the space of unit quaternions is actually slightly nicer than the space of rotations: the space of quaternions maps nicely to a sphere in four dimensions, while the space of rotations isn't simply connected. Put another way, the unit quaternions cover the space of rotations twice, because they cannot cover the space of rotations once and also provide a way to interpolate between paths of rotations. By working with quaternions, we get to work with points on a sphere, instead of points on a real projective plane.

2. An Orthogonal Basis Problem

Here's a system of representing rotations that doesn't work.

Suppose we want to rotate a point around an axis. One way to do this might be to extend the axis into a full coordinate frame, by somehow finding two additional vectors which meet at right angles to the axis and to each other.



Once we have such a coordinate frame[3], we can map our point to our coordinate system, rotate around the axis using a two-dimensional rotation in the plane of our other two vectors, and then transform back into the original coordinate system.[4]

$$R = \begin{pmatrix} | & | & | \\ \boldsymbol{n} & \boldsymbol{b} & \boldsymbol{t} \\ | & | & | \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} | & | & | \\ \boldsymbol{n} & \boldsymbol{b} & \boldsymbol{t} \\ | & | & | \end{pmatrix}^{-1}$$

Ideally, we'd also like our two additional vectors to vary smoothly as we adjust our axis. Put another way, we'd like to find some continuous function which takes as input a normalized vector and

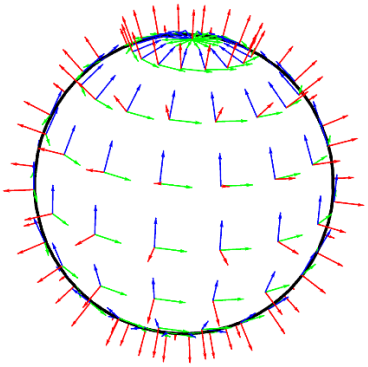
[3] Technical note: With a fixed handedness.

[4] If ***n, b***, and ***t*** are normalized, then this is slightly easier; since the basis transform matrix is orthogonal, we have

$$\begin{pmatrix} | & | & | \\ \boldsymbol{n} & \boldsymbol{b} & \boldsymbol{t} \\ | & | & | \end{pmatrix}^{-1} = \begin{pmatrix} | & | & | \\ \boldsymbol{n} & \boldsymbol{b} & \boldsymbol{t} \\ | & | & | \end{pmatrix}^{T}.$$

outputs the rest of the coordinate system. (The reason we want this function to be continuous is because we'd like to be able to nicely interpolate between rotations; otherwise, although the results might be OK, our internal model of the system would suddenly change its state as we passed over the discontinuity.)

Unfortunately, this is impossible: no matter how we try to construct such a function, we'll always have a discontinuity somewhere in the space of unit vectors. The easiest way to see this is through the Hairy Ball Theorem: if we had such a function, then we'd be able to place a coordinate frame at each point of a sphere, like this (excepting the north and south poles in this illustration):



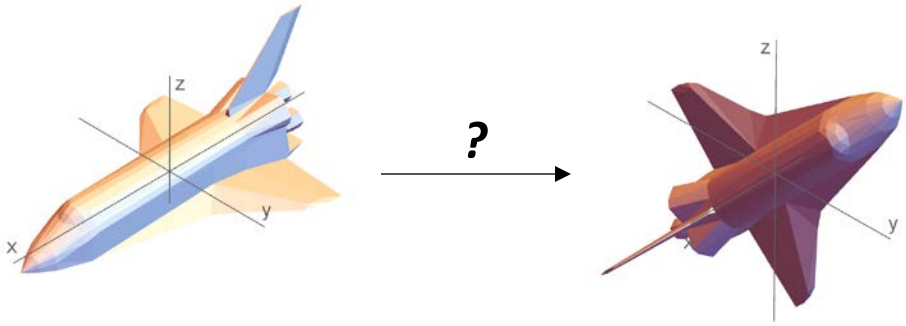
If we could do such a thing, then the green or blue vectors would form a continuous vector field on the surface of the sphere. But this would produce a smooth combing of the sphere, which is impossible by the Hairy Ball Theorem.

3. Euler Angles

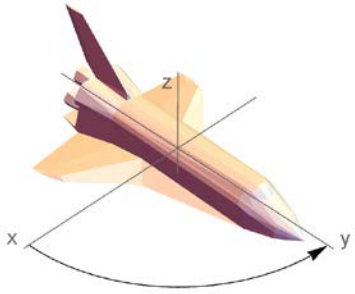
If you've heard of Euler angles before, these singular points that arise when trying to comb the sphere might remind you of *gimbal lock*, a problem that arises in using this system to represent rotations – where at particular choices of axis, you lose a degree of freedom and the entire system freezes up until you rotate it out or through of this particular range of axes. If you haven't heard of Euler angles before, let's rewind a bit.

Euler angles give us a way to represent orientations as a unique product of a yaw, a pitch, and a roll, as follows:

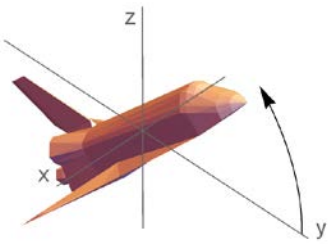
Suppose we have an object which we want to rotate to a particular orientation, which we express (as with quaternions) as an axis and an angle around that axis.



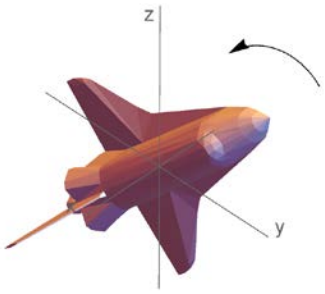
First, we match the lateral bearing (yaw) of the object:



Then, we pitch the object up or down to match the new axis:

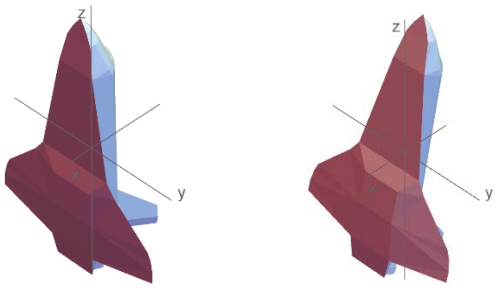


Finally, we rotate (roll) the object around this axis to match the full orientation.



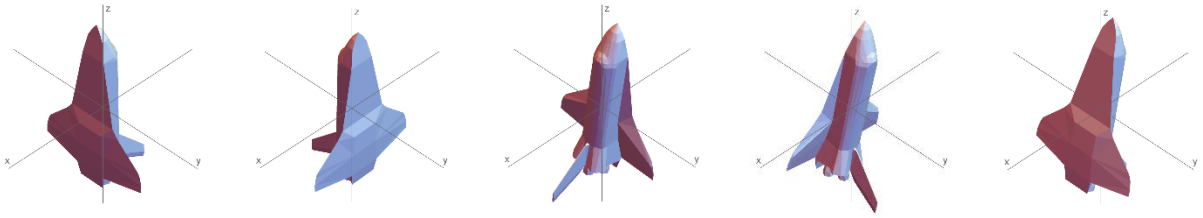
If we limit the yaw, pitch, and roll to some range of values (e.g. $0 \leq yaw, roll < 360°$ and $-90° < pitch < 90°$), these three numbers can then be used to uniquely represent any orientation outside of gimbal lock. (We can also uniquely specify a rotation instead of an orientation by listing the yaw, pitch, and roll that a model undergoes as a result of that rotation.)

However, we have a problem: Regardless of our choice of Euler angle system[5], we'll always be able to find some axis near which slightly different orientations lead to wildly different Euler angles. For instance, consider these two orientations with regards to the above system:



In the above system, the first of these two orientations can be given by a simple 90° pitch upwards. For the second, we need to turn the object 180°, pitch it upwards just less than 90°, and then roll it another 180°.

In particular, this means that our mapping in reverse from orientations to Euler angles is discontinuous – and that as a result, in order to interpolate between two orientations using Euler angles, we might have to take a longer route than necessary:



Gimbal lock has been the cause of a variety of problems in real-world systems; for more information on the ill effects of gimbal lock, see [Hanson, pg. 19-27].

4. The Connectedness of Rotations

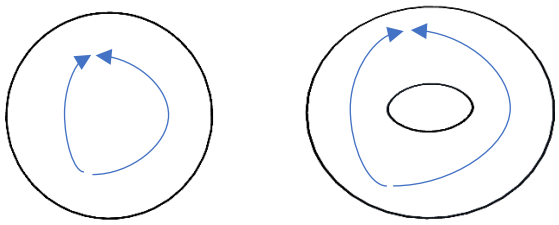
Euler angles, in fact, have problems for reasons beyond the Hairy Ball Theorem – their geometry is that of a four-dimensional torus (which doesn't correspond to the geometry of the space of rotations), and the fact that they have three parameters also prevents them from smoothly representing the space of rotations (if we consider the sets of points we get as we vary the angle of rotation from 0° to 360°, we

[5] We've described the ZYX system of Euler angles above, where we rotate around the Z, Y, and X axes in our local coordinate frame in sequence. (For instance, at the second step, we rotated around the object's local Y axis.) We can also describe Euler angles as a triplet of rotations around each of three axes, at least one different from the rest. We'll always have gimbal lock somewhere, regardless of the Euler angle system we choose; however, in some cases (e.g. ships), we might be able to choose a frame such that we should never rotate to an orientation that would cause gimbal lock.

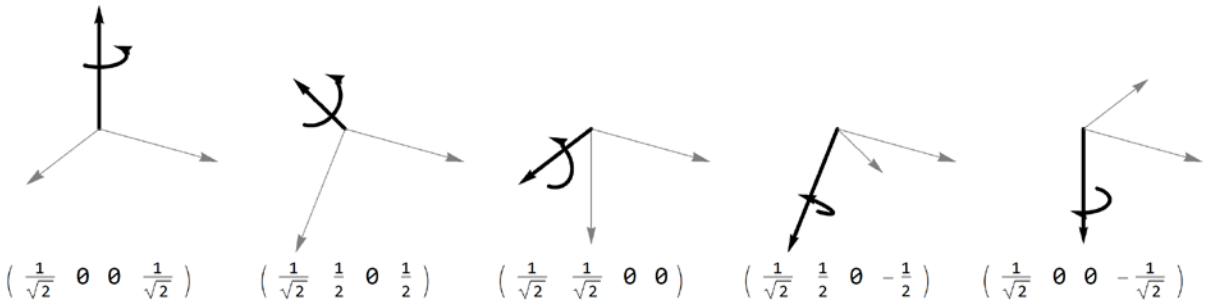
can see that the sets of spheres we get must at some point "turn back" on themselves, which prevents us from having a nice mapping from a region in $\mathbb{R}^3$ to the space of rotations.)

Even if we uniquely parameterized rotations as a pair of an axis and an angle (with some constraints, so that we express each rotation exactly once), we'd still run into problems when talking about interpolation – specifically, when talking about interpolation between paths of rotations.

Consider a space, and draw two continuous paths through the space which begin and end at the same two points. We say that this space is *simply connected* if, no matter which two points or paths we choose, we can always continuously transform the first path into the second. For instance, the sphere is simply connected, while the torus is not simply connected.



As it turns out, the space of rotations isn't simply connected – unless we allow ourselves to represent each rotation twice, in which case we get the quaternions. To see this, consider the cycle given by starting with a 180-degree rotation around the vertical axis, and then continuously turning the axis until it points downward.



Since a 180-degree clockwise rotation is the same as a 180-degree counterclockwise rotation, our path starts and ends at the same rotation. If the space of rotations were simply connected, we would be able to smoothly adjust and contract this loop until we get a single point. However, no matter how we adjust this path of rotations, our axis of rotation must at some point be horizontal. Therefore, we cannot turn this path into a single point, and so our space is not simply connected.

The unit quaternions get around this by representing each rotation in two ways. As a result, we need to rotate our axis by a full 360° in the space of quaternions to get back to the same point (in the space of unit quaternions) we started with. Additionally, we can easily map the unit quaternions to the surface of a four-dimensional sphere and back (since the unit quaternions satisfy $q_w^2 + q_x^2 + q_y^2 + q_z^2 = 1$). Since the four-dimensional sphere, like the three-dimensional sphere[6] is simply connected, the unit quaternions themselves are simply connected, which is nice.

[6] Unlike the two-dimensional sphere (the edge of a circle), which essentially doesn't have enough dimensions to transform paths.

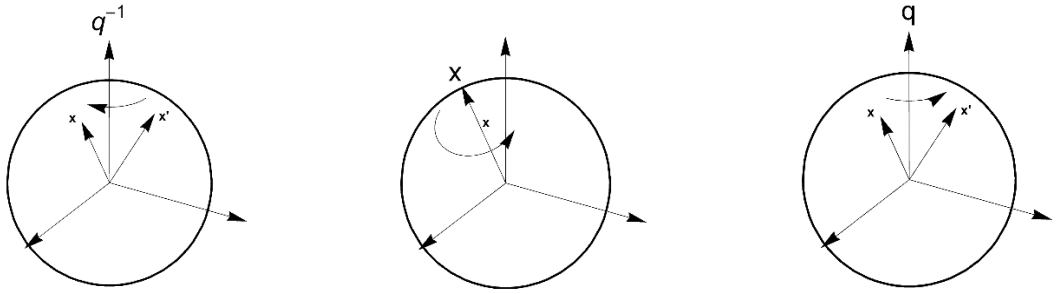
5. Working with Quaternions: 9 Recipes and Tricks You Might Not Have Heard About

Rotating points: We've talked about how the unit quaternion $\left(\cos\frac{\theta}{2}, n_x \sin\frac{\theta}{2}, n_y \sin\frac{\theta}{2}, n_z \sin\frac{\theta}{2}\right)^T$ represents a rotation, and how to find the result of performing two rotations in sequence, but we haven't actually talked about how to rotate a point around an axis.

Let's say we have a point $(x, y, z)^T$ and a quaternion $\boldsymbol{q} = \left(\cos\frac{\theta}{2}, n_x \sin\frac{\theta}{2}, n_y \sin\frac{\theta}{2}, n_z \sin\frac{\theta}{2}\right)^T$ representing the angle and axis around which we want to rotate. We can actually think of the point $\boldsymbol{x} = (x, y, z)^T$ not as a point, but as a quaternion $(0, x, y, z)$, which represents a 180° counterclockwise rotation around the axis $(x, y, z)^T$.

Now, consider the following sequence of rotations:

- Rotate by -θ around $\left(n_x, n_y, n_z\right)^T$.
- Rotate by 180° around **x**.
- Rotate by θ around $\left(n_x, n_y, n_z\right)^T$.



We'd like to figure out what rotation this corresponds to. If we apply this rotation to our desired result – the rotation of $\boldsymbol{x}$ by **q**, which we'll call **x'** – we wind up transforming **x'** to **x**, performing a rotation which leaves $\boldsymbol{x}$ in place, and finally transforming **x** back to **x'**.

On the other hand, if we started with a vector perpendicular to **x'** (let's call it **w'**), we'd transform **w'** to a vector **w** perpendicular to $\boldsymbol{x}$, negate it, and rotate it back, ultimately, to **-w**. Therefore, our sequence of rotations is the same as a 180° rotation around **x'**.

That is, if we express points in $\mathbb{R}^3$ as corresponding 180° unit quaternions, then the quaternion

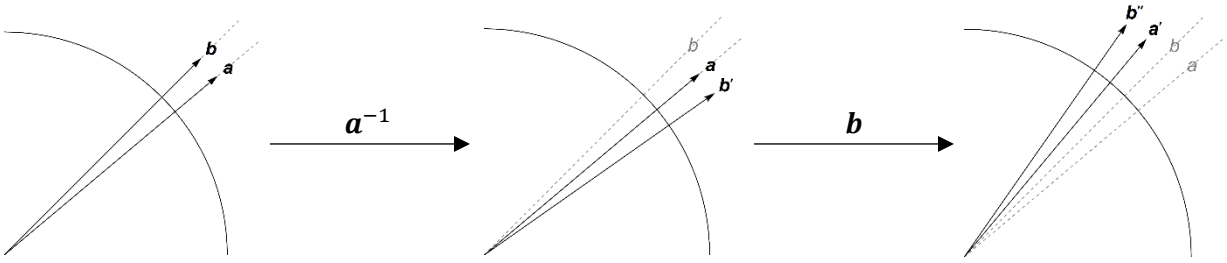
$$\boldsymbol{x'} = \boldsymbol{qxq^{-1}}$$

is the result of rotating the point **x** by **q**.

Rotating between two points: Suppose we have two points **a** and **b**, and we want to find some rotation which rotates the point **a** to the point **b**. Interpreting points as 180° rotations as before, consider the quaternion

$$\boldsymbol{ba^{-1}}.$$

If we apply this rotation to (the point) **a**, we actually wind up on the far side of **b** – twice as far as we meant to go!



Intuitively (we can formalize this without much difficulty), the solution is to rotate half as far: the quaternion

$$\sqrt{ba^{-1}}$$

then gives a smooth and direct rotation from **a** to **b**.

We can think of the square root in much of the same way we thought of powers of quaternions at the start: the square root of an axis-angle unit quaternion corresponds to dividing its angle by 2, which we can compute either by expressing the quaternion in axis-angle form or by using the half-angle formulas for sin and cos. (We also have a sign problem from the square root; in this case, we always want to choose the quaternion with a nonnegative w component.)

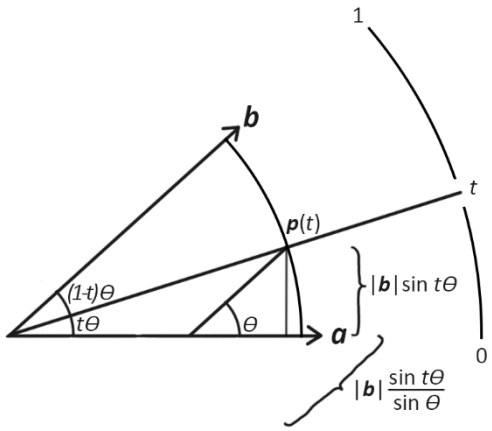
Rotations are orientations. If we have some object, we can fix some initial orientation for the object, and then describe its orientation by writing a rotation which transforms the object's initial orientation into its current orientation. Conversely, if we've fixed some initial orientation for the object, we can uniquely describe a rotation by giving the orientation of the object at the end of the rotation.

Interpolating between quaternions: Quaternions can be directly embedded in four dimensions as the set of points on the unit four-dimensional sphere. As a result, we can measure the distance between quaternions by the distance in four-space, and we can interpolate between quaternions by interpolating between points on the sphere.

In two dimensions, we can interpolate between points **a** and **b** on the unit circle (using a parameter t that ranges from 0 to 1) using the formula

$$\boldsymbol{p}(t) = \frac{\sin\big((1-t)\theta\big)}{\sin(\theta)}\boldsymbol{a} + \frac{\sin(t\theta)}{\sin(\theta)}\boldsymbol{b}$$

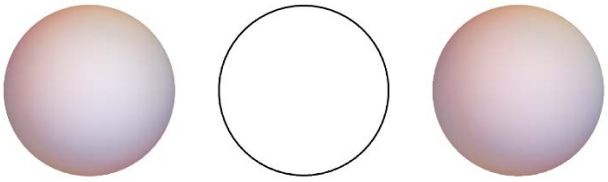
where θ is the angle between **a** and **b**. It's easiest to see this geometrically – for the component of $\boldsymbol{b}$ in the above expression, for instance:



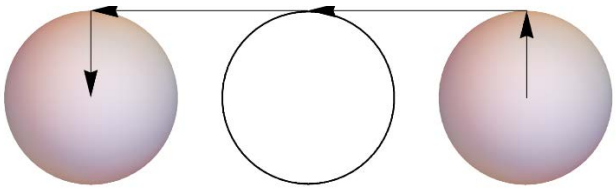
Since we can always think of a plane containing **0, a,** and **b**, the same formula also works in higher dimensions. (This is more commonly known as the *slerp* formula for spherically interpolating between two points.)

We have to be careful, though! Although this formula always gives the shortest way to interpolate between two unit quaternions, since **b** and **-b** represent the same rotation, we might have parity problems using this formula to interpolate between two rotations if we're not careful. In particular, the double-covering property of the unit quaternions combined with the above formula means that we have two ways to rotate between two rotations – and if we don't check in advance, we can wind up traversing the longer of the two ways. The solution when using quaternions to represent rotations is to choose whether to interpolate between **a** and **b** or between **a** and **-b**, depending on which of the two pairs are closer together.

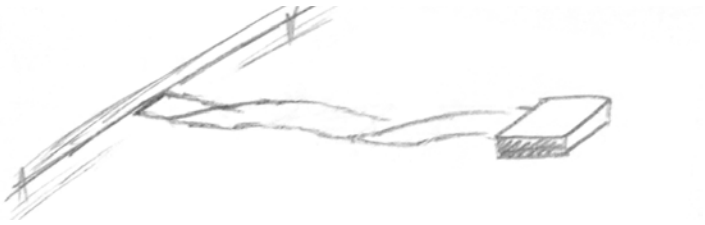
Visualizing quaternions on the sphere in four dimensions: We can lay out the surface of a four-dimensional sphere by separating it into a solid unit ball, a hollow spherical 'equator', and a second solid unit ball. The point $(w, x, y, z)^T$ on the four-dimensional sphere maps to $(x, y, z)^T$ in the left ball if $w < 0$, $(x, y, z)^T$ on the sphere if $w = 0$, and $(x, y, z)^T$ on the right ball if $w > 0$.



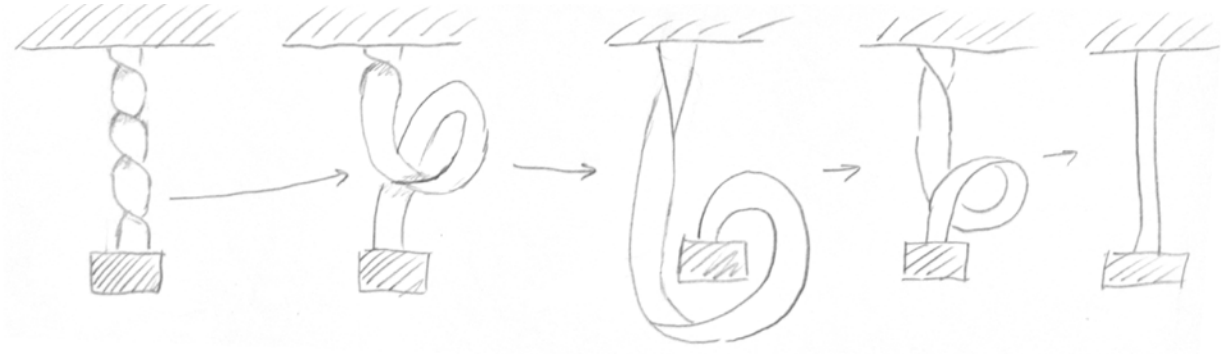
We can also draw paths between quaternions using this approach. For instance, if we were to perform a full 360-degree rotation around $(0, 0, 1)^T$ in the space of quaternions, we'd start at the center on the right (at $(1, 0, 0, 0)^T$), go upwards through the right ball, pass through the sphere at $(0, 0, 0, 1)^T$, then go downwards through the left ball, finishing in the center of the left ball at $(-1, 0, 0, 0)^T$.



The belt trick. Here's a nice way to show that the space of rotations isn't simply connected, while the space of unit quaternions is: Take a long strand of cloth (a belt will also do), fix one end of it to a stationary object, and give the end one full twist. The goal is now to find a way to bend and manipulate the middle of the cloth (possibly passing it over the end) *while keeping the ends stationary* so as to remove the twist in the cloth.



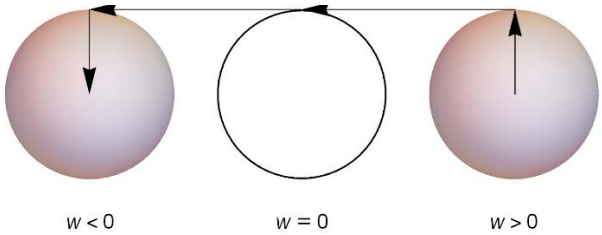
For a full twist, this is impossible – we can turn a full clockwise twist into a full counterclockwise twist, for instance, but we can't untwist the fabric. Surprisingly, though, if we start out with *two* full twists, we can untwist the fabric!



Here's the real trick: Instead of thinking about the fabric as a surface, we can think of the fabric as a series of orientations along a curve. These then form a path through the space of unit quaternions, which we can visualize!

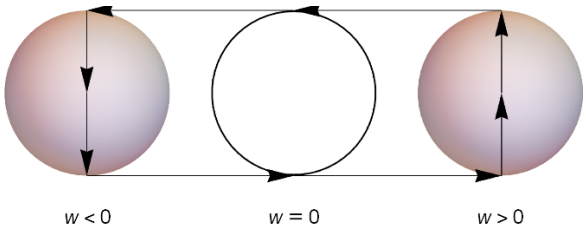
A better way, in fact, is to imagine a series of nested glass spheres around the free endpoint of the fabric, each of which contains some slice of the fabric as it leads inwards. Then we can specify the orientation and position of the fabric at any point by specifying the orientation of the corresponding glass sphere.

For a full turn, we have a series of rotations around a single axis, which gives us roughly the same path in the space of unit quaternions as before:

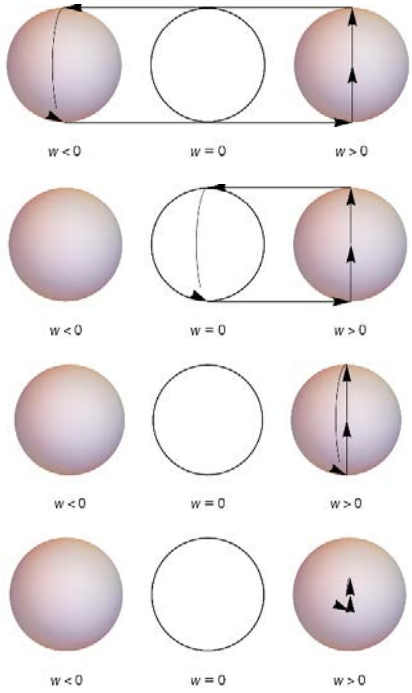


Since we start and end at different unit quaternions, though, we can't transform this path into a point while keeping the endpoints intact, so we cannot untwist the fabric.

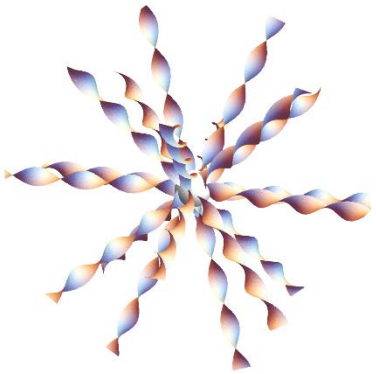
However, when we have two full twists, we have a full loop through the space of unit quaternions:



We can then untwist the fabric by transforming the path on the four-dimensional sphere to a single point:



But we have one more trick. Suppose instead of one twisted piece of fabric, we have an entire sphere of twisted pieces of fabric:



If we think of these pieces of fabric as being embedded in nested glass spheres as before, then no matter how we rotate the glass spheres, the strands of fabric will never intersect. As a result, if we start from an untwisted configuration, twist the spheres to create a double twist in one of the pieces of fabric (in fact, in all of the pieces of fabric), and then reverse the result, we'll be able to show dozens of double twists being untwisted at once without a single intersection in the entire configuration at any point.

This is quite a sight when animated; for more visualizations of this trick, see Andrew Hanson's *Belt Trick* demonstration at https://www.cs.indiana.edu/~hansona/quatvis/Belt-Trick/index.html.

Deriving quaternion composition. If we know that rotations are linear transformations, that an arbitrary rotation can be expressed as a product of rotations about the x and y axes, and that our resulting structure will require a 720-degree rotation to be returned to its initial state, we can sort of rederive the rules for composing quaternions (and in particular, the matrix at the beginning of this paper) as follows:

Let's denote our identity rotation by *1*, a 180-degree rotation about the x axis by *i*, and a 180-degree rotation about the y axis by *j*. The result of performing *j* followed by *i* is a 180-degree rotation about a third axis, which we'll denote by *k*. Since we treat a 360-degree rotation as a sort of alternate form of the identity rotation *1*, we have $i^2=j^2=k^2=-1$. We already have $ij=k$; manipulating this expression gives $jk = i$ and $ki = j$. Finally, we can see that $ji = -k$, since $ji = -(jk)(ki) = -ij = -k$, and we can similarly determine that $kj = -i$ and $ik = -j$.

Since rotations are linear, quaternion composition should be linear as well; therefore, we can split the product $\boldsymbol{pq}$ into a linear sum of $\boldsymbol{p}*1$, $\boldsymbol{p}*i$, $\boldsymbol{p}*j$, and $\boldsymbol{p}*k$. We get

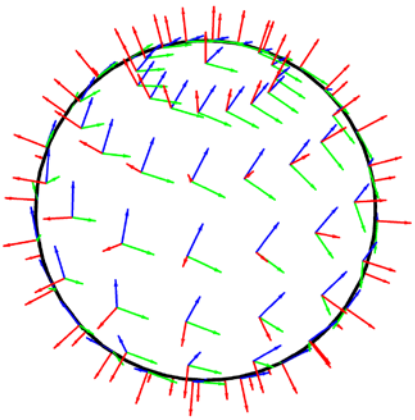
$$\begin{aligned}
(p_w + p_x i + p_y j + p_z k)1 &= p_w + p_x i + p_y j + p_z k \\
(p_w + p_x i + p_y j + p_z k)i &= -p_x + p_w i + p_z j - p_y k \\
(p_x i + p_y j + p_z k + p_w)j &= -p_y - p_z i + p_w j + p_x k \\
(p_x i + p_y j + p_z k + p_w)k &= -p_z + p_y i - p_x j + p_w k
\end{aligned}$$

so if $\boldsymbol{r} = \boldsymbol{pq}$, then

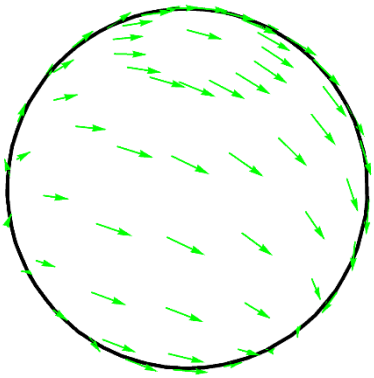
$$\begin{pmatrix} r_w \\ r_x \\ r_y \\ r_z \end{pmatrix} = \begin{pmatrix} p_w & -p_x & -p_y & -p_z \\ p_x & p_w & -p_z & p_y \\ p_y & p_z & p_w & -p_x \\ p_z & -p_y & p_x & p_w \end{pmatrix} \begin{pmatrix} q_w \\ q_x \\ q_y \\ q_z \end{pmatrix}.$$

Almost combing a hairy sphere. Although we know it's impossible to comb a hairy sphere, having some way to construct an orthonormal basis in a mostly continuous way for the points of the sphere (ideally,

from the point alone) is useful in many situations. One such method comes from [Frisvad]; basically, we can start with an orthonormal basis at the top of the sphere, and then rotate this downwards along each line of longitude, covering the entire sphere except for the south pole.



orthonormal bases



tangent vectors only

We can rotate the coordinate frame using a variety of methods; one would be to determine the rotation needed using the formula for rotating between two points above, and then to compute the result of transforming $(1,0,0)^T$ and $(0,1,0)^T$ by this rotation. If $(x,y,z)^T$ is a unit vector with $z \neq -1$, we then have that

$$\boldsymbol{b} = (1 - \frac{x^2}{1+z}, -\frac{xy}{1+z}, -x)$$

and

$$\boldsymbol{t} = (-\frac{xy}{1+z}, 1 - \frac{y^2}{1+z}, -y)$$

are perpendicular to $(x,y,z)^T$, to each other, and form the rest of a coordinate frame containing $(x,y,z)^T$ as one of its axes. When z=-1, we just output the coordinate frame $(0,0,-1)^T, (0,1,0)^T, (1,0,0)^T$.

Unfortunately, as a result of the singularity in this method, we wind up having numerical issues when z is close to -1. There are at least two approaches to solve this problem, both of which work just fine in practice. [Max] finds the optimal cutoff point for determining when we're at the south pole, which works pretty well. [Duff et al.] and [Reynolds] extend the original approach in a nice way: instead of propagating a coordinate frame downwards from the north pole, we can also propagate a coordinate frame upwards from the south pole and meet at the equator! Although we now have a discontinuity along the entire equator, we can define an appropriate coordinate frame at every point of the sphere and arrange the frames so that the frames on the southern hemisphere are just a flipped version of the frames on the northern hemisphere.

Converting to matrices and back. Even if your internal rotation representation is based on unit quaternions, rotation matrices can come in quite handy, whether you need to express a rotation in a matrix format which a computer renderer can easily read, or to constrain your rotation to satisfy some set of constraints.

Since rotations are linear transformations (in particular, the action of a quaternion $\boldsymbol{q}$ on $\mathbb{R}^3$ by $\boldsymbol{x} \to \boldsymbol{qxq^{-1}}$ is linear), we can start out by computing $\boldsymbol{qiq^{-1}}$, $\boldsymbol{qjq^{-1}}$, and $\boldsymbol{qkq^{-1}}$:

$$\boldsymbol{qiq^{-1}} = \left(q_w^2 + q_x^2 - q_y^2 - q_z^2, 2(q_x q_y + q_w q_z), 2(q_x q_z - q_w q_y)\right)^T$$
$$\boldsymbol{qjq^{-1}} = \left(2(q_x q_y - q_w q_z), q_w^2 - q_x^2 + q_y^2 - q_z^2, 2(q_w q_x + q_y q_z)\right)^T$$
$$\boldsymbol{qkq^{-1}} = \left(2(q_w q_y + q_x q_z), 2(q_y q_z - q_w q_x), q_w^2 - q_x^2 - q_y^2 + q_z^2\right)^T$$

Since $\boldsymbol{q}$ is a unit quaternion, we then have that rotating a point by $\boldsymbol{q}$ is equivalent to multiplying by the 3x3 matrix

$$\begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_w q_y + q_x q_z) \\ 2(q_x q_y + q_w q_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) \\ 2(q_x q_z - q_w q_y) & 2(q_w q_x + q_y q_z) & 1 - 2(q_x^2 + q_y^2) \end{pmatrix}.$$

Trying to convert back from a (numerically computed) rotation matrix to a quaternion usually results in an overdetermined system; we have seven constraints and four parameters (with a choice of sign). We can just look at the values from some subset of the matrix, for instance, or use a nonlinear least squares method to try to find optimal values for q_w, q_x, q_y, and q_z.

6. This Paper, but in Higher Dimensions

In two dimensions, rotations have one degree of freedom; there are no axes to choose, and any rotation is parameterized by its angle. In three dimensions, rotations are parameterized by an axis and an angle, for a total of three degrees of freedom. In four dimensions and higher, we cannot rely upon the axis-angle model; instead, we can decompose any rotation into a product of rotations in two-dimensional subspaces.

If you need to compute with rotations in higher dimensions, it might make the most sense to work with rotation matrices (which are the set of orthogonal matrices with determinant 1) directly. In general, in n dimensions orthogonal matrices have n^2 parameters and $n(n+1)/2$ constraints, for a total of $n(n-1)/2$ degrees of freedom.

In fact, since a single unit quaternion has three degrees of freedom and four-dimensional rotations have six degrees of freedom, it turns out we can represent a rotation in four dimensions by a pair of quaternions. In higher dimensions, we can talk about the *octonions*, an eight-dimensional algebraic system with inverses and a norm similar to the quaternions, but *without the associative property* (that is, $(ab)c$ may no longer be equal to $a(bc)$). These can then be used to represent eight-dimensional rotations, although things get complicated.

The Hairy Ball Theorem is true in all odd dimensions (where combing a sphere is well defined); however, we *can* comb the sphere in any even dimension! One easy way to do so is the following: For any vector $(x_1, x_2, \ldots, x_{2n-1}, x_{2n})^T$, the vector

$$(-x_2, x_1, -x_4, x_3, \ldots, -x_{2n}, x_{2n-1})^T$$

is perpendicular to $(x_1, \ldots, x_{2n})^T$. In fact, in four dimensions, we can do even better, and provide a full coordinate frame: if $(w, x, y, z)^T$ is a unit four-vector, then the four vectors

$$(w, x, y, z)^T,$$
$$(-x, w, z, -y)^T,$$
$$(-y, -z, w, x)^T,$$
$$(-z, y, -x, w)^T$$

are all of length 1 and are all orthogonal to each other. Surprisingly, these form exactly the matrix we used to represent quaternions (and is in fact a reshuffled multiplication table on 1, I, j, and k). We can also do this in two dimensions: the vectors (x, y) and (y, -x) have the same length and are perpendicular to each other. Although this cannot be done in six dimensions, while the octonions give us a way to do this in eight dimensions.

Perhaps the most surprising thing is that past eight dimensions, the space of normed division algebras just stops; the real numbers, the complex numbers, the quaternions, and the octonions are the only algebras over the reals which have a norm and allow division by nonzero numbers. This is Hurwitz' theorem, a nice proof of which can be found in Conway and Smith's *On Quaternions and Octonions*.

As it turns out, the space of rotations is never simply connected (for instance, in dimension 2, the space of rotations is diffeomorphic to a disk, which has a hole in it), but in dimensions above 2 we can always do something like we did with quaternions, double-covering the space of rotations in order to get a space which is simply connected.

7. Sources and Further Reading

Quaternions in General:

John H. Conway and Derek Smith. 2003. On Quaternions and Octonions. A K Peters, Ltd., Natick, MA, USA.

Andrew J. Hanson. 2006. Visualizing Quaternions. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Eugene Salamin. 1979. Application of Quaternions to Computation with Rotations. Stanford AI Lab, Stanford, CA, USA.

Finding Orthonormal Bases:

Tom Duff, James Burgess, Per Christensen, Christophe Hery, Andrew Kensler, Max Liani, and Ryusuke Villemin. 2017. Building an Orthonormal Basis, Revisited. *Journal of Computer Graphics Techniques (JCGT)* 6, 1 (March), 1-8.

J. R. Frisvad. 2012. Building an orthonormal basis from a 3D unit vector without normalization. *Journal of Graphics Tools 16*, 3, 151-159.

Nelson Max. 2017. Improved Accuracy when Building an Orthonormal Basis. *Journal of Computer Graphics Techniques (JCGT)*, 6, 1 (March), 9-16.

Marc B. Reynolds. 2016. Orthonormal Basis from Normal via Quaternion Similarity. URL: http://marc-b-reynolds.github.io/quaternions/2016/07/06/Orthonormal.html