A Practical Parameterization of 2 and 3 Degree of Freedom Rotations

F. Sebastian Grassia

May 13, 1997

CMU-CS-97-143

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents the RQ (Reparameterized Quaternion) parameterization for three degree of freedom (DOF) rotations as an alternative to quaternions for the applications of differential control and dynamic simulation. Because it requires just three Euclidean parameters instead of the four non-Euclidean parameters of quaternions, RQ is more efficient, less difficult to incorporate into large systems, and able to make use of Euclidean interpolants, such as cubic splines. While RQ does possess singularities in its parameter space, we show how they can be easily avoided for the intended applications. After discussing RQ's properties and limitations when applied to interpolation and spacetime optimizations, we derive an extension to two DOF rotations, and show how it can be used to model a ball-and-socket joint to a very good approximation.

Keywords: animation, rotations, quaternions, differential control, interpolation

1 Introduction

Two primary goals in computer animation are making objects and characters move realistically and controllably. There are many different approaches and ideologies for achieving these goals, but they must all share a common concern: choosing a representation for the underlying degrees of freedom (DOFs) of the objects they animate. In choosing a representation, we must consider three important factors:

- How well does it model the way the object or joint actually moves?
- How does it behave in the intended application areas? This includes numerical properties and complexity of computing the necessary quantities.
- What level of complexity does it impart to the system we are building?

The objects and characters being animated today have many different kinds of DOFs, some of which determine structural poses, and some of which specify finer, more "organic" quantities like facial expressions and hair. We are concerned with the first of these two groups, which are known as *rigid body DOFs*, and consist of translations and rotations. Rigid body DOFs are the primary concern of most commercial keyframe animation systems, inverse kinematics algorithms, and ongoing research into dynamic simulation and optimization.

Translations can be represented as linear functions of a set of DOFs, and are therefore straightforward to encode, understand, and use. Rotations, however, are nonlinear functions of their DOFs, which increases the complexity of algorithms needed to manipulate and control them. Further, the mathematical space in which rotations are defined is non-Euclidean, which complicates the numerical issues involved in using derivatives of rotational DOFs, which we must do for inverse kinematics, dynamic simulation, and optimization.

The work presented in this paper grew out of a profound dissatisfaction with available representations of rotations for use in our research, which entails differential control, optimization, and interpolation of rotations in the context of human character animation. We will discuss the shortcomings of the parameterizations currently used in graphics in section 2. Then in section 3 we will present a new parameterization for three DOF rotations that, while not without limitations, we believe represents a practical step forward for controlling and simulating with rotations. In section 4 we demonstrate why this is so, and also discuss the limitations of the parameterization when applied to interpolation and spacetime optimization. In section 5 we will extend the parameterization to a two DOF rotation that is useful in modeling ball-and-socket joints, and which has few practical limitations. We also include in the appendices formulae and C code for computing and differentiating with the new parameterization.

2 Background

The applications that are driving both research and industry today include inverse kinematics, dynamic simulation, and spacetime optimization. Although these applications vary considerably in the size and complexity of the motion problems they address, they all require the same basic functionality:

1. the ability to compute positions and orientations of points on body parts as functions of the parameters
2. the existence of and the ability to compute derivatives of these positions and orientations with respect to the parameters
3. the ability to interpolate smoothly and controllably between sequences of parameter keyframes

Regardless of the rotation parameterization and high-level problem structure, rotations are generally converted into 3x3 or 4x4 transformation matrices as a convenient intermediary form. This gives us a common baseline for computation across various parameterizations: we will be interested in computing a

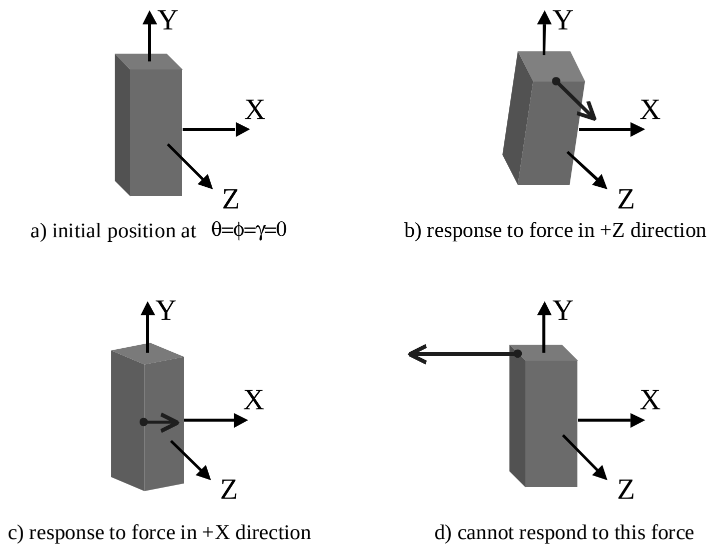

Figure 1: Gimbal lock in YXY Euler angles. The box can only rotate about its center, and the rotation is parameterized by three Euler angles (θ,φ,γ), rotating first about the Y axis, then the X axis, then Y again (this particular sequence is sometimes used by physicists). When all three angles are zero, the first and last rotation axes are identical, and it is impossible to rotate the box about the Z axis, no matter how great a force we apply.

rotation matrix and the partial derivatives of that rotation matrix with respect to its parameters. Therefore, if the parameterization possesses an n element vector of DOFs v, we must be able to compute:

$$\mathbf{R}(v) \quad \text{and} \quad \frac{\partial \mathbf{R}}{\partial v}$$

where $\mathbf{R}$ is a 4x4 matrix and $\partial \mathbf{R}/\partial v$ is a 4x4xn tensor, that is, an n element vector of 4x4 matrices, each of which is the partial derivative of $\mathbf{R}$ with respect to one of the n parameters in v.

Interpolating three DOF rotations is problematic because rotations are non-Euclidean, and all of the most convenient interpolants, such as the family of cubic splines, work well only for Euclidean quantities. A best-case scenario for a parameterization of rotations would allow us to use ordinary cubic splines for interpolation of rotations.

Now that we know the quantities we are interested in computing, we can examine the parameterizations in use today and see why they are unsatisfactory.

2.1 Euler Angles

An Euler angle is a DOF that represents a rotation about one of the coordinate axes. There are three distinct formulae $\mathbf{R}_x$, $\mathbf{R}_y$, and $\mathbf{R}_z$ for computing rotation matrices, depending on which coordinate axis the Euler angle represents. These formulae involve trig functions of the Euler angle, and, although they are nonlinear, their partial derivatives are easy to compute.

The Euler angle is really a one DOF rotation, and in this capacity it functions well in all respects. In practice, when we need a two or three DOF rotation, we replace $\mathbf{R}$ with a sequence of single DOF Euler rotation matrices. There are several such sequences that can be used to represent a three DOF rotation, and this in itself is problematic, since converting from one to the another is nontrivial.

This concatenation of single DOF rotations causes more serious problems because, no matter what sequence we choose, there will always be at least one configuration (*i.e.* values for each of the Euler angles) where the first and last axes of rotation in the sequence align, which causes two of the three partial derivatives to become linearly dependent, resulting in a loss of a rotational degree of freedom. This situation, called gimbal lock, means that when in such a configuration, there is a direction in which we can pull or push with infinite force and yet be unable to induce a rotation because no combination of differential change of the three Euler angles will produce any rotation in that direction. This situation is illustrated in Figure 1 for the sequence YXY of Euler angles, which is capable of representing any orientation, but is gimbal locked at (0,0,0).

Interpolation of multiple DOF rotations is also a problem for Euler angles because rotations confound axes (see [5] for more details), and Euler angles ignore this fact. When interpolating sequences of orientations, the Euler angle model interpolates around each of its three axes simultaneously and independently. This shows up, especially for freely spinning objects, as strange contortions and gyrations.

In short, Euler angles are not a good model for multiple DOF rotations, because they represent such rotations as a sequence of single DOF rotations about coordinate axes. Despite the persistent presence of Euler angles in commercial keyframe systems, they are unsuitable for inverse kinematics, dynamic simulation, and spacetime optimization of three DOF rotations.

2.2 Quaternions

Euler himself showed that any orientation can be achieved by a single rotation about the proper axis (in general *not* one of the coordinate axes). Quaternions, as presented to the graphics community by Shoemake [5], can be used as an elegant implementation of this model that also produce smooth, natural-looking interpolations of sequences of orientations. Quaternions are 4D objects that are thought of as having a 3D-vector component and a scalar component. If the quaternion has unit length, then the scalar component can be interpreted as the cosine of ½ the angle of rotation, and the vector as the axis of rotation scaled by sine of ½ the angle of rotation, *i.e.*

$$q = \left[\sin(\tfrac{1}{2}\theta)\, ax,\ \cos(\tfrac{1}{2}\theta)\right]$$

where θ is the angle of rotation and ax is a unit length axis of rotation.

The formula for computing a rotation matrix $\mathbf{R}$ from the unit quaternion is simple and inexpensive[1], as are the partial derivatives of $\mathbf{R}$ with respect to each of the four quaternion components. Further, the partial derivatives exist and are linearly independent over the entire surface of the unit 4-sphere on which quaternions are defined, which means that unit quaternions are free from gimbal lock when used to control orientations.

However, this behavior does not come without a cost. A quaternion uses four parameters to represent a three DOF rotation, which means that there is always a direction of change (*i.e.* partial derivative) for the quaternion that induces a transformation that is *not* a rotation. This direction is, in fact, along the 4-vector defined by the quaternion, since any change along this direction would cause the length of the quaternion to become non-unit, and thus no longer a rotation.

Several strategies have been developed to deal with this problem. The obvious approach is to add explicit constraints that force quaternions to maintain their unit length. If the intended application supports nonlinear constraints, this can be accomplished with a single scalar constraint for each quaternion, which forces the quaternion to maintain its unit length. While this does work for differential control and simulation, it forces the systems of equations that such applications solve to be larger than necessary, since we have an extra parameter and an extra constraint for each rotation.

Gleicher [3] eliminates the need for the extra constraints by building *autonormalization* into the formula for computing $\mathbf{R}$ from q. In this scheme, q can take on any value except (0,0,0,0), and the

[1] The rotation matrix $\mathbf{M}$ presented in [5] transforms *row* vectors, so is actually the transpose of the matrix $\mathbf{R}$ that we desire for transforming column vectors.

components of q are replaced by (analytically) normalized versions in the formulae for computing $\mathbf{R}$ and its four partial derivatives. The result is that differential change in the quaternion can no longer cause it to cease representing a rotation; however, the quaternion *can* still change along the direction of its length, but doing so now has no tangible effect whatsoever. Thus for each quaternion there is a DOF that induces no change in configuration. This is unacceptable for applications such as dynamic simulation (and some control algorithms) that invert a quantity known as the *mass matrix*, because the ineffectual DOF causes the mass matrix to become singular.

Baraff and Witkin [1] take a different approach, discarding the quaternion parameters as the basis for differentially controlling a quaternion rotation. They reason as follows: given a current orientation represented by a quaternion q, the orientation can differentially change by acquiring an angular velocity ω, which is a 3-vector whose direction gives an axis of rotation and whose magnitude is the rate of rotation in radians / time unit about that axis. The underlying representation for the rotation is still a quaternion, and the formula for computing $\mathbf{R}$ is the same, but we no longer compute the four partial derivatives of $\mathbf{R}$ wrt. q. Instead, we compute three matrices that specify how the orientation will change given a differential angular velocity with components along each of the coordinate axes. This formulation is free from gimbal lock, and is also quite efficient, since there are no extra constraints required, and the number of "DOFs" that appear in the equations of motion is three for each rotation.

There is, however, a heavy cost in code complexity, since the number of parameters defining the rotation (four) is different from the number of DOFs we use to control it (three). Baraff provides formulae for computing the time derivative of a quaternion from a given angular velocity, which must be performed before integrating the differential equations that often arise in the intended applications. To make the most efficient use of this technique, the disparity between number of parameters and number of DOFs must be propagated through large code segments. Furthermore, it cannot be made to work for optimization, so its use is limited to differential control and simulation.

Recall now that in addition to derivatives, we are also interested in interpolation of orientations. The surface of the unit 4-sphere really *is* the correct place to interpolate rotations, because, with antipodes identified, it possesses the same metric and group structure as $SO(3)$, the rotation group. Shoemake [5] constructs *spherical Bezier curves* on the surface of the unit 4-sphere, and Barr *et. al.* [2] use optimization techniques to solve for a series of interpolating quaternions that minimizes tangential acceleration on the surface of the 4-sphere while satisfying angular velocity constraints at the key orientations. However, neither technique admits an analytical closed form (which precludes altogether their use as basis functions for representing time-varying orientations), and the quantities available for controlling the shape of the interpolation are difficult to visualize.

In summary, quaternions are quite well behaved for most possible applications, but they incur an overhead in efficiency and/or code complexity. Furthermore, the available means of interpolation are not easily controlled.

3 RQ Parameterization

What we would really like is a parameterization that behaves like a quaternion but is embedded in Euclidean space such that the number of parameters equals the number of DOFs. Such a parameterization would be simple to control and allow us to use ordinary cubic splines to interpolate rotations. This goal is, of course, unrealizable, as it is a standard exercise in a topology text to show that $SO(3)$ *cannot* be mapped into R^3 without singularities, *i.e.* gimbal lock. Our more realistic goal is to trade off some of the robustness of quaternions for a Euclidean embedding, trying to minimize the loss with respect to the mathematical quantities we need to compute.

We have devised a parameterization called RQ (Reparameterized Quaternion) that meets these criteria. RQ represents a rotation by a 3-vector v, whose direction gives the axis of rotation and whose magnitude is the amount of rotation about that axis (in radians). There are several known methods of translating such an *axis/angle* parameterization directly into a rotation matrix, such as *Rodrigues' formula* and

Cayley's parameterization [4]. However, we use quaternions as an intermediate, since they are computationally equivalent to the direct methods that possess the properties we desire, and are easier to analyze; the formulae for computing the individual quaternion elements as functions of v are as follows:

Let

$$\theta = |v| = \sqrt{v_x v_x + v_y v_y + v_z v_z}$$

Then

$$q = \left[\frac{\sin(\frac{1}{2}\theta)}{\theta} v, \ \cos(\tfrac{1}{2}\theta) \right]$$

There are two immediate observations to make about this parameterization. First, the quaternion will always have unit magnitude regardless of the value of v, which means every point in RQ parameter space represents a rotation – we need take no special measures to insure the quaternion has unit length. Second, there seems to be a singularity in the parameter space at $v = [0,0,0]$, since at this point θ is zero. However, $\sin(\frac{1}{2}\theta)/\theta$ is an *indeterminate form*, which has continuous value and derivative in the limit as θ goes to zero (in fact, it is the well-known sinc function). We can compute the value at the limit using *l'Hôpital's Rule,* which states:

> Suppose $\mathrm{a}(x)$ and $\mathrm{b}(x)$ are two functions that are continuously differentiable in the neighborhood surrounding (but not including) the point d, and $\mathrm{b}(x)$ is nonzero in the neighborhood. Suppose further that
>
> $$\lim_{x \to d} \mathrm{a}(x) = 0 = \lim_{x \to d} \mathrm{b}(x) \qquad \text{If so, then:} \qquad \lim_{x \to d} \frac{\mathrm{a}(x)}{\mathrm{b}(x)} = \lim_{x \to d} \frac{\mathrm{a}'(x)}{\mathrm{b}'(x)}$$

Since the conditions are met, we can use l'Hôpital's Rule to compute the vector part of q in the limit as θ goes to zero by differentiating with respect to θ:

$$q_v = \frac{\frac{\partial}{\partial\theta}\sin(\frac{1}{2}\theta)}{\frac{\partial}{\partial\theta}\theta} v = \tfrac{1}{2}\cos(\tfrac{1}{2}\theta)\, v$$

This gives the expected result that $v = [0,0,0]$ maps to the zero rotation quaternion $q = [0,0,0,1]$. Furthermore, away from zero, the l'Hôpital's Rule formulae are numerically indistinguishable from the original formulae out to an appreciable fraction of a radian. Therefore, as long as we keep the transition point from using one set to the other reasonably small (say, square root of machine precision), q will be a continuous function of v to within machine precision.

3.1 Derivatives

Since we are actually reparameterizing quaternions, we can compute the derivatives of $\mathbf{R}$ with respect to its RQ parameters by applying the chain rule of differentiation. That is, we have $\mathbf{R}(q(v))$ and wish to compute $\partial\mathbf{R}/\partial v$, which we can compute as the product:

$$\frac{\partial \mathbf{R}}{\partial v} = \frac{\partial \mathbf{R}}{\partial q}\frac{\partial q}{\partial v}$$

Since we already know how to compute the partial derivatives $\partial\mathbf{R}/\partial q$, the only new quantities we need are the twelve partial derivatives of the quaternion with respect to its RQ parameters (here $\partial q/\partial v$), which are given in Appendix A. Additionally, Appendix C discusses the supplemental C source code for computing $\mathbf{R}$, $\partial\mathbf{R}/\partial v$, and other quantities presented later in this paper.

3.2 Tradeoffs

So far the RQ parameterization seems to fulfill all of our requirements, implementing the axis/angle rotation model in three Euclidean parameters. Before we can discuss its application to the animation problems we have talked about, we must be clear about what we have given up.

3.2.1 Combining Rotations

One nice feature of quaternions is that there exists a multiplication operator that can be used to combine rotations. If q_1 and q_2 are unit quaternions, then the combined rotation q_3 that is the result of first rotating by q_1 and then by q_2 can be calculated as $q_3 = q_2 \circ q_1$, where '$\circ$' represents quaternion multiplication, as defined in [5]. RQ (and any pure axis/angle parameterization) possesses no analogous operation. We can combine two RQ vectors using any combination of binary Euclidean operators, but in no case will the result (in general) be equivalent to combining the two rotations – to do so we would need to convert the RQ vectors to their corresponding quaternions, perform quaternion multiplication, then convert back, incurring several trig and one inverse trig functions.

Fortunately, this operation is typically not needed in the applications we have talked about. Rotations are changed only by direct parameter manipulation, or incrementally, via their derivatives. When they are combined, it is usually in the context of a transformation hierarchy, where all rotations have already been converted into transformation matrices.

3.2.2 Singularities

For the purposes of control and simulation, the principal advantage of quaternions over Euler angles is their freedom from gimbal lock. We already know that the RQ parameterization must have singularities, so if it is to be useful, we must locate all singularities and show how they can be avoided at a cost that is outweighed by the benefits of RQ.

As previously mentioned, gimbal lock is equivalent to the disappearance or loss of linear independence of rotational derivatives. Therefore we can determine the singularities in RQ space by analytically examining the partial derivatives $\partial q / \partial v$. Doing so reveals that a singularity occurs whenever θ is an integer multiple of 2π. Note that this does *not* apply to θ=0 because the forms derived from l'Hôpital's Rule are well behaved. This makes intuitive sense, since a rotation of 2π about *any* axis is equivalent to no rotation at all – the entire shell of points 2π distant from the origin (and 4π, *etc.*) collapses to the identity in $\mathrm{SO}(3)$. This singularity manifests itself in the following way: whenever $|v| = 2\pi$, differentially changing v along its own axis induces continued rotation about v, but differential movement in any direction perpendicular to v induces a velocity for v *in the tangent plane of the singularity shell*, thus (differentially) keeping v on the shell and inducing no change in orientation. In other words, when v hits one of these shells, it can differentially keep rotating in the direction it is already going, but in no other direction. Fortunately, for the applications of differential control and physical simulation, avoiding these singularities is easy, due to two facts.

First, if we consider just the RQ parameter space inside the first singularity shell, each unique orientation (except the rotation of zero radians) has two possible parameterizations: as a rotation of θ radians about v, and as a rotation of $2\pi - \theta$ radians about $-v$ (quaternions have a similar property, in that antipodes q and $-q$ represent the same orientation).

Second, both control and simulation operate by moving forward through time in small steps, and the possible change in DOFs during each step is small (much less than a radian for angular parameters).

Using these facts it is easy to avoid the singularity: at each time step when the rotation is queried for its value and derivative, we examine $|v|$, and if it is close to 2π, we reparameterize v by $(1 - 2\pi/|v|)v$, which is an equivalent rotation, but with better derivatives. Such *dynamic reparameterization* could, in theory, also be applied to avoiding gimbal lock in Euler angles, but whereas here the reparameterization simply scales the current parameters, the corresponding operation on Euler angles involves switching the formula used to compute the rotation matrix and a sequence of inverse trig functions to determine the new parameters [6].

4 Applications

Now that we have seen how the RQ parameterization works and what its theoretical limitations are, it is time to focus on the reason for its creation: the simplification of algorithms that use parameterized rotations. Of the four applications we have discussed in this paper – control, simulation, optimization, and interpolation – we believe RQ to be ideally suited to the tasks of differential control and dynamic simulation. In the following sections we will explain why, and also discuss the complications that arise in applying RQ to interpolation and optimization.

4.1 Differential Control and Dynamic Simulation

One of the motivating applications for this work is differential control, which enables direct manipulation interfaces, inverse kinematics, and real-time control of robotic manipulators. To control the positions and velocities of objects and end-effectors of kinematic assemblies, the only demands imposed by differential control are that the derivatives $\partial \mathbf{R}/\partial v$ be continuous and free from gimbal lock. Since control is only performed at discrete instants in time, the simple dynamic reparameterization technique presented in section 3.2.2 will assure that these demands are always met.

In dynamic simulation applications, we track not only an object's instantaneous position and pose, but also its linear and angular velocity. Linear velocity is stored as a 3-vector that represents the Cartesian direction and magnitude of the velocity. Angular velocity is also represented as a 3-vector ω, whose meaning is nearly identical to that of an RQ vector, except that its magnitude θ represents the *rate* of rotation about the axis rather than absolute orientation, as in RQ.

To update the position and orientation correctly as the simulation moves forward in time, we need, in addition to the derivatives necessary for differential control, a formula for computing the time derivative of the orientation given the current orientation and the instantaneous angular velocity. Baraff [1] derives such a formula for a quaternion orientation:

$$\dot{q} = \tfrac{1}{2}\hat{\omega} \circ q$$

where $\hat{\omega}$ is the angular velocity vector ω extended with a zero scalar component to make a quaternion.

Since RQ does not possess an equivalent operation to quaternion multiplication, we cannot derive a similar formula for $\dot{v}$. However, since we *can* convert a quaternion into an RQ vector inside the first singularity shell at 2π, we can form $\dot{v}$ in terms of $\dot{q}$ like so:

$$v = \mathrm{V}(q), \qquad \text{Therefore} \qquad \dot{v} = \frac{\partial \mathrm{V}}{\partial q}\dot{q}$$

$\mathrm{V}(q)$ involves one arctrig and one trig function, but when we take the derivative with respect to q and put the entire formula in terms of v, much simplification occurs. Appendix B gives a formula for $\dot{v}$ in terms of v and ω that contains roughly the same number and kind of operations that the quaternion multiplication to compute $\dot{q}$ would have required. Thus RQ can be used in place of quaternions for dynamic simulation with equivalent performance and reduction in code complexity.

We claim that RQ is, in fact, ideally suited to these tasks because, despite the small measure we must take to avoid the singularity, implementation of RQ can be much cleaner than quaternions. Not only is RQ modestly more efficient, since it requires only three DOFs rather than four and no explicit constraints, but it is also truly modular, in that no constraints or information need be propagated to code segments outside the core RQ functions.

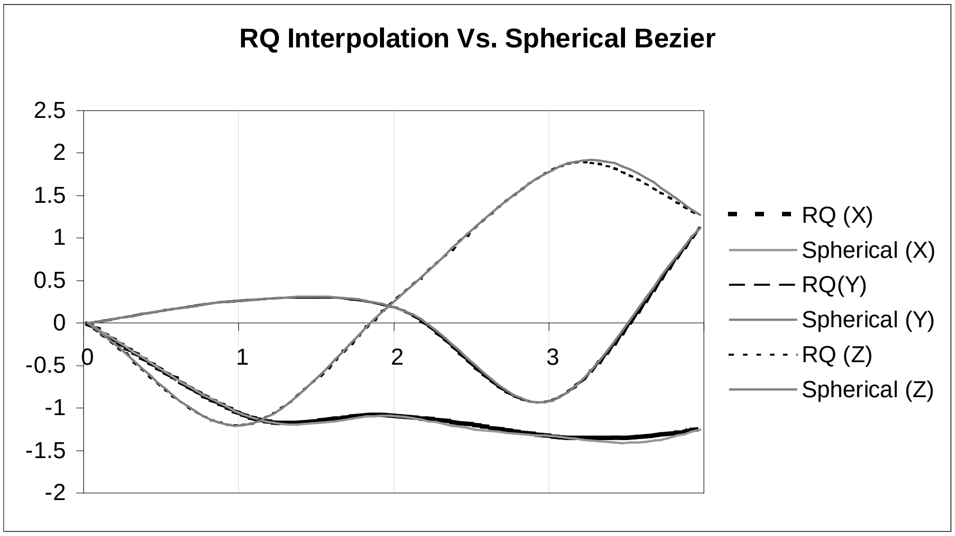

Figure 2: Comparison of interpolated rotations using RQ and spherical Beziers. Five orientations were spaced evenly through four units of time, and the same tangent controls were used in both cases. In the above graph, the "RQ" curves give the interpolated x, y, and z values resulting from Euclidean Bezier interpolation of the RQ parameters; the "Spherical" curves are the results of spherical Bezier interpolation on the unit 4-sphere, converted back into RQ parameters.

4.2 Interpolation

The principal benefit of embedding a rotation in a Euclidean space is the possibility of using Euclidean interpolants on sequences of orientations. Euler angles do not make effective use of Euclidean interpolants because they ignore the interaction of rotation axes. Since RQ is based on the axis/angle model of rotations, we can hope to do better.

However, the singularity shell (which is common to all implementations of the axis/angle model) causes complications because the distance between similar orientations grows without bound as one approaches the singularity. We can illustrate this as follows: let $\gamma = 2\pi - \delta$ for some infinitesimal value of δ. Then if $v_1 = [\gamma, \gamma, \gamma]$ and $v_2 = [-\gamma, -\gamma, -\gamma]$ are two RQ orientations, we note that although they are *extremely* close to each other in SO(3), being only 2δ radians apart, interpolation in RQ space will rotate through nearly 4π radians – probably *not* the desired behavior. Quaternions avoid this wraparound problem because interpolation on the surface of a sphere uses the same distance metric as SO(3), provided consecutive orientations are on the same side of the sphere (*i.e.* their dot product is positive).

Away from the singularity shell, however, the results of using corresponding Euclidean RQ interpolants and non-Euclidean quaternion interpolants are indistinguishable. Figure 2 shows a comparison of the interpolation of five key orientations using both spherical Beziers on quaternion representations of the orientations [5] and regular, Euclidean Beziers on RQ representations. When the quaternions resulting from the spherical Bezier interpolation are converted into RQ parameters, the two methods are observed to correspond quite closely. The fact that we can achieve essentially identical results from interpolating in Euclidean RQ space has two important consequences. First, we can produce natural looking interpolations using *any* sufficiently smooth Euclidean interpolating curve, including the entire family of cubic splines, all of which have efficiently computable closed forms. Second, these Euclidean interpolants have fairly intuitive controls in RQ space; for instance, if we are using a hermite cubic interpolant and we want an object to have greater spin about the y-axis as it is passing through one of the keys, we would simply increase the y component of the tangent control vectors at that key.

So how can we achieve this performance in the face of the problem near the singularity? In the most general case, when orientation keys are allowed to be arbitrarily far apart in parameter space, we cannot. But if we stipulate that n consecutive keys be within one complete revolution of each other (where n is the number of *position* keys necessary to define a spline segment – two for hermites and beziers, four for b-splines), then we can dynamically reparameterize the members (along with tangent controls, if they exist) of each individual segment as a group to ensure that none are near the singularity shell, and then proceed with the interpolation. This fix prevents us from using cardinal and other, related splines that cannot be broken up into segments. Also, to make sure the stipulation is met, we may need to insert "hidden" keys in between the keys specified by the animator if the change in orientation between two consecutive keys is more than one revolution. The need for this last measure is, at least, not exclusive to RQ, since quaternions are incapable of encoding a difference in orientation of 4π or more between consecutive keys.

Finally, we note that in practice, this added complication is often not required. In character animation, actors and objects seldom deviate from the canonical upright position by more than one revolution unless they are in freefall.

4.3 Spacetime Optimization

In spacetime and other optimizations that operate over an entire animation simultaneously, DOFs are not simply angles at one instant in time, but rather rotation-valued functions of time. For instance, the function for a single angle might be a cubic spline defined over the animation time interval, in which case the DOFs are the positions of the spline's control points, and we need to compute derivatives of rotations at various points in time (*i.e.* along the curve) with respect to these control points (simply several further applications of the chain rule to our existing formulae).

Since these functions are essentially interpolating curves, we might think about dealing with the singularity in the same manner as for interpolation. However, we cannot reparameterize the functions dynamically, because doing so will change the shape of the curves, potentially perturbing the optimization out of the state-space well it is currently traversing. Therefore, unless the possible range of rotations is less than 2π (which actually is true when solving for motion displacements rather than motions), we do not recommend the RQ parameterization for spacetime optimizations involving three DOF rotations.

5 Ball-And-Socket Joints

The RQ parameterization can be even more useful for modeling other types of joints than freeform three DOF rotations. Let us consider the type of ball-and-socket joint found at the shoulders and the hips. If we were to map out the legal range of motion of these joints in configuration space using RQ or any other parameterization, we would get a complex, non-regular hyper-surface. However, we can create a joint that approximates the actual motion and limits of motion fairly well using the following scheme, illustrated in Figure 3.

We model the motion of, say, an arm, as a single DOF twist about the principal axis of the outstretched arm, followed by a two DOF swing of the arm that incurs no twist about the arm's axis. The last is important because we wish to place limits on the allowable twist of the arm; this would be very difficult if both rotations could generate twist, but since only the first rotation can twist, the limits can be easily enforced with linear inequality constraints. As for creating suitable limits for the swing rotation, we will discuss more accurate schemes later, but for now we will stipulate a maximum rotation in any direction, so that a rigid arm will be able to sweep out a circular patch on the surface of a sphere.

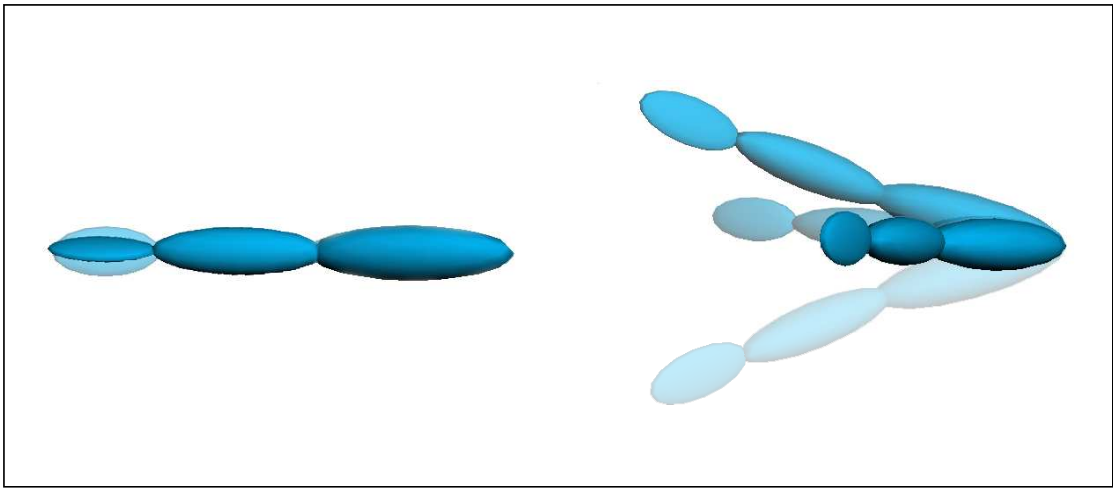

Figure 3: Degrees of Freedom for Ball-And-Socket Joint. The "arm" pictured above uses the ball-and-socket model proposed here at its shoulder joint. In the image on the left, the twist DOF is being exercised, and the arm spins about its axis. On the right the two swing DOFs are used to make the hand swing out a circle, starting at the bottom position and finishing closest to us. Note that in this entire motion there is no spin about the arm's axis.

We claim that this is a good model of ball-and-socket joints because the amount of twist in such joints is fairly independent of the swing angle, and the sweep limits, while never really circular, are often rather ellipsoidal. Although we are constructing a single joint from a sequence of two rotations, there is no possibility of gimbal lock, since the axes of rotation are, by definition, mutually orthogonal.

The twist can be parameterized as either an Euler angle, if the principal axis for the limb happens to be a coordinate axis, or as a single DOF RQ rotation, whose derivation is straightforward given the following presentation of two DOF rotations. Parameterizing the twist-less swing rotation would be tricky using Euler angles, but given an axis/angle model like RQ it is simple, since a necessary and sufficient condition for a rotation that contains no spin about a specified vector is for the rotation axis to be orthogonal to the vector.

This means that to achieve our desired two DOF rotation, all we need is an RQ rotation vector that lies in the plane perpendicular to the major axis of the limb in its canonical (zero rotation) position. We could do this with an explicit constraint, but we can more elegantly create such a rotation by reparameterizing the RQ rotation itself like so: pick any two orthogonal, unit vectors in the perpendicular plane; these vectors, s and t become a 2D basis for our desired swing rotation v, an RQ vector that we compute like so:

$$v = \alpha s + \beta t$$

where α and β are now the two DOFs, which we can form into a 2D RQ vector that we will call r. Now since s and t are unit vectors, the length of r is the magnitude of the swing rotation, So to place a limit on the swing we need only place an inequality constraint on this vector's magnitude:

$$|r| \leq \text{limit}$$

In fact, it is not much more difficult to impose a more accurate, ellipsoidal angular limit. If we know the angular limits of rotation along the major and minor axes of the ellipse, a and b, then we must choose s and t to correspond to the major and minor axes in the plane, and pose the following inequality constraint instead of the one above:

$$\left(\frac{r_\alpha}{a}\right)^2 + \left(\frac{r_\beta}{b}\right)^2 \leq 1$$

Since the two DOF rotation is formed from a reparameterization of the three DOF rotation, all the same formulae apply, but now we are computing the 3D RQ vector v from the 2D RQ vector r. To compute the derivatives of $\mathbf{R}$ with respect to r, we simply apply the chain rule once more:

$$\frac{\partial \mathbf{R}}{\partial r} = \frac{\partial \mathbf{R}}{\partial q}\frac{\partial q}{\partial v}\frac{\partial v}{\partial r} \qquad \text{where} \quad \frac{\partial v}{\partial r_\alpha} = s \quad \text{and} \quad \frac{\partial v}{\partial r_\beta} = t$$

This two DOF rotation is suitable for all the same applications as the three DOF, and additionally optimization. Recall that the three DOF rotation could not be used for optimization because dynamically reparameterizing the control points to avoid the singularity shells was unacceptable. But the first shell occurs at an angular magnitude of 2π, and since the angular motion limits for the types of joints the two DOF rotation models should always be much less than 2π, crossing any of the shells should never be a problem.

6 Conclusion

We have presented a new parameterization of three and two DOF rotations. Although it is technically not as robust as the straight quaternion parameterization, it performs well in practice. In practical terms of flexibility, ease of interaction, and ease of integration into large systems, we have found RQ to outperform quaternions and all other parameterizations discussed here.

At the beginning of this paper, we stated three evaluation criteria by which any parameterization must be judged. We will conclude by measuring RQ against them.

Accuracy in modeling motion. RQ is a faithful implementation of the axis/angle model of rotations. As such it is both a complete representation of orientations, and easily extendible to various forms of restricted rotations. We have demonstrated this by deriving a two DOF variant that is quite useful in modeling a ball-and-socket joint.

Behavior. Since RQ possesses singularities in its parameter space, it will experience gimbal lock at complete revolutions about any axis. We have shown, however, how this situation can be easily avoided for most potential applications, excluding optimization. Derivative computation using RQ entails chain rule evaluation, but we have provided code (discussed in Appendix C) that encapsulates the computation efficiently.

Code Complexity. Using RQ has simplified our own code considerably over various strategies of using quaternions. The "unclean" aspects of RQ (two sets of formulae for computing with it; the need for dynamic reparameterization) can be safely modularized and do not propagate beyond the rotation object.

Acknowledgements

Thanks to David Baraff for helpful discussions on singularities and computing $\dot{v}$, to Matt Mason and Mike Erdmann for providing a handle on the robotics knowledge base in this area, and to David Baraff, Matt Mason, and Zoran Popović for critical reads of this paper.

Bibliography

1. David Baraff. Rigid Body Simulation. In Andrew P. Witkin, organizer, *Physically Based Modeling Course Notes (SIGGRAPH '95),* pages G1-G68, July 1995.
2. Alan H. Barr and Bena Currin and Steven Gabriel and John F. Hughes. Smooth interpolation of orientations with velocity constraints using quaternions. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings),* volume 26, pages 331-320, July 1992.
3. Michael Gleicher and Andrew Witkin. Through the Lens Camera Control. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings),* volume 26, pages 331-340, July 1992.
4. Richard M. Murray and Zexiang Li and S. Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Boca Raton, 1994, pages 22-34,73.
5. Ken Shoemake. Animating Rotations with Quaternion Curves. In Brian A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings),* volume 19, pages 245-254, July 1985.
6. Ken Shoemake. Euler Angle Conversion. In Paul Heckbert, editor, *Graphics Gems IV,* Academic Press, 1994, pages 222-229.

Appendix A: Derivatives

As in section 3, let v be a 3-vector that RQ parameterizes a quaternion $q = [\, q_x, q_y, q_z,\ q_w \,]$ (where q_w is the scalar part), and $\theta = \ |v|\ = \sqrt{v_x v_x + v_y v_y + v_z v_z}$

Now also let i range over the three components of q that make up its vector part, and let j range over the components of v.

The formulae for computing the partial derivatives of q with respect to v are, in the usual case where $\theta >> 0$:

$$\frac{\partial q_w}{\partial v_j} = -\tfrac{1}{2} v_j \frac{\sin(\frac{1}{2}\theta)}{\theta}$$

When $i = j$: $$\frac{\partial q_i}{\partial v_j} = \tfrac{1}{2} v_j^2 \frac{\cos(\frac{1}{2}\theta)}{\theta^2} - v_j^2 \frac{\sin(\frac{1}{2}\theta)}{\theta^3} + \frac{\sin(\frac{1}{2}\theta)}{\theta}$$

When $i \neq j$: $$\frac{\partial q_i}{\partial v_j} = \tfrac{1}{2} v_i v_j \frac{\cos(\frac{1}{2}\theta)}{\theta^2} - v_i v_j \frac{\sin(\frac{1}{2}\theta)}{\theta^3}$$

In the limit as $\theta \to 0$ and we use l'Hôpital's Rule to compute q, we use the following forms to compute the partial derivatives:

$$\frac{\partial q_w}{\partial v_j} = -\tfrac{1}{4} v_j \cos(\tfrac{1}{2}\theta)$$

When $i = j$: $$\frac{\partial q_i}{\partial v_j} = \tfrac{1}{2}\cos(\tfrac{1}{2}\theta)$$

When $i \neq j$: $$\frac{\partial q_i}{\partial v_j} = 0$$

Appendix B: $\dot{v}(v,\omega)$

Assume the same definitions as Appendix A.

In the normal case where $\theta >> 0$ (note that $\cot(\frac{1}{2}\theta) = \cos(\frac{1}{2}\theta)/\sin(\frac{1}{2}\theta)$, and we should have already computed both these functions):

Let $$p = v \times \omega \qquad \gamma = \theta\cot(\tfrac{1}{2}\theta) \qquad \eta = \frac{v \cdot \omega}{\theta}\left(\cot(\tfrac{1}{2}\theta) - \frac{2}{\theta}\right)$$

Then $$\dot{v} = \tfrac{1}{2}\left(\gamma\omega - \eta v + p\right)$$

In the limit as $\theta \to 0$, the formula simplifies to: $\dot{v} = \omega$

Appendix C: Sample Code

Sample C source code for computing the rotation matrix $\mathbf{R}$, its partial derivatives $\partial\mathbf{R}/\partial v$, and $\dot{v}(v,\omega)$ for the two and three DOF versions of RQ rotations can be found at http://www.cs.cmu.edu/~spiff/rq. This code can be used to start computing with RQ immediately, but as it stands, it is not very efficient. We have purposely omitted the caching necessary to make the implementation efficient, because the form of cache storage will depend on the modularization strategy.

In particular, the values `theta, cosp, sinp`, and the quaternion `q` should be cached and stored with `v` when the rotation matrix is calculated from `v`, to be used later when calculating derivatives. Also, as noted in the code, when computing the derivatives of the two DOF rotation, the intermediate derivatives of `q` with respect to a three DOF RQ rotation should be computed only once and cached.