

# **Exploiting Quaternions to Support Expressive Interactive Character Motion**

by

**Michael Patrick Johnson**

B.S., Computer Science, Massachusetts Institute of Technology, 1993.

M.S., Media Arts and Sciences, Massachusetts Institute of Technology,  
1995.

Submitted to the Program in Media Arts and Sciences  
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

**MASSACHUSETTS INSTITUTE OF TECHNOLOGY**

February 2003

© Massachusetts Institute of Technology 2003. All rights reserved.

Author .....

Program in Media Arts and Sciences

October 20, 2002

Certified by .....

Bruce M. Blumberg

Asahi Broadcasting Corporation Career Development Associate Professor

of Media Arts and Sciences

Thesis Supervisor

Accepted by .....

Andrew B. Lippman

Chairman, Departmental Committee on Graduate Students



# **Exploiting Quaternions to Support Expressive Interactive Character Motion**

by  
Michael Patrick Johnson

Submitted to the Program in Media Arts and Sciences  
on October 20, 2002, in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

## **Abstract**

A real-time motion engine for interactive synthetic characters, either virtual or physical, needs to allow expressivity and interactivity of motion in order to maintain the illusion of life. Canned animation examples from an animator or motion capture device are expressive, but not very interactive, often leading to repetition. Conversely, numerical procedural techniques such as Inverse Kinematics (IK) tend to be very interactive, but often appear “robotic” and require parameter tweaking by hand. We argue for the use of hybrid example-based learning techniques to incorporate expert knowledge of character motion in the form of animations into an interactive procedural engine.

Example-based techniques require appropriate distance metrics, statistical analysis and synthesis primitives, along with the ability to blend examples; furthermore, many machine learning techniques are sensitive to the choice of representation. We show that a quaternion representation of the orientation of a joint affords us computational efficiency along with mathematical robustness, such as avoiding gimbal lock in the Euler angle representation. We show how to use quaternions and their exponential mappings to create distance metrics on character poses, perform simple statistical analysis of joint motion limits and blend multiple poses together.

We demonstrate these joint primitives on three techniques which we consider useful for combining animation knowledge with procedural algorithms: 1) pose blending, 2) joint motion statistics and 3) expressive IK. We discuss several projects designed using these primitives and offer insights for programmers building real-time motion engines for expressive interactive characters.

Thesis Supervisor: Bruce M. Blumberg

Title: Asahi Broadcasting Corporation Career Development Associate Professor of Media Arts and Sciences



## Doctoral Dissertation Committee

Thesis Supervisor: \_\_\_\_\_

Bruce M. Blumberg  
Asahi Broadcasting Corporation Career Development  
Associate Professor of Media Arts and Sciences  
Program in Media Arts and Sciences, The Media Lab  
Massachusetts Institute of Technology

Thesis Reader: \_\_\_\_\_

Cynthia Breazeal  
LG Career Development Assistant Professor of Media Arts and Sciences  
Program in Media Arts and Sciences, The Media Lab  
Massachusetts Institute of Technology

Thesis Reader: \_\_\_\_\_

Andrew J. Hanson  
Professor of Computer Science  
Department of Computer Science  
Indiana University



*To my parents,  
for their belief, love and support throughout my 13 years at MIT*





## Acknowledgements

This thesis would not have been possible without the help of many friends, colleagues and mentors over my thirteen years at MIT.

I cannot thank Bruce Blumberg enough for being friend, office-mate, mentor, advisor and idea sounding board for the last seven years. He taught me much of what I know about character animation, animal behavior and a lot about Life. He let me investigate my more bizarre ideas like filling stuffed animals with sensors and sticking lasers into gloves. He also knew when to tether me back down to the practical when I got too far into the theoretical clouds. It has been a great pleasure working with him; I look forward to many more years of friendship and intellectual stimulation.

Next, I'd like to thank my two committee members, Cynthia Breazeal and Andrew Hanson for their patience and support and making this thesis much better than it could have been. Cynthia has been instrumental in my thinking through of expressive inverse kinematics and the special problems associated with physical characters, as well as providing a great testbed for trying out these ideas. Andy has been wonderfully patient and helpful as I learned continuous group theory, as well as a great source of inspiration for visualizing quaternions and seeking intuitive, visual understandings of the underlying mathematics. It's been an honor to work with both of you.

There are so many people I have had the pleasure of working with at the Media Lab since 1990 that I cannot possibly name them all. In particular, I'd like to thank my close friends: Chris Wren, who I met as a UROP when I got to the Lab in 1990 and who has been a friend, room-mate, colleague and all-around great guy to know; Andy Wilson, for many stimulating discussions on interface and animation, as well as help on the statistical portion of this thesis and some fun collaborations; Bill Tomlinson, who has been a great friend since we first watched the Chicago Bulls and both said "Hey, Kukoc looks like Steve Buscemi!" at the same time; Jed Wahl for hardcore gaming nights and enlightening discussions of them; Thad Starner and Trevor Darrell, who taught me how to be hard-core and have fun doing it; The *Autonomous Agents* Crew, in particular Brad Rhodes, Lenny Foner, Alan Wexelblat, Nelson Minar, and Agnieszka Meyro, for friendship, great discussions and making my first years here a pleasure; Amy Bruckman, for great discussions of interactive narrative and being one of the few who never hesitated to tell me when I was being a moron, but in the sweetest way; Ali Azarbayejani, for being one of the smartest guys I know and the best person to hang at SIGGRAPH with; Tony Jebara for many conversations on calculus of manifolds; Michael Boyle Johnson ("Wave") for not forcing me to be called "Particle" when I got to the Lab, for many great conversations on character animation and for teaching me how to get into SIGGRAPH parties; Bill Butera, Yuri Ivanov, Brygg Ullmer, John Underkoffler, Wendy Plesniak, Ravi Pappu and Ari Benbasat for keeping me going when it was looking grim and being great to hang out with.

I'd also like to thank my other mentors since I got to MIT, in particular: Dave Koons, for my first UROP position at the Media Lab where I discovered gimbal lock on a VPL Dataglove; my undergrad advisor and UROP supervisor Marc Raibert for introducing me

to computer graphics, physically-based modeling, robotics and for explaining gimbal lock; my Bachelor's advisor Chris Atkeson for getting me into motor learning and showing me that science can involve toys; My UROP supervisor Andrew Moore for teaching me genetic algorithms and reinforcement learning, my first publication and making me a scientist; my Master's advisor Pattie Maes for her wisdom, brilliance, advice, and friendship; Neil Gershenfeld for teaching me much of the physics and advanced numerical techniques I know; Aaron Bobick, Joe Paradiso and Hiroshi Ishii for taking an interest in my work and making me think differently.

None of this work would have been possible without the phantasmagoric menagerie of *Synthetic Characters* grad students since the beginning Beaver Days: Matt Berlin, Rob Burke, Marc Downie, Scott Eaton, Matt Grimes, Michal Hlavac, Damian Isla, Yuri Ivanov, Chris Kline, Ben Resner, Ken Russell, Bill Tomlinson, and Song-Yee Yoon. Also, I'd like to thank the undergrads and animators that made this possible: Jed Wahl, Adolf Wong, Geoff Beatty, and Jesse Gray. Special thanks to Marc Downie for introducing me to Geometric Algebra and being a mathematical sounding board. Also thanks to our assistant Aileen Kawabe for helping me tie up all the logistic loose ends and being very cool to have around.

Many other friends outside the lab have been great sources of conversation, inspiration and motivation: The Thirsty Magic gaming crew, especially Brian & Jen, Jeff & Dierdre, James and Derek; my long-time friend and roommate Jin Choi for all the late-night gaming nights and support; my friends at the CBC.

Very special thanks go to Linda Peterson, the Guardian Angel and Den Mother of the Media Lab graduate students, who went beyond the call of duty to keep me on track all the way through from the beginning, from my Master's through my General Exams and finally to the end of my Phd.

Finally, I would like to especially thank my mom, dad and two brothers, who have been extremely supportive and loving throughout my many years at MIT. I could not have done it without you.

# Contents

<b>I</b>	<b>Imaginary</b>	<b>25</b>
<b>1</b>	<b>Introduction</b>	<b>27</b>
1.1	Principles and Thesis Statement . . . . .	30
1.2	Scope . . . . .	32
1.3	Related Work Areas and Contributions . . . . .	32
1.3.1	Real-time Motion Engines . . . . .	32
1.3.2	Multi-target pose interpolation . . . . .	33
1.3.3	Example-Based Function Approximation . . . . .	34
1.3.4	Orientation Statistics . . . . .	34
1.3.5	Posture Statistics . . . . .	35
1.3.6	Real-time Inverse Kinematics . . . . .	35
1.4	Thesis Roadmap . . . . .	35
1.5	Summary . . . . .	37
<b>2</b>	<b>Approach: Example-Based Procedural Animation</b>	<b>39</b>
2.1	Interactivity, Expressivity and the Illusion of Life . . . . .	39
2.2	Exploiting an Animator's Knowledge of Expressive Character Motion . . . . .	41
2.2.1	Pose Blending: Multi-Target Interpolation/Extrapolation . . . . .	42
2.2.2	Statistical Analysis and Synthesis of Joint Motion . . . . .	44
2.2.3	Expressive Inverse Kinematics . . . . .	45
2.3	Summary . . . . .	46
<b>3</b>	<b>Rotation, Orientation and Quaternion Background</b>	<b>49</b>
3.1	Rotation, Orientation and Euler's Theorem . . . . .	50
3.1.1	Rotation <i>versus</i> Orientation . . . . .	50
3.1.2	Euler's Theorem and Distance Metrics . . . . .	50
3.1.3	Summary . . . . .	52
3.2	Representing Rotations . . . . .	52
3.2.1	Coordinate Matrix . . . . .	53
3.2.2	Axis-Angle . . . . .	55
3.2.3	Euler Angles . . . . .	57
3.2.4	Representation Summary . . . . .	62
3.3	Quaternions . . . . .	62
3.3.1	Quaternion Hypercomplex Algebra . . . . .	63
3.3.2	Polar Form and Powers . . . . .	67

3.3.3	Topological Structure of Unit Quaternions: Hypersphere $S^3$	69
3.3.4	Exponential Map and Tangent Space	69
3.3.5	Basic Quaternion Calculus and Angular Velocity	72
3.3.6	Interpolation, Slerp and Splines	73
3.3.7	Advantages	74
3.3.8	Disadvantages	75
3.3.9	Recommended, Related and Other reading	76
3.4	Quaternion Algebra and Geometry Summary	77
<b>4</b>	<b>Statistical Kinematic Joint Models</b>	<b>81</b>
4.1	Motivation for Statistical Kinematic Model	82
4.2	Motivation for a Quaternion Representation of Character Joints	84
4.2.1	Properties	84
4.2.2	Special Orthogonal Matrices $SO(3)$	86
4.2.3	Euler Angles	87
4.2.4	Quaternions	89
4.3	Summary of Statistical Kinematic Model Motivation	90
<b>II</b>	<b>Real</b>	<b>92</b>
<b>5</b>	<b>Quaternions for Skeletal Animation</b>	<b>93</b>
5.1	Articulated Skeletal Model	94
5.1.1	Simplifying Assumptions	95
5.1.2	Skeletal Tree Structure	95
5.1.3	Root Joints are Special	96
5.2	Bones, Joints, and Coordinate Frames	96
5.2.1	Coordinate Frame Terminology	96
5.2.2	Joints and Bone Transformations with Quaternions	97
5.2.3	Open Kinematic Chains and Compound Transformations	99
5.3	Postures, Posture Distance, Motions and Animations	101
5.3.1	Postures	101
5.3.2	Posture Distance Metric	102
5.3.3	Motions	102
5.3.4	Animations	103
5.4	Summary	103
<b>6</b>	<b>QuTEM: Statistical Analysis and Synthesis of Joint Motion</b>	<b>105</b>
6.1	QuTEM	106
6.1.1	Motivation	106
6.1.2	QuTEM Definition	108
6.1.3	QuTEM as a Wrapped Gaussian Density	108
6.1.4	Scaled Mode Tangents (SMT), Ellipsoids, and Mahalanobis Distance	111
6.2	QuTEM Parameter Estimation	112
6.2.1	Estimation of the Mean from Data	114

6.2.2	Hemispherization . . . . .	118
6.2.3	Estimation of Unit Quaternion Covariances . . . . .	119
6.2.4	Estimating Constraint Radius . . . . .	122
6.2.5	Summary of QuTEM Parameter Estimation . . . . .	122
6.3	QuTEM Sampling . . . . .	122
6.3.1	QuTEM Sampling Algorithm . . . . .	123
6.3.2	Singular Data Woes . . . . .	123
6.3.3	Summary of Synthesis . . . . .	124
6.4	QuTEM Summary . . . . .	124
<b>7</b>	<b>Multi-variate Unit Quaternion Interpolation</b>	<b>127</b>
7.1	Problem Description . . . . .	127
7.1.1	Interpolation and Extrapolation . . . . .	128
7.1.2	Vector Space vs. Spherical Interpolation . . . . .	129
7.2	Slime: Fixed Tangent Space Quaternion Interpolation . . . . .	129
7.2.1	Motivation: Extension of slerp . . . . .	129
7.2.2	Slime Algorithm Definition . . . . .	130
7.2.3	Slime Properties . . . . .	130
7.2.4	Summary of Slime . . . . .	136
7.3	Sasquatch: Moving Tangent Space Quaternion Interpolation . . . . .	137
7.3.1	Spherical Springs Physical Analogy . . . . .	138
7.3.2	Spherical Metric . . . . .	139
7.3.3	Setting Up the System . . . . .	139
7.3.4	Solving the System for Steady State . . . . .	141
7.3.5	Sasquatch Algorithm . . . . .	141
7.3.6	Interpolation and Extrapolation . . . . .	143
7.3.7	Convergence Results . . . . .	143
7.3.8	Interpolation Visualization . . . . .	143
7.3.9	Summary of Sasquatch . . . . .	144
7.4	QuRBF's: Quaternion-Valued Radial Basis Functions . . . . .	144
7.4.1	Scalar RBF . . . . .	146
7.4.2	Vector-Valued RBF . . . . .	147
7.4.3	Quaternion-Valued RBF's with Slime . . . . .	147
7.4.4	Quaternion-Valued RBF's with Sasquatch . . . . .	148
7.4.5	Quaternion Inputs . . . . .	150
7.5	Summary of Weighted Quaternion Blending . . . . .	150
<b>8</b>	<b>Eigenpostures: Principal Component Analysis of Joint Motion Data</b>	<b>153</b>
8.1	Motivation for Posture Subspace Analysis . . . . .	153
8.2	Principal Component Analysis Overview . . . . .	154
8.2.1	Mathematical Description . . . . .	154
8.2.2	Standard PCA Algorithm Summary . . . . .	155
8.3	PCA on Posture . . . . .	156
8.3.1	Eigenposture Algorithm . . . . .	156
8.3.2	Projection and Reconstruction . . . . .	156

8.4	Initial Eigenposture Results . . . . .	157
8.5	Summary of Eigenpostures . . . . .	157
<b>9</b>	<b>(Toward) Expressive Inverse Kinematics</b>	<b>161</b>
9.1	Approach . . . . .	162
9.2	Joint Constraints with the QuTEM . . . . .	163
9.2.1	Approach . . . . .	163
9.2.2	Goals . . . . .	163
9.2.3	Constraint Satisfaction Operator . . . . .	165
9.2.4	Constraint Projection Operator . . . . .	165
9.2.5	Singular Densities . . . . .	166
9.2.6	Empirical Results . . . . .	167
9.3	Equilibrium Points with the QuTEM . . . . .	167
9.4	QuCCD: Quaternion Cyclic Coordinate Descent . . . . .	169
9.4.1	CCD IK Paradigm . . . . .	169
9.4.2	Unconstrained QuCCD Algorithm . . . . .	172
9.4.3	QuCCD with Constraints . . . . .	174
9.5	Mixing Pose Blending and IK . . . . .	174
9.6	Adding Expressivity with Subspace Models . . . . .	175
9.7	Summary of Expressive IK . . . . .	175
<b>10</b>	<b>Experimental Results and Application Examples</b>	<b>177</b>
10.1	QuTEM Analysis Results . . . . .	177
10.2	Synthesis of New Motion from the QuTEM . . . . .	180
10.3	Sasquatch Experiments . . . . .	181
10.3.1	Monte-Carlo Convergence Trials . . . . .	181
10.3.2	Reduction to slerp . . . . .	182
10.3.3	Attractor Trajectories . . . . .	183
10.4	Slime Results . . . . .	185
10.4.1	<i>Swamped!</i> . . . . .	185
10.4.2	(void*) . . . . .	187
10.4.3	Rufus . . . . .	190
10.4.4	Duncan the Highland Terrier . . . . .	191
10.4.5	Sheep Dog: Trial by Eire . . . . .	192
10.4.6	$\alpha$ -Wolf . . . . .	192
10.4.7	Slime Results Summary . . . . .	194
10.5	Expressive IK Results . . . . .	194
10.6	Results Summary . . . . .	195
<b>11</b>	<b>Related Work</b>	<b>197</b>
11.1	Animation Engines . . . . .	197
11.1.1	Perlin . . . . .	197
11.1.2	Blumberg . . . . .	198
11.1.3	Rose . . . . .	198
11.1.4	Grassia . . . . .	199

11.1.5	Downie: Pose Graph . . . . .	199
11.2	Multi-dimensional Quaternion and Pose Blending . . . . .	200
11.2.1	Grassia: Nested Slerps . . . . .	200
11.2.2	Buss and Filmore: Spherical Weighted Averages . . . . .	201
11.2.3	Lee: Orientation Filters . . . . .	202
11.3	Joint Rotation Statistical Synthesis . . . . .	202
11.3.1	Brand: Style Machines . . . . .	202
11.3.2	Pullen and Bregler . . . . .	202
11.3.3	Lee: Hierarchical Analysis and Synthesis . . . . .	203
11.4	Quaternion Joint Limits . . . . .	203
11.4.1	Grassia . . . . .	203
11.4.2	Lee . . . . .	203
11.4.3	Wilhelms and Van Gelder . . . . .	204
11.4.4	Herda, Urtason, Fua and Hanson . . . . .	204
11.5	Expressive IK . . . . .	204
11.5.1	Blow: Quaternion CCD IK with Joint Limits . . . . .	205
11.5.2	Lee: Quaternion IK with Constraints Using Conjugate Gradient . . . . .	205
11.5.3	Grassia: Quaternion IK for Motion Transformation . . . . .	206
11.5.4	Hecker: Advanced CCD Extensions . . . . .	206
11.5.5	Fod, Mataric and Jenkins: Eigen-movements . . . . .	206
11.5.6	D'Souza, Vijayakumar, Schaal and Atkeson: Locally Weighted Projection Regression for Learning Inverse Kinematics . . . . .	207
11.6	Summary and Recommended Reading . . . . .	208
<b>12</b>	<b>Discussion, Future Work and Conclusions</b>	<b>209</b>
12.1	Discussion . . . . .	209
12.1.1	Pose Metrics . . . . .	209
12.1.2	Multi-variate Unit Quaternion Blending . . . . .	210
12.1.3	Quaternion Statistics . . . . .	211
12.2	Future Work . . . . .	212
12.2.1	Dynamics . . . . .	212
12.2.2	Joint Limits . . . . .	212
12.2.3	QuCCD . . . . .	212
12.2.4	Orientation Statistics . . . . .	213
12.2.5	Posture Statistics . . . . .	213
12.2.6	Expressive IK . . . . .	213
12.2.7	Translational Joints . . . . .	213
12.3	Conclusions . . . . .	214
12.4	Summary of Contributions . . . . .	217
<b>A</b>	<b>Hermitian, Skew-Hermitian and Unitary Matrices</b>	<b>219</b>

<b>B</b>	<b>Multi-variate (Vector) Gaussian Distributions</b>	<b>221</b>
B.1	Definitions . . . . .	221
B.2	Isoprobability Contours and Ellipsoids . . . . .	222
B.3	Mahalanobis distance . . . . .	223
B.4	Sampling a Multi-Variate Gaussian Density . . . . .	225
B.5	Recommended Reading . . . . .	226
<b>C</b>	<b>Quaternion Numerical Calculus</b>	<b>227</b>
C.1	Solving Quaternion ODE's . . . . .	227
C.2	Embedding Euler Integration with Renormalization . . . . .	228
C.3	Intrinsic Euler Integration . . . . .	228
<b>D</b>	<b>Quaternions: Group Theory, Algebra, Topology, Lie Algebra</b>	<b>233</b>
D.1	Vector Space . . . . .	233
D.2	The Rotation Group in $\mathbb{R}^3$ . . . . .	234
D.3	Quaternion Theory . . . . .	235
	D.3.1 Hypercomplex Representation . . . . .	235
	D.3.2 Vector Space Interpretation of Quaternions . . . . .	240
	D.3.3 Quaternion Curves . . . . .	249
D.4	$SU(2)$ . . . . .	253
	D.4.1 Isomorphism . . . . .	253
	D.4.2 Pauli Spin Matrices . . . . .	254
D.5	Lie Groups and Lie Algebras . . . . .	257
D.6	Recommended Reading . . . . .	259



# List of Tables

3.1	Quaternion Algebra Summary . . . . .	78
3.2	Quaternion Algebra Summary II . . . . .	79
3.3	Quaternion Algebra Summary III . . . . .	80



# List of Figures

- 1-1 Virtual and Physical Expressive Interactive Characters. The virtual characters, developed by Blumberg's *Synthetic Characters* Group, are (clockwise from top left): Dobie the learning dog (SIGGRAPH 2002), the Raccoon from *Swamped!* (SIGGRAPH 1998), shy Elliot from (void\*) (SIGGRAPH 1999), Duncan and sheep from Sheep|Dog (Media Lab Europe Opening, 2000), and two wolf pups from  $\alpha$ -Wolf [90] (SIGGRAPH 2001). On the right are some physical robots which were designed to be expressive. They are (clockwise from top left): Breazeal's *Kismet* [13], *Leonardo* from Stan Winston Studios ([www.stanwinston.com](http://www.stanwinston.com)) with skin and without, and the (unskinned) Public Anenome (SIGGRAPH 2002) by Breazeal's Robotic Life Group. . . . . 28
- 1-2 The bones (colored solid) which are animated underly the mesh (grey transparent) skin. Each bone rotates with respect to its parent by a 3D rotation, making a hierarchical skeletal model with the pelvis at the root. . . . . 29
- 2-1 Canned animation clips (motion capture or hand-animated) offer maximal expressivity since they can be fine-tuned, but minimal interactivity since they are specific. Procedural methods (such as Inverse Kinematics) are usually maximally interactive since they offer an algorithmic, general solution, but tend to be very hard to make expressive. Example-based methods based on a hybrid of the two techniques offer the best of both worlds. . . . 40
- 2-2 Blend of two animations, sampled at the same time  $t$  but at different happiness values from -0.1 to 1.1. The examples (red boxes) are the original animations. The frames in between the examples *interpolate* the posture according to the level of happiness. The frames outside the interval  $[0, 1]$  *extrapolate* the examples, making caricatures of the original walks. . . . . 42
- 2-3 Extrapolation and Interpolation. The circles on the axes represent example animations of a happy, normal, and angry verb. The triangular filled region (convex hull) of the examples is the interpolation space. A point outside this space, such as the dark point specifying an angry *and* happy verb is an example of *extrapolation*. By extrapolating well, we can obviate the need for the animator to increase the size of the interpolation space with a new example at this point. As the number of axes increase, this gives us an *exponential* decrease in necessary examples. . . . . 43

3-1	A moving body coordinate system $B$ can represent the orientation of the body with respect to a known world coordinate system $W$ . We ignore translational effects for simplicity. . . . .	51
3-2	Euler's theorem states that the angular displacement of any rigid body can be described as a rotation about some fixed axis ( $\hat{n}$ ) by some angle $\theta$ . . . .	51
3-3	Euler angle illustration: pretend your fingers when held as shown are three moving orthogonal axes. Any orientation in space can be specified by a <i>yaw</i> around the thumb followed by a <i>pitch</i> around the middle finger then a <i>roll</i> around the index finger. . . . .	59
3-4	A gimbal consists of three concentric hoops connected by single degree of freedom pivot joints (each pivot is a physical realization of an <i>Euler angle</i> ) which attach adjacent hoops orthogonally (the outermost black hoop here is considered the "earth" and is fixed in space and cannot rotate.). The left image depicts the gimbal in its "zero" position, with the teapot (colored red to show that it is fixed to the red hoops's coordinate frame and cannot rotate independently of it) in an "unrotated" position, with the three hoop pivots orthogonal and corresponding to axes (red is $\hat{x}$ , green is $\hat{y}$ and blue is $\hat{z}$ ). The middle image illustrates an arbitrary rotation of the teapot and the associated gimbal configuration. The right image shows the inherent problem with three hoop gimbals and any associated Euler angle representation — gimbal lock. Here the teapot's nose is pointing straight up, and two hoops have aligned, removing a degree of freedom. In this configuration, it is impossible to find a smooth, continuous change of the gimbal which will result in a rotation around the teapot's local "up" direction, here shown as a superimposed purple axis. Any attempt to rotate around the purple axis is impossible from this configuration — the gimbal is said to be <i>locked</i> since it has lost a degree of freedom. A real gimbal with a gyro instead of a teapot would shake itself to pieces if it tried to rotate around this locked axis — a very real phenomenon in early navigational systems using Euler angles and real gimbals. . . . .	61
3-5	While walking from his work at Dunsink Observatory to his home in Dublin, Hamilton realized that he needed a third imaginary unit and was so excited that he scratched the quaternion algebra equations onto a rock on the bridge over a canal near the Royal Irish Academy [6]. (Photo credit: Rob Burke 2002) . . . . .	64
3-6	A depiction of the exponential map. Points in the tangent space are mapped onto the sphere by the exponential mapping and vice versa by the logarithmic map, its inverse. . . . .	71
3-7	A depiction of slerp. The two examples are interpolated at constant angular velocity as the parameter changes with constant speed. The exponential portion of slerp can be interpreted with the exponential mapping with respect to one example. In this view, a constant speed line in the tangent space will map to a constant angular velocity curve on the sphere. . . . .	74

5-1	The bones (colored solid) which are animated underly the mesh (grey transparent) skin. Each bone rotates with respect to its parent by a 3D rotation, making a hierarchical skeletal model with the pelvis at the root. . . . .	94
5-2	An articulated figure can be considered as a tree with joints as edges connecting limbs (nodes). The black circle shows the root of the tree, although any point in the structure could be chosen. . . . .	95
5-3	The link transform from a parent bone's coordinate system in a kinematic chain ( $\mathbf{B}_p$ ) to its child ( $\mathbf{B}_i$ ) depicted in 2D. The coordinate system $\mathbf{L}_i$ (and associated grey bone) show the zero rotation (basis) configurarion. The quaternion $\hat{Q}_i$ is the angular displacement from the basis and thus specifies the current orientation of the child bone with respect to the parent's coordinate system. . . . .	98
6-1	A sktech of a bimodal distribution on $S^3$ which exhibits antipodal symmetry. Such distributions are valid distributions over $\text{SO}(3)$ through the double-covering. . . . .	107
6-2	An abstract depiction of the QuTEM. The mean is the tangent space point, the ellipse depicts the $\rho$ Mahalanobis distance constaint surface, and the axes depict the principal axes of the density and their relative variances (covariance). . . . .	109
6-3	The QuTEM models a spherical distribution by estimating a zero-mean Gaussian density in the tangent space to the unit quaternion mean. . . . .	110
6-4	A sketch of a spherical density on $S^3$ constrained to be zero beyond a certain distance from the mean. . . . .	111
6-5	A three-dimensional visualization of the SMT transformation which turns ellipses on the hypersphere into ellipsoids in the tangent space at the center of the ellipse. The left image (a) shows the original spherical ellipse with its center; the right upper (b) shows the ellipse in the tangent space by using the exponential map at the center point (notice the center maps to the origin in the tangent space); the bottom right image (c) shows the result when the space is rescaled by the axis lengths on the ellipse to form a circle in a warped tangent space. Notice all true objects are one dimension higher. . .	113
6-6	Points sampled randomly on an ellipsoid around the origin in $\mathbb{R}^3$ (left) and 3D projection plot of the exponential mapped points (right, created by ignoring the $\hat{z}$ component of the quaternion). Notice the points, although are on the <i>boundary</i> of an ellipsoid on the left, appear to map inside the ellipse on the sphere. This artifact results from the projection, which ignores the $z$ direction. By viewing the sphere along the direction directly pointed at the center, however, we see that the shape is elliptical, as it should be. . . . .	114
6-7	Our data is antipodally symmetric, and therefore we might arbitrarily get the sample $\pm\hat{Q}_i$ . We would like all data on the same local hemisphere of $S^3$ for simplicity. This hemisphere will be defined by the choice of the sign on $\hat{M}$ . To hemispherize data, we simply flip the data to lie on the same hemisphere as the mean choice $\hat{M}$ using a dot product as a test. . . . .	118

7-1	An abstract depiction of the unit quaternion blending building block. The algorithm should take $N$ unit quaternion examples $\hat{Q}_i$ with associated weights $a_i$ and perform a weighted sum of them. These answer is then usually written to a joint controller. . . . .	128
7-2	The slime algorithm maps examples (green spheres) into tangent descriptions (yellow vectors) with respect to a chosen reference quaternion (yellow sphere) by describing the examples in terms of geodesic curves (red great circles) that pass through the reference and the example. The yellow vectors live in a linear space since they correspond to angular velocities of the curves, and therefore can be blended linearly. . . . .	132
7-3	The slime algorithm linearly blends the tangent vector description (yellow vectors) of the geodesics (red great circles) of the examples (green spheres). As an example, the orange vector is an arbitrary weighted blend of the three example vectors. This blend is actually specifying a particular different geodesic through the reference point — the orange great circle. By integrating this blended angular velocity forward unit time from the reference (using the exponential), we get the blended quaternion (orange sphere). . . . .	133
7-4	Choice of the reference quaternion (yellow sphere) affects the interpolation of examples (green spheres) since quaternions are represented as one parameter subgroups (red great arc circles through the examples and reference) through the reference quaternion. As the choice approaches the “average” of the examples, the curves (examples) become more separated from each other and therefore the interpolation becomes better. . . . .	135
7-5	The system of Aristotelian springs with constants $k_i$ connected between the example points (nails), $p_i$ , and the free point (yellow), $q$ . All nails must be on the same local hemisphere. . . . .	138
7-6	An orthogonal projection of the system from above the free point, $q$ . Note that this is <i>not</i> the exponential mapping since orthographic projection does not preserve the spring lengths with respect to the spherical distance metric. . . . .	139
7-7	A 3D orientation field specified as a radial basis function around the examples (corner locations boxed in red). The weight of each example is inversely proportional to its distance from the sample point in the field as described in text. The center image clearly has equal weights on all examples, and is therefore the centroid of the four examples with respect to the spherical metric. . . . .	145
7-8	A Sasquatch RBF of a parameterized walk cycle with two input dimensions, happiness and turning radius. All images are sampled at the start of the walk cycle and the RBF is sampled evenly in all directions. The image was created with six examples, four on the corners and two for normal-left and normal-right. . . . .	151
8-1	Training set RMS reconstruction error (radians) versus number of basis eigenvectors. . . . .	158
8-2	First ten principal components from 5800 frames of a 53 joint dog. . . . .	158

9-1	An abstract visualization of finding whether the query point $\hat{Q}$ on the unit quaternion sphere is inside the constraint ellipse boundary or not and the nearest projected valid point, $\hat{Q}'$ . . . . .	164
9-2	A visualization of the $SMT(\hat{Q})$ sphere which divides out variance differences along the principal directions $\hat{x}$ , $\hat{y}$ and $\hat{z}$ . The mode, $\hat{M}$ is mapped to the origin, and the constraint boundary $\rho$ away from the mean is the surface of the sphere (which is radius $\rho$ ). Therefore, we can perform very fast, simple sphere-point checks and projections on properly mapped data. . . .	166
9-3	Several screenshots of random sampled dog postures on the constraint boundary. The shots were created by creating a uniform rotation for each joint in the dog, then projecting to the nearest point on the constraint surface. Most sampled configurations are reasonable, though in some the “joint-local” nature of these constraints becomes obvious by a body interpenetration. We do not handle these <i>posture</i> constraints yet, but feel the Eigenpostures might be useful here. . . . .	168
9-4	The geometry of a CCD local update step on a three link kinematic chain. The algorithm calculates the vector from the current joint being updated (here 2) to both the effector’s current Cartesian position ( $\mathbf{c}$ ) and the goal’s position ( $\mathbf{d}$ ), expressed in the local coordinate system of the joint ( $\mathbf{B}_2$ ). These vectors can be used to calculate the local angular displacement of joint 2 ( $\Delta_2$ ) which minimizes the error $\ \mathbf{c} - \mathbf{d}\ $ between the goal and effector. Joint 2’s orientation is then updated by rotating it in the displacement’s direction by some percentage, which is expressed as a weight ( $a_i$ ). This completes a single CCD sub-step on Joint 2. The algorithm would then proceed to Joint 3 and perform the same set of operations again on the updated chain. This continues cyclically down the chain until convergence or a stopping criterion is met. . . . .	170
9-5	The CCD algorithm after updating Joint 2 with a full weight of 1.0. The rotation update ( $\Delta_2$ ) must be applied in the parent’s coordinate system since the orientation of the joint is specified as an update. . . . .	171
10-1	The dog in “mean pose” where all of his joints have been set to their mode.	178
10-2	Plots of the mode-tangent descriptions learned from animation data for several joints on a dog model. The scatterplot is the transformed quaternion data and the ellipsoid shows the Mahalanobis distance 1.0 isocontour of the estimated density from the data. . . . .	179
10-3	TEM plot of just the elbow joint of our dog. Notice the structure contains more than one degree of freedom, although it tends to lie in a particular direction. . . . .	180
10-4	Convergence statistics for a 5 point system using intrinsic integration over 100 random trials. Here, $\epsilon = 1.0e - 12$ . The middle curve is the mean and those bracketing it are one standard deviation. . . . .	183

10-5	An illustration of the stable attractor which is the steady state solution to Sasquatch. Here we choose 50 random initial points not too close to each other then integrate the system with $dt = .01$ and plot the resultant trajectories. Here we choose two examples, whose attractor (steady state) is the identity quaternion, 1. Since the data live in $S^3$ , we project out the $z$ component for this plot. . . . .	184
10-6	A plot of the trajectories for the attractor taken as the log at the attractor location (the identity), which is the tangent space $\mathbb{R}^3$ . The attractor in the log is located at the origin. . . . .	185
10-7	The <i>Swamped!</i> project, shown at SIGGRAPH 1998. The interactor directs the chicken character using natural gestures of the plush toy ( <i>sympathetic interface</i> ). The raccoon is autonomous and uses an early slime-based RBF system based on Rose's Verbs and Adverbs work. . . . .	186
10-8	The raccoon character is autonomous. He can blend between several emotional states based on his interactions with the chicken. These states are expressed through the motion with pose-blending. . . . .	187
10-9	Elliot the shy nerd starts off dancing very inhibited. Over time, his dance styles becomes more open as he enjoys himself. . . . .	188
10-10	Eddy the Dude shows off the range of motion of his hip joints with his split move. . . . .	189
10-11	Blend of two animations, sampled at the same time $t$ but at different blend weights, from -0.1 to 1.1. The examples (red boxes) are the original animations, so the algorithm can extrapolate as well as interpolate. . . . .	189
10-12	Rufus, a simple articulated robot dog head with a camera in its eye. Rufus was the first example of using out pose-blending slime algorithm on a physical robot. . . . .	190
10-13	Duncan and the Shepherd. This project was one of the first to begin to look at clicker training the animal. Both the shepherd and dog used an early slime-based blend. . . . .	191
10-14	Sheep—Dog used an acoustic pattern recognition system to direct the dog to herd sheep into a pen using traditional dog-training lingo. . . . .	192
10-15	A shot of the $\alpha$ -Wolf installation . . . . .	193
10-16	A blend along one of the emotional adverb axes. Picture credit to Bill Tomlinson. . . . .	193
10-17	The insides of the "Public Anenome" robot by the Robotic Life Group at the MIT Media Lab that was shown at SIGGRAPH 2002. . . . .	195
10-18	The Anenome with its skin on. . . . .	196



# **Part I**

## **Imaginary**



# Chapter 1

## Introduction

This dissertation will describe a set of building blocks I have found useful in the design and implementation of expressive interactive motion engines for both virtual and physical characters such as those illustrated in Figure 1-1. These building blocks are based on a *quaternion* representation of joint orientation and rotation for an articulated figure model like that depicted in Figure 1-2.

I will show the reader how to use quaternions to solve some of the common problems in a real-time motion engine that usually are intended to be “solved” by an Euler angle parameterization. The problems I will address are:

**Multi-target pose blending** : Morphing between multiple example poses of a character

**Real-time Inverse Kinematics (IK)** : What’s the nearest posture will put my hand there?

**Statistical joint modelling** : How does this joint tend to move?

**Fast, learnable joint limits** : Where does this joint *not* move?

**Pose sub-space analysis** : How do these joints tend to vary together?

**Expressive IK** : What is a posture which will put my hand there without looking like a robot?

I will also show that quaternions are an appropriate choice from a computational point of view. I will also show why any of the Euler angle parameterizations is almost *always* the wrong choice of representation from both group-theoretic and computational arguments. Instead, I will argue that the use of the *logarithmic mapping* of a quaternion into a 3-vector  $\theta\hat{n}$  is preferable, maintaining many of the desirable properties of an Euler angle decomposition (minimal parameter, “linear”) while avoiding most of the undesirable properties, such as the infamous *gimbal lock*. I will show how to use the exponential mapping (which is related to the theory of Lie groups and Lie algebras) to “locally linearize” pose data so that it can be analyzed with standard, powerful analysis methods such as Principal Component Analysis (PCA).

In particular, the set of building blocks I will introduce are:



Figure 1-1: Virtual and Physical Expressive Interactive Characters. The virtual characters, developed by Blumberg's *Synthetic Characters* Group, are (clockwise from top left): Dobie the learning dog (SIGGRAPH 2002), the Raccoon from *Swamped!* (SIGGRAPH 1998), shy Elliot from (void\*) (SIGGRAPH 1999), Duncan and sheep from *Sheep|Dog* (Media Lab Europe Opening, 2000), and two wolf pups from  $\alpha$ -*Wolf* [90] (SIGGRAPH 2001). On the right are some physical robots which were designed to be expressive. They are (clockwise from top left): Breazeal's *Kismet* [13], *Leonardo* from Stan Winston Studios ([www.stanwinston.com](http://www.stanwinston.com)) with skin and without, and the (unskinned) Public Anenome (SIGGRAPH 2002) by Breazeal's Robotic Life Group.

---



Figure 1-2: The bones (colored solid) which are animated underly the mesh (grey transparent) skin. Each bone rotates with respect to its parent by a 3D rotation, making a hierarchical skeletal model with the pelvis at the root.

---

**Pose distance metrics** Appropriate group-theoretic distance metrics on poses for use in any algorithm which requires domain-specific metrics, like most example-based learning methods we focus on.

**slime and sasquatch** Two new algorithms for computing a weighted blend of  $n$  unit quaternions representing rotations in 3D. These are useful for multi-target animation interpolation. Also, most example-based function approximation methods require robust blending primitives of some sort to blend exemplars.

**QuTEM joint model** A statistical model of individual joint motion learnable from example data and consisting of: 1) mean joint coordinate frame, 2) principal axes of joint variation and variances associated with these and 3) hard joint limits described as an isoprobability contour.

**Eigenpostures** A statistical model of posture (coupled joint motion, or multiple quaternions) which best models the variations in animation data and can serve as an “expressive” basis for a character’s motion in algorithms or as the starting point for character-based animation compression algorithms.

**Fast, Learnable Quaternion Joint Limits** How to estimate a convex joint constraint boundary to represent fast, hard joint limits on a quaternion joint representation.

**Quaternion Cyclic Coordinate Descent (QuCCD)** A fast quaternion version of the recent real-time heuristic Cyclic Coordinate Descent (CCD) IK algorithm which can incorporate joint limits.

I will then show how we use these primitives in tackling three main areas of expressive interactive character motion:

**Multi-Target Pose Blending** Blending  $n$  poses together simultaneously. Pose blending can be used to blend  $n$  animations in real-time or blend exemplars in powerful and well-known non-parametric function approximation algorithms like  $k$ -nearest neighbor,  $k$ -means clustering, and locally-weighted regression.

**Statistical Joint Analysis and Synthesis** How to learn a model of the ranges of motion on each joint from a corpus of animation data, how to use these to make pose metrics invariant to these ranges, how to generate new poses (and simple animations) which respect the joint variances and limits, and how to use the model to compute fast, simple joint motion limits.

**Expressive Inverse Kinematics** How to implement a fast heuristic IK algorithm called Cyclic Coordinate Descent (CCD) with quaternions and quaternion joint limits. Also, we sketch several simple ways to use all of our building blocks together to make an IK solver produce less “robotic” looking solutions. For example, we discuss using a learned posture subspace model to constrain a procedural IK solution space and give initial results at coupling pose-blending and CCD.

The rest of this chapter will proceed as follows:

**Section 1.1** lists the design principles we took in this research and presents our thesis statement.

**Section 1.2** defines the scope and audience of the thesis.

**Section 1.3** gives a capsule description of related work areas and summarizes our contributions to each.

**Section 1.4** gives a roadmap through the rest of the thesis with capsule descriptions of each chapter.

**Section 1.5** summarizes this chapter and the contributions of our research.

In this work, I followed a set of design principles, summarized in the next section.

## 1.1 Principles and Thesis Statement

The following set of principles were followed throughout this work:

**Interactive/Real-Time** Interactive characters means real-time. Much of the work in motion editing and “interactive” methods in the computer graphics community are focused on “interactive design tools for animators.” In other words, they are allowed to produce incorrect results, but should be easily tweakable by an animator in “real-time” (about 5hz) to get around these and make a perfect production animation, like

a film. Ultimately, the animator will coerce any tool into making the animation he or she desires, and most algorithms are focused at making these easier for the animator. We are *not* talking about these tools. By “interactive” or “real-time,” we mean an engine that takes commands from the “brain” of an interactive character and must respond in-character, right away, with no glitches and with no input from an animator except for the use of animation examples created off-line. In general, we have milliseconds to decide the next animation frame. In a production animation tool, the animator has minutes or hours for this. This principal also quickly eliminates the ability to use most of the recently popular motion transformation algorithms which use expensive optimization techniques.

**Example-Based** This work started from the simple assertion that “Animators know best how a character should move and are best able to express this by creating canned animation examples, not programming.” Therefore, all the algorithms we developed had to be learnable from animation data or exploit it in some way, such as blending or synthesis from a learned statistical model.

**Let the Animator Work Naturally** Most character engines will enforce a particular articulated figure structure (usually Euler angles) on the animator for them to animate, even if the axes are not simple to animate. We try to avoid these forced models. Rather, we want the animator to work naturally and then we can use analysis and synthesis methods to coerce these into the real-time data structure we need.

**Quaternion-based** Although I motivate the use of quaternions after the fact, it is (arguably) well-known that they are the best computational representation of orientation for rigid bodies without mathematical problems, as we will see in Chapter 3. Unfortunately, the use of quaternions in articulated figure animation is fairly recent. The standard representation of a character’s posture is usually a vector of Euler angles, which entails the use of rotation matrices for performing coordinate transformations, a very common operation for interactive character algorithms which we describe in Chapter 5. Most useful character animation algorithms were therefore based on these less desirable (as we argue in Chapter 4) representations. The few quaternion algorithms were treated as a black box. Instead, we chose to follow a principled approach and extend the toolbox of standard figure animation techniques which we described above to work with a quaternion representation.

Based on these principles, we summarize our thesis statement:

**Thesis Statement:** By exploiting a unit quaternion representation of joint rotation, we can create computational building blocks for the design of expressive interactive character algorithms which afford us:

- Maximally leveraging an animator’s skills
- Computational efficiency in space and time
- Mathematical robustness

## 1.2 Scope

This dissertation covers real-time example-based expressive motion for interactive characters. We will limit ourselves to just kinematics (motion without regard for dynamics) <sup>1</sup>. Furthermore, we will restrict the work to rigid articulated skeletal models with only rotational joints and not mesh deformation techniques.

The intended audience is a senior programmer or engineer given the task of writing a fast, expressive animation engine for interactive characters that needs to incorporate canned animation clips for expressivity, such as in a videogame. For this reason, we try to focus on geometric intuition and insight rather than group theory, while explaining the deeper mathematical reasons for using quaternions rather than the more standard Euler angles for real-time articulated figures. When we approached the problem, there were only few people (for example, Hanson [36, 34, 37]) who focused on intuitive approaches to quaternions for designing new algorithms rather than using them as a “black box.”

In particular, the audience should be sick of the practical problems associated with using Euler angles in interactive applications and tired of the lack of intuitive algorithms and design approaches for creating new quaternion algorithms or extending known ones, which the initial motivation for this thesis.

## 1.3 Related Work Areas and Contributions

This dissertation covers several broad areas of related work. These are:

- Real-time articulated figure animation engines
- Multi-target pose interpolation
- Example-based function approximation methods
- Orientation statistics
- Posture Statistics
- Real-time Inverse Kinematics

We offer new contributions to each area, discussed in turn.

### 1.3.1 Real-time Motion Engines

Traditionally, most real-time motion engines use an Euler angle or homogenous matrix representation of rotation. This is due to the fact that many useful algorithms for real-time motor control, such as Inverse Kinematics, came out of the robotics community where Euler angles are manifested physically as servos. Also, most interpolation algorithms assume that the examples are vectors that form a vector space in order to decompose (factor) the

---

<sup>1</sup>Recent work by Matt Grimes in the *Synthetic Characters* Group has begun to look at extending these ideas to dynamics control.



problem into smaller scalar sub-problems. This leads many designers to use Euler angles since they seem to offer a linear, factored representation that can be used in these algorithms. Unfortunately, rotations do not form a vector space, as we will see, which often leads to strange, hard to understand behavior or “hacks” to try to patch the problems. This quickly eliminates any perceived advantage of using Euler angles.

Instead, we argue for the use of a quaternion representation motion and show how to solve many of the common problems in interactive character animation that usually lead to an Euler angle parameterization or to inefficient conversions from quaternions to Euler angles and back. We discuss these in the next several sections.

Also, we try to provide a comprehensive introduction to quaternions with a focus on computation and intuition. We also collect together many of the ideas which are spread throughout the literature in several fields and give pointers to useful recommended reading.

### 1.3.2 Multi-target pose interpolation

Multi-target pose blending is an extension of the classic multi-target mesh interpolation algorithms used to morph between several examples of a polygon mesh that span a space of geometry. The standard technique, since it operates on mesh vertices (which are true vectors), cannot be used for rotations without modification since they are not. The standard methods for multi-target pose interpolation either use an Euler angle model so that Euclidean methods such as RBF’s may be applied (e.g. Rose [44]) or use nested slerp constructions (e.g. Grassia [30], which scale poorly and which cannot be used simply as a black box to blend  $N$  examples with specific weights. Shoemake’s classic slerp quaternion interpolator handles interpolation of rotation by using quaternions, but can only blend between two examples with one parameter between them. To solve these problems, we offer two new primitives for pose blending, which we introduce now.

The first, Spherical Linear Interpolation of Multiple Examples (slime) is a fast unit quaternion blending primitive which approximately satisfies the rotation group metric, but is not rotationally-invariant since it transforms the unit quaternions to a fixed tangent space to perform the blend within. On the other hand, a fixed tangent space offers us the advantage of speed since we may preprocess quaternions and end up with an algorithm that scales linearly with examples and uses few trigonometric calls. A poor choice of tangent space, however, can cause similar (though mathematically and computationally much better behaved) problems to Euler angles since it is a singular representation. To solve this, we show that the mean across all data of a joint’s orientation is a good choice of fixed blending space since it places the singularity as far from the data as possible, unlike an Euler angle representation, which often places it right in the middle of the data unless complicated preprocessing steps are taken. Also, due to the singularity and the fact that the space is fixed, slime is not appropriate for blending bodies that are allowed to rotate freely. This is not a major disadvantage in practice, however, since almost all physically-plausible character joints *cannot* spin all the way around any axis like Regan’s head in *The Exorcist*.

The second, sasquatch, is an iterative extension to slime which uses a moving tangent space to handle joints that revolve all the way around, such as the root node that lets a character move around in a virtual world. Also, sasquatch affords us rotational-invariance, respects the rotation group metric, offers linear scaling in examples, linear convergence

(one floating point digit per iteration), good parametric behavior and a way to perform pseudo-linear blends on the sphere for more than two examples. In this way, it extends the slerp building block to more than two quaternions, as we intended, while maintaining all of its desirable properties.

### 1.3.3 Example-Based Function Approximation

The use of example data in a procedural context is the domain of *example-based learning* or function approximation. We do not offer any new algorithms in this field, but instead offer a set of domain-specific primitives (pose metrics and synthesis primitives) which are required by these methods. Most such methods are very sensitive to the choice of representation of the data and require domain-specificity for good results. Example-based methods are appropriate here since we want a simple way to encode animation examples in the system, while also allowing for real-time on-line incremental learning on this representation.

### 1.3.4 Orientation Statistics

Orientation statistics is the statistics of orientations of objects in space (see the recent [57] for a comprehensive overview). Often orientation statistics are calculated on a matrix representation of orientation which leads to complicated mathematics. Instead, we can use the unit quaternion representation to simplify the mathematics significantly. This fact seems little known in the literature on orientation statistics as most methods seek to be work for arbitrary dimension, resorting to manifold-tangent methods, differential geometry, or exterior calculus<sup>2</sup>. These methods are all too complicated, inefficient and unnecessary for the quaternion group and its simple spherical topology. Furthermore, the existence of an algebra on the sphere allows us to simplify estimation of parameters.

One way to use a unit quaternion representation is to estimate a Gaussian probability density function in  $\mathbb{R}^4$  conditioned to live on the unit sphere. This result is called the *Bingham distribution* [7]<sup>3</sup>. Although the principal axis estimates are the same on the sphere and in the embedding space (eigenvectors of the sample covariance matrix), the Bingham variances are much harder to estimate. Also, due to the fact that the Bingham variance parameters are not estimated in the rotation group itself means that the parameters do not have a direct physical interpretation in terms of joint angles.

As an alternative to the Bingham distribution and matrix distribution approaches, we offer the QuTEM model which can estimate orientation statistics of joints from data. It uses the Lie group structure of the quaternions (in particular the exponential mapping and Lie algebra, described in Chapter 3 and Appendix D) and the well-known Gaussian estimation and sampling methods. The covariances<sup>4</sup> it estimates have physical meaning (units

---

<sup>2</sup>An exception we found recently is [64] which relates the statistics of  $\mathbf{SO}(3)$ ,  $\mathbf{SO}(4)$  and quaternions.

<sup>3</sup>Although they seem fairly uncommon in the literature, recently they have been gaining in use (see, for example, Matt Antone’s excellent thesis on using them for camera pose recovery from examples [1])

<sup>4</sup>We feel that our QuTEM distribution is mathematically closely related to the Bingham distribution since both end up solving an eigenvector problem on the sample covariance matrix, but have not worked through the mathematical details at this time. We predict that the principal axes of motion and the mean will be extremely related, if not identical, and that the variances will be related by a monotonic function.

are in radians) and can be used as a smooth joint limit constraint manifold for IK. Also, the quaternion mean is a useful primitive for finding a good tangent space in which to linearize data for statistical analysis of pose as well as a good choice of tangent space for performing fast pose blending, as discussed above. Finally, we can use these structure to create dimensionless pose metrics using the standard Mahalanobis distance.

### 1.3.5 Posture Statistics

Finally, we offer a sketch of using the powerful subspace analysis algorithms such as Principal Component Analysis (PCA) which have had great success in analyzing image data in the computer vision community to characterize and model the intrinsic degrees of freedom in example animation data. These methods usually assume a Euclidean space and naive use of quaternion data with them can lead to problematic, non-intuitive results. Instead, we show how the exponential mapping and quaternion mean primitives can be used to linearize the data in an invertible way, allowing us to use PCA without resorting to Euler angles, which is the standard approach to the statistical analysis of posture.

### 1.3.6 Real-time Inverse Kinematics

Inverse kinematics is often the most expensive primitive in any engine. Most engines use an Euler angle representation of rotation, following the robotics community where numerical IK algorithms originated, in order to use linear algebra techniques. Standard Euler angle IK algorithms, since they use a matrix description, scale quadratically in the number of joints, which can be computationally infeasible. As an alternative, we offer a fast quaternion version of the Cyclic Coordinate Descent (CCD) algorithm Euler angle algorithm which has shown recent popularity in the videogame industry since it avoids matrices by using heuristics and therefore is much faster.

Furthermore, we show how to learn fast quaternion joint limits from data and augment our quaternion CCD algorithm to respect these limits. The standard approach to joint limits is to clamp Euler angles inside a certain interval. At the time we began this research, there did not exist any way to do joint limits on quaternions without converting to an Euler angle representation and back. Furthermore, we show how to learn these limits from example data rather than forcing an animator to specify them by hand, which is the case for all the recent quaternion limits except for the excellent recent work of Herda, Urtason, Fua and Hanson [39, 40].

## 1.4 Thesis Roadmap

This dissertation is divided into two parts — Imaginary and Real. Part I, Imaginary, argues for example-based methods, gives background information and motivates the use of quaternions for modeling joints. Part II, Real, then shows how we actually exploit quaternions in addressing the problems we introduced in this chapter.

Part I, Imaginary, proceeds as follows:

**Chapter 1** introduced the three main areas of expressive character motion we will address.

**Chapter 2** motivates the use of example-based methods coupled with procedural algorithms for leveraging the skill of an animator in a real-time expressive interactive character engine.

**Chapter 3** introduces and discusses mathematical background in rotations. It presents several commonly used parameterizations of rotation — rotation matrices, axis-angle, Euler angles. It then introduces quaternions, our choice of representation, from an algebraic and geometric viewpoint, providing required background for the rest of the dissertation.

**Chapter 4** offers a set of criteria for the representation of a statistical model of joint rotation. It then evaluates three of these parameterizations — rotation matrices, Euler angles and quaternions — against these criteria, arguing that quaternions offer both mathematical robustness and computational efficiency.

Part II proceeds as follows:

**Chapter 5** presents the commonly-used rigid bone-joint skeletal model of articulated figure animation and introduces terminology and notation used throughout the remainder of the document. It shows how quaternions can be used to model joints and defines the form of the example animation data.

**Chapter 6** defines the QuTEM statistical joint model, shows how to estimate a QuTEM from examples and shows how to sample new joint orientations from the model.

**Chapter 7** describes the problem of multi-variate unit quaternion weighted interpolation for pose-blending. It presents our two new algorithms — *slime* and *sasquatch*— for pseudo-linear weighted unit quaternion blends. It then describes how these can be used to extend a Euclidean example-based non-linear interpolation function — Radial Basis Functions (RBFs) — to work with quaternion inputs and outputs.

**Chapter 8** presents the problem of posture subspace analysis. It then presents our Eigenpostures algorithm, which extends a Euclidean subspace analysis algorithm — Principal Component Analysis (PCA) — to quaternion joint data.

**Chapter 9** introduces the difficult problem of Expressive Inverse Kinematics (Expressive IK). We show how to implement hard joint limits with the QuTEM. We then describe QuCCD, our quaternion extension to the fast CCD IK algorithm. We also describe how to use the QuTEM as a joint equilibrium point. Finally, we offer initial ideas of how pose-blending, the QuTEM, the QuCCD algorithm and Eigenpostures could be coupled to approach the problem of Expressive IK.

**Chapter 10** presents results on applying some of the building blocks to animation data. We visualize QuTEM models learned on animation data and show how new poses can be synthesized. We then describe several experiments on the *sasquatch* algorithm to demonstrate its behavior and choose parameters. Finally, we describe the projects

which have used the slime algorithm for pose-blending and discuss the issues that motivated the building blocks chronologically.

**Chapter 11** discusses influential and related work in the areas we covered in this research.

**Chapter 12** presents conclusions and directions for future.

We also offer several appendices for required background:

**Appendix A** gives a cursory description of complex matrices which we use in portions of the document.

**Appendix B** gives a quick introduction to multi-variate Gaussian probability densities and terminology.

**Appendix C** describes quaternion differential equations and how we solve them.

**Appendix D** offers a more formal mathematical treatment of the algebra, group theory, and topology of quaternions to serve as a reference or introduction for the mathematically-inclined.

## 1.5 Summary

The chapter presented the following problems of real-time expressive interactive character animation we will discuss in this dissertation:

- Appropriate pose metrics
- Multi-target pose blending
- Statistical joint modelling
- Pose sub-space analysis
- Joint limits learned from data
- Real-time Inverse Kinematics
- Expressive IK

We also introduced the new set of computational and mathematical building blocks we found useful for solving these problems and which we will serve as the main contributions in this thesis:

**Appropriate pose distance metrics** : domain-specific primitives for use in example-based methods

slime **and** sasquatch : two new multi-variate weighted unit quaternion blending primitives

**QuTEM model** : statistical analysis of quaternion-represented joint motion

**Eigenpostures** : posture subspace analysis using a quaternion extension to PCA

**Fast, learnable quaternion joint limits** : How to use the QuTEM to learn fast, hard limits on joint motion from a corpus of animation data

**QuCCD** : a fast quaternion version of the CCD IK algorithm that incorporates quaternion joint limits

**Expressive IK** : a description of the problem of Expressive IK, an initial evaluation of several ways to approach it using our building blocks in conjunction

The next chapter will motivate the use of example-based methods coupled with numerical, procedural algorithms for leveraging the skill of an animator in the design of expressive character engines.

## Chapter 2

# Approach: Example-Based Procedural Animation

This chapter will argue for the use of example-based procedural algorithms which combine the expressivity of motion of a hand-animated animation such as those found in an animated feature film like Pixar’s *Toy Story* with the infinite variability and interactivity of a numerical algorithm such as an Inverse Kinematics <sup>1</sup> (IK) algorithm. Figure 2-1 depicts the structure of our argument abstractly. We will argue that canned animation *clips* are easy to make expressive since an animator can tweak them iteratively until they are, but not very interactive since they are fixed in advance and therefore are only appropriate in the specific context for which they were created. We will then argue that a numerical, algorithmic approach such as IK offers much more interactivity since it can handle continuously changing goals, such as tracking a fluttering butterfly with its head and eyes. On the other hand, we will argue that these algorithms, which are typically hand-programmed, are very hard to make expressive since parameters and heuristics must be tweaked by hand. This is an unnatural way for an animator to work. We then argue for the use of example-based algorithms which offer the best of both worlds, allowing for procedural control to handle interactivity while incorporating expert knowledge of expressive movement from animation clips.

## 2.1 Interactivity, Expressivity and the Illusion of Life

In their book on the Disney approach to hand-drawn animation *The Illusion of Life* [84], Thomas and Johnston define the *illusion of life* as follows:

It is the change in shape that shows what the character is thinking. It is the thinking that gives the illusion of life.

In the case of an autonomous virtual 3D character, the change in shape is defined by its motion and the thinking by the Artificial Intelligence engine driving the character’s motion.

---

<sup>1</sup>Recall that IK can be described as the problem of finding the joint angles of a character that will place some part of its body, such as a dog’s paw, on some specified location in the world, such as on that cat’s tail.

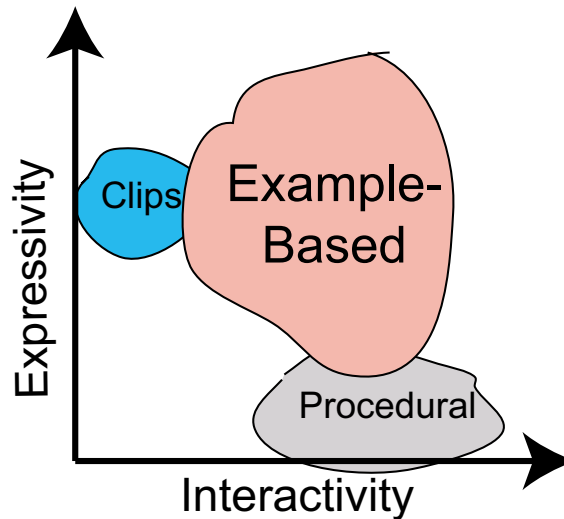


Figure 2-1: Canned animation clips (motion capture or hand-animated) offer maximal expressivity since they can be fine-tuned, but minimal interactivity since they are specific. Procedural methods (such as Inverse Kinematics) are usually maximally interactive since they offer an algorithmic, general solution, but tend to be very hard to make expressive. Example-based methods based on a hybrid of the two techniques offer the best of both worlds.

---

Therefore, in order to create and maintain the illusion of life, the character must think; in order to convey thought, the character must move expressively to show its internal mental state to a viewer at all times. Any time the motion loses this *expressivity* of motion, the illusion of life can be lost. Likewise, if the character does not appear to respond properly to the continuously changing demands of *interactivity* will also appear lifeless.

Loss of *expressivity* in motion could be due to a glitch in the motion such as a velocity discontinuity. It could also be due to an obvious repetitive motion such as a hand-animated walk cycle or karate chop commonly seen in videogames. Even though the animation is hand-crafted to be very expressive, if it does not vary over time in response to changes in the world due to interactions with other creatures or a human participant, it will appear dull and lifeless no matter how great the animation clip is. Also, since a character's emotional state can change continuously, canned animations cannot express this since they were designed for discrete emotional states. For example, if a character is somewhat depressed and a friend slowly cheers him up but there are only clips for happy and despondent, this subtle mental state which changes continuously due to interaction cannot be expressed. Thus, a canned animation clips can be said to be very *expressive* for the specific mental state and context for which it was created, but not very *interactive*. Figure 2-1 depicts this graphically.

Likewise, if a character needs to respond to a sudden change in mental state, such as being surprised, the motion must respond appropriately and immediately to maintain the *interactivity* of motion as well. For example, say the character motor engine simply plays



a canned animation clip from an animator to reach for a doorknob when the brain tells it to. The animation was created with a certain doorknob height and character's mental state implicit in it. If the character then encounters a hobbit-hole door half as high, it needs to reach for the doorknob in the appropriate lower location or it will not appear interactive. To solve this problem, reaching and tracking for virtual characters is usually done with a general procedural algorithm such as Inverse Kinematics (IK) to handle the infinite variability required for maintaining interactivity. Since IK algorithms came out of the classical robotics community where the expressivity of motion does not matter, only where the robot's end-effector ends up, these algorithms often will be ascribed as having "robotic" motion by a human viewer. To deal with this, the programmer must tweak parameters or enter heuristics by hand, which is laborious and usually the result is still not very expressive. Therefore, procedural, numerical algorithms tend to offer maximal *interactivity*, but minimal *expressivity*, as is shown in Figure 2-1.

In order to create a strong illusion of life in a virtual character, we would like to maximize both the expressivity and the interactivity of the motion. Some sort of hybrid *example-based procedural algorithm* which allows the incorporation of expert knowledge in the form of animation clips should be able to offer the best of both worlds, as depicted in Figure 2-1. A hybrid should offer the *continuous variability* needed to maintain interactivity while simultaneously incorporating *expert knowledge* of motion in the form of canned animation examples, either from motion capture or an animation package. We argue that:

**Leveraging the animator's knowledge of expressive character motion in the form of canned example clips into the infinite variability of a numerical procedural algorithm should maximize both *interactivity* and *expressivity* in order to maintain the illusion of life of a character.**

So how can we leverage an animator's knowledge of expressive animation that is implicitly contained in animation examples?

## 2.2 Exploiting an Animator's Knowledge of Expressive Character Motion

The last section argued for using example-based procedural algorithms for leveraging an animator's knowledge implicit in animation examples. This section will describe three ways which we will exploit animation knowledge from expressive examples:

**Pose Blending** : Multi-target interpolation and extrapolation of clips

**Statistical Analysis and Synthesis of Joint Motion** : Modeling how a joint tends to move, estimating joint limits implicit in clips and generating new motions to test whether they are valid

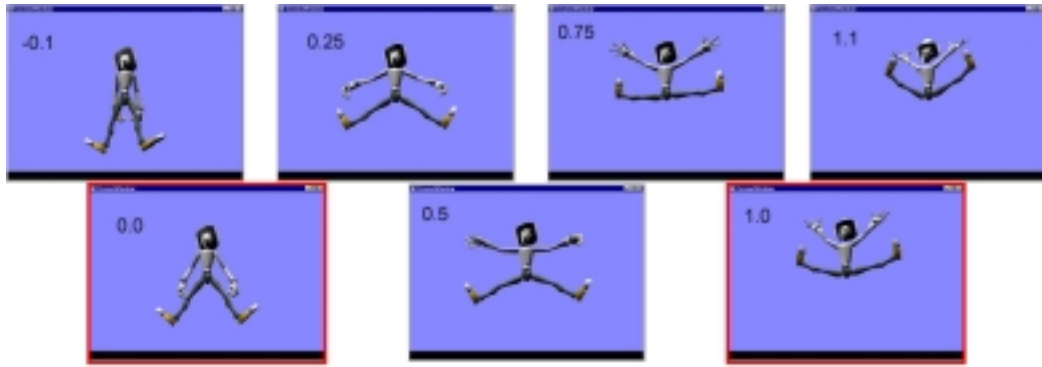


Figure 2-2: Blend of two animations, sampled at the same time  $t$  but at different happiness values from -0.1 to 1.1. The examples (red boxes) are the original animations. The frames in between the examples *interpolate* the posture according to the level of happiness. The frames outside the interval  $[0, 1]$  *extrapolate* the examples, making caricatures of the original walks.

**Expressive Inverse Kinematics (IK)** : Augmenting a numerical “robotic” looking IK solver with a model of body knowledge to find more “natural” and expressive solutions.

### 2.2.1 Pose Blending: Multi-Target Interpolation/Extrapolation

Animation clips give examples of how a character should move in a specific context. Consider Figure 2-2. The character is shown frozen at the top of a jump. The two boxed frames are from a hand-animated jumping animation of the character in a happy mood (1.0) and a sad mood (0.0). These two examples define constraints on the motion of the character by specifying how it should move when it needs to jump and is happy or sad. If we assume that the space of motion (motion-space) is smooth and continuous between these examples, we can use an *interpolation*, or *weighted blending* algorithm to try and estimate poses for a jump and values of happiness between 0 and 1. Figure 2-2 shows samples of frames interpolated by performing a weighted average of the two examples using the value of happiness as a weight. Interpolation algorithms are thus a way to *proceduralize examples directly*. They assume that a weighted average of expressive examples should also be expressive and recognizable. After Rose [44], we call the content of the animation (here a jump) a *verb* and the style (here happiness) an *adverb*.

Some interpolation algorithms are also capable of *extrapolation*, or estimating the nature of the motion outside of the convex hull formed by the examples (see Figure 2-3). In Figure 2-2, the frames sampled at -0.1 and 1.1<sup>2</sup> are examples of extrapolation. They produce plausible looking exaggerations of the examples in order to “caricature” the motion. Thus, a good pose-blending algorithm should allow for extrapolation in order to leverage the animator’s talent, which is one of our design principles espoused in the Introduction.

<sup>2</sup>“Our knobs go to 11.” – *Spinal Tap*

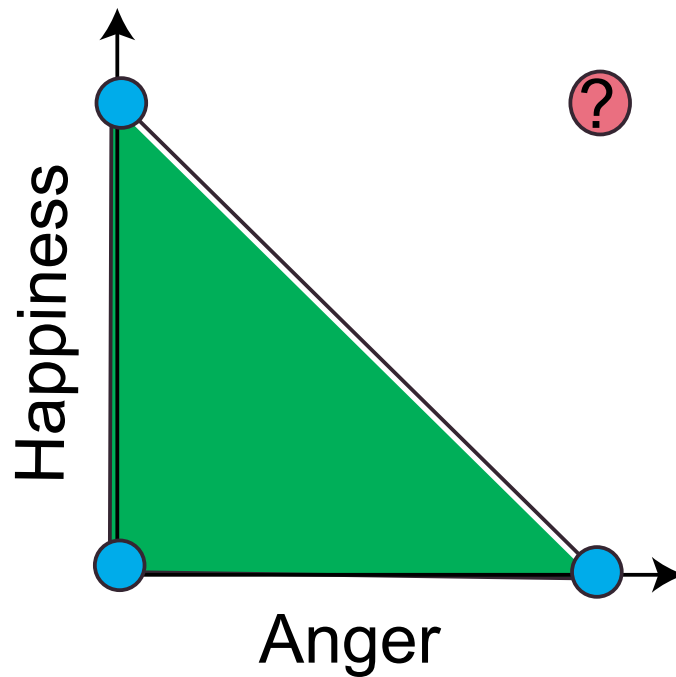


Figure 2-3: Extrapolation and Interpolation. The circles on the axes represent example animations of a happy, normal, and angry verb. The triangular filled region (convex hull) of the examples is the interpolation space. A point outside this space, such as the dark point specifying an angry *and* happy verb is an example of *extrapolation*. By extrapolating well, we can obviate the need for the animator to increase the size of the interpolation space with a new example at this point. As the number of axes increase, this gives us an *exponential* decrease in necessary examples.

---

Furthermore, our character will have more than just one degree of stylistic freedom. For example, in *α-Wolf* (see Section 10.4.6) a wolf pup had three axes of variability in its walk cycles: turning left/right, happiness, and how old the pup was (the pups grow up slowly to be adults). This implies the need for a *multi-target* interpolation algorithm that can blend more than  $N$  animation examples according to some continuous adverb that can be used to calculate weights on examples.

To summarize:

**Leveraging the animator by proceduralizing expressive motion examples with multi-variate interpolation implies the need for appropriate and robust *pose-blending* primitives.**

## 2.2.2 Statistical Analysis and Synthesis of Joint Motion

Clearly, if we keep extrapolating the character's happiness value in Figure 2-2, the character's joints will begin to exhibit unnatural behavior since they will break the implicit *joint motion limits* on the joint. The character's joints should be unable to move further due to the limits on joint mobility to maintain the illusion of life. Unfortunately, an interpolation algorithm has no notion of these constraints and will blithely spin the character's arms backwards.

### Joint Motion Limits

This implies the need for a way to enforce *joint range limits* on the possible mobility of the joint. Then if the interpolated solution tries to break a constraint, we can explicitly force it to remain at the limit. For example, an elbow should not rotate backwards unless the character is preparing for a trip to the hospital or is skilled in yoga. Ideally, we want to leverage the knowledge inherent in the animation clips rather than forcing the programmer to specify them by hand. How can we find these?

Notice that the joint limits are implicit in the animation in terms of the space where there *are no examples nearby*. Lack of examples implies either that:

- The region is a valid subset of motion space for that character where no examples have been seen yet.
- The region is not a valid subset of motion space since it violates some constraint on natural motion of the character.

We hypothesize that we can estimate the joint motion limits by learning a statistical model of the motion over all examples we have seen for the character. If we had such a model, we could then use an isocontour of probability as a constraint boundary! In other words, we can find a region that contains as much of the data (or all of it) as possible. If we find that a posture of the character has too low a likelihood, then we can assume that the

joint should not go there and use this to constrain motion. Since we can estimate it from data, we can then learn the model from the corpus of all animation examples in order to leverage the animator best.

Most statistical analysis methods will assume a *metric* on the data being analyzed. In this case, we will need a distance metric between two joint orientations of the character. The metrics need to be appropriate to the problem domain to be useful. Therefore:

**Statistical joint analysis implies the need for appropriate distance metrics on joint orientations.**

### Posture Constraints

Unfortunately, joint constraints are not enough. A posture could also be invalid if it would cause an interpenetration of the body with itself. Again, these constraints are also implicit in the animation in terms of where the data is not. Whereas joint limits are local, posture constraints imply the need for analysis of the *coupled motion* of joints. If we could find a statistical model of the space of *postures*, we hypothesize that we should be able to use the model to constrain posture to the subspace of examples which we have seen.

As above, metric are needed:

**Statistical analysis of posture will imply the need for appropriate distance metrics between two postures of a character.**

### 2.2.3 Expressive Inverse Kinematics

The essential problem of Inverse Kinematics (IK) is to find a pose of a character's body that results in a particular body part (or parts) ending up at a certain position in the world. Inverse Kinematics shows up in two main roles: animation tools and real-time character engines. In the former case, the animator uses the IK algorithm to help iteratively make keyframes more naturally in situations where constraints are required. In the case of videogame engines, IK is often a procedure applied to some joints in order to procedurally track moving objects with the character's head or (often) weapon. The former can run at "interactive" rates for the animator, like around 5hz. An IK engine for a real-time app needs to run at more like 100hz. Also, production IK tools allow a visual feedback iterative approach for the animator, while a character motor system requires fast and correct placement of the body parts on the fly without any chance for later correction.

Inverse kinematics techniques came out of mechanical engineering and robotics, where often robots were designed to have analytically solvable solutions. Also, these robots were designed to solve engineering and manufacturing tasks, so the content of the motion was

all that mattered — it doesn't matter how you get the torch to the part to weld, just make sure it doesn't hit anything and gets there. Thus, robots have “robotic” motion.

Systems that cannot be solved analytically require a numerical, iterative solution, which makes it a computationally expensive operation in general when there are many characters. These numerical solutions usually encode some simple notion of body knowledge (such as where joints tend to be and a stiffness) and sometimes some kind of joint limit.

In general, a given IK problem will have multiple (potentially infinite) solutions. To a traditional robotic system, the closest solution is usually all that is required. For an interactive, expressive character like a human, however, some of these solutions will appear more “natural”. For example, people's postures are subject to the force of gravity, therefore a lower energy posture is preferable and will look more natural. We argue that the exact manner that a character exploits its redundant kinematic degrees of freedom is a major part of the expressiveness of the character's motion. Therefore, an example-based statistical model of the motion subspace of a character should capture this expressive knowledge to some degree.

Therefore, we argue for an *expressive IK* system which consists of combining a “robotic” content solution with a model of body knowledge that lets us “project” the robotic solution onto the subspace of our character's actual motion. Such a system must handle the following issues:

- Speed
- Joint limits
- Expressive

We argue that we can leverage the procedural power of the numerical methods that exist by augmenting the iterations with a model of body motion knowledge gleaned from an animator's examples. Explicitly, we will argue for the following approach to solving expressive IK:

**Augment the procedural power of numerical search with the expressive power of expert body knowledge.**

## 2.3 Summary

To summarize, this chapter argued that:

- Canned animation examples (clips) are maximally expressive, but minimally interactive
- Numerical, procedural algorithms are maximally interactive, but minimally expressive.

We then stated the main hypothesis of our thesis:

**Leveraging the animator’s knowledge of expressive character motion in the form of canned example clips into the infinite variability of a numerical procedural algorithm should maximize both *interactivity* and *expressivity* in order to maintain the illusion of life of a character.**

We then motivated three ways which we chose to explore the use of example-based procedural methods to leverage the animator:

**Pose Blending** : Multi-target interpolation and extrapolation of clips

**Statistical Analysis and Synthesis of Joint Motion** : Modeling how a joint tends to move, estimating joint limits implicit in clips and generating new motions to test whether they are valid

**Expressive Inverse Kinematics (IK)** : Augmenting a numerical “robotic” looking IK solver with a model of body knowledge to find more “natural” and expressive solutions.

We showed that these methods entailed the need for:

- Appropriate metrics on joint orientation
- Appropriate metrics on postures
- Appropriate pose-blending algorithms that extrapolate well.





# Chapter 3

## Rotation, Orientation and Quaternion Background

This chapter will introduce the theory and issues of mathematically modeling the familiar notion of spatial rotations and rigid body orientations in our physical world (three spatial dimensions). Even though the concept is familiar physically, there are many ways to represent rotation mathematically and computationally, each with its own pros and cons. To appreciate these issues, an understanding of rotation is required. We will describe spatial rotation from first principles by introducing *Euler's theorem* of rotation. We will then describe four rotation representations popular in character animation in some detail in order to illustrate the mathematical issues that arise in using them. These are:

- Coordinate Matrix
- Axis-Angle
- Euler Angles
- Quaternions

We will briefly introduce the first three representations and compare them from a mathematical and computational point of view. In particular, we will focus on the important issues of interpolation, distance metrics, computational speed and mathematical robustness which we motivated in the previous chapter.

We will then focus on a mathematical and geometric introduction of quaternions, which are the basic mathematical representation which we use throughout our work, along with a similar discussion.

We will end the chapter with a multi-page table summarizing the useful formulas for quaternions from an algebraic and geometric viewpoint to serve as a reference. The mathematically-inclined reader can also find a more group theoretic reference on quaternions in Appendix D.

## 3.1 Rotation, Orientation and Euler's Theorem

Rotation of solid rigid bodies (for example, a rock) is intuitive. We grab objects all day and rotate them in different ways as we use them without thinking about it. How do we model this phenomenon mathematically and computationally? It turns out that this problem is far from intuitive. This section will introduce the basic principles of rotation of 3-space. We will explain the relationship between rotations and orientations of rigid bodies. This section will focus on basic concepts and fundamental issues with understanding rotation and orientation and not on any particular representations.

### 3.1.1 Rotation *versus* Orientation

The astute reader may have noticed our distinction between *rotations* and *orientations*. The difference is subtle and should be made clear.

A *rotation* is the action that transforms one vector into another vector. By definition, a rotation 1) preserves the magnitude of a vector and 2) preserves the handedness of the space (in vector algebra terms, it preserves the direction of the cross products between basis vectors). In most of this document, we will be assuming rotations in 3-space. Occasionally, we will discuss rotations of 4-space and will be explicit. Rotations in 3-space have 3 degrees of freedom, so we will need at least three numbers to define them.

An *orientation*, on the other hand, is the attitude of a rigid body in space. The terms are often and easily conflated because orientations are usually represented as a rotation *with respect to* a fixed, known coordinate frame (also called an *inertial frame* or *basis*). Figure 3-1 depicts a rigid body with an attached body (local) coordinate system  $[B]$  which is measured with respect to some fixed world coordinate system  $[W]$  with primes denoting the moving frame. Often the term *angular displacement* is used to make the distinction between rotation and orientation clear in the case of rigid bodies, since displacement implies action. For our purposes, we will ignore the translational component and focus on the rotation component.

### 3.1.2 Euler's Theorem and Distance Metrics

The fundamental principle of rigid body orientation is *Euler's theorem* (Figure 3-2). Euler's theorem can be stated as follows:

**Euler's Theorem:** Every displacement (or orientation with respect to a fixed frame) of a rigid body can be described as a rotation by some angle  $\theta$  around around some fixed axis  $\hat{n}$ .

Intuitively, Euler's theorem just says that if we grab a rock at some orientation in space and we want to rotate it to some other orientation, there *always* exists a *fixed* axis that we can rotate around in order to get to that orientation, and the magnitude of the rotation is the angle. In other words, the axis us tells us *which way* to rotate the object and the angle tells us *how far*.

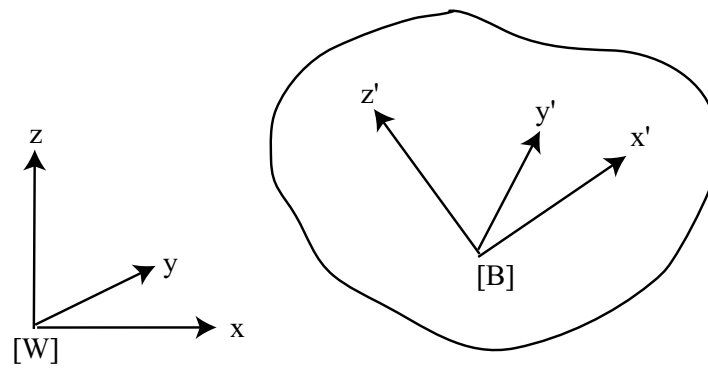


Figure 3-1: A moving body coordinate system  $B$  can represent the orientation of the body with respect to a known world coordinate system  $W$ . We ignore translational effects for simplicity.

---

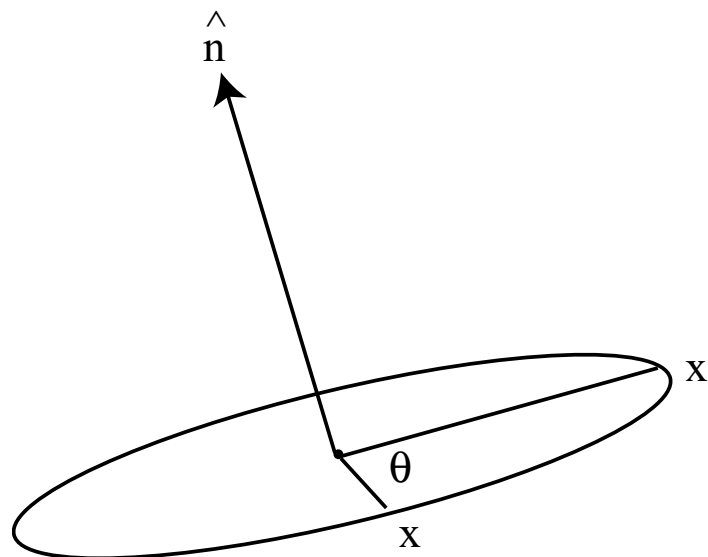


Figure 3-2: Euler's theorem states that the angular displacement of any rigid body can be described as a rotation about some fixed axis ( $\hat{n}$ ) by some angle  $\theta$

---

A useful property of Euler's theorem is that the angle directly gives us an intuitive notion of a distance metric on rotations and therefore orientations! In fact, this angle is the natural group metric for rotations. Therefore, if we want the distance between two orientations, we can directly use the angle of the rotation between the two orientations. We will see that this angle is easier to calculate in some representations than others and will be one motivation for a quaternion approach.

Finally, we note that Euler's theorem implies that spatial rotations have three degrees of freedom — two to specify the axis (since it is normalized, we can use spherical coordinates) and one for the angle. Therefore, the minimum number of parameters to describe a rotation is three.

### Composition and Non-Commutivity

Two rotations can be *composed* together by applying one rotation first and then the next. Euler's theorem assures us that the rotation created by this composition has its own axis and angle decomposition, though getting to this from the factors is dependent on representation. Unfortunately, 3D rotations do not (in general) commute under composition. In other words, if one rotates around some axis and angle to get a new orientation then rotates by a second angle and axis from that new orientation the resulting orientation will *in general be different than if we applied the rotations in the reverse order*. This is not the case for 2D rotations, where the angles can be added in any order.

To visualize non-commutivity intuitively, take your right hand and make coordinate axes like those in Figure 3-3. Point your thumb up and index finger forward. If you rotate by 90 degrees around your thumb (positive angles being counter-clockwise), then 90 degrees around your middle finger's new position, your index finger will be pointing down. If instead you rotate by 90 around your middle finger followed by 90 around the new thumb position, your index finger will point left!

Non-commutivity is a fundamental property of 3D rotations and will be important in some of our later discussions.

### 3.1.3 Summary

This section introduced some terminology and explained Euler's theorem. We described how Euler's theorem immediately gives us a natural distance metric on rotations and therefore between two orientations. The next section will introduce mathematical and computational representations and parameterizations of rotation.

## 3.2 Representing Rotations

Since we will describe a joint as an orientation with one, two or three rotational degrees of freedom, we need a way to represent this fact computationally. In general, there are four main ways that rotations are represented in practice:

- Coordinate Matrix

- Axis-Angle
- Euler Angles
- Quaternions

Ideally, we would like our choice of representation to have several important properties:

**Efficiency** The representation should take up minimal space in memory and be efficient in time for the tasks required. We want to minimize conversions to and from other representations since this is essentially “wasted” computation if we can get around it with the right choice of representation. Also, if our representation has its own algebra, we can perform rotation compositions within the representation.

**Robustness** The representation should be robust. Some representations, such as the Euler angles as we shall see, contain discontinuities that must be handled. We would like our representation to avoid these issues. Also, some representations are *redundant* in that several (or an infinite number, at times) elements can represent the same rotation. This can cause problems if not handled properly.

**Ease of Use and Visualization** Ideally, we want our representation to be simple and easy to visualize. As we saw above, we would like the representation to model the action of Euler’s theorem as simply as possible in order to simplify metrics, visualization and understanding.

The first three representations will be introduced in this section. We will treat quaternions, on which this research is based, in the following section for clarity.

### 3.2.1 Coordinate Matrix

The group of rotations of Euclidean 3-space ( $\mathbb{R}^3$ ) is usually denoted as  $\text{SO}(3)$ , which stands for the group of *special orthogonal* 3 by 3 matrices. Recall that an *orthogonal matrix* consists of orthogonal column vectors which are of unit magnitude — in other words, the columns form an *orthonormal basis* for the rotated space as measured in the *unrotated* space’s coordinate system (frame). Of the orthogonal matrices, called  $\mathbb{O}(3)$ , there are two subsets: those with  $\det = +1$  and  $\det = -1$ , where  $\det$  is the matrix determinant. The subset with negative determinant are *reflections* since they change the handedness of space. The subset of  $\mathbb{O}(3)$  with  $\det = +1$  are called the *special* orthogonal matrices. A rotation matrix  $\mathbf{R} \in \text{SO}(3)$  will transform a column vector  $\mathbf{x} \in \mathbb{R}^3$  to a new column vector

$$\mathbf{y} = \mathbf{R}\mathbf{x}$$

by rotating it and preserving its magnitude.

A coordinate matrix can be trivially produced if a basis for the rotated space with respect to the old one is known — it is just the matrix with the basis in the columns of the matrix. Explicitly,

$$\mathbf{R} = \begin{bmatrix} | & | & | \\ \hat{\mathbf{x}}_{new} & \hat{\mathbf{y}}_{new} & \hat{\mathbf{z}}_{new} \\ | & | & | \end{bmatrix}$$

will rotate a vector in the unrotated basis space into one in the new space defined by the vectors in the columns. In Figure 3-1, the body axes define the body's orientation with respect to the world and would serve as our basis. Note that this definition of the matrix assumes we are using column vectors. Some graphics libraries and texts (for example, [86]) use row vectors, which means the basis is in the rows.

## Composition

The composition of rotations is simply the familiar multiplication of the corresponding matrices:

$$\mathbf{R}_2 \mathbf{R}_1 = \mathbf{R}_{21}$$

where  $\mathbf{R}_1$  will be applied to the column vector first. Note that the order of rotations must be read from right to left, since we are using column vectors.

As we mentioned above, rotations do not commute. Therefore, a matrix representation of rigid body rotation will be non-commutative as well.

Mathematically speaking,

$$\mathbf{R}_1 \mathbf{R}_2 \neq \mathbf{R}_2 \mathbf{R}_1 .$$

This fact is important and can be easy to forget, but has far-reaching implications.

## Euler's theorem and Metric

The axis of rotation of a matrix is the eigenvector with unity eigenvalue, which Euler's theorem decrees must exist. Recall that an eigenvector of a transformation is scaled by the transformation. In the case of rotation it is the set of points that do not move under rotation, which is the definition of an axis of rotation.

The other two eigenvectors will be complex and have complex eigenvalues  $e^{\pm i\theta}$ . They define a plane orthogonal to the axis of rotation. Although we can get to the angle (or metric) through the eigenvalues, we can also rotate an arbitrary unit vector in the plane of rotation by the matrix and then find the resulting angle between the original and resulting vectors using a dot product and arccos.

## Advantages

Mathematically, the matrix representation seems almost perfect since we used it, in a sense, to *define* rotations —  $\mathbf{SO}(3)$  is exactly the mathematical group we want to represent. Matrices in  $\mathbf{SO}(3)$  map 1-to-1 onto angular displacements of rigid bodies. We will see that other representations do not have this 1-to-1 property, including the quaternions.

Matrices are usually familiar since linear algebra is taught early in most college curricula. Many algorithms are based on a matrix representation, so there is a lot of history which can be drawn on as well.

### Disadvantages

Unfortunately, the matrix representation has several computational problems. First, it takes nine parameters to represent a structure with only three inherent degrees of freedom. This means there are six constraints on the matrices that need to be enforced to remove the extra degrees of freedom: the orthonormality of the columns and the determinant being positive. If memory is at a premium, this is an inefficient representation.

Additionally, when many rotations are concatenated numerically, roundoff error will cause the matrix to drift away from special orthogonal, which introduces shearing and scaling effects which are undesirable. We can use the standard Gram-Schmidt algorithm (see, for example, Strang [81]) to “renormalize” the matrix, but this can be computationally expensive if we need to do it often.

Interpolating between matrices in  $\text{SO}(3)$  is tricky due to the multiple constraints. The familiar convex sum interpolator used to interpolate within vector spaces:

$$\alpha \mathbf{R}_1 + (1 - \alpha) \mathbf{R}_2 \tag{3.1}$$

*does not work* on  $\text{SO}(3)$  since in general the interpolated matrices will violate the constraints. Mathematically speaking,  $\text{SO}(3)$  is *not* a vector space (although we use them as affine transformations of vector spaces). A vector space requires linear superpositions of elements to be closed under the space (see Appendix D for a more formal definition). Therefore, we cannot use the many familiar and well-understood vector space algorithms directly on rotations. This fact is one of the most important to remember in our research, so we repeat it:

**The group of rotations of 3-dimensional Euclidean space, known as  $\text{SO}(3)$ , does not form a vector space.**

This is the crux of most people’s problem with designing algorithms for rotations, as we will see. Instead of a vector space, rotations form a *Lie group*, which we introduce briefly in Appendix D along with references for the interested and mathematically-inclined reader.

Numerical integration of a rotation matrix differential equation, which is similar to the interpolation problem, also causes normalization problems due to numerical error.

Finally, matrices are hard to visualize in terms of the action they perform since the axis is an eigenvector, and not directly accessible in the representation.

### 3.2.2 Axis-Angle

One way to parameterize  $\text{SO}(3)$  is by using Euler’s theorem directly and representing a rotation as the pair  $(\hat{\mathbf{n}}, \theta)$ . This is called an *axis-angle* representation and most graphics

libraries offer conversions between matrices and axis-angle.

### Composition

It is not simple to compose rotations in this notation without converting in and out of some other representation such as a coordinate matrix and creating the formula in that manner. This can be computationally expensive.

### Euler's Theorem and Metric

A nice feature of axis-angle is that it directly represents Euler's theorem. Therefore, we can immediately use the angle portion of a rotation between two orientations as the metric. However, the lack of a simple composition rule makes this metric computationally expensive since we need to convert to and from a matrix.

### Advantages

The main advantage of the axis-angle representation is that it is as close to Euler's theorem as we can get. It directly represents the action of rotation. This makes it quite appealing from an intuitive point of view.

### Disadvantages

Renormalization does not immediately seem to be a problem since we can normalize the axis if it drifts from unity magnitude. This method does not address what happens when numerical error creeps into the angle portion, however. Essentially, it is ignored.

Another issue is that an infinite number of angle choices (multiples of  $2\pi$ ) represent the same rotation. To avoid confusion, the convention that the axis is a unit magnitude vector and the angle is in the interval  $[-\pi, \pi]$  is often chosen. Even with this convention, two axis-angle pairs still refer to the same rotation. Specifically,  $(-\hat{n}, -\theta)$  refers to the same rotation as  $(\hat{n}, \theta)$ .

Looking more carefully, there is also a nasty redundancy in the fact that a zero rotation around *any* axis is the same exact rotation, the identity! In other words, the representation of the identity element of  $\text{SO}(3)$  is not unique — in fact there is an uncountably infinite number of them — which can cause serious problems in algorithms as rotations approach the identity element. Often special case conditions are used near the identity to get around this, like defining a zero rotation around the  $\hat{x}$  axis to be the identity rotation when the angle approaches zero, but this can introduce discontinuities, exactly what we are attempting to avoid.

Axis-angle may seem safe and simple for doing interpolation naively using a linear interpolation (the convex sum in Equation 3.1) between the four components of the representation. This approach, like the matrix version, is problematic. First, and most obvious, the components of the representation are not in the same units, so applying the same scale to them is suspicious at best! Almost certainly, one will not get the shortest path interpolation between the two points (the interpolated axis-angles can be converted to  $\text{SO}(3)$  and a natural distance metric there can be used to prove this). Second, one needs to deal with the



wrap-around of the angle if one wants semi-unique representatives for the rotation (it could also just be allowed to range over all of  $\mathbb{R}$ , but this can be problematic computationally due to numerical overflow).

If we choose to keep the angle in a fixed range, the interpolation cannot be continuous — at some point it needs to “jump” through the boundary from  $-pi$  to  $pi$  or from 0 to  $2\pi$ . These discontinuities wreak havoc on most interpolation and numerical integration schemes that are unaware of them.

### 3.2.3 Euler Angles

A common way to represent a rotation in an animation system is to factor it into three sequential rotations around the principal orthogonal axes ( $\hat{x}$ ,  $\hat{y}$ ,  $\hat{z}$ ) and represent the rotation as the triple  $(\theta_1, \theta_2, \theta_3)$ , with each angle being around some particular axis. This is based on the fact that the rotation has three degrees of freedom, so three angles should specify any rotation.

Each of these matrices has a simple form:

$$\begin{aligned} \mathbf{X} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x \\ 0 & \sin \theta_x & \cos \theta_x \end{bmatrix} \\ \mathbf{Y} &= \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y \\ 0 & 1 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y \end{bmatrix} \\ \mathbf{Z} &= \begin{bmatrix} \cos \theta_z & -\sin \theta_z & 0 \\ \sin \theta_z & \cos \theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Note that the matrices will be transposed if a row vector basis is used.

Any product of three of these matrices such that no two consecutive matrices have the same axis is usually called an *Euler angle set* in the robotics ([18]) and graphics ([77],[86]) communities <sup>1</sup> There are twelve possible products: X-Y-Z, X-Y-X, Y-Z-X, Y-Z-Y, Z-X-Y, Z-X-Z, X-Z-Y, X-Z-X, Y-X-Z, Y-X-Y, Z-Y-X, Z-Y-Z. These are usually read in the order the rotations are applied, and not the order of matrix multiplication, which can be confusing.

Consider the factorization Z-Y-X, which means we rotate around Z then Y then X. There are in fact two ways to think about this, which leads to confusion:

- Fixed axis
- Moving axis

---

<sup>1</sup>This is actually a misnomer which can lead to confusion. Euler angles were invented by physicists to solve certain problems such as the precessing gyroscope. In general, physicists refer to either Z-X-Z or Z-Y-Z as *Euler angles* by convention. The aerospace, graphics, and robotics communities borrowed these from physicists, but along the way the name has become used for any of the twelve factorizations (see, for example, Shoemaker [77] or Watt and Watt [86]).

A *fixed axis* viewpoint states that you rotate the object around the world  $\hat{z}$ , then the world  $\hat{y}$ , then the world  $\hat{x}$ .

A *moving axis* viewpoint states that you rotate around the local (body) axes: first around the local  $\hat{z}$ , then the newly rotated  $\hat{y}$ , which we denote  $\hat{y}'$ , then around the newly rotated  $\hat{z}$ , which is denoted  $\hat{z}''$ . Figure 3-3 shows how to think about moving axis intuitively. If you imagine local coordinate axes on your fingers, then you would rotate around your thumb ( $\hat{z}$ ) first, then around the new position of your middle finger ( $\hat{y}'$ ), then around the new position of your index finger ( $\hat{x}''$ ).

Usually only the moving axis formulation is called an Euler angle set, and the other a *fixed angle set*, after Craig [18]. Surprisingly, the two viewpoints turn out to only reverse the order of matrix multiplication of the three factors! Specifically,

**Moving axis Z-Y-X = Fixed axis X-Y-Z**

To prove this, consider the action of the rotations on an arbitrary frame,  $\mathbf{B}$ . In order to rotate Z-Y-X in world space, we first rotate around the  $\hat{z}$  axis, call the rotation  $\mathbf{R}_1$ , to get a new frame,  $\mathbf{B}'$ . In order to next apply the second rotation around the world  $\hat{y}$ , we need to use a change of basis operator to specify the rotation in the original frame,  $\mathbf{B}$ . The resulting rotation we need to apply to  $\mathbf{B}'$  to rotate around  $\hat{y}$  in  $\mathbf{B}$  is thus:

$$\mathbf{R}_2 = \mathbf{Z}\mathbf{Y}\mathbf{Z}^{-1}$$

Read from right-to-left, we “undo” the  $\mathbf{Z}$  rotation to get back into the world frame, then apply the  $\mathbf{Y}$  rotation there, then “redo” the  $\mathbf{Z}$  rotation. But this clearly has the effect of reversing the order of the multiplications, since

$$\mathbf{R}_2\mathbf{R}_1 = \mathbf{Z}\mathbf{Y}\mathbf{Z}^{-1}\mathbf{Z}$$

and the two right hand factors will cancel to leave

$$\mathbf{R}_2\mathbf{R}_1 = \mathbf{Z}\mathbf{Y}$$

This composite rotation takes  $\mathbf{B}$  into  $\mathbf{B}''$ .

Finally, to apply the rotation around  $\hat{x}$  in world space, the required rotation is:

$$\mathbf{R}_3 = \mathbf{Z}\mathbf{Y}\mathbf{X}\mathbf{Y}^{-1}\mathbf{Z}^{-1}$$

which when applied to our other two rotations gives us

$$\mathbf{R}_3\mathbf{R}_2\mathbf{R}_1 = \mathbf{Z}\mathbf{Y}\mathbf{X}$$

due to similar cancellation. But this is the same result that we would get if we rotated around  $\hat{x}$  first, then the local  $\hat{y}$ , then the local  $\hat{z}$ ! Therefore, moving axis Z-Y-X is the same as fixed axis X-Y-Z.

Intuitively, in order to perform a rotation in world space we need to “reach in” and multiply it on the right.

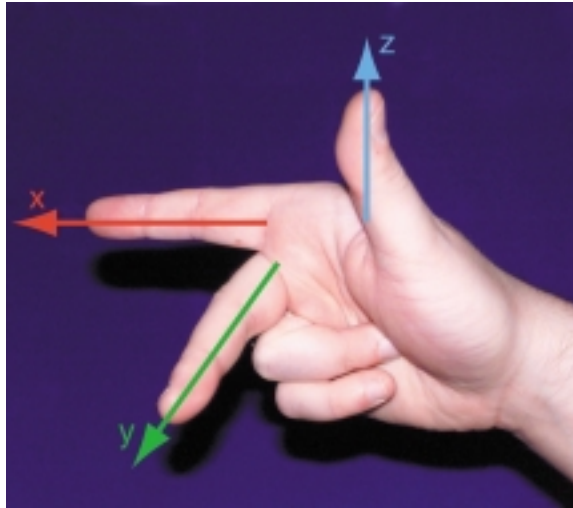


Figure 3-3: Euler angle illustration: pretend your fingers when held as shown are three moving orthogonal axes. Any orientation in space can be specified by a *yaw* around the thumb followed by a *pitch* around the middle finger then a *roll* around the index finger.

---

### Euler's Theorem and Metrics

Euler angles are quite far removed from Euler's theorem, ironically. Although in the case of 2D rotations they are the same, in 3D this is not the case. In order to find the equivalent axis-angle of an Euler set, and therefore the natural metric between two orientations, we need to:

1. Create the three factor matrices from the angle.
2. Multiply the three matrices together.
3. Extract the axis-angle from the resulting matrix.

Computationally, this is too expensive. Usually the standard Euclidean vector distance metric is used as if the Euler angle triple were a vector (it is not, as we discuss below). In other words, given two Euler angle triples arranged in a vector,  $\theta_1$  and  $\theta_2$ , the standard metric used is:

$$d = \|\theta_1 - \theta_2\|$$

This “metric” ignores the coupling between the components of the Euler angles, and is therefore appropriate only for nearby orientations. This metric will also be badly behaved whenever one of the angles jumps through a discontinuity, such as from  $2\pi$  to 0, since the components actually live on circles ( $S^1$ ) and not in a vector space, which the Euclidean metric assumes.

## Advantages

One main advantage of using Euler angles is that people can understand them quickly and can specify an orientation with them fairly well except in certain cases, as we describe below. They also have a long history in physics and can make certain integrals over the space of rotation easier to do.

The main reason they seem to still show up in animation packages like 3D Studio Max is that they allow the animator to view and tweak animation using *function curves*, which are 2D plots of each angle over time.

Euler angles are minimal — only three parameters, so seem to be efficient. As we show below, however, there is a distinct advantage to using four parameters instead of three.

Finally, the fact that the angles are used directly implies that no normalization needs to be done on the angles (although to make the triple unique ranges must be specified, as we see below).

## Disadvantages

The principal disadvantage of Euler angles is that mathematically there is an inherent singularity in any minimal (3 parameter) parameterization of  $\text{SO}(3)$ . Clearly, there is not a singularity in the rotation group since we can rotate a free body in space physically without bumping into any singularities! This singularity results from the loss of a degree of freedom in the *representation itself*, called a *coordinate singularity* (see [60] for a clear description of coordinate singularities versus singularities in the geometric structure itself).

As a concrete example, consider the following set of rotations: rotate  $\pi/2$  around  $\hat{z}$ , then  $\pi/2$  around  $\hat{y}$ . Your  $\hat{x}$  axis has aligned with the original  $\hat{z}$  axis! (In the hand notation, your index finger is now pointing in the same direction that your thumb started in). Any orientation that can be gotten by adding a roll in this new configuration could have been produced by initially rotating around the  $\hat{z}$  axis instead! Mathematically, the extra degree of freedom has collapsed.

**Gimbal Lock** This coordinate singularity is commonly referred to as *gimbal lock* for historical reasons. A *gimbal* is a physical device consisting of concentric hoops with pivots connecting adjacent hoops, allowing them to rotate within each other (see Figure 3-4). A gimbal with three rings attached orthogonally as in the figure is in fact a *physical realization* of a moving Z-Y-X Euler angle description.<sup>2</sup> Gimbals are often used to hold gyroscopes in attitude sensors in the aeronautical industry. Since gyros want to stay fixed in space, a gimbal connected to an airplane body or satellite can allow a gyro to actually stay fixed in space — the gimbal will move around it in order to keep the gyro at that orientation. The Euler angle values can then be read trivially off the pivots with simple electronics like shaft encoders.

Figure 3-4 shows a locked gimbal on the right — here the teapot cannot be rotated around its local “up” direction. Even worse, as one approaches gimbal lock, the singularity usually causes numerical ill-conditioning, often evidenced physically by the gimbal wig-

---

<sup>2</sup>The author did not really understand gimbal lock until he played with a real one and locked it up himself.

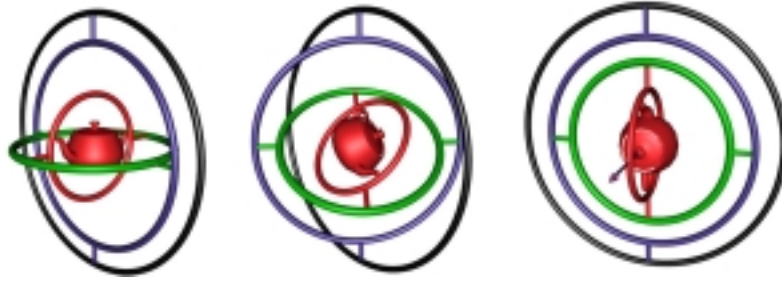


Figure 3-4: A gimbal consists of three concentric hoops connected by single degree of freedom pivot joints (each pivot is a physical realization of an *Euler angle*) which attach adjacent hoops orthogonally (the outermost black hoop here is considered the “earth” and is fixed in space and cannot rotate.). The left image depicts the gimbal in its “zero” position, with the teapot (colored red to show that it is fixed to the red hoops’s coordinate frame and cannot rotate independently of it) in an “unrotated” position, with the three hoop pivots orthogonal and corresponding to axes (red is  $\hat{x}$ , green is  $\hat{y}$  and blue is  $\hat{z}$ ). The middle image illustrates an arbitrary rotation of the teapot and the associated gimbal configuration. The right image shows the inherent problem with three hoop gimbals and any associated Euler angle representation — gimbal lock. Here the teapot’s nose is pointing straight up, and two hoops have aligned, removing a degree of freedom. In this configuration, it is impossible to find a smooth, continuous change of the gimbal which will result in a rotation around the teapot’s local “up” direction, here shown as a superimposed purple axis. Any attempt to rotate around the purple axis is impossible from this configuration — the gimbal is said to be *locked* since it has lost a degree of freedom. A real gimbal with a gyro instead of a teapot would shake itself to pieces if it tried to rotate around this locked axis — a very real phenomenon in early navigational systems using Euler angles and real gimbals.

gling madly around as it operates near the singularity<sup>3</sup>. Gimbal lock is why the aerospace industry was one of the first to switch to using quaternions to represent orientation — satellites, rockets and airplanes are not happy when their navigational gyro gimbals lock up and are likely to crash<sup>4</sup>

Gimbal lock will occur somewhere in *any* fixed choice of axes. The only way around it is to add a fourth gimbal ring and actively drive the other rings away from lock, but this is ad hoc and adds complexity. We will show below that quaternions add a fourth parameter in a principled manner.

**Interpolation** Gimbal lock wreaks havoc on any interpolation scheme or numerical integrator which tries to smoothly interpolate through the singularity. Usually it is evidenced by extremely poor numerical performance, or the system jittering (most early computer graphics cameras or airplane simulations using Euler angles spin wildly when pointed straight

<sup>3</sup>Personal communication with Robert Nicholls, Lincoln Labs, MIT.

<sup>4</sup>For an interesting report on this problem in the early Apollo program, see [42] which describes how the inertial navigation system for the Apollo Lunar Excursion Module suffered from gimbal lock. The pilots were taught to steer away from the singularity, as was dramatized in the movie *Apollo 13*.

up). This is the case for numerical integration as well. Furthermore, if one interpolates Euler angles using the standard convex sum, the resulting path when viewed in  $\text{SO}(3)$  will not take the shortest path between the endpoints as it does in a vector space, which is usually what we want (see Watt and Watt [86] for some nice pictures of some of the paths that can occur). Also, extrapolation will be poor for the same reasons.

**Factorization of  $\text{SO}(3)$**  The essential mathematical problem with an Euler angle formulation is that it tries to do a global *factorization* of the rotation group  $\text{SO}(3)$  into a subset of  $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ , or  $\mathbb{R}^3$  (subset due to the angle interval constraints). Mathematically, however,  $\text{SO}(3)$  is a minimal group and cannot be factored! In other words, *any* such factorization of  $\text{SO}(3)$  into “smaller chunks” will have problems somewhere — there is simply no way around this.

Hanson [36] describes the factorization problem from a synthesis point of view, which is simpler to understand and important in its own right, so we stress it:

**If you rotate something around  $\hat{x}$  and then around  $\hat{y}$ , there will always be a component of  $\hat{z}$  rotation in the result.**

This property can be proven to oneself (and can also be used to prove that  $\text{SO}(3)$  is unfactorable) by multiplying an  $\hat{x}$  and a  $\hat{y}$  rotation matrix together and then extracting the axis-angle description from the resulting matrix (by finding the eigenvector with eigenvalue 1, for example) — the axis will have a  $\hat{z}$  component, meaning that there is some  $\hat{z}$  rotation in the result, even though we thought we added none. Hence, the components of a factorization are coupled and cannot change independently without causing problems. If it were a valid factorization, they would not be coupled and would transform independently.

### 3.2.4 Representation Summary

This section discussed several representations of rotation and the issues involved with computational use of these representations. The next section will introduce the quaternion representation of rotation which we use extensively in our research.

## 3.3 Quaternions

Quaternions were discovered on October 16, 1843 by the great Irish mathematician Sir William Rowan Hamilton as he was walking along the canals by the Royal Irish Academy in Dublin, Ireland with his wife. For many years, he had been searching for a way to multiply and divide “triples” of real numbers (what we call a 3-vector today) by extending the complex numbers, which allow the division of doubles (sets of two reals), into three dimensions. On that day, he realized “in a flash of insight” that he needed three imaginary units and one real instead of a real and two imaginary units. So excited was he by the

discovery that he carved the fundamental quaternion algebra equations into a rock with his knife [6]. Today, the spot is commemorated by a plaque shown in Figure 3-5.

This section will introduce Hamilton's quaternions, the representation of rotation we use in this research, from an algebraic and geometric point of view, with an emphasis on intuition. A more formal mathematical treatment can be also found in Appendix D. The section will proceed as follows:

**Section 3.3.1** will introduce quaternions as an extension of complex numbers and give the basic formulae for the quaternion algebra.

**Section 3.3.2** will describe the polar form of quaternions and describe how they can be used to model 3D rotations.

**Section 3.3.3** will describe the topological space of the *unit* quaternion group, the hypersphere in four dimensions, denoted  $S^3$  (since it has three intrinsic degrees of freedom). This topology will allow us to use spherical geometry in order to design algorithms.

**Section 3.3.4** will describe the exponential mapping of the quaternion group and its relation to tangent spaces on the hypersphere. The exponential map and its inverse the logarithmic mapping will be important tools in designing our algorithms.

**Section 3.3.5** will give a brief introduction to quaternion calculus as we will use it in this document.

**Section 3.3.6** introduces the basic building block used in quaternion spline interpolation, *slerp* (spherical linear interpolation).

**Section 3.3.9** will give the interested reader pointers to recommended reading for other approaches to quaternions. It will also give a summary of the other guises the quaternion group goes by in other fields.

### 3.3.1 Quaternion Hypercomplex Algebra

As Hamilton originally discovered, the quaternions are an extension of the complex numbers into four dimensions<sup>5</sup>, with a real part and three distinct imaginary parts. Higher dimensional complex numbers such as quaternions are called *hypercomplex*. Many properties of quaternions can be discovered by extending the familiar theorems of complex analysis [70] by simple analogy to quaternions.

---

<sup>5</sup>There does not exist a three-dimensional version of complex numbers, which was the main stumbling block for Hamilton — even dimensions are required.



Figure 3-5: While walking from his work at Dunsink Observatory to his home in Dublin, Hamilton realized that he needed a third imaginary unit and was so excited that he scratched the quaternion algebra equations onto a rock on the bridge over a canal near the Royal Irish Academy [6]. (Photo credit: Rob Burke 2002)

### Definition

Mathematically, a quaternion can be written in the form

$$Q = w + xi + yj + zk$$

where  $w, x, y, z \in \mathbb{R}$  and  $i, j, k$  are each distinct imaginary numbers such that

$$i^2 = j^2 = k^2 = ijk = -1$$

and pairs multiply similarly to a cross product in a right-handed manner:

$$ij = -ji = k$$

$$jk = -kj = i$$

$$ki = -ik = j$$

Hamilton called the pure real term a *scalar* and the collective imaginary portion a *vector* [33], which is where the current terminology originated. The collection of the four real coefficients  $(w, x, y, z)$  he termed a *quaternion*. We will denote the group of quaternions as  $\mathbb{H}$ , such that  $Q \in \mathbb{H}$  denotes a quaternion with arbitrary magnitude.

A more modern and shorthand notation for a quaternion which mirrors more closely



the traditional complex number notation is to define it <sup>6</sup> as the formal sum of a real scalar and real 3-vector:

$$Q \triangleq w + \mathbf{v}$$

where  $\mathbf{v} \in \mathbb{R}^3$ . Expanding this out by components gives

$$w + \mathbf{v} = w + x\hat{\mathbf{i}} + y\hat{\mathbf{j}} + z\hat{\mathbf{k}}$$

and the  $\hat{\mathbf{i}}, \hat{\mathbf{j}}$  and  $\hat{\mathbf{k}}$  here can be thought of as an imaginary basis for the vector portion of the sum:

$$\hat{\mathbf{i}} \triangleq 0 + 1i + 0j + 0k$$

$$\hat{\mathbf{j}} \triangleq 0 + 0i + 1j + 0k$$

$$\hat{\mathbf{k}} \triangleq 0 + 0i + 0j + 1k$$

Hamilton called a quaternion with zero real part a *pure* quaternion since it is purely imaginary. A vector  $\mathbf{x}$  in  $\mathbb{R}^3$  can be represented as the pure quaternion  $0 + \mathbf{x}$ , which will be useful below. Similarly, the reals are the set of pure scalar quaternions.

## Basic Operations

The *conjugate* of a quaternion, denoted by a star (\*) superscript, simply negates the imaginary part as with a normal complex number:

$$Q^* = w - \mathbf{v}$$

The *magnitude* of a quaternion is simply the product of a quaternion with its conjugate, as with complex numbers:

$$|Q|^2 = QQ^* \tag{3.2}$$

$$= Q^*Q \tag{3.3}$$

$$= w^2 + \mathbf{v} \cdot \mathbf{v} \tag{3.4}$$

A quaternion with unit magnitude is called a *unit quaternion*. We will make extensive use of them in this document and describe them in more detail below.

## Addition

Quaternions can be added and subtracted commutatively in the standard way by performing the operation component-wise. It will be important later to note that if we add two unit quaternions, we will not get another unit quaternion, so addition is only closed over the entire quaternion group.

---

<sup>6</sup>The symbol  $\triangleq$  is means “defined as equal”

## Multiplication

Multiplication of quaternions follows by simply doing a normal Cartesian product of the two quaternions as if they were polynomials of the terms  $i, j, k$  and then performing reductions of the higher order products of imaginary terms ( $i^2, ij$ , etc.) using Hamilton's set of algebraic rules above. This manipulation gives an explicit quaternion multiplication formula useful for computation:

$$\begin{aligned} Q_1 Q_2 = & ((w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) + \\ & (y_1 z_2 - y_2 z_1 + w_1 x_2 + w_2 x_1) i + \\ & (x_2 z_1 - x_1 z_2 + w_1 y_2 + w_2 y_1) j + \\ & (x_1 y_2 - x_2 y_1 + w_1 z_2 + w_2 z_1) k) \end{aligned}$$

A simple corollary of this definition is that a pure unit quaternion (having zero scalar part and a unit magnitude vector part) squares to -1 under the quaternion multiplication! Explicitly,

$$(0 + \hat{\mathbf{v}})^2 = (0 + \hat{\mathbf{v}})(0 + \hat{\mathbf{v}}) = -1 \quad \forall \hat{\mathbf{v}} \in \mathbb{R}^3 .$$

This property makes the correspondence to the complex numbers obvious, but with the imaginary component being a vector rather than a scalar.

The quaternion product can also be written in terms of vector algebra notation as:

$$PQ = (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, \mathbf{v}_1 \times \mathbf{v}_2 + w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1)$$

People familiar with the vector algebra will notice that the cross product term implies immediately that the quaternion multiplication is not commutative, as we expect. We see that the quaternion product contains both the dot (scalar) and cross (vector) products separately in the quaternion. Although neither the dot or cross product is invertible by itself, their sum is invertible!

The quaternion with unity scalar is clearly the multiplicative *identity* element since multiplying it has no effect on a quaternion.

## Inverse (Division)

The *inverse* of a quaternion is defined as

$$Q^{-1} = \frac{1}{|Q|} Q^*$$

just as with complex numbers. All quaternions except the zero quaternion have a unique inverse.

## Unit Quaternions

Quaternions with unit magnitude form a subgroup of the full quaternion group. In the next section, we will see that the unit quaternions are all that are required to represent rotation,

although we need the full quaternion group in order to define rotations and derivatives. To make the notation clear, we will denote a *unit* quaternion as  $\hat{Q} \in \hat{\mathbb{H}}$ . The “hat” implies unit magnitude.

A useful property of unit quaternions is that their inverse is simply the conjugate.

$$\hat{Q}^{-1} = \hat{Q}^*$$

### 3.3.2 Polar Form and Powers

Again in correspondence with the complex numbers, a quaternion  $q \in \mathbb{H}$  permits the polar representation

$$Q = r e^{\hat{\mathbf{n}} \frac{\theta}{2}} = r \left[ \cos \frac{\theta}{2} + \hat{\mathbf{n}} \sin \frac{\theta}{2} \right]$$

where  $\hat{\mathbf{n}}$  is a pure quaternion. Here,  $r$  is the *magnitude*,  $\frac{\theta}{2}$  is called the *angle* of the quaternion, and  $\hat{\mathbf{n}}$  is called the *axis*. It is useful to write the angle as  $\frac{\theta}{2}$  rather than  $\theta$ , as we shall see shortly when we show how to use quaternions to rotate vectors. The exponential must be taken as the formal power series:

$$e^{\frac{\theta}{2} \hat{\mathbf{n}}} = 1 + \frac{\theta}{2} \hat{\mathbf{n}} + \frac{(\frac{\theta}{2} \hat{\mathbf{n}})^2}{2!} + \frac{(\frac{\theta}{2} \hat{\mathbf{n}})^3}{3!} + \frac{(\frac{\theta}{2} \hat{\mathbf{n}})^4}{4!} + \frac{(\frac{\theta}{2} \hat{\mathbf{n}})^5}{5!} + \dots$$

for the formula to make sense (as with matrix exponentials) and reducing terms with the fact that  $\hat{\mathbf{n}}^2 = -1$ . We will make extensive use of the exponential form of quaternions and describe this in more detail below.

DeMoivre’s power theorem also carries to the quaternions:

$$Q^t = r^t e^{\hat{\mathbf{n}} t \frac{\theta}{2}} = r^t \left[ \cos(t \frac{\theta}{2}) + \hat{\mathbf{n}} \sin(t \frac{\theta}{2}) \right]$$

One must be careful using the exponential form of quaternions since the product is not commutative. Therefore, standard rules of exponentials learned from high school do not apply! This can be a potential source of errors in derivations.

An intuitive way to think about exponentiation of a *unit* quaternion is that it calculates a point that it is the fraction  $t$  along the great circle from the identity (1) to the quaternion  $\hat{Q}$ . This can be found by using the trigonometric form of deMoivre’s formula as well. Since  $\hat{\mathbf{n}}$  is orthogonal to this great circle, if  $t$  is varied with constant speed, we will move along this great circle at constant angular speed as well. This is the basis for fundamental quaternion interpolator, *slerp*, described below.

### Rotations of 3-Vectors

A quaternion can be used to represent a rotation of normal Euclidean 3-space,  $\mathbb{R}^3$ . Recall that we can interpret a vector as a pure imaginary quaternion. Consider the following quadratic product:

$$\mathbf{y} = Q \mathbf{x} Q^{-1}$$

with both  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$  interpreted as pure quaternions and  $Q \in \mathbb{H}$ . It can be proven that this triple product will *always* produce a pure quaternion for any unit quaternion  $q$  and any pure quaternion  $\mathbf{x}$ . Most importantly,  $\mathbf{y}$  will be the vector  $\mathbf{x}$  rotated by  $\theta$  radians around the axis  $\hat{\mathbf{n}}$ ! (The interested reader should see Appendix D for a more formal treatment of this.)

Notice that the actual rotation is by  $\theta$  and *not*  $\frac{\theta}{2}$ , which is why we introduced the half-angle in the first place. An intuitive way to think about this half-angle is that since the quaternion is multiplied by the vector twice, the angle is “applied” twice, so only half is needed for each side of the multiplication <sup>7</sup> The half-angle also means that we need to rotate a quaternion by multiples of  $4\pi$  to get it back to where it started, not  $2\pi$  like normal rotations.

Since the inverse divides by the magnitude, the subgroup of *unit* quaternions is enough to represent rotations. For this document, we will only use unit quaternions to represent rotations. This reduces the rotation formula to the fundamental formula for quaternion rotation:

$$\mathbf{y} = \mathbf{R}(\theta, \hat{\mathbf{n}}) = \hat{Q}\mathbf{x}\hat{Q}^*$$

where  $\mathbf{R}(\theta, \hat{\mathbf{n}})$  is the rotation matrix that rotates by  $\theta$  around  $\hat{\mathbf{n}}$ .

### Double covering

An important property of the rotation formula is that both  $\hat{Q}$  and its negation  $-\hat{Q}$  will produce the same rotation. This is an important fact to be remembered in algorithm design as we will see later, so we make it clear:

**Both a unit quaternion  $\hat{Q}$  and its negative  $-\hat{Q}$  represent the same rotation of a vector. This is called a *dual-valued* or *double-covering* representation.**

### Useful Rotation Formulae

Quaternions allow us to find the shortest rotation between two orientations (Euler’s theorem) trivially. If  $\hat{P}$  and  $\hat{Q}$  represent two orientations, then the product  $\hat{P}^*\hat{Q}$  gives us a quaternion that will rotate  $\hat{P}$  into  $\hat{Q}$ . Thus, we can intuitively think about multiplying one quaternion by the conjugate of the other as “subtraction,” though remembering that it is *not* commutative.

Similarly, the shortest rotation that takes one unit vector  $\hat{\mathbf{x}}$  into another  $\hat{\mathbf{y}}$  is simply found by the vector product  $(\hat{\mathbf{x}}^*\hat{\mathbf{y}})^{1/2}$ . In fact, any unit quaternion can be written as the product of two unit vectors in this way.

This product is actually the foundation for the deeper theory of *Clifford* or *geometric* algebras ([17, 41, 32]) of which quaternions are one example.

---

<sup>7</sup>Hanson also provides a mathematical description of why this half-angle creeps into the formula [36] — it results from a square root in the frame equations.

## Composition of Rotations

The composition of two rotations each represented as a unit quaternion is simply the product of the two quaternions. In other words, if we want a quaternion which represents first a rotation  $Q_1$  followed by a rotation  $Q_2$ , we need the quaternion  $Q_2Q_1$ . Notice that the composition order happens from right to left, as is the case with rotation matrices acting on column vectors. Indeed, it often helps to avoid problems with forgetting about non-commutivity to in fact think of quaternions as matrices, since most readers are already familiar with the non-commutivity of matrix multiplication.

## Summary of Quaternion Algebra

In the last section, we learned that:

- Quaternions extend complex numbers to four dimensions with many formula carrying over through analogy.
- Quaternions allow us to *divide* vectors as well as multiply them (unlike dot and cross).
- Unit quaternions allow us to simply represent rotations of vectors.
- Both a unit quaternion and its negative represent the same rotation.

### 3.3.3 Topological Structure of Unit Quaternions: Hypersphere $S^3$

Unit quaternions ( $\mathbb{H}$ ) are often represented computationally as unit vectors in  $\mathbb{R}^4$ . This representation is the surface of a hypersphere in 4 dimensions! This sphere is also known as  $S^3$ , since the surface has three degrees of internal freedom, though it is embedded in a four-dimensional space. When taking this geometric point of view, the negative of a quaternion is called its *antipode*. The fact that both a quaternion and its antipode refer to the same rotation is called *antipodal symmetry*.

Thinking of the unit quaternions as living on a hypersphere is the most useful property of the quaternions as it allows for the use of visualization and geometric reasoning for algorithm design and lets us not consider quaternions as a “black box.” Many calculations that would be difficult to do analytically in the quaternion algebra are potentially much simpler using geometric reasoning and hyperspherical trigonometry. Algorithm design can also use this geometric property as a starting point, using construction schemes on the sphere rather than the algebra directly. We shall use this fact throughout this document.

### 3.3.4 Exponential Map and Tangent Space

We introduced the exponential above in the polar form. This section will describe the exponential mapping and its inverse the logarithmic mapping in more detail since we will use it throughout our work, as do several other graphics researchers recently [52, 29, 54, 55]. The exponential and logarithmic maps will let us map vectors into unit quaternions and vice versa. We will see that this is related to the tangent space to the hypersphere and

that the log map can be used to locally linearize the quaternions in an analytic and invertible way.

### Definition

Any unit quaternion can be written as the exponential of a pure vector:

$$\hat{Q} = e^{\mathbf{x}} = e^{\frac{\theta}{2}\hat{\mathbf{n}}}$$

Likewise, we can define the logarithm of a quaternion as the inverse of the exponential:

$$\ln \hat{Q} = \ln e^{\frac{\theta}{2}\hat{\mathbf{n}}} = \frac{\theta}{2}\hat{\mathbf{n}}$$

By restricting the magnitude of the vector ( $\frac{\theta}{2}$ ) to the range  $[0, \pi]$ , we can get an almost unique mapping from the *solid ball* of radius  $\pi$  to a unit quaternion. In this case the origin will map to the identity element. Each point *inside* the ball represents a *unique* rotation. Notice, however, that *antipodal points* on the *surface* of the ball are identical rotations since a rotation around any axis by  $\pi$ , no matter what the sign of the axis, is the same rotation. This representation can be used directly, though care must be taken at the surface of the ball since it introduces a discontinuity (see [29] for more details on this). We expect this to be the case since, as we mentioned above, any three parameter representation (which the log vector is) must have a singularity somewhere.

Computationally, the most robust way to implement the quaternion exponential mapping is using the equation:

$$\mathbf{w} = 0 + \frac{\text{Vector}(q)}{\text{sinc}(\frac{\theta}{2})}$$

where

$$\frac{\theta}{2} = \arccos(\text{Scalar}(q))$$

and  $\text{Scalar}(q)$  is the scalar component of the quaternion,  $\text{Vector}(q)$  is the vector part, and  $\text{sinc}$  is the “sinc” function  $\sin(x)/x$  whose limit at  $x = 0$  exists.<sup>8</sup> This equation also makes the mapping very clear — the log is taken by dividing the vector part through by a (scalar) sinc function and zeroing the scalar part. It is simple to check if we use the polar form of the quaternion ( $\cos \frac{\theta}{2} + \hat{\mathbf{n}} \sin \frac{\theta}{2}$ ):

$$\mathbf{w} = 0 + \frac{\hat{\mathbf{n}} \sin \frac{\theta}{2}}{\text{sinc}(\frac{\theta}{2})}$$

Expanding  $\text{sinc} = \sin(x)/x$  gives

$$\mathbf{w} = 0 + \frac{\frac{\theta}{2}\hat{\mathbf{n}} \sin(\frac{\theta}{2})}{\sin(\frac{\theta}{2})}$$

---

<sup>8</sup>Care must be taken in an implementation to avoid divide by zero. We use a lookup table interpolation near  $x = 0$  and perform the division explicitly beyond this range.

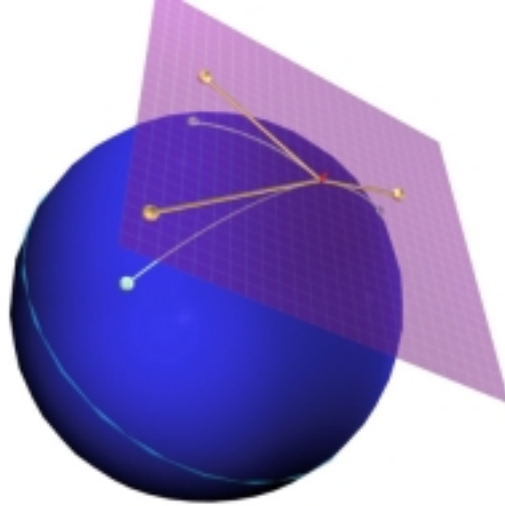


Figure 3-6: A depiction of the exponential map. Points in the tangent space are mapped onto the sphere by the exponential mapping and vice versa by the logarithmic map, its inverse.

and reduces to the desired

$$\mathbf{w} = \frac{\theta}{2} \hat{\mathbf{n}} .$$

This gives us a simple way to convert from axis-angle to unit quaternions and back, which as we saw above is important for metrics based on Euler's theorem.

### Tangent Space to $S^3$ : $\mathbf{T}S^3$

A powerful way of thinking about this mapping is that  $\mathbf{w}$  lives in the 3-dimensional tangent space at the identity of the quaternion hypersphere. This tangent space is denoted  $\tan 1S^3$ . Lack of a subscript will implicitly mean the tangent space is at the identity. Figure 3-6 depicts the mapping. Put succinctly:

**The logarithm maps a point on the sphere into a point in the tangent space at the identity.**

In order to map a unit quaternion  $\hat{Q}$  into the tangent space at some other location on the sphere  $\hat{P}$ , we simply rotate the sphere to align  $\hat{P}$  with the identity and then take the log:

$$\boxed{\ln_{\hat{P}}(\hat{Q}) = \ln(\hat{P}^* \hat{Q})}$$

Most of our algorithms will use this basic formula, so we box it to make it clear. The subscript denotes that we are taking the log at a different point than the identity.

Furthermore, this is also an invertible mapping, giving us:

$$\boxed{\exp_{\hat{P}}\left(\frac{\theta}{2}\hat{\mathbf{n}}\right) = \hat{P}e^{\frac{\theta}{2}\hat{\mathbf{n}}}}$$

The tangent space is  $\mathbb{R}^3$  and is in fact a vector space. Therefore, the exponential and log maps can serve as a local “linearization” (approximation) of the unit quaternion group, mapping unit quaternions into tangent vectors. These maps are also extremely related to the quaternion calculus and basic quaternion interpolator (slerp), described below. We discuss their relation to *Lie algebras* in Appendix D.

Tangent space mappings will allow us to more easily visualize and design interactive manipulation algorithms, such as Hanson’s “Rolling Ball” algorithms for manipulating quaternions [34, 37, 36].

### Properties

The exponential map has a few properties which we will leverage. First, it preserves the spherical distance from any point on the sphere to the identity. In other words, the magnitude of the log vector will be the same as the spherical distance from the identity to the mapped quaternion. This allows us to construct a simple metric between quaternions:

$$\boxed{\text{dist}(\hat{P}, \hat{Q}) = 2\|\ln(\hat{P}^*\hat{Q})\|}$$

Notice that this gives the natural metric between two quaternions ( $\theta$ ) directly!

Second, it preserves the angle made between two quaternions and the identity. In other words, if the identity is thought of as the north pole of the Earth, two lines of constant longitude emanating from the north pole will logmap to two vectors that have the same angle between them as the longitude lines make at the north pole.

Intuitively, these properties imply that a spherical circle centered around the tangent point on the sphere will map to a sphere ( $S^2$ ) in the tangent space. Likewise, ellipses map to ellipsoids. Squares, however, will become distorted. These effects are exactly what we see when we look down on the north pole of a globe as well.

Finally, we note that since the logmap is effectively a local linearization, it is best near the center of the map.

### 3.3.5 Basic Quaternion Calculus and Angular Velocity

This section will introduce the basic quaternion calculus formula which we will use in the design and understanding of some of our algorithms and explain its relation to the familiar instantaneous angular velocity in mechanics.

The time derivative of a unit quaternion  $\hat{Q}(t)$  is:

$$\boxed{\frac{d}{dt}\hat{Q}(t) = \dot{\hat{Q}} = \frac{1}{2}\hat{Q}(t)\omega'(t) = \frac{1}{2}\omega(t)\hat{Q}(t)}$$



where  $\omega \in \mathbb{R}^3$  is the angular velocity of the quaternion with respect to the basis frame (identity element) and  $\omega' \in \mathbb{R}^3$  is the local angular velocity in the frame at  $\hat{Q}$ . (We will not prove this here, but see [52, 53, 19].) Notice that angular velocity is a true vector quantity.

There are several points to mention. First, the factor of  $\frac{1}{2}$  handles the fact that the quaternion curve will move half as fast as the corresponding  $\text{SO}(3)$  curve due to the half angle in the rotation formula. Second, since the derivative is expressed in the quaternion algebra, it is a quaternion, although *not* unit itself.

Intuitively, a derivative at a point is a tangent vector at that point. Since our derivative is of a unit quaternion  $\hat{Q}(t)$ , it must be tangent to  $S^3$  at  $\hat{Q}$ . Since angular velocity is a pure vector in the quaternion algebra, it must have no component in the identity direction. Therefore, it is orthogonal to the real axis and can be thought to live in the tangent space at the identity (real axis). By then multiplying by the location  $\hat{Q}$ , we effectively “rotate” the local angular velocity to the tangent space at  $\hat{Q}$  so that the derivative is expressed in the inertial basis.

Another way to look at the derivative is to consider a fixed unit quaternion  $\hat{Q}_0$  exponentiated by time:

$$\hat{Q}(t) = \hat{Q}_0^t$$

which can be expressed as

$$\hat{Q}^t = e^{t \ln \hat{Q}_0}$$

and the derivative in time is then

$$\dot{\hat{Q}}(t) = \hat{Q}(t) \ln \hat{Q}_0$$

which is the differential equation for a constant angular velocity curve that passes through the identity at  $t = 0$  and  $\hat{Q}_0$  at  $t = 1$ .

Finally, we discuss numerical integration of quaternion ordinary differential equations (ODE) in Appendix C.

### 3.3.6 Interpolation, Slerp and Splines

Several interpolation techniques currently exist for doing interpolation on a sphere. An advantage of the quaternion representation is that these interpolation techniques, unlike the ones we saw above, are smooth and continuous over the entire sphere and do not exhibit anomalous singularities (gimbal lock).

The most important of these spherical interpolation techniques, introduced to the graphics community by Shoemake [73], is *slerp*, which is short for *spherical linear interpolation*. It can be defined in the quaternion algebra as:

$$\boxed{\text{slerp}(\hat{Q}_1, \hat{Q}_2, t) = \hat{Q}_1 (\hat{Q}_1^* \hat{Q}_2)^t}$$

Slerp is the hyperspherical version of the familiar convex sum in a vector space (often called *lerp*) and interpolates at constant angular velocity along the shortest path (a great circle) from  $p$  to  $q$  as  $t$  ranges from 0 to 1 at constant parametric speed. Slerp can be thought of

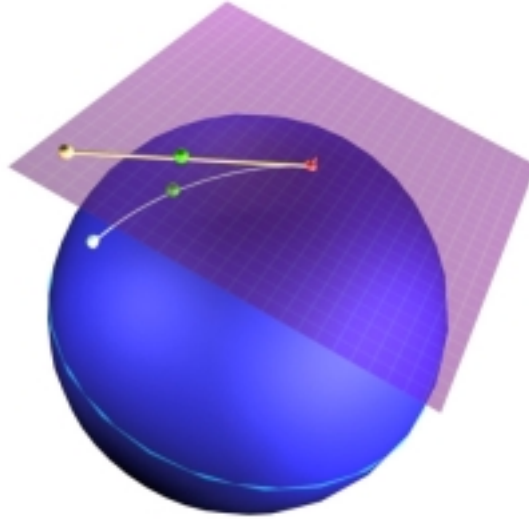


Figure 3-7: A depiction of slerp. The two examples are interpolated at constant angular velocity as the parameter changes with constant speed. The exponential portion of slerp can be interpreted with the exponential mapping with respect to one example. In this view, a constant speed line in the tangent space will map to a constant angular velocity curve on the sphere.

as “walking along the equator at constant speed.” Figure 3-7 depicts slerp. As the tangent point is moved at constant speed in the tangent space, its mapping on the sphere moves at constant angular speed.

Care must be taken in the use of slerp due to the antipodal symmetry of unit quaternions when representing rotations in  $SO(3)$ . Since both  $\hat{Q}$  and  $-\hat{Q}$  refer to the same rotation, we need to find the *shortest path in  $SO(3)$* . To handle this, the simple heuristic that both quaternions must be on the same side of the sphere is used. If it were on the other side, the geodesic path would appear to “take the long way around” to get to the other orientation.

Given that we have a spherical analogue of the lerp geodesic, a spline construction scheme can be used to generate Catmull-Rom, Cubic Hermite, and other splines on  $S^3$ . The interested reader should see Schlag’s *Graphics Gem* [72] for more information. A useful source is [52] who show a general construction scheme for analytic spline using the exponential map. Our work is similar to this, except we will be performing multi-variate interpolation rather than just temporal.

### 3.3.7 Advantages

So what do we get for accepting this unfamiliar object? Nothing short of the grail: lack of a singularity in the representation. This property follows from a theorem in topology which is amusingly called the Hairy Ball Theorem [73] (our discussion closely follows Shoemake’s description in the reference). It states that it is only possible to create a continuous tangent

vector field (which can be thought of as “hair”) on a sphere of odd surface dimension (though possibly embedded in an even dimensional space).

This theorem is obvious for a one-dimensional sphere, or circle ( $S^1$ ). The “hair” can clearly be “combed” in one direction around the circle without discontinuities. It is also possible on  $S^3$ , where the quaternions live. Surprisingly, however, it is *not* possible on a sphere of two-dimensions ( $S^2$ ), which are the spheres we are familiar with as balls and balloons and directions in space. Put simply, you cannot comb the hairs on a tennis ball without getting a cowlick or “bald spot” somewhere on the surface. In other words, there *must* be a discontinuity in hair direction or a spot where the tangent must vanish entirely to zero (at a whorl, say). But on a hypertennis ball (four-dimensional), it would be possible to comb it without such a problem.

This theorem implies that no matter how a point is moving continuously around the sphere (in our case representing a smooth change in the orientation of an object in  $\mathbb{R}^3$ ), there is *no* spot where we will get “stuck” on a singularity trying to move in a direction we cannot (over a cowlick or whorl) as in a minimal three-coordinate representation. Put succinctly:

**Quaternions do not suffer from gimbal lock or coordinate singularities.**

Another added advantage for numerical calculations is that quaternion multiplication uses fewer multiplies than matrix multiplication, making it more computationally efficient for composition. Another bonus is that numerical drift away from unit magnitude is easily removed by renormalizing the quaternion in the obvious manner of dividing by the magnitude. Some researchers instead use the entire quaternion group (of any magnitude) and perform the normalization only when calculating a rotation triple product on a vector [26].

The full group is useful in designing algorithms as well. For example, Shoemake suggests taking the square root of a quaternion as:

$$\hat{Q}^{\frac{1}{2}} = \frac{(1 + \hat{Q})}{\|1 + \hat{Q}\|}$$

which uses the addition operator of the full quaternion group rather than by the more obvious application of the exponential form (we describe the natural log of a quaternion below):

$$\hat{Q}^{\frac{1}{2}} = e^{\frac{1}{2} \ln(\hat{Q})}$$

### 3.3.8 Disadvantages

The main disadvantage is that quaternions use mathematics that is less familiar to most people, so they require a little extra work to understand and work with. The worst drawback is that since quaternions live on a sphere, one *cannot* use the Euclidean vector space interpolation methods such as B-splines without modification. These techniques need to

terminology	field	usual denotation
quaternion	computer graphics	subgroup of $\text{SO}(4)$
Euler parameters	mech/aero engineering	subgroup of $\text{SO}(4)$
spin group	quantum physics	$\text{SU}(2)$

be reformulated to work on a sphere. Some of these methods have been extended already (see, for example, [52, 4, 67, 26, 73, 19]).

It is important to remember that this is *not* just a drawback of quaternions, but is in fact *inherent in the nature of the rotation group itself*, as described above. Since we are forced to deal with this fact anyway, quaternions allow us to use spherical geometric reasoning in algorithm construction and visualization.

### 3.3.9 Recommended, Related and Other reading

Quaternions masquerade under many different names in the literature as different fields re-discovered the need for them. For example, the quaternions are isomorphic to the special unitary 2 by 2 complex matrices,  $\text{SU}(2)$ , which is a spin group in quantum physics. A lot of intuition about quaternions can therefore be gained by learning about  $\text{SU}(2)$ . Artin's *Algebra* [2] gives a great introduction to  $\text{SU}(2)$  and the relation of the algebra to  $S^3$ . Mechanical and aerospace engineering often use the term *Euler parameters* for quaternions, which is unfortunate since they are very different from the Euler angles. Many fields use the fact that a subgroup of  $\text{SO}(4)$  can model the quaternion algebra linearly. Table 3.3.9 summarizes some of the quaternion aliases to help the reader in a keyword search.

Many books exist which are helpful in learning about the classical groups, such as the rotation group  $\text{SO}(3)$ , as well as the mathematics which is useful for handling quaternions.

A *great* reference book which the author wishes he had five years ago is the recent book by Gallier [23] which covers affine spaces, homogenous coordinates, Lie group and algebras and many other geometric ideas with an eye toward computational issues rather than pure mathematics.

There are many useful articles in the *Graphics Gems* series on useful techniques for quaternions, from random rotations to 2D input devices, to theory: [56, 28, 74, 72, 34, 27, 61, 75, 35, 76, 77, 78, 37].

McCarthy's introduction to kinematics is also useful for understanding clifford algebras, Plucker coordinates, and other structures which are useful in kinematics [59].

Kuipers has a great introductory book on quaternions [53]. Dam's tech report [19] contains a clear presentation of rotation representation and derives some of the properties of slerp and the cubic Hermite (squad), as well as introducing a new interpolation scheme (spring). They also correct a bug in Shoemake's derivation of squad.

For the more mathematically inclined, a recent book explores the calculus of quaternion and Clifford algebras with an eye towards application (such as a quaternionic neural net solution), but it is not for uninitiated! [32]. Sattinger and Weaver's classic introduction to Lie groups was one of the more useful to the author for understanding representations of

rotation though it is mostly of use to quantum physicists [71]. Weyl [88] is also a classic on group theory. A great introduction to  $SU(2)$  and the exponential mapping can be found in [2].

For those unfamiliar with tensor and vector calculus, [91] is a useful start. Saff [70] is a reasonable introduction to complex analysis in general, but not quaternions. Bartels *et al* [5] is a great source for spline construction and interpolation theory. Finally, Nikravesh [62] offers an useful discussion of using quaternions in a numerical simulation of rigid body dynamics, including issues with integrating quaternion matrix equations.

A good book for learning differential geometry is Burke's [14]. *Gravitation*, a big black tome, has useful sections on the spinor representation of rotation as well as some great visualization tools and insights into the nature of curved spaces, tensor calculus, differential geometry, and the rotation group.

Finally, Geometric Algebra is a superset of the quaternions which is becoming popular in many fields as it pulls together the representation of physical groups into a unified framework to promote sharing of ideas. Many excellent papers on quaternion techniques ranging from quaternion wavelets to quaternion neural nets and applications ranging from quantum, Kalman filters, computer vision, robotics, and optics can be found in [17]. The SIGGRAPH community recently began looking at them in a course as well [68].

### 3.4 Quaternion Algebra and Geometry Summary

We summarize the main formulas and their geometric interpretation in terms of spheres and great circles in the following tables. The graphics are meant to show how quaternion operations are very related to geometric operations on spheres in the same way that unit complex numbers amount to operations on a unit circle. In some of the images, the axis coming out of the page is  $\hat{n}$ , showing that we are taking an orthogonal projection of the one-parameter subgroup orthogonal to  $\hat{n}$ . In others, we explicitly show the *real* axis along the horizontal (with the group identity, 1) and the entire vector imaginary portion collapsed abstractly into the vertical axis as if it were a complex number <sup>9</sup>.

---

<sup>9</sup>In fact, it actually is. The Clifford algebra pseudoscalar  $I$  times a vector acts much like the complex unit  $i$  times a scalar (see [17])

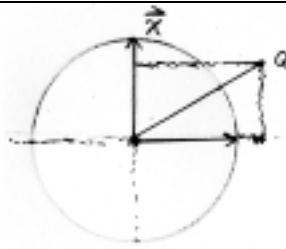
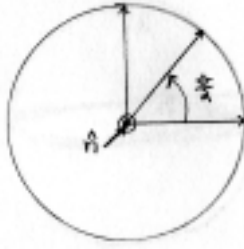
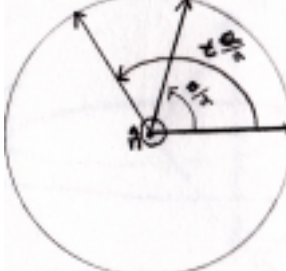
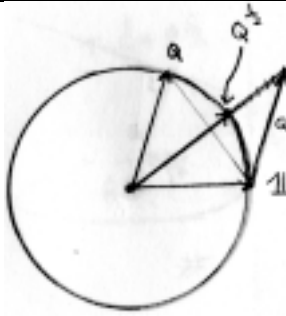

Name	Formula	Geometry
Rectangular	$w + x\hat{\mathbf{i}} + y\hat{\mathbf{j}} + z\hat{\mathbf{k}} = w + \mathbf{x}$	
Polar/Exponential	$e^{\frac{\theta}{2}\hat{\mathbf{n}}} = \cos(\frac{\theta}{2}) + \sin(\frac{\theta}{2})\hat{\mathbf{n}}$	
deMoivre (power)	$Q^t = e^{t\frac{\theta}{2}\hat{\mathbf{n}}} = \cos(t\frac{\theta}{2}) + \sin(t\frac{\theta}{2})\hat{\mathbf{n}}$	
Sqrt (geometric)	$\frac{(1+Q)}{\ (1+Q)\ }$	
Conjugate	$Q^* = w - \mathbf{x} = e^{-\frac{\theta}{2}\hat{\mathbf{n}}}$	

Table 3.1: Quaternion Algebra Summary

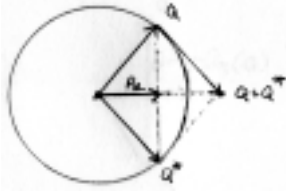
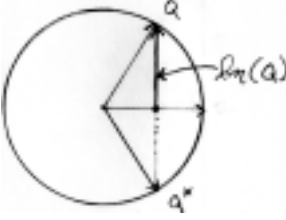
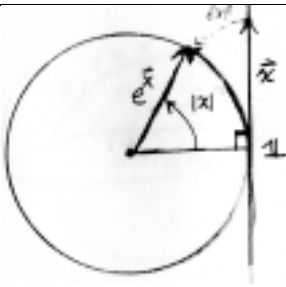
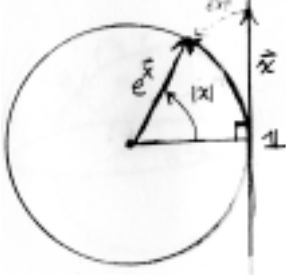
Name	Formula	Geometry
Scalar (real)	$\Re(Q) = \frac{Q+Q^*}{2}$	
Vector (imaginary)	$\Im(Q) = \frac{Q-Q^*}{2}$	
Modulus (magnitude)	$(QQ^*)^{\frac{1}{2}} = (Q^*Q)^{\frac{1}{2}}$	
Expmap	$e^{\mathbf{x}} = \cos(\ \mathbf{x}\ ) + \text{sinc}(\ \mathbf{x}\ )\mathbf{x}$	
Logmap	$\ln(Q) = 0 + \frac{1}{\text{sinc} \cos^{-1}(\Re(Q))} \Im(Q)$	

Table 3.2: Quaternion Algebra Summary II

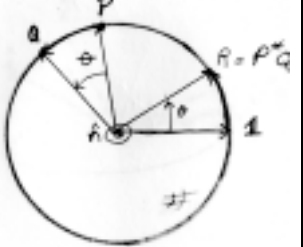
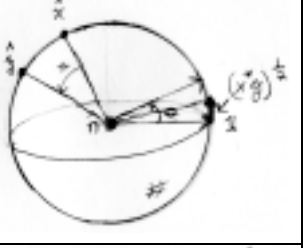
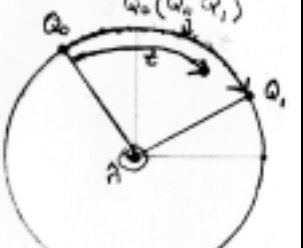
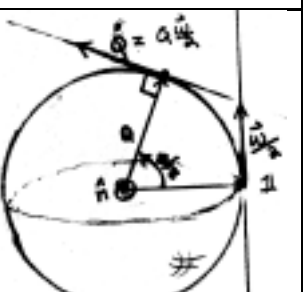
Name	Formula	Geometry
Rotation Difference	$R_{P,Q} = (P^*Q)$	
Vector Difference	$R_{\hat{x},\hat{y}} = (\hat{x}^*\hat{y})^{\frac{1}{2}}$	
Slerp Geodesic slerp( $Q_0, Q_1, t$ )	$Q_0(Q_0^*Q_1)^t = Q_0e^{t\ln(Q_0^*Q_1)}$	
Derivative	$\dot{Q} = Q(t) \ln(Q(t)) = \frac{1}{2}Q\omega' = \frac{1}{2}\omega Q$	
Change of basis invariant	$R^*e^{\frac{\theta}{2}\hat{n}}R = e^{\frac{\theta}{2}R^*\hat{n}R}$	N/A

Table 3.3: Quaternion Algebra Summary III



# Chapter 4

## Statistical Kinematic Joint Models

The last chapter introduced the mathematical issues with representing orientations and rotations, which are the essential components to modeling character joint motion. It introduced several representations, focusing on the quaternion, which we saw had a lot of nice properties in terms of efficiency, interpolation and metrics.

This chapter will motivate an abstract statistical kinematic joint model consisting of the following parts:

- Joint equilibrium (center) point
- Joint motion limits around the center
- Joint motion probabilistic model around the center.

We then give a set of properties we would like the joint model to possess based on our overall problem of learning a motion manifold from examples. These are:

- Scale-Invariance
- Convexity
- Constraints
- Singularity-Free
- Intuitive
- Fast and Efficient
- Common currency
- Able to Interpolate

We then compare the following choices of representation against these properties:

- Special Orthogonal Matrices ( $SO(3)$ )

- Euler angles ( $S^1 \times S^1 \times S^1$ ).
- Quaternions ( $S^3$ )

We conclude that the best representation of a joint uses a quaternion representation as follows:

- Use a quaternion for representing the *current* rotation of the joint with respect to the bone it is connected to.
- Use a quaternion statistical model learned over the *entire* corpus of animation to model joint motion probabilistic behavior and limits.

The rest of the chapter will proceed as follows:

**Section 4.1** motivates the need for a statistical model of joint motion.

**Section 4.2** motivates the use of quaternions as the representation of orientation of a joint for a statistical model by comparing it against Euler angles and  $SO(3)$  matrices according to a set of desirable criteria for a statistical model.

**Section 4.3** summarizes the argument for the use of a quaternion representation of joints.

## 4.1 Motivation for Statistical Kinematic Model

Consider the rotation of your wrist. It tends to move around some average center point, approximately where it lines up with your forearm. It can rotate left-right over a small range and up/down over a wider range. It can roll (twist) around the axis of the forearm as well. The twist degree of freedom is often considered to be at the wrist itself in computational skeletal models, giving the wrist three degrees of freedom, even though clearly through the action of the tendons is occurring along the forearm. Other models place the twist degree of freedom at the *elbow*, which most people consider to be one degree of freedom, giving the wrist and the elbow each 2 DOF to model the 4 DOF of the wrist to upper arm chain. Clearly, all of these models are abstract descriptions of what is actually going on in the underlying muscles and tendons. Actual organic joints can be quite complex, exhibiting disturbing phenomena such as hysteresis, or path memory. We will ignore a lot of these lower level effects and focus on the common observable rotations and how they move bones.

On the other hand, we cannot *entirely* ignore this issue. A very interesting thing we noticed is that in switching from a *rigid mesh* for virtual characters (in which joint rotations rotate fixed chunks of geometry connected to the bones) to a *skinning* approach (where a seamless mesh deforms in a weighted manner according to the bone location) can change how an animator chooses to make this decision! Consider the wrist example again: as the wrist twists, the skin of the forearm twists. To an animator, this means the bone inside is twisting, which happens at the elbow. If they apply twist at the wrist instead, the forearm won't move as easily. We were surprised by early statistical analyses which showed that

elbows were 2 DOF! Therefore, by allowing a model that can find inherent degrees of freedom of the joint we can avoid imposing artistic limits on the animator by forcing a certain rotation structure up front that the animator must adhere to. Instead we argue for letting the animator do what is most natural and then having us extract the proper data later, as we argued for earlier.

Another interesting thing to notice about organic joints is that since they have tendons connecting them to the bones, they in general cannot spin around in any direction by a full 360 degrees like Regan's head in *The Exorcist* or a robotic joint. Consider your wrist again. Wiggle it around randomly in all directions.<sup>1</sup> Not only does it have a central location, but the edge of the constraint boundary (joint limit in any direction) is fairly smooth over most of the range, meaning that the boundary is close to *convex*.

So what can we gather from this simple empirical study? We will argue that to model an organic joint like a wrist, the minimal set of statistics we will need are:

**Mean, Center, Average, Equilibrium** The center of the joint as a coordinate frame. This can be thought of as some average “minimal energy” location over all possible muscle equilibria points when considered as springs.

**Convex Joint Limits** The joint will tend to have a limited amount of rotation away from the center of the joint. This boundary seems to be fairly convex, lacking “corners” where we get stuck<sup>2</sup>

**Variances around the mean** Joint limits and variances are clearly very related, and we will use the same structure for them. Again, we might need a more complicated *mixture model* to handle joints in certain cases. Since kernel-based mixture models are based on sums of these second order statistics models (which serve as local kernels), having a clear single second-order statistical kernel model is required first.

In summary, we will use the following simplifying assumptions in our statistical model:

- Joint data is local is fairly local in the rotation group and therefore single joint statistics can be modeled with a mean and variances about the mean.
- Joint limits can be expressed in terms of limits in the principal variance directions and form a convex constraint surface.
- Joints do not spin all the way around in any direction<sup>3</sup>, therefore the joint will live in a closed subspace of the rotation group.

The next section will discuss joint properties in a little more detail and argue that quaternions are the best way to model this kind of joint data.

---

<sup>1</sup>Schaal at USC calls this “motor babbling”. In yoga class we call it “vibration.” The effect is the same: to force the joint into all its configurations. I imagine this is what a baby is doing when it wiggles randomly.

<sup>2</sup>These do exist, however. The shoulder joint exhibits hysteresis, or path memory, or posture. It is possible to take a certain path into a configuration where one can't get back out in certain directions. Thus, real joints *do* exhibit singularities in certain places. Since we are worried about modeling expressive motion as quickly as possible, we will ignore these strange cases which rarely happen on average. In modeling a human being performing yoga, we will need to have a better statistical model.

<sup>3</sup>Excluding the special top level “root node” discussed in the next chapter which lets the character's center of mass rotate and tumble arbitrarily in space.

## 4.2 Motivation for a Quaternion Representation of Character Joints

Why use a quaternion statistical model rather than something else for modeling the statistical kinematics of organic joints? We argue that an appropriate joint model which leverages animator knowledge through analysis and synthesis of examples should have the following properties:

- Scale-Invariance
- Simple, Fast, Convex Range Constraints
- Singularity-Free
- Intuitive
- Fast and Efficient
- Common currency
- Able to interpolate

Furthermore, there are three clear representation alternatives out there now:

- Special Orthogonal Matrices
- Euler angles
- Quaternions

First, we will describe the properties we feel the joint representation needs. Then we will address these properties for each of these representations.

### 4.2.1 Properties

**Scale invariance** First, the model should ideally allow *scale-invariance*. This means that in a distance metric between two joint rotations, we would like to have the distance in dimensionless units. For example, a wrist joint rotates further in certain directions than others. We would like to consider the relative distance of the joint rotation with respect to its full range, for example as a percentage, or weight, of rotation in each direction. Scale-invariance is often required by statistical algorithms.

**Joint Range Limits** A model should allow for hard joint limits. Extrapolation techniques or inverse kinematics techniques that will go beyond the example data will need a way to be constrained to avoid unnatural poses, such as an elbow going backwards. Hard constraints are often at odds with convexity, although if we use a density with ellipsoidal isocontours we can choose a particular contour (standard deviation) as a hard constraint surface. This choice allows us the benefits of a hard surface while not losing the convex continuous gradient of the underlying density. These constraints should be easy to learn from data. Since we want to model arbitrary organic characters, we cannot make assumptions from biometric data — we need to collect our own. Having to tweak the joint constraints on a model by hand is not an effective way to leverage an animator’s skills. Additionally, since constraint checks often occur inside intensive iterations, they need to be as fast as possible. Finally, we would like the limit model to be convex, or form a smooth boundary. This will be advantageous for many optimization techniques that can get “stuck” on the corners of a non-convex constraint boundary.

**Singularity-free** A model should not have *coordinate singularities*. A coordinate singularity is where the representation goes to zero (or infinity) due to some mathematical reason which has nothing to do with the thing modeled (joint rotation). For example, a gimbal (see Figure 3-4) can be used physically to describe rotations. Unfortunately, there will always exist some configuration where the gimbal gets stuck due to a coordinate singularity (loss of a degree of freedom) as we saw in Chapter 3. Clearly, rotations themselves do not lock up — tumbling rocks don’t suddenly lock up at some orientation. In fact, lack of singularity will be the best reason to choose quaternions. Often naive users of Euler angles will assume that “glitches” in interpolation and integration of Euler angles are bugs in their code, whereas in fact they are an inherent mathematical flaw which needs to be addressed properly.

**Intuitive** Ideally, we would like our representation to be as intuitive as possible. Since we are representing something about which people have a lifetime of intuition — rotation of joints — we would like our representation to be as close to the geometry as possible, with nothing extra. The proper representation will allow an intuitive understanding of and increased ability to design appropriate and efficient algorithms for handling rotation computationally.

**Efficiency** Since we are modeling interactive characters, we need to make the calculation of motion as fast and small as possible without sacrificing mathematical robustness or simplicity. Euler’s theorem gives us a target for complexity: 3 parameters is a minimal representation. Since speed is much more important for maintaining the illusion of life in a character, time efficiency will be chosen over space when required.

**Common Currency** Related to efficiency, we would like a unified common currency for describing rotations to avoid the numerical problems and computational overhead with converting between representations in algorithms. Also, this will let us blend the result of different algorithms in the same manner irregardless of how the individual algorithms

calculate their answers. Since we need a common currency, we should try to get the best one from a mathematical and computational point of view.

**Interpolation/Extrapolation** Finally, we would like the representation to allow for simple, very fast interpolation. Ideally, it should allow for extrapolation as well, which will let us leverage the animator better by requiring less examples from him to “patch up” problems caused by the representation.

Next, we compare the following common choices of representation with respect to these properties:

- Special Orthogonal Matrices  $\text{SO}(3)$
- Euler angles
- Quaternions

## 4.2.2 Special Orthogonal Matrices $\text{SO}(3)$

We discussed coordinate matrices for  $\text{SO}(3)$  in the previous chapter. Recall that a coordinate matrix’s columns form a basis for the rotated joint with respect to the frame it is measured with respect to.

**Scale-Invariance** Metrics on coordinate matrices are well-known (see [59]). These treat the matrix as a large vector, however, which makes it harder to see how the matrix metric relates to the rotation. However, much of statistics is based in the linear algebra so much is known about finding principal axes using singular value techniques (see, for example, Therrien [83]).

**Constraints** Joint constraints are not obvious in  $\text{SO}(3)$ . Since it is not a vector space, we cannot factor the space into orthogonal components and constrain in each direction independently. We need to consider the entire representation simultaneously, coupling and all. How to do this is not clear since there are already constraints in the matrix to keep it special orthogonal. In practice, joint constraints are usually encoded as ranges on an Euler parameterization of the matrix.

**Singularity-Free**  $\text{SO}(3)$  is singularity free. Only three degrees of freedom are needed to specify a rotation. A  $3 \times 3$  has 9 entries with 6 constraints to maintain orthogonality and positive determinant.

**Intuitive** Matrices are familiar, but I would not call them immediately intuitive. They look at how *coordinates* change under the action of a rotation, rather than on the geometric, coordinate-free invariants of the motion. On the other hand, we can think of them as being a coordinate frame in terms of three orthogonal axes in the columns. For this reason, matrices can be intuitive for entering or extracting a rotation directly in terms of the effect of the rotation on each basis element.

**Efficiency** Matrices are often quite fast because they are usually implemented in hardware. In software, it is clear that we can do better since there is much redundancy in the representation due to the six constraints. This redundancy gets effectively squared when we compose rotations through matrix multiplication. Also, numerical roundoff issues require us to renormalize the matrices to avoid shearing effects, which involves invoking Gram-Schmidt or some other matrix renormalizer like polar decomposition. Clearly, a representation which had fewer redundancies and constraints would be more computationally efficient.

**Common currency** Since  $\text{SO}(3)$  is in fact the group we are seeking to represent, a coordinate matrix is a natural common currency. Matrices are the most popular common currency in many graphics systems. Spectral methods allow us to extract the translational and rotational parts of the matrix if needed, so we can convert to other representations, although these methods involve eigenvector calculations which can be expensive.

**Interpolation** It is not clear how to blend  $n$  matrices in  $\text{SO}(3)$  into a weighted average that is still in  $\text{SO}(3)$ . Adding them with the weights linearly then renormalizing in general will *not* work since they are not a vector space, as we saw. The 6 constraints make interpolation quite tricky, though integration is well-known for  $\text{SO}(3)$ .

To summarize, matrices are common, familiar, well-understood and intuitive in some situations. Mathematically, they are in some sense the group we want to represent computationally since they map one-to-one onto rotations of rigid bodies. Unfortunately, the redundancies in the representation make it computationally sub-optimal as well as difficult to use in geometric algorithms, since multiple coupled constraints needed to be handled simultaneously.

### 4.2.3 Euler Angles

Euler angles allow a triple of three real numbers to represent a rotation, so they are minimal in the sense of having the minimal degrees of freedom. To be used in practice computationally, however, they must be turned into  $\text{SO}(3)$  matrices, making them susceptible to all the arguments for and against  $\text{SO}(3)$ .

**Scale-Invariance** Since Euler angles “factor” the 3 rotational degrees of freedom into orthogonal axis directions, it is fairly easy to divide out by the ranges in each direction. However, this naive approach, while seeming to imply scale-invariance, will not be since it ignores the coupling of the components and the underlying natural metric on rotations. In other words, if we arrange Euler angles into a vector in  $\mathbb{R}^3$  and use the standard Euclidean metric on them, we will get very ill-behaved metric properties globally, though locally they appear reasonable.

**Constraints** Hard joint limits on Euler angles have been around for a long time and are commonly used in kinematic and physics engines. Even for algorithms that represent rotations differently, say with an  $\text{SO}(3)$  matrix or quaternion, rotation constraints

are often added by converting to an Euler angle set, applying the constraints, then mapping back into a matrix. This is a terrible idea for several reasons. First, there is computational overhead in the trigonometric calls and matrix multiplies. Secondly, extracting Euler angles from a rotation matrix is in general ill-posed numerically. Finally, the constraint boundary formed by three clamped Euler angles will *not* be convex when mapped into the rotation group. To avoid these “corners,” expensive non-linear programming methods often get used to add constraints to IK algorithms [3]. Unfortunately, constraining Euler angles to create joint limits is standard practice.

**Singularity-Free** The mapping from Euler angles into rotation matrices in  $SO(3)$  has a singularity as we saw in the last chapter. Gimbal lock is illustrated in Figure 3-4. There is no way around this.

**Intuitive** Euler angles have been around so long since they are intuitive to explain to animators and programmers. There is no free lunch here, however. The first time the animator encounters gimbal lock when trying to specify an orientation all intuition is gone. Furthermore, since there are 12 different choices of Euler set and many wildly varying conventions<sup>4</sup> it often takes a lot of trial and error to figure out which set data came from unless this information is provided. The main reason Euler angles are not intuitive is that they try to ignore the coupling between the axes in a global way (by choosing fixed axes), which is not possible.

**Efficiency** Euler angles are maximally efficient in space since there are three parameters. Unfortunately, since there is no simple formula for composing two rotations represented as an Euler angle set (no algebra), we need to convert to matrices, multiply them out, then extract the angles again if we want to compose several rotations. This is computationally inefficient. Furthermore, the singularity issues with Euler angles often require extra computation to look for singular states and avoid them. Avoiding them entirely in the representation seems like a much better idea.

**Common currency** Euler angles are a very poor choice of common currency due to the fact that there are 12 sets and they contain an inherent singularity which others may or may not handle properly.

**Interpolation** Often interpolation is done naively with Euler angles since they appear to be vectors. As we have noted, the singularities cause problems if an interpolation passes near it. Also, linear interpolations between two Euler angle vectors can produce wildly varying curves in the actual rotation group.

To summarize, although Euler angles naively appear to be memory-efficient, intuitive, and allow us to use standard vector space linear algebra techniques, they suffer from theoretical issues that outweigh any potential advantage. Furthermore, they suffer from computational issues both from these singularities and from the fact that they are converted into matrices anyway, eliminating much of the potential speed benefits. Therefore, Euler angles are a poor choice of representation.

---

<sup>4</sup>Almost every book I picked up disagreed on the conventions and some on the definitions for what an Euler angle set was.



## 4.2.4 Quaternions

Strangely, quaternions pre-date these other representations, but lost out historically. Recently, they are coming back with increased popularity in many fields under the field of Geometric (Clifford) algebra. We argue that quaternions are an optimal trade-off of computational efficiency, mathematical elegance, and ultimately for correct intuition about rotation groups, though at first they are very nonintuitive since they are so unfamiliar.

**Scale-Invariance** Quaternions are very close to the inherent group metric between rotations, which we want to represent. In fact, geodesics (great circles) on the quaternion hypersphere are the shortest paths between rotations (in  $\text{SO}(3)$ ). Therefore, our metrics have the best chance of being scale-invariant. The big problem is that in general rotation ranges (and therefore the effective scale) are not as easy to think about as in a vector space due to the coupling between the quaternion components. Statistical methods for scale-invariance on curved manifolds are of recent research interest, but much of the existing theoretical statistics work for manifolds is very opaque and too general for our problem.

**Constraints** Unfortunately, there was no clear way to do joint range constraints on a quaternion representation in the literature when we began this work. Most applications convert the quaternions to a matrix, then to an Euler set, then back again, which is horribly inefficient. Therefore, it was an open question for us to model joint ranges and also 1 and 2 DOF joints with a quaternion. Since then, several similar approaches have appeared, which we discuss in Chapter 11. As for constraints on the representation, our quaternions must stay unitary. Since they are easily represented as unit vectors in  $\mathbb{R}^4$ , we can simply divide by their magnitude to renormalize numerical drift. This is much easier than renormalizing a rotation matrix with 6 constraints to satisfy, rather than one. Quaternions live on a hypersphere, which is a convex space. If we choose our statistical model and constraint surface to be a smooth contour on this surface, we will have a convex constraint model. This geometric isomorphism between  $S^3$  and the quaternions will therefore be a useful visualization tool for us.

**Singularity-Free** By far the main advantage of quaternions is that they represent rotations in a singularity-free manner. Integration and interpolation can proceed on the unit quaternion sphere without fear of gimbal lock or degrees of freedom vanishing. Therefore, we can avoid extra machinery for looking for and handling these problems.

**Intuitive** Quaternions are not that intuitive to most people when they see them. One reason is that quaternions are an example of spherical geometry, which is non-Euclidean and therefore probably not taught in a standard engineering curriculum. Furthermore, it is 4-dimensional, making it much harder to visualize for many people. On the other hand, quaternions are very close to Euler's theorem, allowing us to view the action of a quaternion by looking at the logarithm, as we saw in the last chapter. Since the log is in  $\mathbb{R}^3$ , this gives us a visualization tool for visualizing quaternions [36]. Quaternions are also non-intuitive due to the half-angle that appears — they represent rotation as

$\frac{\theta}{2}$ , not  $\theta$ . This is due to the *antipodal equivalence* of quaternions for representing real rotations — the quaternion  $\hat{Q}$  and  $-\hat{Q}$  refer to the exact same rotation of  $\mathbb{R}^3$ . Therefore, the *sign* of the quaternion does not matter. Computationally, we need to handle this symmetry. This is one of the biggest issues with errors in quaternion algorithms.

**Efficiency** Quaternions represent rotations with four numbers, which is only one more than is required. Since the extra parameter gives us freedom from singularities, however, the extra floating point number is well worth it. Also, quaternion multiplication has less multiplies than matrix multiplication, so composition of rotations is efficient.

**Common currency** Quaternions are useful as a common currency for rotation for all the above reasons except intuition, which is becoming less of an issue every year. They are a better common currency than a matrix from an efficiency standpoint. The main reason they are the best choice of common currency is that there is an algebra, so we need not convert to other representations until we need to render. This is a major advantage.

**Interpolation** As we saw in Chapter 3, quaternions offer us a simple interpolator, *slerp*, which can be used to construct spline families through geometric construction [72]. The lack of singularities means our interpolator will be easier to implement. Finally, the existence of one-parameter subgroups (great circles) that can be parameterized by the exponential map will be useful for good extrapolation behavior, as we will see in Chapter 7.

Due to the geometric significance, efficiency, and robustness of the quaternion representation, we chose to use it as our joint model. This led to several outstanding problems we needed to address which we had not seen in the interactive computer graphics or robotics literature previously. These were:

- Weighted blend of  $n$  unit quaternion examples.
- Joint limits with quaternions
- Statistical probability densities for quaternion data
- Inverse kinematics with joint limits and quaternion representation

In the past few years since we began this work, interest in this area has been growing and several researchers have begun to look at these issues as well. We cover these in Chapter 11.

## 4.3 Summary of Statistical Kinematic Model Motivation

In this chapter, we argued for a quaternion representation for our statistical example-based model for joints. We argued that quaternions were the best choice for a statistical model representation for the following reasons:

- Minimal singularity-free representation
- Computationally efficient
- Potentially more intuitive (closer to geometry) than any other method.

The first two ideas seem fairly clear after our discussion. The latter we hope to argue for and demonstrate in this work.

This concludes Part One: Imaginary of this dissertation. The Part Two: Real portion of this thesis will show how to use these abstract concepts and ideas to actually look at real data and characters. We hope to validate the arguments we made in Part One through examples.

The next chapter will introduce the skeletal articulated figure model common to most animation packages and engines and show how to implement it with quaternions. We will show how to use quaternions to model the *pose* of a character as a tuple of  $n$  quaternions, one for each joint, and various operations such as distance metrics on pose.

## **Part II**

### **Real**

# Chapter 5

## Quaternions for Skeletal Animation

As we demonstrated in the last two chapters, quaternions are an efficient and non-singular representation for rotations in three-dimensional space. So how do we use them for modeling a character's joints?

This chapter will introduce the standard articulated skeletal model for rigid-body character animation, using quaternions as the representation for rotation of joints. We will also introduce terminology, concepts and formal notation that will be used throughout the rest of this document for describing the kinematic configuration of a synthetic character.

Specifically, the reader will learn the following in this chapter:

- Joint-Bone Skeletal Model
- The Quaternion Joint Model
- Forward Kinematic Equations with Quaternions
- Postures, Posture Metrics and Animations

The chapter will proceed as follows:

**Section 5.1** introduces the articulated skeletal model which consists of a tree structure of bones connected by joints. We also discuss simplifying assumptions we make.

**Section 5.2** introduces coordinate frame terminology and notation. It then presents the concept of open kinematic chains through a skeleton and the forward kinematic calculations. We show how coordinate frames along a kinematic chain can be computed efficiently in terms of quaternions and vectors rather than using the traditional 4x4 homogenous matrix representation.

**Section 5.3** will introduce *postures* (tuples of unit quaternions) for representing the joint rotational degrees of freedom of a figure and how to perform metrics on postures in the quaternion algebra. It will also define the concept of *motion* (time-derivative of posture) and describe the form that example data describing motion (animation) that we assume.

**Section 5.4** summarizes the main points of the chapter.



Figure 5-1: The bones (colored solid) which are animated underly the mesh (grey transparent) skin. Each bone rotates with respect to its parent by a 3D rotation, making a hierarchical skeletal model with the pelvis at the root.

---

## 5.1 Articulated Skeletal Model

Rigid skeletal models of characters are standard these days in animation packages and videogame engines. A *skeletal model* consists of a hierarchy of rigid *bones* (or *links*) which are connected together by *joints* (see Figure 1-2). Bones are rigid bodies in the sense that they cannot bend or change length, so can be described with just a single parameter – length. Joints connect bones together and allow them to move with respect to (which we will shorten to w.r.t.) each other, either rotationally or translationally. Joints have an *attachment point* on the bone to which they connect and between one and three rotational degrees of freedom (DOFs) and one to three translational degrees of freedom. The *orientation* of a joint will be described as an *angular displacement* with respect to the bone it is attached to. If we allow translational DOFs in the joint, we get a general *spatial displacement*. Under this definition, we see that bones describe *coordinate frames* (rotation plus translation) and joints *rigid transformations* between them, which we discuss further below.

Using displacements implies that if just an elbow is rotated, all the bones below it in the tree (hands, fingers, lower arm) rotate with respect to the elbow’s parent bone (upper arm) as well.<sup>1</sup>

---

<sup>1</sup> Although this definition might seem obvious to someone familiar with an interactive animation program, some physics systems represent bones as floating rigid bodies defined with respect to the world coordinate system and consider the joints as constraints on the relative motion of the bones. The definition of rotations as *relative* rather than *absolute* simplifies the kinematic mathematics while also allowing familiar behavior for designers used to working with keyframe animation systems.

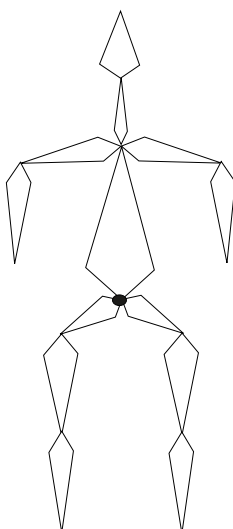


Figure 5-2: An articulated figure can be considered as a tree with joints as edges connecting limbs (nodes). The black circle shows the root of the tree, although any point in the structure could be chosen.

---

### 5.1.1 Simplifying Assumptions

We make several simplifying assumptions. First, we will restrict our model to only *rotational* joints and not discuss translational, or prismatic, joints which are often modeled in robotic simulations. This assumption is reasonable for most situations in character animation. Also, translational effects are the easier of the two, being a standard vector space.

Furthermore, some simulations also model different types of rotational joints, such as revolute joints (1 DOF), universal joints (2 DOF) and ball-and-socket joints (3 DOF). Other systems force 3 DOF joints to be created from three hinge joints with zero length bones between them, which is essentially just an Euler angle set. This is often done to use linear algebra techniques as if the joint angles formed a vector.

Instead, we will consider all rotational joints to be full ball-and-socket joints (3 DOF). This restriction lets us model the rotation of all joints with a single unit quaternion for simplicity. Note that a 1 DOF joint will be a quaternion with a fixed axis. We will see later how to find the inherent joint DOFs from example animation data automatically and how to constrain the quaternion rotation to these degrees of freedom.

### 5.1.2 Skeletal Tree Structure

A valid skeleton is considered as a *tree* in the graph theory sense, with the joints considered as *edges* and the bones as *nodes* (see Figure 5-2). This might seem a little weird at first, since normally we think of bones having length and joints not. Since bones can have more than one joint attached to them, but not vice versa, we see that bones must be nodes.

A tree must have a single *root node*, usually placed at the pelvis. Since it has no joints

above it, a character can only move around an environment if we attach it to some fixed inertial coordinate system called the *world* by a full six degree of freedom rigid body joint. As mentioned above, all of the other *internal* joints have only rotational degrees of freedom. For this document, we will also ignore the root joint translation since it is a more familiar Euclidean space and can be factored out.

Also, a unique acyclic *path* can be found between any two bones in the tree consisting of all the bones and joints along the way. Clearly, the collection of joint displacements along a path define a compound transformation between the bone coordinate systems. We now formalize these notions.

### 5.1.3 Root Joints are Special

The root joint needs to be handled specially. Internal joints, since they are relative rotations from their parent, can have animations played out on them directly and will look correct as long as the animations were defined with respect to the same basis posture of the character (Figure 1-2 shows the dog's basis). The root joint, however, defines the orientation of the character in the world frame, which is an arbitrary choice. For this reason, we cannot play a root animation directly on the character, since the orientation will be defined by where the animator chose to start and end the animation.

For example, a walk cycle turning to the left may start from the orientation at the identity, and end up going 90 degrees to the left. To deal with this, the root joint orientation must be handled differentially. In other words, we will need to find the derivative of the motion and integrate it forward from the current orientation rather than just set the orientation absolutely from the animation data. In general, we will only be discussing internal joints for this document. We will make points about handling the root node as needed.

## 5.2 Bones, Joints, and Coordinate Frames

This section will introduce our notation for describing coordinate systems and vectors defined in different coordinate bases. It will then show how the joint parameters (the quaternion representing orientation and the attachment point to the parent) represent the transform between two bones.

### 5.2.1 Coordinate Frame Terminology

A bone can be used to define a *coordinate frame*, denoted  $B$ , which describes the orientation and translation of one frame with respect to another frame. For this section, we will usually denote a coordinate frame as a familiar  $4 \times 4$  *homogenous matrix* which relates the coordinates in one frame to those in the basis of another by acting on homogenous column vectors. We denote the transformation (equivalently, spatial displacement) that describes one coordinate frame  $B$  with respect to another coordinate frame  $A$  as  ${}^A_B D$ . We will denote a vector  $x$  defined in a coordinate frame  $B$  as  ${}^B x$ . A point  $x$  in  $B$  ( ${}^B x$ ) can be described in the basis of  $A$  by matrix multiplication:



$${}^A\mathbf{x} = {}^A\mathbf{D}^B\mathbf{x}.$$

This formula reveals the choice of notation (which follows Craig [18]): the superscript which describes the basis can be thought of syntactically “cancelling out” with the subscript defining the relative frame, leaving only the superscript which describes the basis of the new point. This syntactic sugar is useful for avoiding errors in transformation equations.

A displacement can be factored into a rotational component and a translational component:

$${}^A\mathbf{D}^B = {}^A\mathbf{T}^B {}^A\mathbf{R}^B$$

which describes the orientation of  $\mathbf{B}$  in the frame  $\mathbf{A}$  as a rotation matrix and the origin as a translation matrix made from the origin’s coordinates in the reference frame. Thus, a frame can also be described as the pair  $({}^A\mathbf{R}^B, {}^A\mathbf{o}_B)$  where  ${}^A\mathbf{R}^B$  is a  $3 \times 3$  rotation matrix and  ${}^A\mathbf{o}_B$  is the origin of  $\mathbf{B}$  written in the basis of  $\mathbf{A}$ . In this notation, we can write the coordinate transform as

$${}^A\mathbf{x} = {}^A\mathbf{R}^B {}^B\mathbf{x} + {}^A\mathbf{o}_B$$

### Compound Transforms and Intermediate Frames

If two frames  $\mathbf{A}$  and  $\mathbf{C}$  are related by an intermediate coordinate frame  $\mathbf{B}$ , then a vector  ${}^C\mathbf{x}$  can be expressed in coordinate frame  $\mathbf{A}$  by the compound transformation

$${}^A\mathbf{x} = {}^A\mathbf{D}^B {}^B\mathbf{D}^C {}^C\mathbf{x}$$

where again the frame scripts cancel between transformations as well. This equation is important since we can often use an intermediate coordinate system to relate two arbitrary frames as long as we know each frame’s displacement relative to the intermediate one.

Since displacements form a group, we can write also factor the compound transformation in terms of the pair representation  $({}^A\mathbf{R}^B, {}^A\mathbf{o}_B)$  as

$$({}^A\mathbf{R}^C, {}^A\mathbf{o}_C) = ({}^A\mathbf{R}^B {}^B\mathbf{R}^C, {}^A\mathbf{R}^B {}^B\mathbf{o}_C + {}^A\mathbf{o}_B) \quad (5.1)$$

which can be checked by simple algebra. Clearly this also extends to arbitrary products of connected intermediate transforms, which we will use below.

Since quaternions represent rotations and  $\mathbf{R}$  is a rotation matrix in  $\text{SO}(3)$  (homogenous coordinates are not needed in the pair notation), we can equivalently describe a coordinate frame as a unit quaternion and a vector. We now show how we can use this fact to model joints and bone transformations with quaternions.

## 5.2.2 Joints and Bone Transformations with Quaternions

**Definition 1** A joint is represented as the pair  $(\hat{Q}, \mathbf{a})$  where  $\hat{Q}$  is the current orientation defined with respect to the parent and  $\mathbf{a}$  is the joint’s attachment point defined in the parent coordinate system.

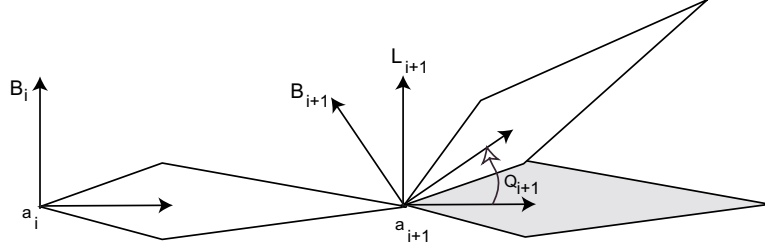


Figure 5-3: The link transform from a parent bone's coordinate system in a kinematic chain ( $B_p$ ) to its child ( $B_i$ ) depicted in 2D. The coordinate system  $L_i$  (and associated grey bone) show the zero rotation (basis) configuration. The quaternion  $\hat{Q}_i$  is the angular displacement from the basis and thus specifies the current orientation of the child bone with respect to the parent's coordinate system.

Figure 5-3 depicts the attachment of two bones in a tree such as that shown in Figure 5-2 by a joint. For simplicity of presentation, we have assumed that the attachment point of a bone to its parent is coincident with its origin. If the origin of the bone is located somewhere else, then we must know the attachment point's location in both frames and perform extra translations between the attachment point and bone origins (see Craig [18] or McCarthy [59] for details).

Let  ${}^p\mathbf{a}_c$  be the attachment point of the child bone  $B_c$  in the coordinate system of its parent bone  $B_p$  and  ${}^p\hat{Q}_c$  be the unit quaternion defining the child's orientation with respect to the parent. The transform  ${}^p_c\mathbf{D}$  that takes a child point  ${}^c\mathbf{x}$  into its description in the parent frame  ${}^p\mathbf{x}$  is the displacement

$${}^p_c\mathbf{D} = {}^p_c\mathbf{T} {}^p_c\mathbf{R}$$

which can be written as

$${}^p\mathbf{x} = {}^p_c\mathbf{T} {}^p_c\mathbf{R} {}^c\mathbf{x} = {}^p_c\mathbf{R} {}^c\mathbf{x} + {}^p\mathbf{a}_c$$

where  ${}^p_c\mathbf{T}$  is the translation matrix created from  ${}^p\mathbf{a}_c$  and  ${}^p_c\mathbf{R}$  is the rotation matrix made from the unit quaternion  $\hat{Q}_c$ . This equation can be thought of as taking a point in the child frame and rotating it into a frame with the same orientation of the parent but located at the attachment point and then translating to make the frame coincident with the parent.

In our model of a joint as the pair  ${}^p\mathbf{a}_c$  and  ${}^p\hat{Q}_c$ , this equation can be interpreted simply in the quaternion algebra as

$$\boxed{{}^p\mathbf{x} = {}^p\hat{Q}_c {}^c\mathbf{x} {}^p\hat{Q}_c^* + {}^p\mathbf{a}_c} \quad (5.2)$$

The inverse transformation that takes a point in parent coordinates into child coordinates

$${}^c_p\mathbf{D} = ({}^p_c\mathbf{T} {}^p_c\mathbf{R})^{-1}$$

which can be written as

$${}^p\mathbf{R}^{-1} {}^c\mathbf{T}^{-1}$$

or equivalently

$${}^c\mathbf{D} = {}^c\mathbf{R} {}^p\mathbf{T}.$$

Recall that the inverse of a rotation is its transpose

$${}^c\mathbf{R} = {}^p\mathbf{R}^T$$

and that the inverse of a translation created from a vector  $\mathbf{a}$  is simply the translation created from  $-\mathbf{a}$ . Simple algebra shows that we can write the inverse of the joint transformation as

$${}^c\mathbf{x} = {}^c\mathbf{D} {}^p\mathbf{x} = {}^p\mathbf{R}^T {}^p\mathbf{x} - {}^p\mathbf{R}^T \mathbf{a}_c \quad (5.3)$$

where syntactic cancellation assures us that we can subtract the vectors. We can factor the rotation out to get

$${}^c\mathbf{R} ({}^p\mathbf{x} - \mathbf{a}_c)$$

which gives us the inverse transform equation

$${}^c\mathbf{x} = {}^c\mathbf{R} ({}^p\mathbf{x} - \mathbf{a}_c) \quad (5.4)$$

which lets us immediately write

$$\boxed{{}^c\mathbf{x} = {}^p\hat{Q}_c^* ({}^p\mathbf{x} - \mathbf{a}_c) {}^p\hat{Q}_c} \quad (5.5)$$

strictly in the quaternion algebra.

**Computational Issues** This formulation takes us less memory and is good for rotating a small number of vectors as will often be the case in an inverse kinematics algorithm (such as ours in Chapter 9). If a large number of vectors must be transformed, however, converting the quaternion to a  $3 \times 3$  rotation matrix and using Equations 5.3 and 5.4 should probably be used if space is not an issue since there will be less total multiplies. Homogenous matrices are most often only used to simplify the linear algebra or when other effects are needed, such as scale.

### 5.2.3 Open Kinematic Chains and Compound Transformations

A path through a skeletal tree defines a *kinematic chain* between the bones at the endpoints [59]. The first bone in the chain,  $B_1$ , is called the *base* and the last one,  $B_n$  is called the *end*. An *open* chain does not have any cycles in it, while a *closed* chain does. We will restrict ourselves to open chains since a tree structure will not have any cycles in it by definition. Since we know all the parent-child transforms for each joint in the chain, we can create a compound transform relating the endpoint bones by composing all the joint transformations down the chain.

Let  $B_i$  denote the  $i$ th bone coordinate frame in a kinematic chain and  $(\hat{Q}_i, \mathbf{a}_i)$  be the joint parameters connecting it to its parent bone frame  $B_{i-1}$ . Let  ${}^i_{i-1}\mathbf{D}$  be the transform from child coordinates to parent coordinates created from the joint parameters as above. Then the transformation relating a bone  $B_j$  to a bone  $B_k$ ,  ${}^k_j\mathbf{D}$ , can be found by composing the transformations along the chain between  $k$  and  $j$ :

$${}^k_j\mathbf{D} = {}^k_{k+1}\mathbf{D} {}^{k+1}_{k+2}\mathbf{D} \cdots {}^{i-2}_{i-1}\mathbf{D} {}^{i-1}_i\mathbf{D} \quad (5.6)$$

This transformation is often written using the base of the chain,  $B_1$ , as an intermediate frame:

$${}^k_j\mathbf{D} = {}^k_1\mathbf{D} {}^1_j\mathbf{D}$$

or

$${}^k_j\mathbf{D} = {}^1_k\mathbf{D}^{-1} {}^1_j\mathbf{D} . \quad (5.7)$$

### Forward Kinematics

The transformation from the base of a kinematic chain to the end of the chain  ${}^1_N\mathbf{D}$  is called the *forward kinematics equation* of the chain. Since the joint attachment points are fixed, the forward kinematics are only a function of the joint orientations along the chain with the attachment locations specifying the length of the bones. These can be considered implicit fixed parameters of the function so we can write:

$$\boxed{{}^1\mathbf{x} = f(\underline{\hat{Q}})}$$

where  $\underline{\hat{Q}}$  is the ordered set of joint orientations along the chain <sup>2</sup>.

The *inverse kinematics* problem tries to find the posture that achieves a certain *end effector* (point in the last bone) location in the world:

$$\boxed{\underline{\hat{Q}} = f^{-1}({}^1\mathbf{x}_{\text{goal}})}$$

We will describe our solution to this problem in Section 9.4.

### Quaternion Forward Kinematics

If we use the factored notation  $({}^k_j\mathbf{R}, {}^k_j\mathbf{o}_j)$  for the compound transform along a chain, we can find  ${}^k_j\mathbf{D}$  efficiently by recursively going down the chain and composing the joint transform as we go using Equation 5.1 to calculate the compound frame transform  $({}^1_i\mathbf{R}, {}^1_i\mathbf{o}_i)$  and using the efficient inverse formula

$$({}^1_i\mathbf{R}, {}^1_i\mathbf{o}_i)^{-1} = ({}^1_i\mathbf{R}^T, -{}^1_i\mathbf{R}^T {}^1_i\mathbf{o}_i) .$$

This equation can be written in the quaternion algebra as

---

<sup>2</sup>We use the underbar to denote an ordered set, or tuple, of quaternion values.

$$\boxed{({}^1\hat{Q}_i, {}^1\mathbf{o}_i)^{-1} = ({}^1\hat{Q}_i^*, -{}^1\hat{Q}_i^* {}^1\mathbf{o}_i {}^1\hat{Q}_i)} \quad (5.8)$$

for efficiency. Since  ${}^j\mathbf{D}$  and  ${}^k\mathbf{D}$  share many intermediate frames  ${}^i\mathbf{D}$ , it is computationally efficient to cache both the forward and inverse transforms at each bone in the chain to avoid redundant computation.

These quaternion operations can be used to efficiently compute the coordinates of a point  ${}^j\mathbf{x}$  in any bone frame  $\mathbf{B}_j$  along the chain with respect to any other bone in the frame  $\mathbf{B}_k$  from the joint parameters as each joint. We will use these operations in Chapter 9.

## 5.3 Postures, Posture Distance, Motions and Animations

Since the translational parameters of the joint transform are fixed by the geometry of the figure, we often will not need them and instead will focus on the rotational degrees of freedom. It will be useful to collect these into a single data structure. We will call the collection of all the rotational degrees of freedom of the skeletal model a *posture*.

### 5.3.1 Postures

**Definition 2** The **posture** of a character, denoted  $\hat{P}$ , is represented as a tuple of unit quaternions  $\hat{P}_i$  containing the rotational degrees of freedom of the skeleton from a depth-first left-right traversal of the tree.

We will refer to the  $i$ th quaternion in the posture,  $\hat{P}_i$  as the  $i$ th joint. The root joint which connects the character to the world will be denoted as  $\hat{Q}_0$  and will only be used in certain situations.

A skeletal model has a distinguished *identity posture* or *basis posture* in which the character is considered to have zero rotation on all the joints, or the identity quaternion,  $\hat{1}$ . Although the actual resulting kinematic configuration of the character in this posture can be arbitrary depending on the character modeler's choices, often the modeler is asked to use a particular configuration of the skeleton as reference point for animations. In the case of humanoid characters, the coordinate system of the bone geometry (and therefore the joints that link to them) is usually chosen such that the identity posture is the configuration of the character with arms stretched out to the side, palms down, and legs straight and slightly apart. In the case of our dog character, the reference posture is similar, but with all the dogs legs on the ground and head looking straight ahead (see Figure 5-1).

### Posture Algebra and Calculus

The algebra and basic calculus of postures follows immediately due to the quaternion product group representation. All quaternion operations can be performed component-wise independently on the posture quaternions.

### 5.3.2 Posture Distance Metric

We can perform a weighted Euclidean norm component-wise:

$$\text{dist}_{\mathbf{w}}(\hat{\underline{A}}, \hat{\underline{B}}) = \left( \sum_{i=1}^n w_i \|\ln(\hat{A}_i^* \hat{B}_i)\|^2 \right)^{\frac{1}{2}}$$

where the  $w_i$  are scalar weights on the components which allow us to weight the contribution from each component quaternion.

One thing to note is that the metric is dependent on  $n$ , the dimension of the quatuple. For this reason, it is often desirable to use a metric with weights that sum to unity. The Root Mean Square (RMS) metric is simply obtained with  $w_i = \frac{1}{n}$ .

This metric can also be found using the power operator:

$$\text{dist}_{\mathbf{w}}(\hat{\underline{A}}, \hat{\underline{B}}) = \left( \sum_{i=1}^n \|\ln((\hat{A}_i^* \hat{B}_i)^{w_i})\|^2 \right)^{\frac{1}{2}}$$

since the scalar power  $w_i$  gets pulled down by the log.

Several important points must be made here. First, this metric is valid over the entire quaternion sphere. This means that it does not take into account antipodal symmetry. Similar to the use of slerp, each of the joint quaternions must be pre-processed so that they live on the same local hemisphere of  $S^3$ . For two samples this is a trivial check. We describe how to do this for  $n$  samples in Chapter 7.

Second, this metric uses the intrinsic metric on the sphere. In other words, it is as if you were a person living on the sphere who could only make measurements along the surface. This will give us the angle between orientations of joints which Euler's theorem specifies must exist. Therefore, this is a natural metric.

Finally, it should be noted that the chord-length between two hemisphere-local unit quaternions is a good approximation of this metric when the arguments are nearby on the sphere, but we prefer to use the geodesic metric since the units are in radians and this will make specifying weights more meaningful.

### 5.3.3 Motions

When a character moves, its pose changes over time. This trajectory of poses is usually called a motion:

**Definition 3** *A motion (or posture trajectory) is a continuous trajectory of poses parameterized by a single time parameter,  $t$ .*

In general, we also want our motions to have some other subjective qualities, such as smoothness, which can be handled in many ways, e.g. cubic splines. We can also take derivatives of a motion in the standard manner by component-wise differentiation, and denote this with the standard overdot,  $\dot{\underline{p}}$ . The space of all motions is called *motion-space*.

### 5.3.4 Animations

An *animation* refers to a particular, known trajectory of postures, such as that created by a motion capture device or by a commercial package. An animation is represented computationally as a list of postures and the corresponding time which the pose occurs in the animation. These postures are usually called *keyframes*, or *samples*. The associated times are often called *keytimes*. To formalize this slightly, we define a keyframe animation as the ordered set of poses and associated keytimes. Note that angular velocity information is contained within implicitly within the interframe displacements.

For simplicity, samples will be considered *uniformly sampled*, or having the same duration of time separating samples. For uniformly sampled animations, the keyframe animation will be represented by an ordered set of postures along with the sample rate,  $\Delta t$ . For this case, the keytimes are integral multiples of  $\Delta t$ :

$$t = i \Delta t$$

so we can simply refer to keyframe poses by an index.

## 5.4 Summary

This chapter described the terminology and computational structures involved in skeletal animation technology, and how we can use quaternions to model the poses of a character. We define a quatuple and introduce distance metrics on them which we will use throughout this thesis.

The reader should have learned:

- The joint-bone skeletal model for articulated figure animation
- Coordinate and kinematic frames on the hierarchy efficiently in terms of quaternions
- Representation of posture as a tuple of quaternions
- A basic metric on postures
- How animation data is represented as an ordered set of uniformly sampled postures and a time increment  $\Delta t$  between them

The next chapter will show how to learn a statistical model of joint motion from a corpus of animations and how to generate new poses according to this model.





## Chapter 6

# QuTEM: Statistical Analysis and Synthesis of Joint Motion

This chapter will describe a simple statistical model of the motion of individual joints whose orientations are modeled as unit quaternions. The model is called the QuTEM (Quaternion Tangent Ellipsoid at the Mean) and contains the following data:

- Mean unit quaternion
- Covariance about the mean (principal motion axes and associated variances)
- Constraint boundary (finite support radius)

We will see that the QuTEM will be a Gaussian distribution in the tangent space at the mean value.

In this chapter, the reader will learn:

- How to find the mean orientation from unit quaternion data concentrated around one of the modes on  $S^3$ .
- How to use the choice of mean representative to flip all data efficiently to a hemisphere of  $S^3$  to deal with antipodal symmetry.
- How to find principal axes and associated variances around the mean representative and estimate inherent joint degrees of freedom.
- How to calculate the Mahalanobis distance of some query unit quaternion to the mean representative.
- How to find a maximal isoprobability contour to model data on a local convex subset of the sphere (finite support, or constraint, radius).
- How to sample new points from the distributions.

This chapter will focus on definitions, algorithms and discussion. Results from experiments on real data can be found in Section 10.1.

The rest of the chapter will proceed as follows:

**Section 6.1** describes the QuTEM model of joint motion statistics and how it can be used as a proportional probability density function.

**Section 6.2** presents estimation algorithms for each of the parameters.

**Section 6.3** shows how to sample new quaternions using the QuTEM density.

**Section 6.4** summarizes the uses of the QuTEM and discusses several outstanding problems.

## 6.1 QuTEM

This section will describe the QuTEM model of joint motion, estimation of parameters (mean, covariance, support radius), and sampling new joint orientations from the distribution.

### 6.1.1 Motivation

As we saw in Chapter 4, we desire a model of joint motion which has smooth isoprobability contours so that we can use the model for joint constraints. Also, the model should be easily learnable from data and allow us scale-invariant (with respect to motion ranges) metrics.

Gaussian (also called *normal*) probability density functions (p.d.f.'s) are the standard workhorse of probabilistic analysis (see Appendix B for a quick overview, or [8] or [83] for a more in-depth treatment). Since the maximum likelihood estimation (MLE) of parameters and sampling from a Gaussian is straightforward and well-known, and also since the isoprobability contours of a Gaussian are ellipsoids, we chose to use a Gaussian distribution to model the motion of a joint.

There are two basic ways to use a Gaussian distribution to model unit quaternion data:

- Embedding Approach ( $\mathbb{R}^4$ )
- Wrapping Approach ( $T_{\hat{M}} S^3$ )

In the *embedding* approach, a Gaussian is estimated in  $\mathbb{R}^4$ , the space in which the unit quaternion manifold ( $S^3$ ) is embedded, and conditioned to live on the sphere. This approach leads to the *Bingham distribution*, which is described in [7, 57, 47, 50, 51, 58] and used recently by several computer vision researchers [1, 16]. Maximum likelihood estimation of the variances leads to handling *confluent hypergeometric functions of matrix argument*, which is tricky.

Instead, we chose to use the *wrapping approach* (see [57]) where a distribution on tangent vectors at the mean is wrapped onto the sphere to make a spherical distribution. This choice was made for simplicity, since the estimation of parameters is simpler, which is desirable.

Notice that for unit quaternion data representing rotations ( $SO(3)$ ) the distribution will exhibit *antipodal symmetry*. Since we assume our data is locally-concentrated on the sphere (since joints tend to live in around some equilibrium point), the distribution will be *bimodal*,

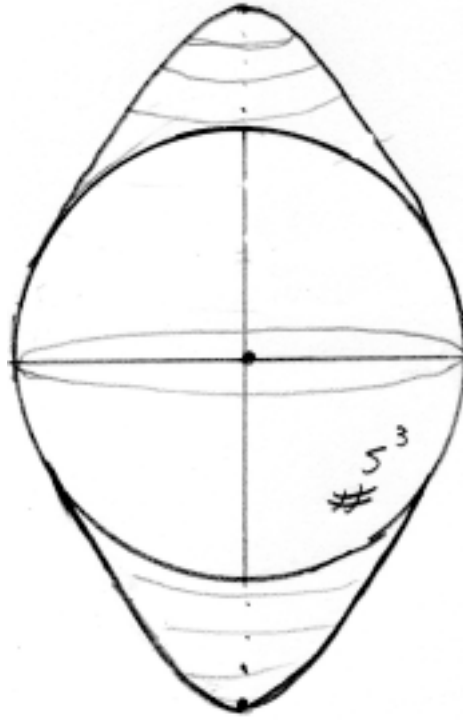


Figure 6-1: A sketch of a bimodal distribution on  $S^3$  which exhibits antipodal symmetry. Such distributions are valid distributions over  $SO(3)$  through the double-covering.

having a maximum on opposite sides of the sphere at the modes  $\pm \hat{M}$  (see Figure 6-1) To handle this, we choose to model only one hemisphere of the distribution (a single choice of sign for  $\hat{M}$ ) and flip any quaternion to be on this hemisphere before the distribution is used.

Finally, we note that tangent vectors to  $S^3$  are easily described in terms of the logarithmic map, as we saw in Chapter 3. Since the exponential map preserves distances and angles from the center of the map, hyperellipses around the map center on  $S^3$  will map to 3D ellipsoids in the tangent space ( $\mathbb{R}^3$ ). Furthermore, a point on a particular isocontour of the distribution on the sphere will map to a point the same distance from the origin in the tangent space. For this reason, we can use the exponential and logarithmic maps in a straightforward manner to pre-process our data analytically without losing information.

To summarize the main idea of the approach:

**Wrap a zero-mean Gaussian in  $\mathbb{R}^3$  onto a hemisphere of  $S^3$  using the exponential map at one of the modes.**

To summarize the assumptions we make:

- Data will form a fairly concentrated bimodal distribution on the sphere exhibiting antipodal symmetry since data can be considered to be compact around some equilibrium point of the joint.
- Antipodal symmetry will be handled by modeling one hemisphere of  $S^3$  and flipping data to that side before using the model.
- For the purposes of this document, we will not need normalized densities (densities that integrate to unity over  $S^3$ ), but will instead look at *proportional densities*.

### 6.1.2 QuTEM Definition

**Definition 4** *The Quaternion Tangent Ellipsoid at the Mean (QuTEM) model is defined as the tuple  $(\hat{M}, \hat{R}, \mathbf{v}, \rho)$  where  $\hat{M} \in \hat{\mathbb{H}}$  is the **mean representative**,  $\hat{R} \in \hat{\mathbb{H}}$  is the **principal axes rotation**,  $\mathbf{v} \in \mathbb{R}^3$  is the **variance vector**, and  $\rho \in \mathbb{R}$  is the **constraint distance**.*

In the model,  $\hat{M}$  is the choice of the mean representative. This defines the hemisphere of  $S^3$  we model, since the other side is the same due to symmetry. To use the density, query quaternions need to be flipped to the hemisphere defined by  $\hat{M}$ , as we show below.  $\hat{R}$  is the rotation of the tangent space that aligns the principal axes of the data with the basis axes and  $\mathbf{v}$  is the 3-vector of variances associated with these axes. These parameters model the covariance about the mean. Finally,  $\rho$  is the Mahalanobis distance (standard deviation value, see Appendix B) from the mean beyond which the density is defined to be zero. This allows the model to handle the fact that organic joints will have constraints that do not allow them to range over the entire  $S^3$ .

Figure 6-2 gives an abstract depiction of the QuTEM model in one lower dimension.

### 6.1.3 QuTEM as a Wrapped Gaussian Density

This section shows how to calculate the proportional probability density value of a particular query unit quaternion  $\hat{Q}$  from the QuTEM model  $(\hat{M}, \hat{R}, \mathbf{v}, \rho)$ .

The QuTEM is a zero-mean Gaussian vector density in the tangent space at one of the quaternion modes wrapped onto a hemisphere of  $S^3$  using the exponential mapping (see Figure 6-3). Recall that the density function for a Gaussian distributed random vector is:

$$N(\mathbf{x}; \mathbf{m}, \mathbf{K}) = \frac{1}{(2\pi)^{n/2} |\mathbf{K}|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T \mathbf{K}^{-1} (\mathbf{x}-\mathbf{m})}$$

where  $\mathbf{x}$  is the random vector,  $\mathbf{m}$  is the mean vector, and  $\mathbf{K}$  is the covariance matrix of the data (see Appendix B for a quick review of multi-variate Gaussian densities or a reference such as [8, 65]). Note that here  $\mathbf{K}$  is assumed to be positive semi-definite (all eigenvalues real and greater than or equal to zero). The pseudo-inverse (see, for example, Strang [81])

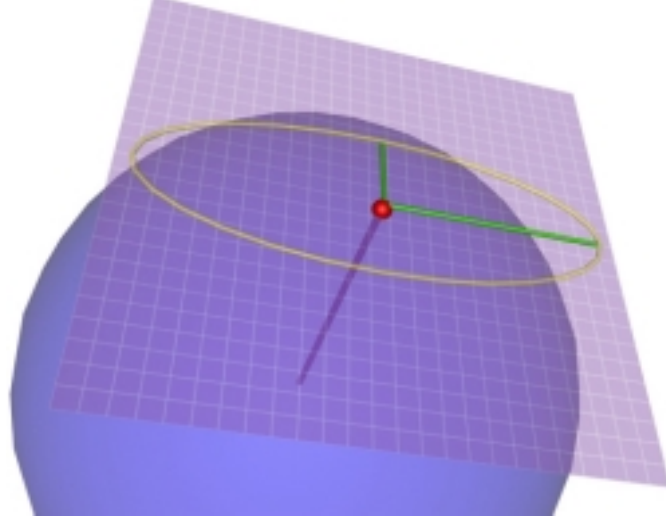


Figure 6-2: An abstract depiction of the QuTEM. The mean is the tangent space point, the ellipse depicts the  $\rho$  Mahalanobis distance constraint surface, and the axes depict the principal axes of the density and their relative variances (covariance).

must be used if  $\mathbf{K}$  is singular, which will happen if the data has less than three rotational degrees of freedom.

Given the exponential mapping of unit quaternions which lets us map to a tangent vector at some point on  $S^3$ , we can define our unit quaternion density directly in terms of the vector Gaussian distribution. Let  $\mathbf{R} \in \text{SO}(3)$  denote the rotation matrix associated with the unit quaternion  $\hat{R}$ . Let  $\mathbf{V}$  be the diagonal matrix created from the vector of variances  $\mathbf{v}$ . Then we define the covariance matrix ( $\mathbf{K}$ ) for a zero-mean normal distribution in terms of the matrix representation as:

$$\mathbf{K} = \mathbf{R}\mathbf{V}\mathbf{R}^T$$

and its inverse

$$\mathbf{K}^{-1} = \mathbf{R}^T\mathbf{V}^{-1}\mathbf{R}$$

where the pseudo-inverse is used if  $\mathbf{V}$  is singular.

Since we are assuming only measurements on the sphere, we cannot directly “subtract off the mean” as in the vector Gaussian density. Instead, we need to use the logarithmic mapping at the mode representative  $\hat{M}$  to perform this.

Let  $\mathbf{q} = \ln(\hat{M}^*\hat{Q})$  be the *mode-tangent* vector of  $\hat{Q}$  at  $\hat{M}$ . The QuTEM density is then just a vector Gaussian density on the mode-tangent vector  $\mathbf{q}$ :

$$p(\hat{Q}; \hat{M}, \hat{R}, \mathbf{v}, \rho) = ce^{-\frac{1}{2}\mathbf{q}^T\mathbf{K}^{-1}\mathbf{q}}$$

where  $c$  is a normalization parameter to make the density integrate to unity over  $S^3$ . For the purposes of this document, we will not need to find this term and assume it to be unity.

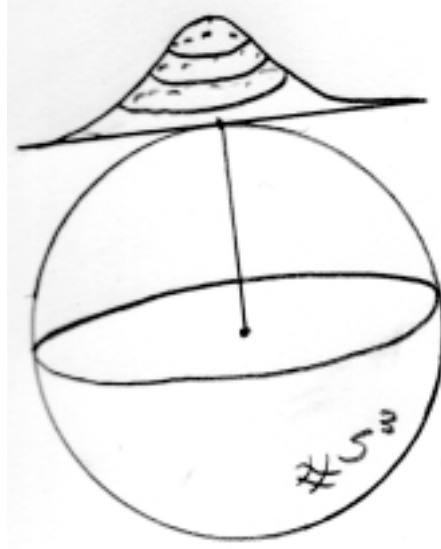


Figure 6-3: The QuTEM models a spherical distribution by estimating a zero-mean Gaussian density in the tangent space to the unit quaternion mean.

Note that this makes the density not integrate to unity over  $S^3$ , but in practice often only proportional densities are needed so this is not a severe limitation.

We can expand the density out in terms of quaternion algebra as:

$$p(\hat{Q}; \hat{M}, \hat{R}, \mathbf{v}, \rho) = ce^{-\frac{1}{2}(\ln(\hat{R}^* \hat{M}^* \hat{Q} \hat{R}))^* \mathbf{V}^{-1} (\ln(\hat{R}^* \hat{M}^* \hat{Q} \hat{R}))} \quad (6.1)$$

where  $\mathbf{V}^{-1}$  denotes the inverse (or pseudoinverse <sup>1</sup> if  $\mathbf{V}$  is singular) of the diagonal matrix of the variances made from the vector  $\mathbf{v}$ .

Finally, we address the extra parameter,  $\rho$ . Since we are modeling data which we have assumed does not range over the entire sphere, we need to force the density to zero outside of some constraint region. Since we argued previously for using a smooth constraint boundary, we decided to choose a particular isodensity contour of the tangent space Gaussian distribution beyond which the density is zero. In other words, this amounts to finding a standard deviation value away from the mean (Mahalanobis distance) beyond which we define the density to be zero. Figure 6-4 depicts what the distribution will look like when wrapped to a hemisphere of  $S^3$ . Since joints tend live on the constraint boundary some portion of the time, this is a reasonable model.

Since the quadratic in the exponent of the Gaussian distribution is the Mahalanobis distance squared, we can factor the formula into a simple test to see if we are outside the valid density region or not. In our notation, this gives the constraint equation:

<sup>1</sup>Recall that the pseudoinverse, which is related to the Singular Value Decomposition, ignores variations in singular directions (see, for example, Strang [81]) and can be created from the diagonal matrix with the simple rule that  $\frac{1}{0} \rightarrow 0$  for each singular variance in the pseudoinverse.

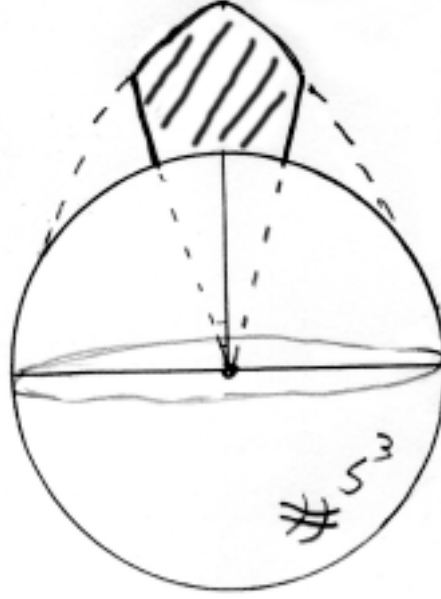


Figure 6-4: A sketch of a spherical density on  $S^3$  constrained to be zero beyond a certain distance from the mean.

$$p(\hat{Q}) = 0 \quad \text{if } \|\mathbf{V}^{-1/2} \ln(\hat{R}^* \hat{M}^* \hat{Q} \hat{R})\| > \rho$$

and otherwise is the same as Equation 6.1.

#### 6.1.4 Scaled Mode Tangents (SMT), Ellipsoids, and Mahalanobis Distance

We will find it useful for the remainder of the discussion and for intuition to collect the multiple stage quaternion transformation used in the definition of the density into a single transformation from a unit quaternion into its “Scaled Mode Tangent” description (which we shorten to SMT) with respect to the QuTEM. Formally, mixing notation a little for convenience,

$$\boxed{\mathbf{s} = \text{SMT}(\hat{Q}; \hat{M}, \hat{R}, \mathbf{v}) = \mathbf{V}^{-1/2} \mathbf{U}^T \ln(\hat{M}^* \hat{Q})} \quad (6.2)$$

or entirely in the quaternion algebra:

$$\text{SMT}(\hat{Q}) = \text{Scale}\left(\frac{1}{\sqrt{v_1}}, \frac{1}{\sqrt{v_2}}, \frac{1}{\sqrt{v_3}}, \hat{R}^* \ln(\hat{M}^* \hat{Q}) \hat{R}\right)$$

where we have collected the non-uniform scale of the vector portion of the quaternion into a function of the three scales factors (one for each component respectively) and the vector in  $\mathbb{R}^3$ . The SMT transformation converts a unit quaternion into a *unit-variance tangent vector*

at the mode representative. This implies that the *magnitude* of the SMT of a unit quaternion is its Mahalanobis distance from the mean. Therefore, the constraint boundary defined by  $\rho$  can be described in terms of the maximal allowable magnitude for the Mahalanobis distance beyond which the density is forced to zero.

The SMT transformation is invertible. A scaled mode tangent vector of a quaternion can be converted back into a unit quaternion by undoing each stage of the transformation — scaling the space back into an ellipsoid, rotating back into the original basis from the principal axes basis, exponentiating the vector onto the sphere and multiplying by the mode to place it at the correct location. We will use this extensively in the section on constraint projection (Section 9.2).

Geometrically, the SMT transformation maps all the points on a spherical ellipse around some center point on the sphere, call it  $\hat{M}$ , into a sphere in the tangent space at  $\hat{M}$ . Figure 6-5 shows the transformation's steps applied to a spherical ellipse on  $S^2$ , the familiar sphere embedded in  $\mathbb{R}^3$ . An ellipse on the sphere around some point (shown as the red point on the sphere) is mapped into an ellipse in the 2-dimensional tangent space with the point as the origin.

Plotting the mapping for full degree of freedom quaternions is trickier. Figure 6-6 shows the exponential mapping image of points sampled randomly on an ellipsoidal surface (a particular choice of Mahalanobis distance) in  $\mathbb{R}^3$ . In order to depict the hypersphere, we have ignored the  $z$  component of the quaternion, which effectively projects it onto the  $z = 0$  hyperplane. This allows us to view the image of the map in three-dimensions, but at a cost — the ellipsoidal boundary maps into a hyperspherical ellipse, but when we projection plot it onto a unit sphere in  $\mathbb{R}^3$ , it will not lie on the sphere (since it really lies on the sphere in  $\mathbb{R}^4$ ) and points will appear to be *inside* the ellipse. We can, however, see that the mapping still preserves ellipses by viewing it directly over the center of the mapping.

Note that

$$d_{qmahala}(\hat{Q}) = \|SMT(\hat{Q}; \hat{M}, \hat{R}, \mathbf{v})\| \quad (6.3)$$

This form is the most efficient for implementations, and we shall see in Chapter 9 that we can use the SMT transformation to model joint constraints as a simple point-sphere boundary test.

This form simplifies the density as well:

$$p(\hat{Q}) = c e^{-\frac{1}{2}(d_{qmahala}(\hat{Q}))^2}$$

## 6.2 QuTEM Parameter Estimation

This section describes how to estimate each of the QuTEM parameters from example animation data.



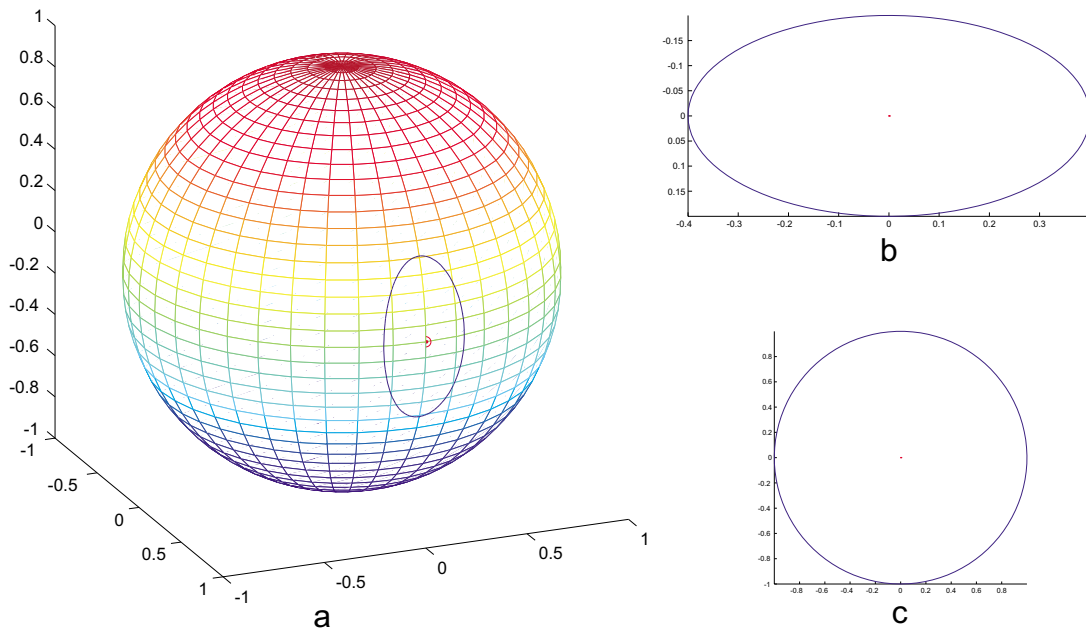


Figure 6-5: A three-dimensional visualization of the SMT transformation which turns ellipses on the hypersphere into ellipsoids in the tangent space at the center of the ellipse. The left image (a) shows the original spherical ellipse with its center; the right upper (b) shows the ellipse in the tangent space by using the exponential map at the center point (notice the center maps to the origin in the tangent space); the bottom right image (c) shows the result when the space is rescaled by the axis lengths on the ellipse to form a circle in a warped tangent space. Notice all true objects are one dimension higher.

---

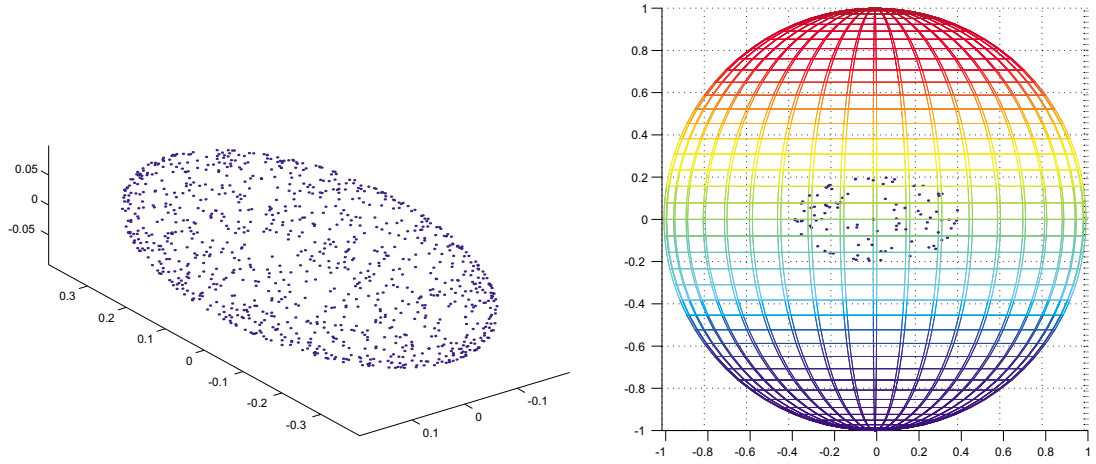


Figure 6-6: Points sampled randomly on an ellipsoid around the origin in  $\mathbb{R}^3$  (left) and 3D projection plot of the exponential mapped points (right, created by ignoring the  $\hat{z}$  component of the quaternion). Notice the points, although are on the *boundary* of an ellipsoid on the left, appear to map inside the ellipse on the sphere. This artifact results from the projection, which ignores the  $z$  direction. By viewing the sphere along the direction directly pointed at the center, however, we see that the shape is elliptical, as it should be.

## 6.2.1 Estimation of the Mean from Data

We have defined the QuTEM object in terms of parameters, but our ultimate goal is to learn the QuTEM from data. The first of these we will need is the mode (maximum density value, which we will also call *mean* throughout the remainder of the document since they are the same in the case of a Gaussian.). This section shows how to estimate the mode representative  $\hat{M}$  from a set of unit quaternion data. We will show that the solution is simply an eigenvector of the sample covariance matrix of the quaternion data represented as unit column vectors in  $\mathbb{R}^4$ . To motivate the approach, we shall introduce the problem in terms of the Euclidean analogue of finding the mean of vector data.

### Approach

The mean of Euclidean vector data is usually found with the familiar average over their coordinate values:

$$\bar{\mathbf{x}} = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_i$$

For unit quaternion data, this formula does not in general work for two reasons. First, it ignores antipodal symmetry. This means that a datapoint could be either  $\hat{Q}$  or  $-\hat{Q}$ . Clearly, the weighted Euclidean sum will not work here, since the mean we get will depend on the sign of the data, which we do not desire. Also consider the case with two data points,  $\hat{Q}$  and

$-\hat{Q}$ . Their coordinate mean will be zero, which is not on the sphere. Second, the formula does not respect the unit quaternion group. We could renormalize the result to lie on the sphere, but consider again two antipodal samples. Since their mean is zero, we cannot renormalize them.

Instead, we will estimate the mode of unit quaternion data by finding the unit quaternion that minimizes squared distances between it and all the data points. This seems appropriate since it can be shown that the Euclidean mean coordinate formula above is the point that minimized the sum of squared Euclidean distances between it and the samples.

Recall that the intrinsic metric for two unit quaternions is:

$$\text{dist}_{S^3}(\hat{Q}_i, \hat{Q}_j) = \|\ln(\hat{Q}_i^* \hat{Q}_j)\|$$

Recall that this distance metric covers the entire hypersphere and does have the antipodal symmetry for a metric on rotations. To handle this, we need to take the minimum of this metric over the choice of relative signs of the quaternions to get the metric for  $\text{SO}(3)$ :

$$\text{dist}_{|S^3|}(\hat{P}, \hat{Q}) \stackrel{\text{def}}{=} 2 \min_{\hat{A}=\hat{Q}, -\hat{Q}} \min_{S^3} \text{dist}(\hat{P}, \hat{A}) .$$

Note that the same distance function can also be written using the embedding space and linear algebra notations with absolute value:

$$\text{dist}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_j) = 2 \arccos(|\hat{\mathbf{q}}_i \cdot \hat{\mathbf{q}}_j|)$$

where  $(\cdot)$  denotes the dot product of the quaternions as unit vectors in  $\mathbb{R}^4$ . Since the dot product of two unit vectors is the cosine of the angle between them, the absolute value ignores the sign before returning the angle.

Therefore, the estimation of the mode representative quaternion as amounts to a quadratic nonlinear minimization problem:

$$\bar{\hat{Q}} = \arg \min_{\hat{P}} \sum_{i=1}^N \min_{|S^3|} \text{dist}(\hat{P}, \hat{Q}_i)^2$$

In other words, the quaternion  $\bar{\hat{Q}}$  which minimizes the sum of squared distances between itself and all the data (taking into account antipodal symmetry), is defined as the mean of a set of unit quaternion data representing spatial rotations. So how do we solve for  $\bar{\hat{Q}}$ ?

## Finding the Mean

Unfortunately, the distance function contains non-linear elements, both with the inverse cosine required for the quaternion log and the absolute value function that makes or distances orientations. This makes analytic minimization tricky. To simplify the problem, we use the fact that the function we are minimizing is quadratic (and monotonic). For this reason, we can replace this minimization with the minimization of a monotonic function of the distance and get the same answer. Since our distance is just  $\theta$ , the shortest arclength between the quaternions, we can replace  $\theta$  with  $1 - \cos(\theta)$ , which is monotonic over the interval  $[0, \pi/2]$ , as is required. Minimizing  $\theta$  is clearly equivalent to minimizing  $1 - \cos(\theta)$ .

We can simplify this even further by noticing that minimizing  $1 - \cos(\theta)$  is the same as *maximizing*  $\cos(\theta)$  over the interval, since we just invert a monotonic function. But  $\cos(\theta)$  is just the dot product of two unit vectors! Also, the absolute value can be removed by noticing that the square of the cosine is monotonic as well. These simplifications lead us to the equivalent but simpler problem:

$$\hat{Q} = \arg \max_{\hat{A} \in S^3} \sum_{i=1}^N (\hat{A} \cdot \hat{Q}_i)^2$$

This formula make intuitive sense as well — we want to maximize the *directional match* between the unit quaternions, which the dot product does. The square means that we will ignore the sign of the dot product — both  $\cos(\theta)$  and  $-\cos(\theta)$  contribute the same amount to the error function, exactly as desired.

We can solve this equation for  $\hat{Q}$  by performing a *constrained minimization* on the sum of squared distances (see for example Strang [80]). In other words, we need to find the quaternion that minimizes the function subject to the constraint that it be unit magnitude. We handle this by adding a Lagrange multiplier ( $\lambda$ ) and implicit constraint equation. This gives us the final minimization equation which we need to solve for  $\hat{P}$ :

$$E(\hat{P}) = \sum_{i=1}^N (\hat{P} \cdot \hat{Q}_i)^2 + \lambda((\hat{P} \cdot \hat{P})^2 - 1)$$

where the extra term with the Lagrange multiplier  $\lambda$  is the constraint term that keeps the magnitude of the quaternion unit and can be thought of a causing extra “energy” to be added to the system for points off the sphere.

For simplicity, in the rest of this derivation we will use the linear algebra of the embedding space ( $\mathbb{R}^4$ ) rather than the quaternion algebra. This gives us the vector equation:

$$E(\mathbf{p}) = \sum_{i=1}^N (\mathbf{p}^T \mathbf{q}_i)^2 + \lambda(\mathbf{p}^T \mathbf{p} - 1) .$$

Rewriting the square using symmetry of the vector magnitude we get

$$E(\mathbf{p}) = \sum_{i=1}^N [(\mathbf{p}^T \mathbf{q}_i)(\mathbf{q}_i^T \mathbf{p})] + \lambda(\mathbf{p}^T \mathbf{p} - 1)$$

from which we can factor out the  $\mathbf{p}$  terms since they do not depend on the sum:

$$E(\mathbf{p}) = \mathbf{p}^T \left( \sum_{i=1}^N \mathbf{q}_i^T \mathbf{q}_i \right) \mathbf{p} + \lambda(\mathbf{p}^T \mathbf{p} - 1)$$

In this form, we notice that the summation can be rewritten as simply the *outer product* matrix of the data vectors. Specifically, if we create a data matrix  $\mathbf{Q}$  whose columns are the quaternion examples, we get a  $4 \times N$  matrix of data. The outer product of this matrix with itself is the summation we require:

$$E(\mathbf{p}) = \mathbf{p}^T \mathbf{Q} \mathbf{Q}^T \mathbf{p} + \lambda(\mathbf{p}^T \mathbf{p} - 1)$$

This is the ultimate formula we need to solve. A necessary condition to be a minimum is that the derivative of  $E(\mathbf{p})$  with respect to the argument  $(\mathbf{p})$  must be zero. Specifically,

$$\frac{\partial}{\partial \mathbf{p}} E(\mathbf{p}) = 2\mathbf{Q} \mathbf{Q}^T \mathbf{p} + 2\lambda \mathbf{p} = 0$$

We recognize this as a familiar eigenvector problem:

$$\mathbf{A} \mathbf{x} = \lambda \mathbf{x}$$

where  $\mathbf{A}$  is the outer product of the data matrix  $\mathbf{Q} \mathbf{Q}^T$ . Notice that  $\mathbf{A}$  is  $4 \times 4$ , so the problem is computationally easy to solve. In order to maximize the original function and get the mean, we need the eigenvector with the *maximum* eigenvalue, since that will maximize the directional match function when plugged in and not the other eigenvectors<sup>2</sup>. We use a Singular Value Decomposition (SVD) of the  $\mathbf{A}$  matrix in order to find the eigenvector associated with the largest eigenvalue, call it  $\mu$ . Since either  $\pm\mu$  is a solution (SVD will return an arbitrary one), we choose the one nearer the identity and use  $\mu$  as the estimate for the mode representative of our QuTEM, denoted  $\hat{M}$ .

### Algorithm and Summary

To summarize, the mean (mode) of unit quaternion-represented orientation data for  $\mathbb{R}^3$  is the maximal eigenvector of the standard  $4 \times 4$  scatter matrix formed by the outer product of the data matrix. Since the objective function is symmetric its solution has two possible eigenvector solutions  $\pm\mu$ .

The algorithm is summarized as follows:

1. Let  $\hat{\mathbf{q}}_i$  be the column vector representation of the unit quaternion sample  $\hat{Q}_i$ .
2. Let the  $4 \times N$  matrix  $\mathbf{Q}$  be the data matrix with column  $i$  being the 4-vector  $\hat{\mathbf{q}}_i$  for the  $i$ th sample.
3. Let  $\mathbf{S} = \mathbf{Q} \mathbf{Q}^T$
4. Let  $\hat{\mathbf{e}}_i$  be an eigenvector of  $\mathbf{S}$  with real eigenvalue  $a_i$ .
5. Choose one of the two eigenvectors  $\pm\hat{\mathbf{e}}_*$  associated with the maximal eigenvalue  $a_*$  as the estimate of the mean,  $\hat{M}$ .

---

<sup>2</sup>This analytic result has also shown up in different guise in the computer vision community, where Horn does a similar calculation in a stereogrammetry problem [43].

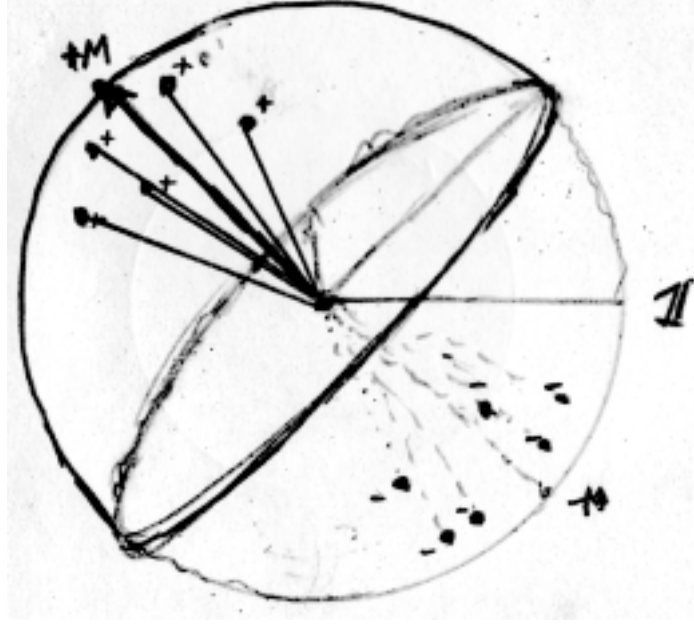


Figure 6-7: Our data is antipodally symmetric, and therefore we might arbitrarily get the sample  $\pm \hat{Q}_i$ . We would like all data on the same local hemisphere of  $S^3$  for simplicity. This hemisphere will be defined by the choice of the sign on  $\hat{M}$ . To hemispherize data, we simply flip the data to lie on the same hemisphere as the mean choice  $\hat{M}$  using a dot product as a test.

## 6.2.2 Hemispherization

This section will describe how to handle the antipodal symmetry (double-covering) of  $\text{SO}(3)$  with  $S^3$ . There are many times we need all unit quaternion data on a local hemisphere of  $S^3$ , for example if we want to use the simple geodesic distance metric or locally linearize the data.

We noted above that the QuTEM density will have antipodal symmetry due to the double-covering of unit quaternions to rotations in  $\text{SO}(3)$ . Since we choose to handle this by assuming that the density is valid only in the hypersphere associated with the mode representative  $\hat{M}$ , we need a way to map our data (and any query points) onto this hemisphere if it is not before applying the density formulas or SMT transformation.

Figure 6-7 shows how we want to choose the sign of each unit quaternion datapoint so that the processed data are in a local hemisphere of  $S^3$ . Note that we *cannot* just choose a global hemisphere, such as the one nearest the identity, as should be clear in the illustration. As the data is rotated closer to the identity, parts of it would pass through the hemisphere boundary orthogonal to the identity, but not others.

## Algorithm

Hemispherizing  $N$  unit quaternions which represent orientations of  $\mathbb{R}^3$  can be done using the same calculation as the mean. Since by construction the mean representative was calculated by minimizing distances from it to the data, if we flip all data to be on the same side as the choice of mean  $\hat{M}$ , they will all lie on a local hemisphere of  $S^3$  defined by  $\hat{M}$ .

Formally, let  $\hat{q}_i$  be the column vector representation of the  $i$ th unit quaternion sample  $\hat{Q}_i$ . Let the  $4 \times N$  matrix  $\mathbf{Q}$  be the data matrix with column  $i$  being the 4-vector  $\hat{q}_i$  for the  $i$ th sample. Then the algorithm is:

1. Find the mean representative of the example data, call it  $\hat{M}$ .
2. For each example  $\hat{Q}_i$ :
  - (a) If  $\hat{M} \cdot \hat{Q}_i < 0$  then  $\hat{Q}_i \leftarrow -\hat{Q}_i$ .

After this operation, the unit quaternions live on the same local hemisphere.

## Hemispherize Discussion and Summary

In general, most researchers simply “flip the signs until it converges.” In other words, pairwise distances between quaternions are compared using dot products and the sign on one example changed if the dot product is negative (other side of the sphere). Since the process needs to be restarted each time a flip is done, this can be expensive if there is a lot of data. Ultimately, there are  $2^N$  choices of sign to be tested.

For small amounts of data, this is often fine, but to process a lot of data, we suggest using our method of finding the mean first. Solving for the mean involves an outer product of a  $4 \times N$  matrix. Creating the outer product matrix takes  $O(n^2)$  operations since it is a matrix multiplication. Finding the eigenvector takes constant time ( $O(1)$ ) since it is a fixed size  $4 \times 4$  symmetric matrix. Testing the sign of each datum with the mean takes  $O(n)$  operations. Therefore, our hemispherize operator takes  $O(n^2)$  to compute whereas the naive brute force approach takes  $O(2^n)$ .

Finally, it is important to note that since the choice of mode representative is contained in our QuTEM model, we can simply flip a new query unit quaternion to the same hemisphere as the mean using a simple dot product calculation.

### 6.2.3 Estimation of Unit Quaternion Covariances

Now that we can find the mean of unit quaternion data representing orientations, we need to look at its *covariance* around the mean, or second moment of the distribution. The covariance describes the principal axes and associated variances of the data. In the QuTEM, we saw that the parameters  $\hat{R}$  and  $\mathbf{v}$  encode the covariance for the density. This section describes how we estimate these parameters by transforming the data to the tangent space at the mean and applying standard Gaussian vector estimation to the “linearized” data.

The approach in this section is straightforward:

- Transform the data into the tangent space at the mean using the exponential map.
- Apply standard Maximum Likelihood Estimation techniques to estimate a zero-mean Gaussian density on the transformed data.

### Estimation of Spherical Variances

In terms of unit quaternion algebra, the Euclidean operation of “subtracting off the mean”  $\hat{M}$  from a sample  $\hat{Q}_i$  translates to the unitary operation of rotating the sample so that the identity aligns with the mean:

$$\hat{P}_i = \hat{M}^* \hat{Q}_i$$

Next, we apply the logarithmic map to the mean-aligned data:

$$\mathbf{w}_i = 2 \ln(\hat{Q}_i) .$$

Recall that the exponential mapping (and inverse) preserves the distance and direction from the center of the map and maps into the tangent space at the center. This means that the transformed data  $\mathbf{w}_i$  will live in the tangent space at the mean and can be thought of as zero-mean Euclidean data with units in terms of the intrinsic group metric ( $\theta$ ). This transformation leaves us a standard Euclidean estimation problem for a Gaussian density.

To solve this, first we create the column data matrix  $\mathbf{W}$  from the transformed examples  $\mathbf{w}_i$ , which is a  $3 \times N$  matrix. We then create the  $3 \times 3$  *sample covariance matrix* in the standard manner using the outer product of the data:

$$\tilde{\mathbf{K}} = \frac{1}{N-1} \mathbf{W} \mathbf{W}^T .$$

The (real) eigenvectors of this resulting  $3 \times 3$  symmetric positive semi-definite matrix, call them  $\mathbf{u}_i$ , correspond to the principal axes of an ellipsoid in 3-space, and therefore form an orthonormal basis aligned with the principal axes of the density. The corresponding eigenvalues are the associated variances in the eigenvector direction. We write this in the linear algebra as:

$$\mathbf{K} = \mathbf{U} \mathbf{\Lambda} \mathbf{U}^T$$

with  $\mathbf{U} \in \mathbf{SO}(3)$  containing the column eigenvectors and  $\mathbf{\Lambda}$  is a diagonal matrix with the eigenvalues  $\lambda_i$  on the diagonal corresponding to the angular variance for the eigenvector  $\mathbf{u}_i$ .

We can align the principal axes with the coordinate basis axes by simply rotating them by the eigenvector basis matrix:

$$\mathbf{x}_i \leftarrow \mathbf{U} \mathbf{w}_i$$

As we saw before, however, a rotation of a 3-vector is simply a unit quaternion quadratic product, so the orthogonal matrix  $\mathbf{U}$  rotation can be converted into quaternion algebra as

$$\mathbf{x}_i = \hat{U} \mathbf{w}_i \hat{U}^* .$$



If we write the entire set of resulting transformations in the quaternion algebra we get a transformation from a quaternion to a principal-axis aligned 3-vector which can be thought of as living in the tangent space at the mode. Therefore, this is an example of a mode-tangent decomposition. Let's be explicit about this transformation. It is:

$$\mathbf{x}_i = \hat{U} \ln(\hat{M}^* \hat{Q}) \hat{U}^*$$

which can also be written as

$$\mathbf{x}_i = \ln(\hat{U} \hat{M}^* \hat{Q} \hat{U}^*)$$

since changing the basis of the vector does not change an invariant subspace (eigenvector) of a transformation.

Notice that these are exactly the parameters of the QuTEM we were seeking. The parameter for the tangent space rotation from the mean-aligned data, denoted  $\hat{R}$  in the QuTEM, is simply the tangent space rotation to principal axes,  $\hat{U}$ . This unit quaternion is found from the estimated  $\text{SO}(3)$  eigenvector matrix in the standard manner. The vector of variances associated with these axes, called  $\mathbf{v}$  in the QuTEM, is formed from the the eigenvalues of the sample covariance matrix.

## Handling Singularities in the Analysis

The sample covariance matrix is only guaranteed to be positive semi-definite, which implies that its eigenvalues  $\lambda_i \geq 0$ . Eigenvalues of zero correspond to directions with zero variance. Therefore, if the data has less than 3 DOF then the singular directions will have eigenvalues of zero. Data gotten from a 1 DOF joint such as an elbow will have only one non-zero eigenvalue since it can only move around a fixed axis. Therefore, the eigenvalues of the sample covariance of the transformed data at the mean give us a direct, intuitive and computational way to automatically discover the underlying degrees of freedom of the data. We can leverage this in algorithms by making sure they respect these degrees of freedom. For example, if we model 1 DOF hinge joints as quaternions with fixed axis then we can use special case scalar solutions (like an Euler angle) instead of the full quaternion to solve the problem correctly.

The eigenvalues of the embedded covariance in  $\mathbb{R}^4$  do not directly have this property, as we mentioned above, being related by a Bessel function, which is why we sought to avoid them.

## Summary of Covariance Estimation Algorithm

To summarize the estimation process for the parameters, we use the following algorithm:

1. Let  $\hat{M}$  be the quaternion mean of the data  $\hat{Q}_i$ .
2. Hemispherize the examples  $\hat{Q}_i$  using  $\hat{M}$ .
3. Let  $\hat{P}_i \leftarrow \hat{M}^* \hat{Q}_i$ .
4. Let  $\mathbf{w}_i \leftarrow \ln(\hat{P}_i)$ .
5. Construct the data matrix  $\mathbf{W}$  with  $\mathbf{w}_i$  as the  $i$ th column of  $\mathbf{W}$ .
6. Let  $\mathbf{K} \leftarrow \frac{1}{N-1} \mathbf{W} \mathbf{W}^T$ .
7. Perform an eigenvector decomposition (SVD) of  $\mathbf{K}$  into  $\hat{\mathbf{U}}$  and a 3-vector  $\nu$  of the eigenvalues (diagonal).
8. Convert  $\hat{\mathbf{U}}$  into its equivalent unit quaternion representation  $\hat{u}$ .
9. Store the values  $\hat{U}$  and  $\nu$  as the parameters of the QuTEM  $\hat{R}$  and  $\mathbf{v}$ .

## 6.2.4 Estimating Constraint Radius

Estimating a QuTEM support radius using this formulation is simple. The constraint radius is simply the maximum Mahalanobis distance (measured as standard deviations from the mean) of the hemispherized samples in the unconstrained model:

$$\rho = \max_i \text{dist}_{Mahalanobis}(\hat{Q}_i, \hat{M})$$

In other words, the standard deviation of the furthest example is used as the constraint. The density is defined to be zero beyond this range.

## 6.2.5 Summary of QuTEM Parameter Estimation

This section described how to estimate the four parameters of the QuTEM model ( $\hat{M}, \hat{R}, \mathbf{v}, \rho$ ) from example data by using the exponential mapping to transform data into a Euclidean space and then estimating a Gaussian density there using standard techniques. We also described how to handle antipodal symmetry. The next section will describe how to *generate* samples from a QuTEM model given that we know the parameters.

## 6.3 QuTEM Sampling

We have found synthesis of new data from the QuTEM density useful in two main ways:

- Generating new test data for Monte-carlo simulations of algorithms by hand-entry of QuTEM parameters

- Generating new joint orientations similar to example animation data

The former will be used to test our pose blending algorithms in Chapter 7. The latter application was tested for learning a “Perlin noise” model for character motion (see [63]) from example data described in Chapter 10, though more work needs to be done in this area.

Since we used the standard Gaussian vector density at the heart of our model, synthesis is straightforward using standard techniques and the exponential map.

### 6.3.1 QuTEM Sampling Algorithm

We can generate new random quaternions by sampling the tangent Gaussian density and then mapping the result onto  $S^3$  with the exponential map at the mode. Sampling from a vector Gaussian distribution is performed by the Box-Muller method and is covered in Appendix B.

To sample a new unit quaternion from the QuTEM density, first generate a sample in the mode-tangent space, call it  $\mathbf{w}$ , generated from the density with covariance parameters  $\hat{R}$  and  $\mathbf{v}$  from a QuTEM. Since this vector is with respect to the mean  $\hat{M}$ , we can use the exponential map to put it on the sphere at the right spot:

$$\hat{Q} = \hat{M} e^{\mathbf{w}}$$

It is crucial to notice that this tangent sample (unlike the transformed data in the estimation case) will actually exist in the *entire* space of  $\mathbb{R}^3$  (the entire tangent space) and not just in the ball of radius  $\pi/2$ , since the Gaussian has infinite extent. When we wrap it back onto the sphere, the point could potentially end up *anywhere* on the sphere.

Since we are concerned with modelling closed regions on the sphere (maximally, the entire hemisphere around the mode representative), we actually can use a simple rejection method to sample points. For example, to keep the samples on the hemisphere, we can reject sampled tangent points not in a proper-sized ball in the tangent space. Specifically, if the sampled point  $\mathbf{w}$  is outside the solid ball of radius  $\pi/2$ , we can simply throw it out and try again. In general, this method will be efficient since the estimated variances of the tangent space Gaussian will be in this length range since they are estimated from data in this range. Therefore, the majority of sampled points will fall within several standard deviations of the mean which will likely also be inside the solid ball of radius  $\pi/2$ , which can be seen by simple geometry. Therefore, few of these rejected points will be rejected.

We also apply rejection sampling using the constraint radius  $\rho$ . If the generated tangent sample’s Mahalanobis distance is outside of the radius  $\rho$ , the point is thrown out and another tried.

### 6.3.2 Singular Data Woes

This rejection method is sometimes grossly inefficient in practice since the TEM radius really is in terms of standard deviations. For fixed or near fixed joints, which effectively have zero variance in all directions and whose only valid sample is effectively the mode

itself, our estimation procedure would model the joint as a mode with a tiny lower bound variance in all directions (since it cannot be zero), but with a tiny (or even zero) constraint radius. Unfortunately, rejection sampling for these cases is extremely inefficient<sup>3</sup>. To handle these special cases, we use the fact that we *know* that the data is singular (since we have stored the variances explicitly as the weight vector  $\mathbf{a}$ , singular directions have weight 0). Hence, we just do not sample singular components, but rather just set them to zero (the mean in our case).

### 6.3.3 Summary of Synthesis

Sampling is done by the following algorithm:

1. Generate a sample  $\mathbf{w}$  in the tangent space Gaussian distribution using QuTEM parameters and the Box-Muller algorithm.
2. Exponential map  $\mathbf{w}$  onto  $S^3$  to get an identity mean quaternion  $\hat{Q} \leftarrow e^{\mathbf{w}}$ .
3. Map it to the QuTEM mean by rotating the sample by the mean  $\hat{Q} \leftarrow \hat{M}\hat{Q}$ .
4. Return  $\hat{Q}$ .

The sample is rejected if it is outside the constraint radius  $\rho$ .

## 6.4 QuTEM Summary

This chapter presented a simple approach to unit quaternion statistical analysis and synthesis by coupling the quaternion exponential mapping and a standard vector Gaussian probability density. The resulting model, called the QuTEM, models the mean, covariance and finite support region of the density.

The QuTEM is a useful building block. It can be used to:

- Learn a probability density from data
- Sample new points from this density
- Hemispherize data
- Use covariance as a distance metric that “divides out” differing variances

The first three points we discussed in the chapter in some detail. The final point we only quickly touched on since we have not used it in depth in our work, but we feel it will be useful for future work. The main remaining issue is that the Mahalanobis distance is defined between a point and the mean, and not two arbitrary points. It is not immediately clear how to handle this properly. In initial tests we convert the examples into their SMT (scaled mode

---

<sup>3</sup>The author notes that it results in an infinite loop when  $\rho = 0$ , which is very bad.

tangent) descriptions and then used the standard Euclidean norm on the resulting vectors which we think is a reasonable approach which needs to be looked at more carefully.

Now that we can analyze quaternion statistics we need to be able to synthesize new examples with a blend operator as well as a statistical operator. Pose-blending involves the weighted blend of  $n$  quaternions and is covered in the next chapter. We will use the QuTEM to solve certain problems we encountered there, as we shall see.



# Chapter 7

## Multi-variate Unit Quaternion Interpolation

This chapter will describe two new algorithms for performing a weighted blend of  $N$  unit quaternions:

- Slime
- Sasquatch

Such a blending operator is needed to create the *multi-target pose blending* building block we motivated in Chapter 2 in order to interpolate between multiple example animations.

The chapter will proceed as follows:

**Section 7.1** will introduce the problem and discuss some of the properties we desire the operators to have.

**Section 7.2** presents the slime algorithm discusses its properties.

**Section 7.3** presents the sasquatch algorithm, which improves on several problems encountered with slime in practice at the cost of slower performance.

**Section 7.4** describes how these two operators can be used to allow an example-based non-linear vector space function approximation algorithm called Radial Basis Functions (RBFs) to work with unit quaternions.

**Section 7.5** summarizes the chapter and main conclusions.

### 7.1 Problem Description

Formally, we can describe the problem of pose-blending as follows: Given a set of  $N$  example postures,  $\mathcal{P} = \{\hat{P}_i\}$ , and a weight vector  $\mathbf{a} \in \mathbb{R}^N$  whose  $a_i$  component specifies the desired contribution from the  $i$ th posture, create a posture  $\hat{B}$  according to the weight

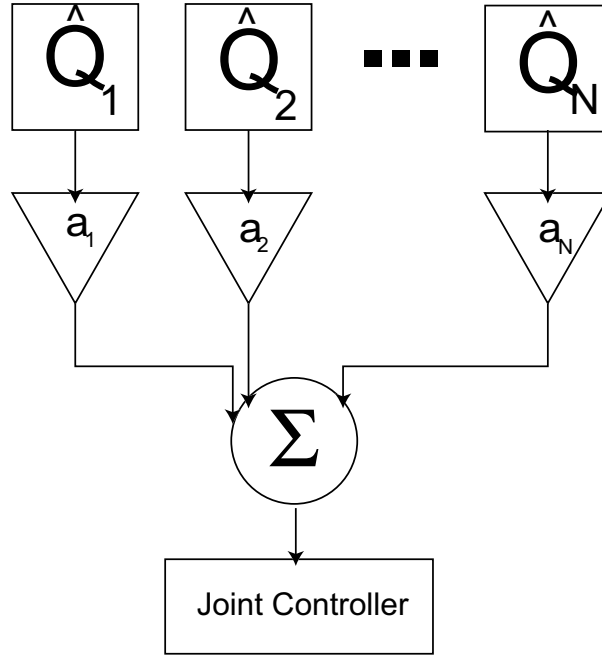


Figure 7-1: An abstract depiction of the unit quaternion blending building block. The algorithm should take  $N$  unit quaternion examples  $\hat{Q}_i$  with associated weights  $a_i$  and perform a weighted sum of them. This answer is then usually written to a joint controller.

---

vector. Figure 7-1 gives an abstract depiction of the problem. We will use the function QWA (Quaternion Weighted Average) to denote such an operator:

$$\text{QWA}(\mathcal{P}, \mathbf{a}) = \underline{\hat{B}}$$

### 7.1.1 Interpolation and Extrapolation

We would like our operators to have the following properties:

- Interpolation
- Extrapolation

The blend function should *interpolate* the examples for the standard basis weight vectors  $\mathbf{e}_i$  (recall,  $\mathbf{e}_i$  has zero entries for all except the  $i$ th, which is 1). Mathematically, the interpolation constraint is:

$$\text{QWA}(\mathcal{P}, \mathbf{e}_i) = \underline{\hat{P}}_i$$

In other words, if the only non-zero weight is on a particular example, we wish to get that example out. Some blending methods only approximate the examples.



Often weight vectors are required to sum to one in order to stay inside the convex hull of the examples, which leads to pure interpolation behavior. If we relax this constraint, we can allow some *extrapolation*, or making a guess as to the answer outside of the region for which we have data. This is useful to make caricatures of motion, such as making a happy walk even more happy. Also, if we have good extrapolation behavior, it is likely that fewer examples will be needed. This is important for leveraging the animator’s skill, as we argued in Chapter 2.

### 7.1.2 Vector Space vs. Spherical Interpolation

Unfortunately, as we saw in Chapter 3, quaternions do not live in a vector space, so the standard linear algebra technique of producing a linear combination of the quaternion examples:

$$\hat{B} = \sum_{i=1}^N a_i \hat{Q}_i$$

will not produce an answer on the sphere. Also, antipodal symmetry implies we need to hemispherize the data, as we saw in Chapter 6.

We can renormalize the solution to get around this, but we would also need to handle antipodal symmetry. Furthermore, this renormalization means that a constant speed change in the weight vector does not produce a constant angular velocity change with respect to the spherical metric (see Chapter 3). This was the original motivation for Shoemake’s famous *slerp* function, which performs a constant angular velocity interpolation of two examples as the single parameter changes with constant speed. We desire this behavior in our extension to  $N$  quaternions. This section chapter will consider two algorithms that were motivated as an multi-variate (more than one interpolation parameter) extensions to *slerp*.

## 7.2 Slime: Fixed Tangent Space Quaternion Interpolation

This section describes the first of our two quaternion weighted blending operators, *slime*.

### 7.2.1 Motivation: Extension of *slerp*

In order to lead into our *slime* algorithm, we first return quickly to Shoemake’s *slerp* function, which served as a starting point for our algorithm. Recall that *slerp* essentially defines a constant angular velocity geodesic curve (great circle) on the quaternion hypersphere in terms of a reference point (the first example) and a second point which defines the direction of the curve from the reference (hence the angular velocity).

The geodesic traced by *slerp* can be parameterized in the exponential form:

$$\text{slerp}(\hat{Q}_0, \hat{Q}_1, t) = \hat{Q}_0 e^{t \ln(\hat{Q}_0^* \hat{Q}_1)} \quad (7.1)$$

or to make the angular velocity portion clear:

$$\text{slerp}(\hat{Q}_0, \hat{Q}_1, t) = \hat{Q}_0 e^{t \omega} \quad (7.2)$$

where  $\omega$  is  $\ln(\hat{Q}_0^* \hat{Q}_1)$ .

In this way, we can think of *slerp* as using the exponential map to represent one quaternion with respect to another (the reference) in terms of the angular velocity of a unit time curve between them. Therefore, the reference quaternion shows up on the left as the “zero point” as well as inside the logarithm. Also, in our formalization, we would actually consider *slerp* as generating only a blend of *one* example, since its single parameter can be thought of as a weight vector of dimension one. Figure 3-7 depicts this graphically.

How do we extend *slerp* to more than one example quaternion to blend at once? The first thing we need to realize is that we will still need a reference quaternion. This reference quaternion is the particular location on the sphere whose tangent space we will be using (we will see below that it is similar to the mean of the vector space interpolation scheme). As we saw earlier, each tangent space on the sphere is a *different space* even though they have similar algebraic structure, so the decision of a reference quaternion is extremely important. We will see that all examples need to be represented in the same tangent space. We shall return to this decision later.

We saw in Section 3.3.4 that the logarithmic map creates a linear vector space (since angular velocities are true vector quantities). For this reason, we can use *map* at a *fixed* reference point to create a *locally-linear* space in which to blend the quaternions with the standard Euclidean weighted sum. The blended tangent space element can then be mapped back onto the sphere with the exponential map.

## 7.2.2 Slime Algorithm Definition

Let us formalize these intuitive notions into an algorithm definition:

**Definition 5** Given a set of  $N$  unit quaternion examples  $\mathcal{Q} = \{\hat{Q}_i\}$ , a reference quaternion  $\hat{P}$ , and a weight vector  $\mathbf{a} \in \mathbb{R}^N$ :

$$\text{slime}(\mathbf{a}; \hat{P}, \mathcal{Q}) = \hat{P} e^{\sum_{i=1}^N a_i \ln(\hat{P}^* \hat{Q}_i)}$$

## 7.2.3 Slime Properties

Several points need to be addressed about *slime*. First, we need to show that it satisfies the interpolation constraints (we drop the parameters for clarity):

**Property 1** *slime satisfies the interpolation constraints*

$$\text{slime}(\mathbf{e}_i) = Q_i .$$

This follows trivially by substituting the standard basis vectors  $\mathbf{e}_i$  into the formula for *slime*— the only contribution is from  $\ln(\hat{P}^* \hat{Q}_i)$  and when exponentiated the reference  $\hat{P}$  and its conjugate cancel leaving  $Q_i$  as desired.

In general, we will not force our weight vectors to sum to one so that we can perform extrapolation on quaternions as well. Therefore, we look at another special case of weight vector, the zero vector  $\mathbf{0}$ , which is the answer we would get if we did not ask for a contribution from any example.

**Property 2** *The zero contribution blend is the reference quaternion:*

$$\text{slime}(\mathbf{0}) = \hat{P}$$

This follows since  $e^0 = \hat{1}$ . This property is useful since it gives us a criterion for choosing a reference quaternion for our blend. Since the zero contribution answer is the reference point, we can choose the reference point by deciding what our “zero contribution” quaternion should be. Slerp clearly chooses the first of the examples as the zero contribution since this is desired for a univariate blend.

### Geometric Interpretation of Slime

Figure 7-2 and Figure 7-3 depict the slime algorithm graphically. The yellow vectors are in the tangent space at the reference quaternion (also yellow for consistency) and illustrate the logarithm of the red geodesics (great circles), or equivalently, the angular velocities of unit time spherical curves from the reference to each to example (in green). The interpolation is performed linearly on the yellow vectors to get a blended angular velocity vector for the curve through the reference, as shown by the orange vector in Figure 7-3. This linearly interpolated velocity can then be integrated back onto the sphere since it describes the orange great circle. By integrating forward unit time, we get the blended example (orange sphere) which lives on the quaternion group.

### Extrapolation Discussion

The geometric interpretation makes it clear how extrapolation works in slime. Consider some weight vector,  $\mathbf{a}$ , which we are using to blend. This vector will specify some particular tangent vector at the reference point. In vague terms, if we want to extrapolate, we want to “keep going in that direction” away from the reference. Since any scalar multiple of  $\mathbf{a}$  will also be in the same direction from the reference as  $\mathbf{a}$ , we see that it lies on the same one-parameter subgroup (great circle) of the sphere. The magnitude of  $\mathbf{a}$  specifies how far along the curve to go. Therefore, as we extrapolate further in the same direction in the tangent space, we are moving along a geodesic in the quaternion group. In other words, as we extrapolate in a straight lines using the exponential map coordinates, we move in a *straight line (great circle) from the reference on the sphere as well*. This property is highly desirable and makes intuitive sense. It is *not* the case with an Euler angle parameterization of the same examples and using a vector space weighted blend, however.

This property is related to the idea of *canonical coordinates of the first kind* (see Sattinger and Weaver [71] or Gallier [23]) in the Lie group theory. Canonical coordinate systems of the first kind are coordinates  $\theta_i$  such that the constant velocity curves  $\theta_i(t) = t\mathbf{a}_i$  in the coordinate system map to one-parameter subgroups when lifted back to the group.

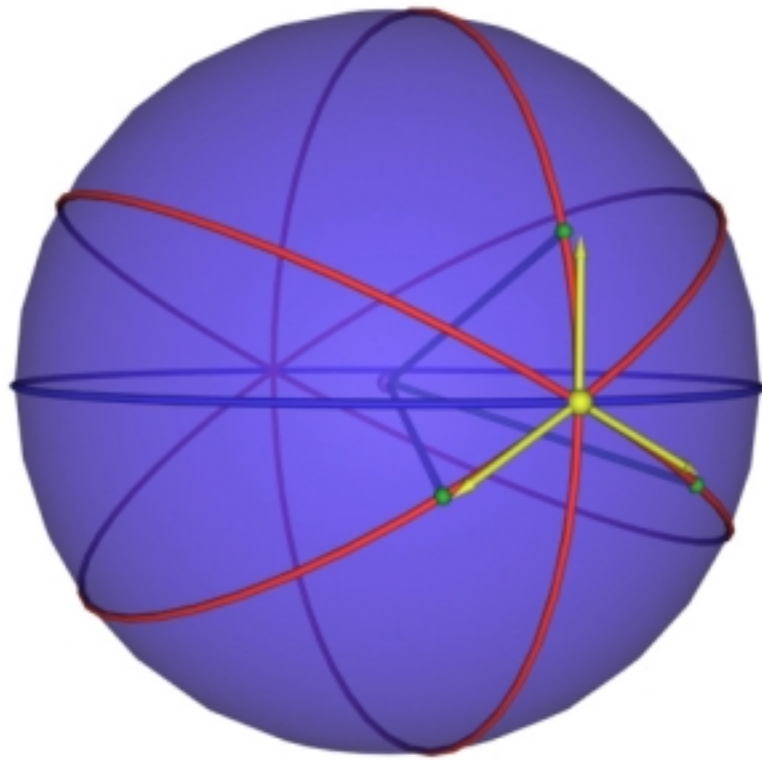


Figure 7-2: The slime algorithm maps examples (green spheres) into tangent descriptions (yellow vectors) with respect to a chosen reference quaternion (yellow sphere) by describing the examples in terms of geodesic curves (red great circles) that pass through the reference and the example. The yellow vectors live in a linear space since they correspond to angular velocities of the curves, and therefore can be blended linearly.

---

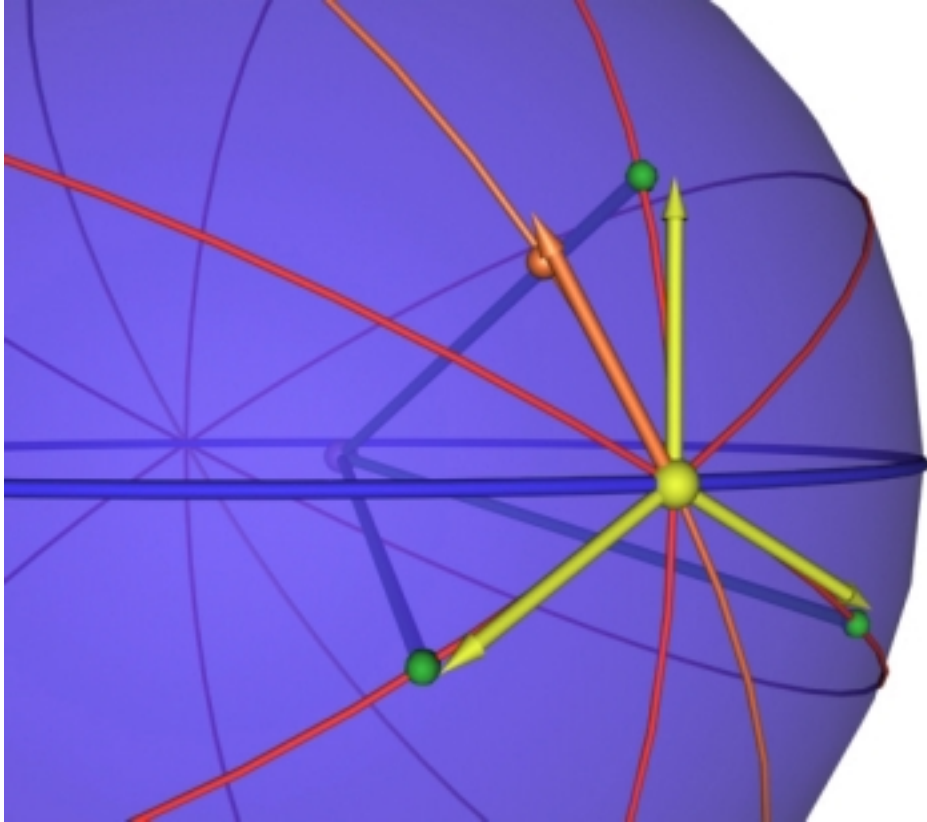


Figure 7-3: The slime algorithm linearly blends the tangent vector description (yellow vectors) of the geodesics (red great circles) of the examples (green spheres). As an example, the orange vector is an arbitrary weighted blend of the three example vectors. This blend is actually specifying a particular different geodesic through the reference point — the orange great circle. By integrating this blended angular velocity forward unit time from the reference (using the exponential), we get the blended quaternion (orange sphere).

---

Clearly, the exponential map (tangent space) coordinate system  $(\theta \hat{\mathbf{n}})$  has this property since any constant velocity line through the origin in the coordinates:

$$\mathbf{w}(t) = \frac{1}{2}t\omega$$

for  $t \in \mathbb{R}$ ,  $\mathbf{w} \in \mathbb{R}^3$ , and  $\omega \in \mathbb{R}^3$ . The coordinates are clearly the individual components of the  $\mathbf{w}$  vector,  $w_i$ . When lifted to the group by exponentiating, we get the familiar geodesic curve:

$$\hat{Q}(t) = e^{\frac{1}{2}t\omega}$$

which as we saw describes the great circle through the identity at  $t = 0$  and with angular velocity  $\frac{1}{2}\omega$ <sup>1</sup>

Euler angle coordinate systems for  $\text{SO}(3)$ , however, do not form a canonical coordinate system. We do not form a proof here, but refer the reader to the Lie group theory. For this reason, however, we can argue analytically that the extrapolation behavior of the Euler parameterization will not be as good since it will not extrapolate along a geodesic from the reference.

To summarize,

**Slime is better at extrapolation than an Euler angle blend since it moves along one-parameter subgroups.**

### Slime Coordinate Singularities

One immediate problem with the slime algorithm is that since it uses a fixed tangent space for blending the quaternions which is only 3-dimensional, we introduce a singularity into the algorithm. In particular, the singularity will occur on the great circle orthogonal to the reference quaternion, since opposite points on this circle are identified in the exponential map. In terms of the tangent space, this occurs in a spherical shell of radius  $\frac{\theta}{2} = \frac{\pi}{2}$ , since the great circle is 90 degrees from the reference.

In practice, the singularity amounts to the slime algorithm being useful only for blending realistic character joints that are not allowed to spin a full 360 degrees in any direction. Since the joint cannot spin 360 degrees, we can map any quaternion path for the joint into a local hemisphere that never crosses the shell by choosing the right quaternion reference. It seems intuitively obvious that the mean of all animation data for the joint would be an optimal choice, since it places the singularity as far from the data as possible. We saw how to calculate this using the QuTEM model in Chapter 6. We discuss this choice further below.

---

<sup>1</sup>We are being a little sloppy with the factor of 2 in the angular velocity terms here. In fact, since we use it merely as a representation here, and not in terms of actual derivatives of curves, we can often ignore this factor, as long as we are consistent.

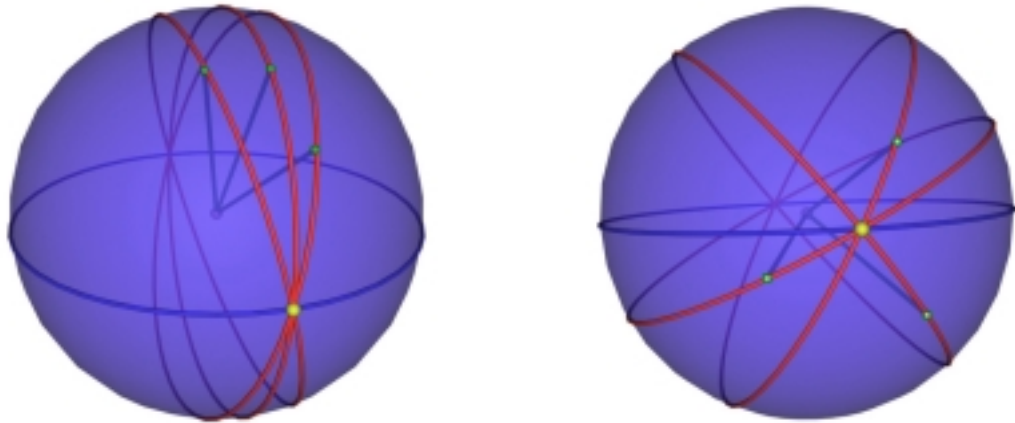


Figure 7-4: Choice of the reference quaternion (yellow sphere) affects the interpolation of examples (green spheres) since quaternions are represented as one parameter subgroups (red great arc circles through the examples and reference) through the reference quaternion. As the choice approaches the “average” of the examples, the curves (examples) become more separated from each other and therefore the interpolation becomes better.

Finally, we mention again that root joints are special. Since the root joint is a rigid body, it *is* allowed to rotate in one direction by 360 degrees. For this joint, we need a different algorithm that avoids this problem. This will motivate our discussion of *sasquatch* below.

### Choosing a Reference Quaternion

Choice of the reference quaternion is important since this quaternion specifies the tangent space which is being used to blend the examples. Different choices will result in different blends for the same weight vector. This property can be clearly seen in Figure 7-4.

So what is a good choice for a fixed reference quaternion? We will consider three choices here:

- One of the examples
- The mean of all examples over all animations
- The identity,  $\hat{1}$

**An Example** The first of these is similar to *slerp*, which uses one of the examples as the reference. Unfortunately, for more than two examples, this does not work as easily. If we were to choose one of the examples as the reference, however, then the example would become the new zero point. This effect occurs because multiplying by the conjugate of the reference before taking the  $\ln$  rotates the examples so that the reference point aligns with the identity element. Since  $\ln(1) = 0$  (we use the vector form for the  $\ln$  result), no blend weight will have an effect on the example — it effectively becomes the origin of the

coordinate system in which the blend occurs, much in the same way as the mean in the vector space case.

This effect might be exactly what is desired, however! If there is a known “neutral” example which could be interpreted as corresponding to the zero weight vector, this could be used as the reference point. In this case the weight vector would not include a component for the reference since this is defined as the blend of the  $\mathbf{0}$  point.

**Mean Over All Animation Data** This argument leads us into a second, much better, choice of reference quaternion of the examples, the mean over all motion examples of the joint <sup>2</sup>. In some sense, this “average” of the rotation examples is an optimal choice since the tangent vector descriptions will be as orthogonal as possible. Also, for a blend weight of zero, the average rotation seems like an intuitively reasonable answer — in the absence of any variation or any explicitly given neutral pose, return the average.

Finally, since the singularity in the logarithmic map occurs on the great circle orthogonal to the map center, choosing the mean as the reference point places the singularity *as far from the data as possible*. Also, since the approximation is better near the map center, we should minimize the average error in the approximation. <sup>3</sup>

**Identity** Lastly, as we discussed in Chapter 5, we can force our animator to use a certain distinguished configuration of the character as the identity posture. This requirement turned out to be useful since it gives us a natural choice for a reference pose for blending, the identity.

What does using the identity as the reference get us? First of all, the blending formula in Equation 5 immediately is simplified

$$\text{slime}(\mathbf{a}; \hat{\mathbf{1}}, \mathcal{Q}) = e^{\sum_{i=1}^N a_i \ln(\hat{Q}_i)} \quad (7.3)$$

since we remove  $N + 1$  quaternion multiplies if  $N$  is the number of examples.

Furthermore, if the coordinate systems of the bones in the skeleton are chosen such that the average pose of the character over its animations *is itself the identity pose*, we have the best of all worlds — we gain the computational efficiency of using the identity as a reference as well as the robustness of using the mean to separate the examples.

## 7.2.4 Summary of Slime

Slime chooses a fixed, global tangent space to blend quaternions using the standard Euclidean weighted average. This algorithm is therefore an approximation which is better around the tangent space quaternion. By choosing the mean over all example orientations of the joint, we can place the resulting singularity as far from the data as possible — in fact,

---

<sup>2</sup>Note that since quaternions do not live in a vector space, we cannot use the normal formula for finding the average of a vector directly. We discussed the average of a quaternion in further depth in Section 6.2.1.

<sup>3</sup>We have proven this empirically by comparing slime and sasquatch solutions over many randomly chosen weights, quaternion examples, and choices of reference quaternion and showed that the minimum average approximation error (difference between the slime and sasquatch solutions) occurs at when the reference point is the mean, as we expected.



for internal joints with a constraint boundary for joint limits this singularity will not be outside the valid probability density for the joint. We can do a “reset transform” on the geometry in order to make the mean over all data be the identity quaternion  $\hat{1}$  by representing the data in the principal bases of the probability distribution as a preprocessing step.

Slime is at the core of most of our successful pose-blending results, which we present in Chapter 10.

Finally, the take-home points about the slime algorithm are:

- Slime is best for internal joints with local compact ranges, and not for rigid body blending since it introduces a singular subspace.
- The singular subspace can be chosen to be as far from the data as possible (orthogonal to the mean over all data) by using the QuTEM mean.
- Slime only approximately extends slerp to more than two quaternions in the sense that constant speed changes in the parameters only lead to constant angular velocity curves if the curve passes through the reference point.
- Slime is fast.
- Since the reference is fixed, we can store preprocess the examples into their log form if they do not change over time. This means we only have to calculate the exponential map and not the logarithmic map for a speed increase. We use this in practice often.

### 7.3 Sasquatch: Moving Tangent Space Quaternion Interpolation

This section describes an iterative extension to slime called sasquatch<sup>4</sup> which removes the singular subspace so that it may be used on fully rotating joints like the root. We will show how a weighted blend on the sphere can be thought of as the steady state solution of a physical system of nails and first-order “springs” pulling a free marble around on the sphere. We will see that the spring constants (relative to each other) can be used as blend weights. The system will converge on a weighted blend that respects the spherical metric, unlike other approaches (including slime).

Slime had some problems with singularities, making it a poor choice for the root joint (which can travel all over the quaternion sphere rather than being locally contained). This may not be a problem for many cases where there is a preferred direction that tends to keep rotation data locally-bunched, as is the case for humanoids that always walk upright or an “up vector” as in a camera. Indeed, we got a lot of mileage from *slime* until the singularity finally was manifested when our virtual dog needed a roll-over animation.

---

<sup>4</sup>Sasquatch actually stands for Spherical “Aristotelian Springs” for QUATernion blending Constrained to a Hemisphere.

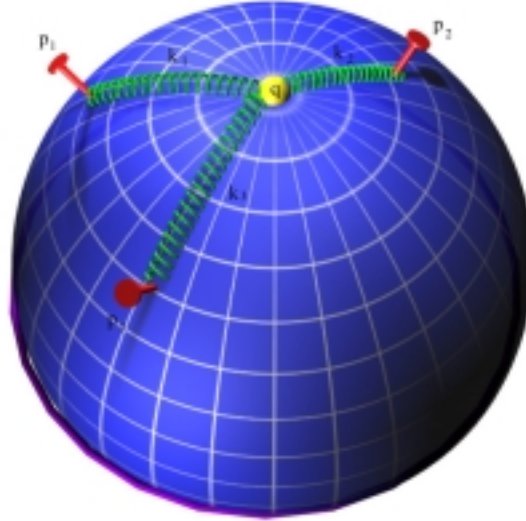


Figure 7-5: The system of Aristotelian springs with constants  $k_i$  connected between the example points (nails),  $p_i$ , and the free point (yellow),  $q$ . All nails must be on the same local hemisphere.

### 7.3.1 Spherical Springs Physical Analogy

A fruitful way to think about the problem of spherical weighted averages is as a physical system of springs on the sphere whose equilibrium point is our desired weighted average. Imagine that the quaternion examples are nails banged into a unit sphere (again, on the same local hemisphere). Further imagine that attached to each nail is a “spring.”<sup>5</sup> The ends of each spring are all attached to a tiny marble, which is free to slide around on the surface of the sphere. The spring constants can be chosen to be the weights of our weighted average. For this work, we will assume that the weights must sum to one (they can be trivially renormalized such that they do by dividing through by their actual sum).

Figures 7-5 and 7-6) illustrate this system. This physical system obviously will apply “forces” to the marble based on the spherical distance between the marble and each nail, weighted by the spring constant. If we set up the system to some initial marble configuration and let it settle to equilibrium, the steady state should be what we desire — the weighted average of the points, inside the convex hull (based on spherical polygons) of the examples. Also, it should be obvious from physical considerations that a solution *must* exist if the example all live within the same local open hemisphere<sup>6</sup>. Also, it should be fairly clear that this solution is unique if the examples all live inside the open local hemisphere.

<sup>5</sup>The quotes on spring will be explained below. As a spoiler, we note that these will be Aristotelian (first-order) springs rather than Newtonian springs for simplicity. See below.

<sup>6</sup>The open hemisphere excludes the thin set of points on the great circle which defines the hemisphere. Clearly, symmetrical examples here could lead to multiple solutions. For example, three point evenly spaced around the great circle with identical weights will have a solution on both poles of the sphere, assuming the great circle is the equator.

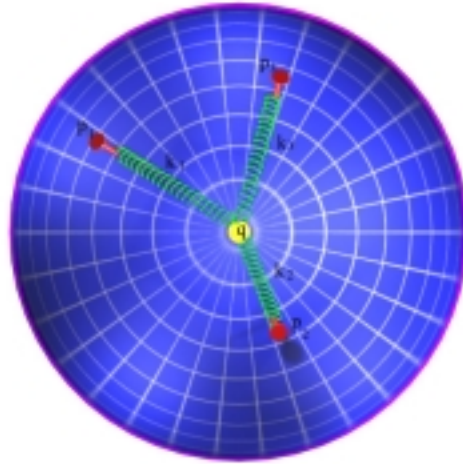


Figure 7-6: An orthogonal projection of the system from above the free point,  $q$ . Note that this is *not* the exponential mapping since orthographic projection does not preserve the spring lengths with respect to the spherical distance metric.

### 7.3.2 Spherical Metric

The first thing we require is a distance metric on the sphere. We already have one with quaternions: in particular, the shortest arclength along the geodesic (great circle for the sphere) between two points is a natural measure of distance between points on the sphere. If all the examples lie on the same hemisphere, we can use the exponential map to simply calculate this spherical distance by converting it into a Euclidean distance. In particular,

$$\text{dist}_{\mathbb{S}^3}(\hat{A}, \hat{B}) = \|\ln(\hat{B}^* \hat{A})\| = \|\ln(\hat{A}^* \hat{B})\| \quad (7.4)$$

where  $\|\cdot\|$  is the familiar Euclidean  $L_2$  metric. It is important to note that this metric is valid only for unit quaternions on a local hemisphere which are up to  $\frac{\pi}{2}$  away from each other on the sphere, as we saw in Chapter 3.

### 7.3.3 Setting Up the System

Now that we have a spherical metric over the hemisphere, we need to find an Ordinary Differential Equation (ODE) for the system. It is well-known that the quaternion derivative is the product of the location of the derivative on the unit sphere (a unit quaternion) and a purely imaginary quaternion which has arbitrary magnitude, as we showed in Chapter 3. This imaginary quaternion can be associated with the familiar angular velocity vector, expressed in either local or inertial coordinates. Formally,

$$\dot{\hat{Q}} = \frac{1}{2}\omega\hat{Q} = \frac{1}{2}\hat{Q}\omega' \quad (7.5)$$

where  $\omega$  is the angular velocity in the global coordinate system and  $\omega'$  is the angular velocity in local (body) coordinates of a rotation in  $\text{SO}(3)$ . The angular velocity is a vector quantity and is purely imaginary — it has no scalar component. Also, it is not a unit quaternion, but has magnitude equal to the angular speed. The multiplication is the standard quaternion multiplication. Therefore, it is clear that the quaternion derivative is also not unit. Rather than using the angular velocity in terms of the rotation group  $\text{SO}(3)$ , we will absorb the  $\frac{1}{2}$  term into the velocity term, which then describes angular velocities in  $\mathbb{H}$  rather than  $\text{SO}(3)$ . Explicitly:

$$\dot{Q} = \dot{Q} = \omega \hat{Q} = \hat{Q} \omega' \quad (7.6)$$

where  $\dot{Q}$  is expressed with respect to the inertial frame coordinates.

How do we find  $\dot{Q}$ ? The unit quaternion  $\hat{Q}$  is the location of the local coordinate system, which in our case is the location of the marble on the sphere. Therefore, we need to find the angular velocity in terms of our nails and springs. In Aristotelian physics, the angular velocity is proportional to the displacement. Rewriting Equation 7.6, we get the *tangent operator* [59]:

$$\hat{Q}^* \dot{Q} = \omega' \quad (7.7)$$

which is a vector quantity. Since each nail and spring will pull independently on the marble, we can simply sum the contributions to the local angular velocity of the marble from each of the nails. To formalize this, let:

$$\omega'_m = \sum_{i=1}^N \omega'_i \quad (7.8)$$

where  $\omega'_m$  is the local angular velocity of the marble.

We need to calculate the local angular velocity contribution from each nail. To be explicit, let the quaternion example points (nails) be labelled as  $\hat{P}_i$ . Let the weight for the  $i$ th spring be  $k_i$ . The distance between  $\hat{Q}$  and  $\hat{P}_i$  is simply found by our spherical distance metric from Equation 7.4. The exponential map gives us tangent vectors anchored at the center of the map (in this case the point  $\hat{Q}$ ) in the local coordinate system whose lengths are the spherical distance. Therefore, we can simply weight the tangent vector with the spring gain:

$$\omega'_i = k_i \ln(\hat{Q}^* \hat{P}_i) \quad (7.9)$$

to get the angular velocity contribution from the  $i$ th nail. The magnitude of  $\omega'_i$  is

$$\|\omega'_i\| = \|k_i \ln(\hat{Q}^* \hat{P}_i)\| = k_i \|\ln(\hat{Q}^* \hat{P}_i)\| = k_i \text{dist}_{S^3}(\hat{Q}, \hat{P}_i) \quad (7.10)$$

which makes explicit that the magnitude of the force is actually the spherical displacement of the marble from the nail, weighted by the spring constant.

To get the total local angular velocity, we add up the contributions, giving us:

$$\omega'_m = \sum_{i=1}^N k_i \ln(\hat{Q}^* \hat{P}_i) \quad (7.11)$$

Now we can substitute into Equation 7.7 and rearrange to get the final quaternion ODE

$$\dot{\hat{Q}} = \hat{Q} \sum_{i=1}^N k_i \ln(\hat{Q}^* \hat{P}_i) . \quad (7.12)$$

which we need to solve for its steady state solution (as  $t \rightarrow \infty$ ).

### 7.3.4 Solving the System for Steady State

To find the steady state solution, we can use two methods:

- Renormalized Euler integration in  $\mathbb{R}^4$
- Euler integrate the angular velocity vector using the exponential and logarithmic maps

The first is the standard Euler numerical integration formula

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \dot{\mathbf{x}}_t \Delta t$$

which will always step off the sphere since the derivative is tangent to it. To handle this, the quaternion is simply renormalized after each step (see, for example, [62] for more details).

The second approach is to Euler integrate the angular velocity using the exponential mapping:

$$\hat{Q}(t + \Delta t) = \hat{Q}(t) e^{\omega'(t) \Delta t} . \quad (7.13)$$

where the  $\omega'(t)$  is the local angular velocity, which we can be found from the derivative with the tangent operator.

We discuss the numerical solution of Quaternion ODE's in more detail in Appendix C.) Since the intrinsic solution takes larger steps and stays within the group, we choose to use it for efficiency and elegance of the algorithm. To solve for steady state, we simply take as large step sizes ( $\Delta t$ ) as possible until the solution converges.

### 7.3.5 Sasquatch Algorithm

Let  $\mathcal{P} = \{(\hat{P}_i, k_i)\}$  be the set of pairs of quaternion points and their weight. The sasquatch algorithm then proceeds as follows:

```

Sasquatch( $\mathcal{P}, \hat{Q}_0, \epsilon, S, \Delta t$ )
{
  1. Let  $\hat{Q} = \hat{Q}_0$ 
  2. Hemispherize( $\{\hat{P}_i\}$ ) to lie nearest to  $\hat{Q}$ 
  3. Loop with  $j = 1$  to  $S$ :
    (a) Let  $\omega_j = \sum_{i=1}^N k_i \ln(\hat{Q}_{j-1}^* \hat{P}_i)$ .
    (b) Let  $\hat{R}_j = e^{\omega_j \Delta t}$ .
    (c) Let  $\hat{Q}_j = \hat{Q}_{j-1} \hat{R}_j$ 
    (d) If  $\|\omega_j\| < \epsilon$  then EndLoop.
  4. Return  $\hat{Q}_j$  as the blend.
}

```

The algorithm also has several free parameters —  $\Delta t$ ,  $S$ , and  $\epsilon$ , as well as the initial starting point for the ODE,  $\hat{Q}_0$ . We discuss the choice of each in turn.

### Precision Parameters

Both  $S$  and  $\epsilon$  refer to how long the iteration continues. The iteration stops if it has run too many iterations (up to a maximum of  $S$  times), or when the magnitude of the update (the angular speed) on the iteration falls below  $\epsilon$ . Therefore,  $\epsilon$  can be used to set the number of significant digits desired in the answer, or  $S$  can be used to force an explicit maximum on the number of computations, which is important for real-time applications.

### Timestep Choice

The parameter  $\Delta t$  is the Euler integration timestep. Normally, we would want this to be small to reduce errors in the calculation, particularly in the case where the trajectory of the system, denoted in our algorithm by the series of quaternions  $\hat{Q}_j$ , is desired. Since we are only interested in the steady state solution, we want this step to be as large as possible *while still maintaining convergence to the steady state*. This fact implies that there exists for each system there exists some  $\Delta t$  which will converge as quickly as possible to the correct answer. The analytic calculation of this value is beyond the scope of this document, but we can find a good value by testing convergence speeds over many random ensembles and finding the fastest value on average. We describe this experiment in Section 10.3.1. We found that a value of about 1.175 gave the best performance.

### Choice of Initial Value

Speed of convergence to steady state, as well as potentially convergence itself, will depend on where the system is started. If we are close to the steady state already, we should

take much less iterations to converge within precision than if we start much farther away. Minimally, since we constrain the spring constants to sum to unity and be non-negative, the solution obviously must exist inside the convex hull of the examples (where the convex hull is defined as a spherical polygon). Therefore, our initial choice should lie within this hull. Since we are seeking the spherical analogue of the Euclidean weighted average, a fine starting point is simply the Euclidean weighted average of the points in the embedding space, using the spring constants as the weights, renormalized onto the sphere. Formally, we define

$$\hat{Q}_0 = \frac{\sum_{i=1}^N k_i \mathbf{p}_i}{\|\sum_{i=1}^N k_i \mathbf{p}_i\|} \quad (7.14)$$

where  $\mathbf{p}_i \in \mathbb{R}^4$  is the unit quaternion  $\hat{P}_i$  interpreted as a vector quantity.

### 7.3.6 Interpolation and Extrapolation

Since we explicitly force our weight vector to sum to unity, `sasquatch` can only interpolate examples and not extrapolate, which is a drawback. One potential way around this might be to use `slime` at the mean of the examples to explicitly extrapolate a new set of examples which can then be passed into `sasquatch`.

### 7.3.7 Convergence Results

`Sasquatch` converges linearly, which should be clear since we use only a first derivative. Also, `sasquatch` should always converge for a proper choice of timestep since all weights are positive and there are no “corners” to get stuck on.

We will present empirical convergence results of `sasquatch` in Chapter 10. There we will demonstrate:

- An optimal choice of timestep based on empirical data collected over many ensembles.
- The property that `sasquatch` reduces to the same answer as `slerp` for the case of two examples.
- Several plots of some attractor trajectories to visualize convergence behavior.

### 7.3.8 Interpolation Visualization

In order to visualize the output of `sasquatch`, we created a 2-dimensional *orientation field*, where each point in a square has a unique orientation associated with it. We used four examples to create a square interpolation space, with the examples on the corners. Then, we used the `Sasquatch` algorithm to interpolate orientations between the examples. The space of weight vectors for four examples is four-dimensional, but we can reduce it to a two dimensional field by parameterizing the weights according to a monotonic function of

the distance to the examples. Specifically, the weight value at a point  $\mathbf{x} = (x, y)$  in the square maps to the following weight vector:

$$w_i = \text{Clamp}(1 - (\|\mathbf{x} - \mathbf{c}_i\|))$$

where  $\mathbf{c}_i$  is the 2-D location of the center of the  $i$ th example on the square. For our example, the corners are the corners of a unit square:  $\{(0, 0), (1, 0), (0, 1), (1, 1)\}$ . The distance metric is the standard Euclidean  $L^2$  norm. Clamp clamps the value of the weight to be non-negative — negative weights are forced to zero. Finally, we enforce the unity sum on weights by dividing through by the actual sum. The results of this applied to a game die are shown in Figure 7-7.

We show how Sasquatch can be used to blend entire postures in Chapter 10.

### 7.3.9 Summary of Sasquatch

This section described `sasquatch`, a new algorithm for calculating a weighted blend of  $N$  unit quaternions. Sasquatch is an iterative algorithm based on the steady state solution of a differential equation. We showed how to set up the equation and how to solve it. We presented the algorithm and discussed the choice of the free parameters. We discussed interpolation and extrapolation behavior and showed that the solution is also rotationally-invariant. We then visualized the output of `sasquatch` on a rigid body.

To summarize the main points:

- Sasquatch can currently only interpolate data points and needs to be extended to extrapolate.
- Sasquatch produces a solution that respects the spherical metric since it explicitly minimizes the weighted distance to each example in the calculation of the steady state (minimum energy solution).
- Sasquatch is rotationally-invariant.
- Sasquatch can handle joints which vary over all  $S^3$  so is suitable for root joints and rigid bodies.
- Sasquatch is iterative, so is not as fast as `slime`.

The next section gives an overview of how to use both `slime` and `sasquatch` with a non-linear function approximator rather than a simple fixed weight combination of the examples.

## 7.4 QuRBF's: Quaternion-Valued Radial Basis Functions

Now that we have two ways of performing a weighted blend on the quaternion hypersphere, we can extend standard linear algorithms using this “pseudo-linear” blending function. Here we will describe how the standard vector space Radial Basis Function (RBF) function



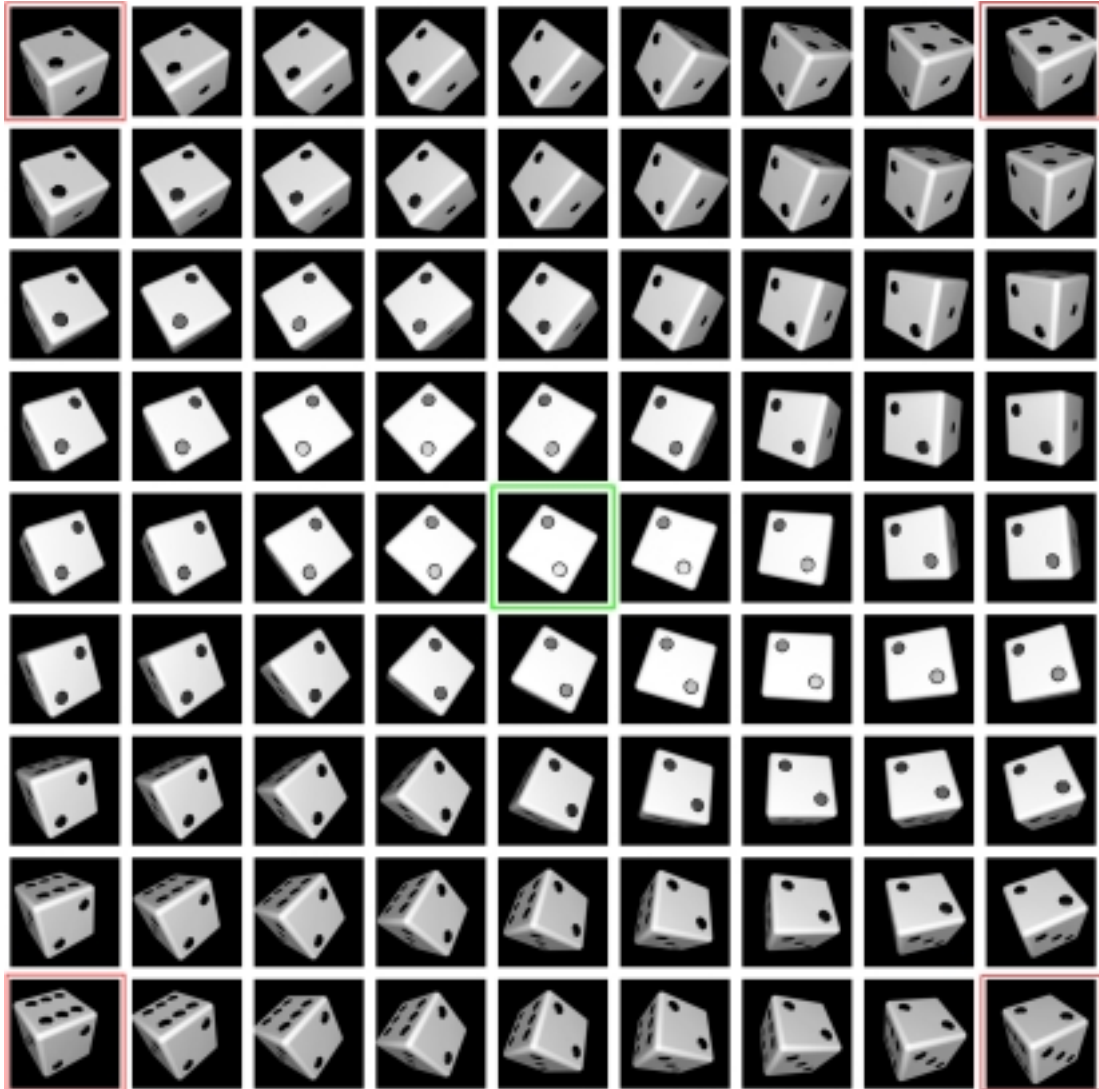


Figure 7-7: A 3D orientation field specified as a radial basis function around the examples (corner locations boxed in red). The weight of each example is inversely proportional to its distance from the sample point in the field as described in text. The center image clearly has equal weights on all examples, and is therefore the centroid of the four examples with respect to the spherical metric.

---

approximation architecture (see [8] or [24] for an introduction to RBF's) can be simply extended to quaternion-valued functions, rather than assuming the output space is a vector space and renormalizing.

Formally, we propose a function approximator  $f$  of the form:

$$f : \mathbb{R}^N \rightarrow \mathbb{H}$$

To motivate this discussion, we first quickly show the scalar RBF, then the obvious Euclidean vector-space extension, followed by our extension to quaternions using our sasquatch algorithm.

### 7.4.1 Scalar RBF

The standard formulation of RBF's seeks a function approximator which is a weighted sum of monotonic functions of the distance to each example, similarly to the method we used to make 7-7. Each basis function is centered on one of  $K$  scalar input examples (here  $x_i$ ), each of which has an associated observation of the true function,  $y_i$ . This gives us:

$$y = f(x) = \sum_{i=1}^K a_i B(x - x_i) \quad (7.15)$$

where  $B(r)$  is the radial basis kernel function. For our purposes, we usually use a Gaussian function for  $B(r)$ :

$$B(r) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{r^2}{2\sigma^2}}$$

where the characteristic width of the Gaussian (where it falls near to zero) is specified by  $\sigma$ . The width can be chosen in several ways. We use the simple heuristic of finding the average pair-wise distance between all input examples and using  $\frac{2}{3}$  of this value, which seems to work well in practice. Using this value, the influence range of the basis function usually falls to zero on near other examples, which is the behavior we desire.

Since the basis functions are given parameters of the algorithm, as well as the example centers  $x_i$ , we seek a vector of weights  $a_i$  which satisfy the interpolation constraints (observation of the function):

$$f(x_i) = y_i$$

Plugging each of these constraint equations into Equation 7.15 gives us the system of equations

$$y_i = \sum_{j=1}^K a_j B(x_j - x_i)$$

which leads to the symmetric matrix equation:

$$\mathbf{B}\mathbf{a} = \mathbf{y} \quad (7.16)$$

where  $\mathbf{y}$  is the vector of output observations  $y_i$ ,  $\mathbf{a}$  is the vector of unknown basis weights  $a_i$  and  $\mathbf{B}$  is a symmetric matrix of pairwise basis functions values between the  $i$ th and  $j$ th examples. In other words,  $B_{ij} = B_{ji} = B(x_i - x_j)$ . We invert this system using standard techniques to solve for  $\mathbf{a}$  that will interpolate the observations. Notice that in the case of over- or under-constrained systems, where we choose to have more or less basis functions than the number of examples (for memory compression, reduction of overfitting, etc), we get a rectangular system to solve. Standard practice is to use a Singular Value Decomposition (SVD), or pseudo-inverse, on the system of equations to find the least squares or minimum norm solutions for  $\mathbf{a}$ .

## 7.4.2 Vector-Valued RBF

The extension from scalar to vector-valued RBF's is fairly straightforward. Formally, we seek a vector-valued function  $\mathbf{f} : \mathbb{R}^N \rightarrow \mathbb{R}^M$  where  $N$  is the input dimension and  $M$  the output dimension. Again, we are given a set of  $K$  observations of the form  $\{(\mathbf{x}_i, \mathbf{y}_i)\}$  which we must interpolate. Since the bases are radial in input dimension, we assume a distance metric on the input vectors. Here, we shall assume the standard Euclidean norm,  $\|\mathbf{x} - \mathbf{y}\|$ . Notice that any radial distance function on inputs may be used here, as long as it is monotonic.

In order to approximate vector outputs, the system is decomposed (since it is a vector space) into a weighted sum over a basis for the output space. In other words, a separate *component* function approximator is learned for *each* output dimension. In other words, we assume the vector function  $\mathbf{f}$  is of the form:

$$\mathbf{f}(\mathbf{x}) = f_i(\mathbf{x})\mathbf{e}_i \quad (7.17)$$

where  $\mathbf{e}_i$  is the standard basis for  $\mathbb{R}^n$  where the vector is 1 for the  $i$ th component and 0 otherwise. Now we need only learn  $M$  scalar RBF's: one for each output basis vector, designed to interpolate just the  $i$ th output component. Hence, we solve  $M$  systems of the form of Equation 7.16 and assemble the results as a weighted sum over basis elements.

## 7.4.3 Quaternion-Valued RBF's with Slime

Most of our early work in posture blending used the exponential mapping at near the mean or identity pose in order to locally-linearize rotations. The complete algorithm is straightforward given the primitives we have:

1. Hemispherize the data output examples  $\{\hat{Q}_i\}$  to align with the QuTEM mean representative.
2. Logmap the data into a 3-vector in the tangent space at the mean:  $\tau \leftarrow \ln(\hat{M}^* \hat{Q}_i)$ .
3. Learn a standard vector-valued RBF  $f(\mathbf{x}_i) = \tau_i$  from the inputs into the transformed, linearized examples  $\{\tau_i\}$ .
4. Use the RBF to approximate a new  $\tau = f(\mathbf{x})$  based on the query  $\mathbf{x}$ .
5. Exponential map the vector back onto the sphere at the mean:  $\hat{Q}(\mathbf{x}) \stackrel{\text{def}}{=} \hat{M} e^\tau$

An advantage of this approach is that it is fast for internal joints. Another advantage is that all of our data examples live in a fixed 3-dimensional output space for each joint, and therefore  $3n$ -dimensional space for a character with  $n$  joints. The method does not depend on the number of examples,  $k$ .

This work was first presented at SIGGRAPH 1999 in a Technical Sketch [45] and has been used in several successful installations which we describe in Chapter 10.

#### 7.4.4 Quaternion-Valued RBF's with Sasquatch

To simply extend RBF's to work with the more general sasquatch blending operator on quaternions, we need to learn the *weights* on the sasquatch blend from the inputs.

We can express a quaternion-valued function as simply a quaternion weighted sum over these examples using sasquatch. Formally, we seek a function to approximate a set of examples  $\{(\mathbf{x}_i \in \mathbb{R}^N, \hat{Q}_i \in \hat{\mathbb{H}})\}$ . We express this function as a quaternion weighted sum over some set of quaternions which we feel span the output space as the weights in sasquatch vary.

$$\hat{F}(\mathbf{x}) = \text{sasquatch}(f_j, \hat{Q}_j) \quad (7.18)$$

where we have defined our component-functions as calculating the weight on the  $j$ th output “basis” quaternion. Each  $f_j$  is simply another scalar RBF which we calculate in the standard manner as a weighted sum over the *input* basis functions:

$$f_j(\mathbf{x}) = \sum_{i=1}^K a_{ij} B(\|\mathbf{x} - \mathbf{x}_i\|)$$

Putting this all together gives us equations of the form:

$$\hat{F}(\mathbf{x}) = \text{sasquatch}\left(\sum_{i=1}^K a_{ij} B_i(\mathbf{x}), \hat{Q}_j\right)$$

Plugging in constraint equations over the  $K$  examples  $(\mathbf{x}_k, \hat{Q}_k)$  gives us the system:

$$\hat{Q}_k = \hat{F}(\mathbf{x}) = \text{sasquatch}\left(\sum_{i=1}^K a_{ij} B_{ik}, \hat{Q}_j\right)$$

where we write  $B_{ij}$  again for  $B(d(\mathbf{x}_i, \mathbf{x}_j))$ , and  $d(\mathbf{x}, \mathbf{y})$  is a monotonic distance function over the input space, which we have taken to be Euclidean for now.

To solve this non-linear function, we will assume that the output space is covered by our examples, and hence use all the output examples as a basis for the output space. In this case, since we are using an RBF to calculate the sasquatch weight for each of these output examples, we can say that we wish the following hold:

$$\sum_{i=1}^K a_{ij} B_{ik} = \delta_{ik}$$

where  $\delta_{ij}$  is the Kronecker delta and is 1 if  $i = k$  and 0 otherwise. This constraint says that we want the weight on the  $k$ th output example to be 1 if the input is the  $k$ th input example,  $\mathbf{x}_k$ , and zero for all other output examples. This system can be written in matrix notation as

$$\mathbf{AB} = \mathbf{I} \tag{7.19}$$

which states that our weights are the inverse of the example weight matrix  $\mathbf{B}$ . Since  $\mathbf{B}$  is symmetric positive semi-definite, it might be singular. We can invert this matrix using the standard SVD pseudo-inverse techniques (see Strang [81]).

## Algorithm

To summarize the algorithm:

1. Learn a standard RBF mapping from input examples  $\{\mathbf{x}_i\}$  to real scalar weights  $w_i$  on each of the  $K$  quaternion output examples  $\{\hat{Q}_i\}$  such that the weight (contribution)  $w_j$  of the  $j$ th output quaternion  $\hat{Q}_i$  on input example  $\mathbf{x}_i$  is  $\delta_{ij}$  (the Kronecker delta).
2. Interpolate a new weight  $w_j$  for each output example  $j$  with the RBF given a query point  $\mathbf{x}$ .
3. Blend the examples together using sasquatch with the interpolated weights.

## Discussion

An important problem with this formulation of RBFs with Sasquatch is that the problem scales linearly in the number of examples rather than being constant as in the slime-based RBF. Unfortunately, we have found this makes it intractable for reasonably-sized problems. More research needs to be done here.

## Interpolation Results on Posture

Figure 7-8 illustrates an example of a Sasquatch RBF with two input dimensions applied to a walk cycle of a dog — left/right turning and a happiness value. Six examples define the convex hull of this space, as is required of sasquatch. These are: happy-left, happy-right, sad-left, sad-right, happy-straight and sad-straight.

Although we have done these early tests on sasquatch, it has not been used in a production system yet.

### 7.4.5 Quaternion Inputs

It should be clear that the spherical metric from Equation 7.4 is in fact radial and monotonic. Hence, we can use it as a distance function in our basis function as well! This extension allows us to approximate functions of the form:

$$f : \hat{\mathbb{H}} \rightarrow \hat{\mathbb{H}}$$

Although we have not used quaternions as inputs in practice, we feel that they will be useful for an example-based learning approach too inverse kinematics since the current posture of the character can be used as an input to the approximator.

## 7.5 Summary of Weighted Quaternion Blending

In this chapter, we described two new algorithms for blending  $N$  unit quaternions according to a weight vector, slime and sasquatch. We demonstrated the following about slime:

- It is fast (constant time).
- It extrapolates well.
- It has a singular subspace and therefore should be used for locally compact data.
- It only approximately respects the spherical distance metric, but is much better than an Euler angle interpolation.

We also discussed the following about sasquatch:

- It converges linearly.
- It is valid anywhere on the sphere since it does not require a fixed tangent space (i.e. it has no singularities).
- It reduces to slerp in the case of two examples.
- It respects the spherical distance metric.

We then showed how to implement quaternion Radial Basis Functions (RBF) using both slime and sasquatch. We present results on using both of these RBF approaches in Chapter 10. The slime version has proven useful in practice, but the sasquatch version needs to be investigated further.

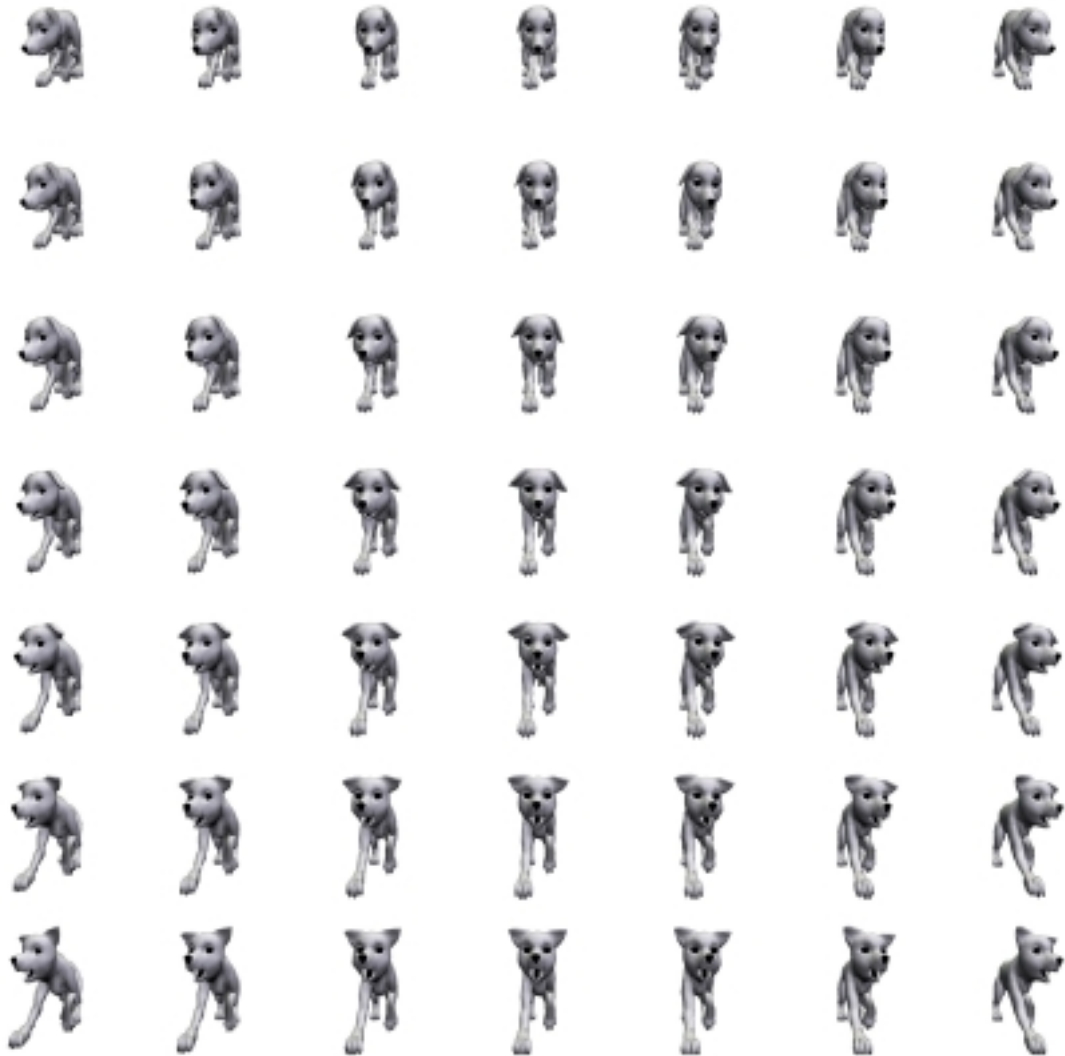


Figure 7-8: A Sasquatch RBF of a parameterized walk cycle with two input dimensions, happiness and turning radius. All images are sampled at the start of the walk cycle and the RBF is sampled evenly in all directions. The image was created with six examples, four on the corners and two for normal-left and normal-right.

---





## Chapter 8

# Eigenpostures: Principal Component Analysis of Joint Motion Data

This chapter will describe the Principal Component Analysis (PCA) algorithm for finding a subspace for the inherent variations in a set of data. We will call the basis vectors of this subspace *eigenpostures* after Turk’s “eigenfaces” paper [85] which used PCA on recognition of face images.

The chapter proceeds as follows:

**Section 8.1** will motivate the reasons for desiring such a subspace.

**Section 8.2** will present the standard algorithm which assumes Euclidean data.

**Section 8.3** describes how to use the QuTEM model and quaternion exponential mapping to process motion data for use in the standard PCA algorithm.

**Section 8.4** presents results from an initial evaluation of the technique on our corpus of dog animation.

## 8.1 Motivation for Posture Subspace Analysis

One large problem with computational motion engines is that animation can take up a lot of memory as more expressivity, and therefore more examples, are required. Storing all these examples can be prohibitively expensive, so a means of compression or *dimensionality reduction* would be useful for reducing this memory footprint. If we can find an invertible transformation of our data to some smaller dimension space, we can use it as a lossy compression method. Furthermore, if we want to perform learning on the motion data, having a smaller dimension learning space will make the learning faster. Additionally (as we discuss further in Chapter 9), if we had such a basis we could use it to project a novel posture created in some procedural manner (such as inverse kinematics or a learning algorithm) onto the basis in order to bring it “closer to the data” in order to minimize “unnatural” postures that these algorithms sometime generate.

## 8.2 Principal Component Analysis Overview

*Principal component analysis* (PCA) is a simple and powerful unsupervised method for performing dimensionality reduction on a collection of vector data [8, 24, 85]. PCA essentially finds a set of orthonormal basis vectors (called the principal components) which define a linear subspace of the data space which models the intrinsic variations in the data. Once such a basis is found, each data vector is then *projected* onto this subspace to create a corresponding *weight vector* with the dimension of this subspace. The full dimension data vector can then be reconstructed by a linear combination of the basis vectors using the weights. Notice that this reconstruction will be exact if the dimension of the data is the same as the subspace. For PCA to be useful, however, the dimension of this subspace (number of basis vectors) must be smaller than the dimensionality of the data vectors. This approach leads to a lossy compression of the data, with a residual error between the reconstructed vector and original data vector. The goal of PCA is to find the set of basis vectors which minimizes this reconstruction error (in a least squares sense). In this way, PCA can be used to approximate the inherent dimensionality in the data and find an encoding which consists of a small set of basis vectors and a (smaller) weight vector for each example.

It can be shown that the eigenvectors of the sample covariance matrix of the data form exactly such a basis, with their corresponding eigenvalues giving a measure of the magnitude of the variation in that direction [8]. Therefore, by ordering the eigenvectors according to their eigenvalues, we can choose a subset of eigenvectors which lead to an acceptable reconstruction error.

### 8.2.1 Mathematical Description

Mathematically, let  $\{\mathbf{x}_i\}$  be a set of  $N$  data vectors in  $\mathbb{R}^D$ . Let  $\bar{\mathbf{x}}$  be the mean of the data and  $\mathbf{y}_i = \mathbf{x}_i - \bar{\mathbf{x}}$  be the zero-mean versions of the data vectors. Let  $\mathbf{Y} = [\mathbf{y}_1 \ \mathbf{y}_2 \ \cdots \ \mathbf{y}_N]$  be the data matrix with the zero-mean data in its columns. Recall that the sample covariance matrix is then

$$\mathbf{K} = \frac{1}{N-1} \mathbf{Y} \mathbf{Y}^T$$

and will be  $D \times D$  as well as symmetric positive definite. Let  $\mathbf{u}_j$  be the eigenvector of  $\mathbf{K}$  with corresponding eigenvalue  $\lambda_j$  such that  $\lambda_j > \lambda_{j+1}$  (lower indices are larger eigenvalues).

**Projection** Now choose some number  $M < D$  of eigenvectors to form an orthonormal basis for the projection subspace. Any new vector  $\mathbf{x}$  (either in the original dataset or a new query point) can be projected onto this subspace using the inner product to find the weight of the example in that direction. Thus,

$$w_k = \mathbf{u}_k^T (\mathbf{x} - \bar{\mathbf{x}})$$

gives the weight  $w_k$  of the example  $\mathbf{z}$  on the eigenvector  $\mathbf{u}_k$ . Finding the weight for each of the  $M$  subspace vectors gives a weight vector  $\mathbf{w} \in \mathbb{R}^M$  for the example  $\mathbf{z}$ . Since  $M < D$ ,

this transformed example will take up less memory than the original example  $\mathbf{z}$ , which has dimension  $D$ .

**Reconstruction** In order to reconstruct from a weight vector  $\mathbf{w} \in \mathbb{R}^M$  back into a vector in the original space (call it  $\mathbf{x} \in \mathbb{R}^D$ ), a simple linear combination of the eigenvectors (plus the stored mean) is used:

$$\mathbf{x} = \bar{\mathbf{x}} + \sum_{k=1}^M w_k \mathbf{u}_k .$$

**Reconstruction Error** Finally, we can check the *reconstruction error* of a particular data vector  $\mathbf{x}$  by projecting it onto the subspace and then reconstructing it and finding the residual. Let  $\mathbf{x}'$  be the reconstruction of the vector  $\mathbf{x}$ . Then the residual is simply the Euclidean norm between the two:

$$r = \|\mathbf{x} - \mathbf{x}'\| .$$

**Finding the Basis Dimension** Reconstruction error is useful for calculating the number  $M$  of eigenvectors to use. An average error measure over the entire data set is computed for increasing values of  $M \leq D$  until the error falls below a certain threshold, specified as a parameter. In practice, this error curve is drops exponentially as  $M$  increases, reaching zero when  $M = D$ .

### 8.2.2 Standard PCA Algorithm Summary

To summarize, the standard PCA algorithm proceeds as follows. Again, let  $\{\mathbf{x}_i\}$  be the  $N$  example data vectors.

1. Calculate the sample mean  $\bar{\mathbf{x}} = \frac{1}{N} \sum \mathbf{x}_i$ .
2. Subtract off the mean from the examples to get  $\mathbf{y}_i = \mathbf{x}_i - \bar{\mathbf{x}}$ .
3. Arrange the zero-mean data  $\mathbf{y}_i$  in the columns of a matrix  $\mathbf{Y}$ .
4. Create the sample covariance matrix  $\mathbf{K} = \frac{1}{N-1} \mathbf{Y} \mathbf{Y}^T$ .
5. Find the eigenvectors  $\mathbf{u}_k$  and eigenvalues  $\lambda_k$  of  $\mathbf{K}$  such that  $\lambda_k > \lambda_{k+1}$ .
6. Choose some number  $M$  of eigenvectors to serve as the basis using some in-sample error metric.
7. Return the sample mean  $\bar{\mathbf{x}}$  and the  $M$  eigenvectors  $\mathbf{u}_k$ .

Again, we note that once the basis and sample mean is found, projection and reconstruction are simple linear operations.

## 8.3 PCA on Posture

The last section described the standard PCA algorithm. One issue with PCA is that it is a linear algorithm, assuming Euclidean data and trying to find a linear transformation of the data which best describes the inherent degrees of freedom in the data. As we noted above, quaternions are not Euclidean, so naive use of the PCA algorithm on quaternion-valued vectors (such as used to represent posture) can lead to strange behavior, as the author discovered in practice. This section will describe how to use the quaternion mean of the data described in Section 6.2.1 and the exponential mapping to perform PCA on quaternion-valued data.

### 8.3.1 Eigenposture Algorithm

Say that we have a set of  $N$  postures of a character  $\{\hat{P}_i\}$ , each with  $M$  joints. Let  $\hat{P}_{ij}$  denote the  $j$ th joint of the  $i$ th example posture. Our “quaternion-ized” PCA algorithm then proceeds as follows:

1. For each joint  $j$ , find the sample quaternion mean  $\hat{M}_j$ .
2. Hemispherize all examples to lie on the choice  $\hat{M}_j$ .
3. For each data point  $i$  and each joint  $j$ , perform the logarithmic map at the mean to get a vector  $\mathbf{q}_{ij} = \ln(\hat{M}_j^* \hat{P}_{ij})$ .
4. For each data point  $i$ , collect the linearized joint vectors  $\mathbf{q}_{ij}$  into a block column vector  $\mathbf{x}_i$  containing the log-mapped components for all joints in order.
5. Perform standard PCA on the linearized data  $\mathbf{x}_i$  to get a basis of eigenvectors  $\mathbf{u}_k$ .
6. Return the linear basis  $\{\mathbf{u}_k\}$  and mean posture  $\hat{M}$  (quaternion-valued).

Note that the quaternion mean is found as specified in Section 6.2.1 and the data hemispherized to handle double-covering. The algorithm simply transforms the quaternion-valued posture tuple into a “linearized” *posture vector* which PCA is then carried out on in the normal manner.

### 8.3.2 Projection and Reconstruction

The projection and reconstruction operations follow immediately. In order to project a new query posture onto the subspace, the mean is rotated out, the posture linearized into a vector, then the vector is projected onto the subspace to return the weight vector. Likewise, reconstruction is the inverse of this, combining the linear basis with the weight vector, then exponentially mapping the result back to the mean posture. Finally, the reconstruction error between the resulting postures can be calculated using the posture metric in Chapter 5.

## 8.4 Initial Eigenposture Results

We did several initial evaluation experiments of PCA on posture data which we describe briefly in this section. We performed our posture PCA algorithm on a corpus of dog animations containing 5800 posture samples of the dog performing various motions such as walking, begging, running, sitting, paw-shaking and looking around. The data contains 53 quaternion-valued joints, meaning there are  $4 \times 53 = 212$  components, or  $3 \times 53 = 159$  possible rotational degrees of freedom in the linearized posture vector.

The results of these evaluations are shown in Figure 8-1 and Figure 8-2. Figure 8-1 shows the root-mean-square (RMS) reconstruction error (in radians) over all postures in the training set as the number of basis eigenvectors is increased. As we noted above, since the covariance matrix is the outer product of the posture vectors, it will be  $159 \times 159$ . Therefore, the maximal rank of the data is 159. The results show that for an RMS error of about .01 radians, about 80 eigenvectors are needed, or about a half compression rate. These results are not as good as we hoped, unfortunately. They seem to show that there is some structure in the data, but that the individual joints tend to move independently much of the time, which limits the usefulness of these techniques for large corpi of motion data. Another potential issue worth investigating is the addition of angular velocity (the time derivative of posture) to the posture vector to see if this reveals more structure in the data (but at the expense of doubling the data dimension).

The first ten principal components are depicted in Figure 8-2. To view the linear basis eigenvectors, we simply exponentially map them back to the mean for each joint to get a posture tuple which we can view. One issue in using PCA on posture data is that the linearization is not as effective on the root joint since it might spin all the way around and whose basis is effectively chosen arbitrarily by the animator. Also, as is common with PCA, the eigenvectors tend to look fairly uninformative since PCA is finding a rotation that makes the data uncorrelated with no regard for local structure such as the fact that the coupling in the data is hierarchical (joints on the same limb tend to be more coupled).

In general, results were mixed. We feel that this is an interesting area for more exploration. Since PCA is a linear algorithm, it finds an orthogonal basis for a subspace. If the data intrinsically actually lives on a curving manifold and not a linear subspace, it is known that PCA will overestimate the data dimensionality. Nonlinear techniques might be worth trying on the data (see [24] for a good overview of these data characterization issues). One simple thing that can be done is to try a cluster-PCA algorithm. Here, the data is clustered and then PCA performed within the clusters.

We also performed several initial visualizations of our data using a recent method for optimally mapping a curved manifold's intrinsic dimensions into a Euclidean space called Isomap [82]. Although this is beyond the scope of this document, we feel that this algorithm might be useful for finding a character's motion manifold.

## 8.5 Summary of Eigenpostures

We described the standard principal component analysis algorithm for dimensionality reduction of vector data. We then showed how to use some of the QuTEM building blocks

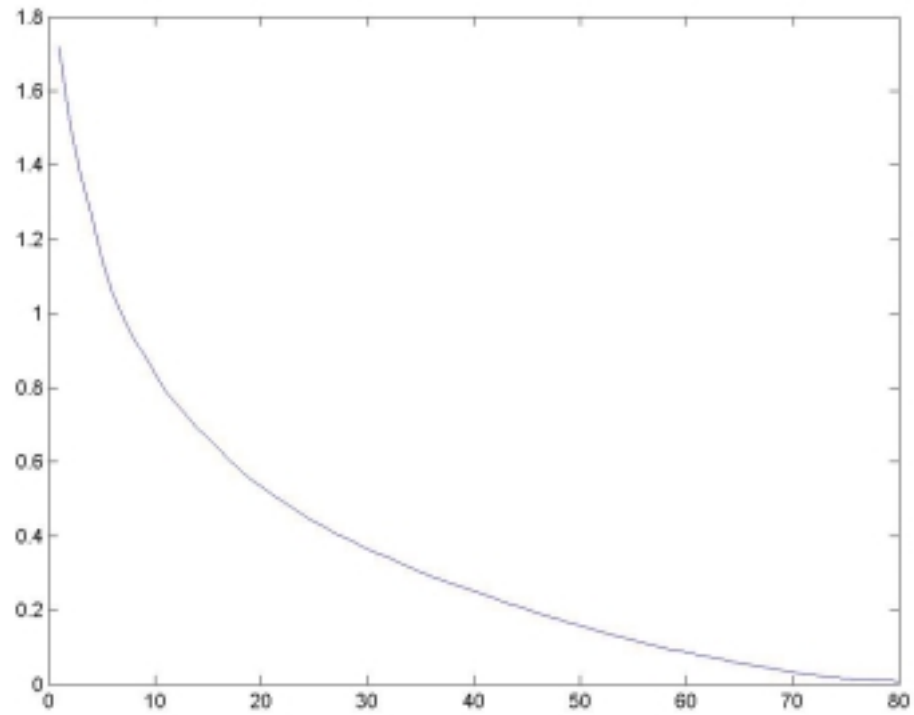


Figure 8-1: Training set RMS reconstruction error (radians) versus number of basis eigenvectors.

---



Figure 8-2: First ten principal components from 5800 frames of a 53 joint dog.

---

(mean and hemispherize) and the quaternion exponential map in order to linearize the data for use in a standard PCA algorithm. We presented initial results on animation data which and concluded that the results were inconclusive and more work needs to be done. We suggested the use of some of the newer non-linear dimensionality reduction techniques that explicitly look for curved manifolds.





# Chapter 9

## (Toward) Expressive Inverse Kinematics

This chapter will introduce the difficult problem of *Expressive Inverse Kinematics* (Expressive IK). The goal of any IK algorithm is to solve the “put my paw on that spot” problem which is often encountered in real-time interactive character engines. Unfortunately, most standard IK algorithms produce robotic-looking solutions, which is not surprising since they were developed in the robotic community where the style of the motion is not important, only the resulting configuration.

A character, on the other hand, must constantly express its internal state through its motion. Therefore, the problem of Expressive IK can be exemplified as “put my paw there but I am really tired.” The resulting motion should convey this internal state in an expressive manner. Furthermore, different characters move in different styles, so that the way one dog puts its paw on a spot is different than another. In our example-based approach, this implies that an Expressive IK engine should find solutions that “look like” that character.

This chapter will describe our progress towards a full Expressive IK algorithm, although due to time constraints we were not able to fully realize the entire approach. The chapter will cover the following building blocks for a real-time quaternion IK engine:

- Fast Joint Limits
- Fast numerical IK solver
- Equilibrium points

We will describe each of these in more detail below.

Finally, we will sketch out two ideas for augmenting this standard “robotic-looking” numerical solver with a model of the character’s actual motion subspace:

- A hybrid of pose-blending and CCD
- Using a model of the character’s motion subspace learned from a corpus of animation data (such as Eigenpostures) to augment a CCD approach.

The rest of the chapter will proceed as follow:

**Section 9.1** describes the basic approach we will take to tackling the problem of expressive IK.

**Section 9.2** will describe our fast model of joint motion constraints learnable from example data. The model will be based on the QuTEM (Chapter 6).

**Section 9.3** will describe how the QuTEM mean (or any other reference posture) can be used as a heuristic to try and make solutions more “natural” looking.

**Section 9.4** will describe our unit quaternion extension to the currently popular Cyclic Coordinate Descent (CCD) real-time IK algorithm.

**Section 9.5** discusses how a hybrid of pose-blending and CCD can be used to reduce the number of examples needed in a purely pose-blending approach. This approach was used on the physical Anenome robot.

**Section 9.6** will sketch out how a statistical analysis of posture from examples (such as our Eigenpostures, described in Chapter 8) could be used in a similar manner to heuristically pull or project an unnatural (or unexpressive) algorithmic solution into the subspace of motion that the character lives in.

**Section 9.7** summarizes the contributions of the chapter.

## 9.1 Approach

To approach the problem of Expressive IK, we chose to start from a recently popular real-time IK solver called Coordinate Cyclic Descent (CCD) [87]. Unfortunately, most standard IK algorithms (including the original CCD algorithm in [87]) assume an Euler angle representation of joint orientation. Since we use a unit quaternion representation of joints, we need an extension CCD to unit quaternions. We describe our extension — QuCCD — in Section 9.4.

Furthermore, many standard IK algorithms often exploit the following:

- Joint motion range limits
- Joint equilibrium point (or center) to model muscle tension
- Heuristics for choosing a particular solution from an infinite subspace in order to find the most “natural” posture, such as lowest energy

Range motion range limits are important to avoid unnatural body configurations in the IK solution, such as an elbow bending backwards or a shoulder bending too far in any direction.

Equilibrium points are often used to “pull” the IK solver back towards the “center” of a joint’s motion range. This can be useful for avoiding numerical drift and for coaxing the IK solver to choose a solution nearer the joint’s center than the joint constraint boundary, which often looks more natural and can sometimes speed up solutions by keeping it from bouncing along the constraint boundary.

Other heuristics on the posture can be added to choose between multiple solutions to find the most “natural” solution. For example, many IK solvers will just find the nearest

solution in terms of displacement magnitude (smallest rotation). This can still lead to postures which also look awkward or unnatural. Another common heuristic is to minimize some energy metric on posture to choose the most natural solution (see, for example, Grasia [30] or Hecker’s GDC video [38]). Unfortunately, if the character happens to be excited, this might not be what is desired. We believe that an example-based approach to finding these heuristics will be better than trying them by hand.

Due to time constraints and the initially mixed results from Eigenpostures (see Chapter 8), we were not able to incorporate the Eigenpostures into our IK algorithm. We feel, however, that this is one of the most exciting areas for future work and will lead towards much more expressive IK solvers.

## 9.2 Joint Constraints with the QuTEM

Usually, joints are modeled using an Euler angle parameterization where the joint limits are explicit intervals over which the Euler angles are allowed to vary. If the joint tries to go outside of its range, the angle can simply be clamped into the interval. This approach is seductively simple, since it involves only scalar comparisons and clamping, and each degree of freedom can be thought of separately. Essentially, this constraint approach “unwraps” the Euler angle (which is  $S^1$ ) into a line and clamps the interval there.

Unit quaternions, however, live on a sphere, which makes this approach problematic. We could represent the sphere using spherical coordinates, but this factorization can create “corners” on the resulting constraint boundary. Such edges, as we argued in Chapter 4, can cause numerical optimization procedures (such as most IK algorithms) to get “stuck.” Due to the lack of a good unit quaternion constraint model, other researchers<sup>1</sup> who use a unit quaternion representation for joints convert to an Euler angle description, clamp the angles there, and then convert back to a quaternion. This is expensive if it must be done every iteration of an IK algorithm, however, since it involves several matrix multiplications and trigonometric functions.

### 9.2.1 Approach

We chose to approach the problem geometrically. We model unit quaternion joint constraints as an elliptical boundary on  $S^3$  (see Figure 9-1). We already saw how the logarithmic mapping converts ellipses on the sphere (isodensity contours) into ellipsoids in the tangent space at the ellipse center in our discussion of the QuTEM model in Chapter 6. Also, we saw how we can learn such a boundary from data, which we argued is required to leverage the animator. For these reasons, the QuTEM will let us implement this model directly.

### 9.2.2 Goals

To handle joint constraints, we will need two operations on the QuTEM model:

---

<sup>1</sup>Jeff Lander (personal communication, 1999).

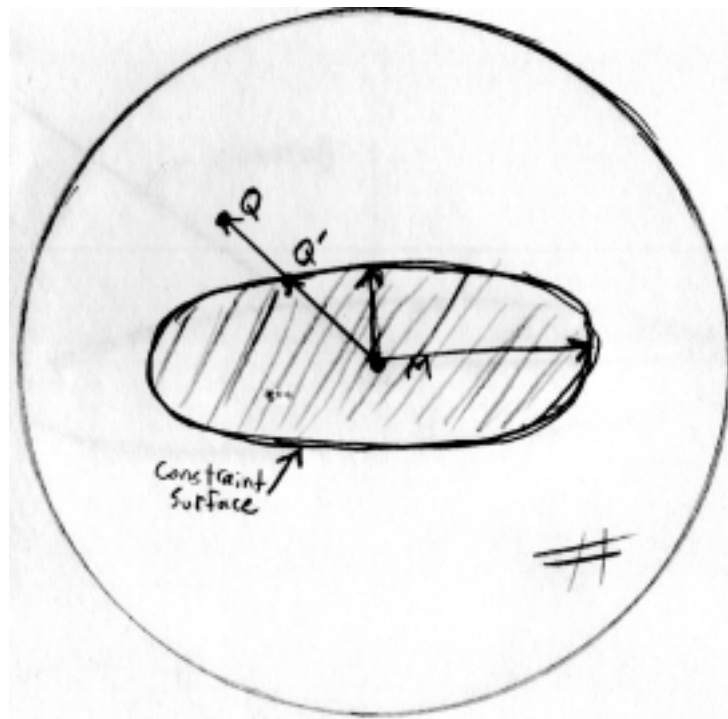


Figure 9-1: An abstract visualization of finding whether the query point  $\hat{Q}$  on the unit quaternion sphere is inside the constraint ellipse boundary or not and the nearest projected valid point,  $\hat{Q}'$ .

---

**Constraint Satisfaction** Is  $\hat{Q}$  inside the constraint boundary?

**Constraint Projection** If  $\hat{Q}$  is not inside the constraint boundary, what is the nearest point to  $\hat{Q}$  that *is* within the boundary?

We describe how to compute each from a learned QuTEM model for a joint.

### 9.2.3 Constraint Satisfaction Operator

We define the constraint boundary to be the isodensity contour of the QuTEM with Mahalanobis distance of  $\rho$  from the center of the joint (the QuTEM mean  $\hat{M}$ ). Since we defined the density to be zero outside this region, the joint is not allowed to go there. We now show perform the test.

Recall that our SMT (Scaled Mode Tangent) transformation defined in Equation 6.2 turns a unit quaternion into a unit variance vector in the tangent space at the mode of the Gaussian, and that its magnitude is simply the quaternion Mahalanobis distance. Therefore, an ellipse (and its interior) on the sphere maps to a solid ball (filled sphere) of some radius when transformed using the SMT. The transformed constraint boundary is obvious — it is simply a sphere with radius  $\rho$  (see Figure 9-2)! We can therefore define our constraint satisfaction predicate as a simple sphere-point test on the transformed query point using the QuTEM. If the transformed point is inside the sphere, it is valid, otherwise not. Simply put, only quaternions within  $\rho$  standard deviations of the mode ( $\hat{M}$ ) are valid.

The test is then defined as follows: First, hemispherize  $\hat{Q}$  to be on the same hemisphere at the QuTEM mean,  $\hat{M}$ . Then the following formula tests to see if the query point is within the constraint radius:

$$\text{ConstraintSatisfied}(\hat{Q}) = \|\text{SMT}(\hat{Q})\| \leq \rho$$

This check is fairly efficient, involving several quaternion multiplications and the sinc call to evaluate the log, so is suitable for use inside an iterative IK algorithm.

### 9.2.4 Constraint Projection Operator

We can also use the SMT transformation and its inverse to project an invalid point  $\hat{Q}$  into the valid region. Formally, we define a `ProjectOntoConstraintBoundary` function which takes a quaternion and returns the nearest quaternion which on the constraint surface:

$$\text{ProjectOntoConstraintBoundary}(\hat{Q}) = \text{SMT}^{-1} \left( \rho \frac{\text{SMT}(\hat{Q})}{\|\text{SMT}(\hat{Q})\|} \right)$$

where again need to hemispherize the query first.

This operator simply changes the magnitude of the unit variance to  $\rho$  so that it lies on the boundary of the sphere, then inverts the SMT transform to put modified unit variance tangent vector back onto  $S^3$  at the correct location. Figure 9-2 illustrates this operation.

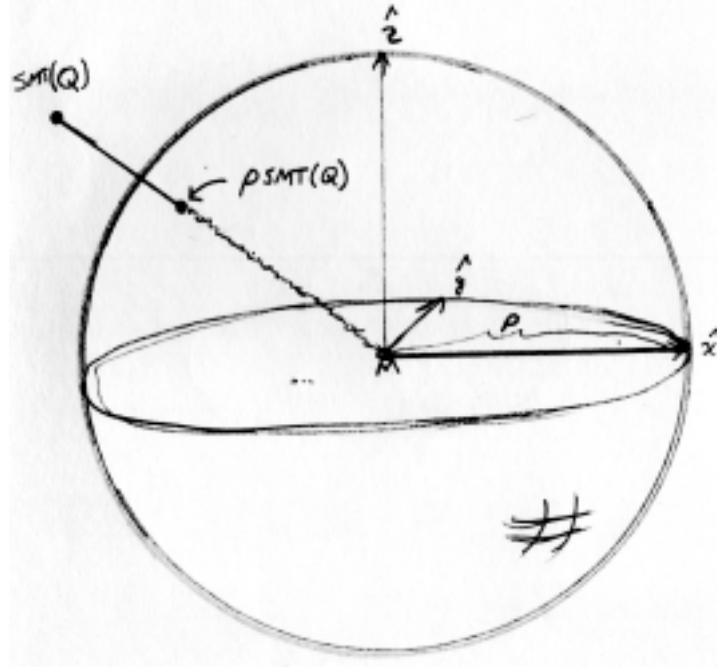


Figure 9-2: A visualization of the  $SMT(\hat{Q})$  sphere which divides out variance differences along the principal directions  $\hat{x}$ ,  $\hat{y}$  and  $\hat{z}$ . The mode,  $\hat{M}$  is mapped to the origin, and the constraint boundary  $\rho$  away from the mean is the surface of the sphere (which is radius  $\rho$ ). Therefore, we can perform very fast, simple sphere-point checks and projections on properly mapped data.

In practice, we often project to a radius of  $\rho - \epsilon$ , so that the point is just inside the constraint boundary to avoid numerical roundoff issues with projected points not satisfying the constraint satisfaction predicate.

### 9.2.5 Singular Densities

For joints with only one degree of freedom, the QuTEM density will be singular. In this case, we must use the pseudoinverse of the covariance matrix. Since the singular directions will be scaled to zero by the pseudoinverse, this will also project points that move off the great circle that defines the joint's degrees of freedom back onto it. For example, if an IK algorithm tries to bend an elbow about a different axis than its fixed axis, the projection will place it back on the appropriate great circle around the elbow's fixed axis.

## 9.2.6 Empirical Results

We tested these operations on a QuTEM learned from a the corpus of all dog animations. First, we generate a *uniform* rotation for every joint <sup>2</sup>. Next, we project *all* the quaternion samples onto the constraint surface (in this case, we also project points that happen to be already inside the surface as well). In this manner, we can randomly sample the constraint boundary. The constrained posture samples can be rendered to see if the resulting constraints are “reasonable.” Figure 9-3 shows several random such samples on our dog. In general, the constraints are visually good — elbows and knees are constrained to one direction, as desired, and other joints seem equally valid.

One issue that becomes immediately obvious from this method, however, is that this joint constraint model is local to joints and does not know about *posture constraints*. A posture constraint is an invalid posture for the character, such as one that causes a body penetration. These constraints are much harder to handle since the boundaries are non-convex and depend on the geometry of the character. Often collision detection routines are used to handle these constraints. We feel that using a statistical model of pose such as our Eigenpostures (see Chapter 8) could help with this problem. If we had a model of the character’s motion subspace learned from examples, we could project a posture that causes an interpenetration onto this surface since, by definition, the surface is learned from positive examples where no interpenetration occurs. More work needs to be done here, however, to validate this approach.

## 9.3 Equilibrium Points with the QuTEM

One problem often found in numerical integration or inverse kinematics algorithms is that we often want the joint to slowly pull itself back towards its equilibrium point, or center. For example, some IK algorithms will add in an error terms based on distance from this joint center in order to constrain the usually under-constrained IK problem.

We can use following simple update rule to “pull” the solution towards the QuTEM mean  $\hat{M}$  by an amount proportional to distance:

$$\hat{Q} \leftarrow \hat{Q}(\hat{Q}^* \hat{M})^\alpha$$

where  $\alpha$  is the “strength” of the pull. For  $\alpha = 0$ , the system will not pull at all. For  $\alpha = 1$ , we jump immediately to the mean  $\hat{M}$ . The parameter is currently chosen empirically.

We have not used this building block extensively, but have found it useful for pulling back numerical drift in numerical integration of the root node, which causes characters to slowly list to one side as they walk.

---

<sup>2</sup>Note that our QuTEM model cannot represent uniform distributions easily without infinite variances. Uniform rotations are simple to generate in quaternions by rejection sampling a unit cube to get uniform points inside the unit sphere in  $\mathbb{R}^4$ , then normalizing them to the surface, as discussed by Shoemake [75]. Notice we can’t sample inside the cube and renormalize as the cube’s corner directions will get more density, but rather need to rejection sample to get points uniformly in a sphere of arbitrary size, then normalize the result.



Figure 9-3: Several screenshots of random sampled dog postures on the constraint boundary. The shots were created by creating a uniform rotation for each joint in the dog, then projecting to the nearest point on the constraint surface. Most sampled configurations are reasonable, though in some the “joint-local” nature of these constraints becomes obvious by a body interpenetration. We do not handle these *posture* constraints yet, but feel the Eigenpostures might be useful here.

---



Another use of the equilibrium point is as a “soft” joint constraint. If the joint begins to go outside of its range, a non-zero  $\alpha$  based on the distance from the constraint boundary can be applied to slowly pull the joint back inside the region. Soft boundaries could be used to avoid “bouncing” off the boundary as often in the inner loop of an iterative IK algorithm, but we have not looked into this yet.

## 9.4 QuCCD: Quaternion Cyclic Coordinate Descent

CCD is a recent heuristic iterative technique for solving simple IK problems in real-time for articulated characters [87, 9, 38]. It has begun to attract much attention in the computer game and interactive character communities due to its relative speed compared to the more traditional *Jacobian-based methods* (see, for example, [79, 3], or Welman’s excellent thesis [87] which compares the two methods.). For these reasons, we chose to use CCD rather than a Jacobian method. The standard description of CCD, however, assumes an Euler angle and  $SO(3)$  representation of joints [87]. This section will describe the basic paradigm of the algorithm and then present our unit quaternion extension of it.

### 9.4.1 CCD IK Paradigm

Recall that the basic problem of IK is as follows: Given an open kinematic chain (see Chapter 5) of bones connected by joints and an *end effector* (a paw, for example), find a set of joint displacements from the current posture of the chain that places the effector at some desired *goal* location in space.

CCD proceeds by iteratively solving a local subproblem at each joint along the chain (see Figure 9-4). Specifically, it calculates the rotation of each joint that will get the effector as close to the goal as possible while leaving all other joints fixed. It then updates the joint orientation by a weighted version of this rotation (see Figure 9-5), where the weights are usually chosen heuristically.<sup>3</sup> The weights can be thought of as *joint stiffness* constants, so that smaller weights imply stiffer joints.

The algorithm starts at one end of the chain<sup>4</sup>, solving locally for a rotation of that joint and performing the weighted update. It then continues performing the local minimizations at each joint in order down the chain. Since it often takes multiple passes down the entire chain to converge, deals only locally with coordinates, and essentially performs a heuristic gradient descent on the effector error, the algorithm is called *cyclic coordinate descent*.

To summarize the basic CCD paradigm:

---

<sup>3</sup>Unfortunately, the weights are a free parameter that must be specified by hand to achieve “good” results empirically. It is future work to someone estimate appropriate values of these from data.

<sup>4</sup>Welman starts from the distal end; we usually start from the base. The solution will depend on which choice is made.

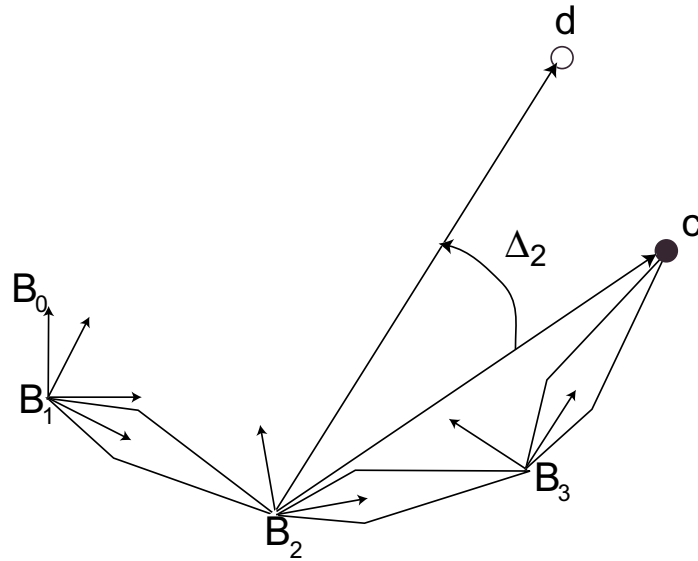


Figure 9-4: The geometry of a CCD local update step on a three link kinematic chain. The algorithm calculates the vector from the current joint being updated (here 2) to both the effector's current Cartesian position (c) and the goal's position (d), expressed in the local coordinate system of the joint ( $B_2$ ). These vectors can be used to calculate the local angular displacement of joint 2 ( $\Delta_2$ ) which minimizes the error  $\|c - d\|$  between the goal and effector. Joint 2's orientation is then updated by rotating it in the displacement's direction by some percentage, which is expressed as a weight ( $a_i$ ). This completes a single CCD sub-step on Joint 2. The algorithm would then proceed to Joint 3 and perform the same set of operations again on the updated chain. This continues cyclically down the chain until convergence or a stopping criterion is met.

1. Loop until the error is under a threshold (convergence) or a maximum number of iterations is performed:
  - (a) For each joint in the chain in order from one end to the other:
    - i. Find a rotation of the joint that locally minimizes the distance between the goal and the current effector position.
    - ii. Update the joint orientation by a weighted version of this rotation.

We will not describe the Euler angle subproblem minimizations here, but refer the reader to Welman's thesis.

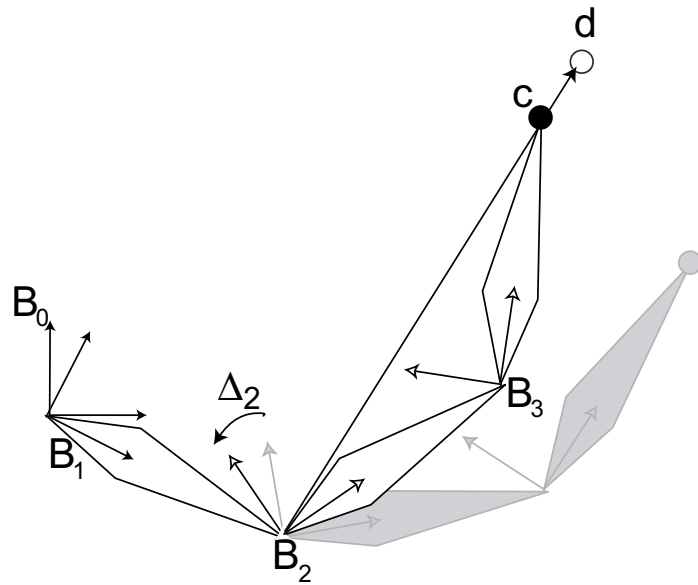


Figure 9-5: The CCD algorithm after updating Joint 2 with a full weight of 1.0. The rotation update ( $\Delta_2$ ) must be applied in the parent's coordinate system since the orientation of the joint is specified as an update.

---

## 9.4.2 Unconstrained QuCCD Algorithm

This section describes our QuCCD algorithm which extends the CCD paradigm to a unit quaternion joint description. This section will express the algorithm without regard for joint constraints, which we will discuss separately in Section 9.4.3.

Assume we have an open kinematic chain consisting of a path through the skeletal tree (see Chapter 5) containing  $N$  bones upon which we want to perform inverse kinematics. Let  ${}^N\mathbf{c}$  denote the (fixed) location of the end-effector in the last frame frame  $N$  in which it is embedded and  ${}^0\mathbf{d}$  denote the desired location of the effector in frame 0 (as is usually the case for IK problems). Let  ${}^j_k\mathbf{F}$  be the forward kinematic transform that takes a point in a bone frame  $j$  and expresses with respect to bone frame  $k$ . Let  $\hat{Q}_j$  denote the quaternion representing the rotation of bone  $j$  with respect to its parent in the chain,  $j - 1$ .

In order to calculate the subproblem solution, CCD requires  $\mathbf{c}$  and  $\mathbf{d}$  both expressed at the current joint in the update cycle, say  $i$ , or  ${}^i\mathbf{c}$  and  ${}^i\mathbf{d}$ . We can calculate these values using  ${}^j_k\mathbf{F}$  using

$${}^i\mathbf{c} = {}^N_i\mathbf{F} {}^N\mathbf{c}$$

and

$${}^i\mathbf{d} = {}^0_i\mathbf{F} {}^0\mathbf{d} .$$

Figure 9-4 depicts this geometry for an  $N = 3$  three link chain. In this case,  $i = 2$

Let  ${}^i\hat{\mathbf{d}}$  and  ${}^i\hat{\mathbf{c}}$  be normalized versions of the vectors. Then the rotation that minimizes the angle between these vectors follows immediately from the quaternion vector product we saw in Chapter 3:

$$\hat{R}_i = ({}^i\hat{\mathbf{c}}^* {}^i\hat{\mathbf{d}})^{\frac{1}{2}}$$

which is a unit quaternion rotation that will take vector  $\hat{\mathbf{c}}$  into  $\hat{\mathbf{d}}$  along the shortest path.

Note that  $\hat{R}_i$  is also expressed in the local, rotated coordinate frame  $i$ , but the orientation of  $i$ ,  $\hat{Q}_i$ , is expressed with respect to its parent frame,  $i - 1$ . In order to update the joint quaternion  $\hat{Q}_i$ , we therefore need to express  $\hat{R}_i$  in the parent frame using the transform  ${}^i_{i-1}\mathbf{F}$ . Since the translational portion of the transform will have no effect on a rotation, we can ignore it. Therefore, we can perform a change of basis on the update  $\hat{R}$  by rotating so the parent and child orientations align, applying the update rotation there, and then rotating back into the local frame:

$$\hat{\Delta}_i = \hat{Q}_i \hat{R}_i \hat{Q}_i^* .$$

Using this rule and scaling the update rotation by exponentiating it to the joint weight  $a_i$ , we get an update rule for the joint quaternion:

$$\hat{Q}_i \leftarrow (\hat{Q}_i \hat{R}_i^{a_i} \hat{Q}_i^*) \hat{Q}_i$$

Note that we left-multiply here since we are rotating in the local frame, as we saw in Chapter 3. Also notice, however, that similar cancellation occurs on the right, leaving:

$$\hat{Q}_i \leftarrow \hat{Q}_i \hat{R}_i^{a_i} \quad (9.1)$$

which is quite simple and elegant, although we can make this faster computationally by using the geometric formula for the square root of a quaternion rather than the exponential map version:

$$\hat{P}^{\frac{1}{2}} = \frac{1 + \hat{P}}{\|1 + \hat{P}\|}$$

Plugging all of these pieces together gives us the following simple update rule based on the current and desired effector positions expressed in the coordinates of the local joint:

$$\hat{Q}_i \leftarrow \hat{Q}_i \left( \frac{1 + {}^i\hat{\mathbf{c}} * {}^i\hat{\mathbf{d}}}{\|1 + {}^i\hat{\mathbf{c}} * {}^i\hat{\mathbf{d}}\|} \right)^{a_i} \quad (9.2)$$

### QuCCD Subproblem Algorithm Summary

To summarize the QuCCD solution to the local joint ( $i$ ) subproblem:

1. Calculate the current effector position vector in the local frame:  
 ${}^i\hat{\mathbf{c}} = \text{Normalize}({}^N\mathbf{F}^N\mathbf{c})$
2. Calculate the desired effector position vector in the local frame:  
 ${}^i\hat{\mathbf{d}} = \text{Normalize}({}^i\mathbf{F}^0\mathbf{d})$
3. Update the joint:  $\hat{Q}_i \leftarrow \hat{Q}_i \left( \frac{1 + {}^i\hat{\mathbf{c}} * {}^i\hat{\mathbf{d}}}{\|1 + {}^i\hat{\mathbf{c}} * {}^i\hat{\mathbf{d}}\|} \right)^{a_i}$

The complete unconstrained QuCCD algorithm follows by using the CCD paradigm and solving the subproblems with this algorithm.

### Discussion and Future Work

**Joint Weights** The main free parameters of the CCD algorithm are the joint weights. We use a simple heuristic rule that makes the base joints stiffer since they have to move more mass. Learning these weights from data or making them functions of time might be a useful extension for allowing expressivity into the algorithm. For example, if a character gets his leg hurt, the joint stiffness could be increased so it seems like he is favoring it.

**Singularities** A nice property of such a geometric approach is that there very few coordinate singularities, unlike the standard Jacobian methods which become ill-defined as the Jacobian becomes singular. The main singularity in the algorithm occurs when  $\mathbf{c} = -\mathbf{d}$  since there are an infinite number of ways to get from  $\mathbf{c}$  to  $\mathbf{d}$ . A simple check can be used to check for this case and choose an arbitrary path as it gets close.

**Other IK Constraints** Welman also shows how to implement more than just this *point constraint*, which tries to place one point on another. Another useful constraint is a *point-orientation constraint* which also specifies the desired orientation the end effector should have at the point. We leave this extension as future work, but expect a similarly simple solution.

**Other Extension to CCD** As CCD is becoming more popular, other researchers are finding other extensions to it as well. We discuss some of these in Chapter 11, including some work on ways to get around the convergence problems with very tight joint constraints, and ways to handle branching chains.

The next section will discuss adding joint constraints to this unconstrained solution.

### 9.4.3 QuCCD with Constraints

The unconstrained CCD algorithm can be augmented with our joint constraints in several ways. The simplest one, which we used for our work, is to simply perform the constraint satisfaction test after solving each sub-problem. If a constraint is violated, the invalid point is projected onto the boundary, then the next joint is solved.

Although this approach works much of the time, it has several problems. Unfortunately, we discovered that this method tends to produce very slow convergence for 1 DOF joints which are constantly bumping into a boundary. The problem is that the CCD step does not take the constraints into account directly and will step in invalid directions, then get pulled back, making for slow progress. Both Hecker [38] and Blow [9] note similar issues and discuss several solutions.

The most direct approach is probably to find a new solution to the subproblem that takes the constraints into account. We feel that it is possible to set up a joint-local error metric which penalizes constraint violations and solve this analytically. The Mahalanobis distance (see Chapter 6, for example, would be a likely candidate for this approach. Initial calculations seem to imply an extended eigenvector solution to the problem, but we leave this extension for future work.

## 9.5 Mixing Pose Blending and IK

A useful technique for either adding expressivity to an IK algorithm or adding procedural generalization to a standard pose-blending algorithm is to make a hybrid of the two.

For example, a pose-blending space can be created which gives examples of what the posture should look like for several goal points. Then adverb parameters can be chosen to correspond to the goal point's Cartesian location <sup>5</sup>. This blend approach, since it must use the known set of examples, is not procedurally general on its own as we argued in Chapter 2. Also, the solution will often have a residual error between the desired effector location (goal) and the resulting effector position from the pose-blending solution. Since

---

<sup>5</sup>Rose investigates using pose-blending for IK in his PhD, which was influential in our approach here.

in both cases we get a solution that is in some sense “close” to the answer we desire, we can use our IK algorithm to “clean up” the solution.

Section 10.5 describes the use of this technique on a physical robot.

## 9.6 Adding Expressivity with Subspace Models

One problem with the QuCCD algorithm (and most other numerical IK solutions) is that it simply is looking at coordinates without regard for any body knowledge of the character. Solutions often appear “unnatural” since it chooses the nearest solution it finds. To handle this, many IK systems add in heuristics to the optimization to make the algorithm choose more natural looking postures. Some of these are energy minimization or the equilibrium points we discussed above. Most of these are hand-coded and need to be tweaked. Furthermore, they do not explicitly capture what makes one character move like Mickey Mouse and not Donald Duck.

On the other hand, if we could learn a mathematical model of the manifold of motion-space that a particular character “lives” on from data, we could project an unnatural IK solution onto this manifold, or pull it towards it like an equilibrium point. In other words, we could take a “Mickey Mouse-looking” IK solution and convert it to a “Donald Duck” solution.

To approach this problem, we did an initial evaluation of one such subspace analysis technique, Principal Component Analysis (PCA), to find a minimal linear sub-space of the character’s motion learned from data, which we described in Chapter 8). Due to time constraints and the mixed results of the linear PCA approach to Eigenpostures, we have not been able to integrate it with the QuCCD algorithm, and leave this for future work. We predict that these manifold and sub-space techniques will be very useful for augmenting numerical IK solutions.

It is worth comparing this “inverse” approach to the more “forward” approach of specifying a pose-blending to IK as we described above. In the pose-blending case, our animator creates the examples needed to specify the space at particular locations to span the space. On the other hand, if we just have a large corpus of animations such as motion capture data, we cannot use this approach. In this case, learning the subspace is exactly what we want. For this reason, both approach are potentially useful.

## 9.7 Summary of Expressive IK

This chapter presented the basic IK problem of putting an end effector on a goal point subject to joint constraints. In particular, our contributions were:

- A fast joint constraint model using unit quaternions that can be learned from examples.
- How to implement joint equilibrium points with the QuTEM
- An extension of the basic Euler angle based CCD IK algorithm to a unit quaternion representation.

- A description of using a mixture of both pose-blending and CCD in order to reduce the number of examples in a pure pose-blending approach to IK.
- A sketch of how a subspace or manifold learned from example data, such as our Eigenpostures, could be used to make a robotic-looking solution more expressive.



# Chapter 10

## Experimental Results and Application Examples

This chapter will describe several experiments and results on the algorithms and ideas we presented in this thesis. The chapter will proceed as follows:

**Section 10.1** illustrates several QuTEM models learned on dog animation data and discusses advantages and disadvantages of the model.

**Section 10.2** describes a simple way to generate new animations “similar” to an example. This is only an initial evaluation.

**Section 10.3** describes several different convergence experiments of the sasquatch algorithm, including choice of timestep, reduction to slerp, and a visualization of the attractor.

**Section 10.4** describes the many projects which have successfully used the slime algorithm for pose-blending.

**Section 10.5** describes an initial evaluation of mixing pose-blending and CCD on a physical robot, the Anenome.

**Section 10.6** summarizes the chapter.

### 10.1 QuTEM Analysis Results

We used the estimation procedures described in Chapter 6 to estimate a QuTEM for each of 38 joints on an earlier dog model (see Figure 5-1) using 26 animations containing about 1200 points. The animations were created by a skilled animator for a particular installation. The full set of animations used was: Beg, BegLow, SolicitPlay <sup>1</sup>, StandToLie, LieToStand, StandToSit, SitToStand, ChaseTailLeft, ChaseTailRight, TightTurnLeft, TightTurnRight, TightTurnLeftToStand, TightTurnRightToStand, Lie, LookUp, Shake, ShakeHigh, ShakeLow, TurnLeft, TurnRight, WalkLeft, WalkRight, Sit, WalkStraight, CatchTreat, Sniff.

---

<sup>1</sup>Crouching down as dogs do when they want to play.



Figure 10-1: The dog in “mean pose” where all of his joints have been set to their mode.

---

The animations tend to fall into verb/adverb groups, with several variants on particular animations. Some are transitions between cyclic animations.

Figure 10-1 depicts the mean pose graphically, where the dog’s joints are all set to the QuTEM mean. This appears correct, and is in fact close to the canonical pose.

The QuTEM covariances are slightly harder to visualize, but since the tangent space description of the quaternion data is three-dimensional, as are the ellipsoids which describe the covariance of this data, we can plot both of them to visualize the ellipsoids that comprise our model. Figure 10-2 contains plots of several joint QuTEM. The plots were made by transforming the data into the tangent space at the mean with the logarithmic map and plotting the resulting points. We also show the ellipsoid of Mahalanobis distance 1.0 (plotted wireframe so the data is visible). Notice that our constraint radius ( $\rho$ ) would scale this ellipsoid until it just contains our all of the data.

**Discussion** One clear issue is that the data does not seem Gaussian on initial inspection, but seems to contain more structure. Since the QuTEM is learned from a set of animations whose data is concatenated together, the original curves of the animation are visible in the scatterplot. Each individual animation tends to cluster data more closely together such that a smaller ellipsoidal model with a different mean would capture each of the animations better than learning a model of the entire joint from multiple animations. This structure implies that the ranges for a joint are *non-stationary*, and depend on some other variable.

For this data, some of the animations are simply adverb variations on the same basic

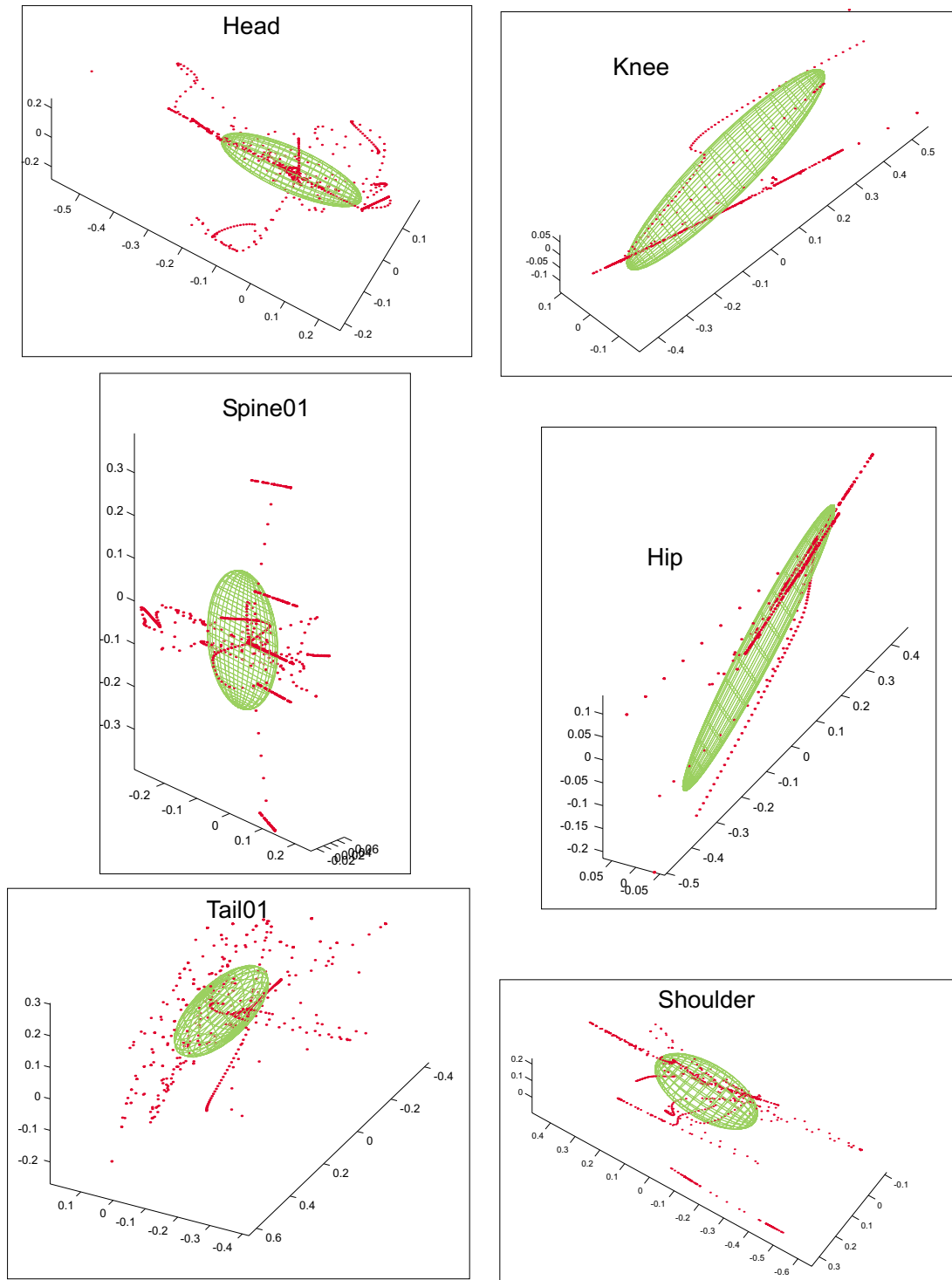


Figure 10-2: Plots of the mode-tangent descriptions learned from animation data for several joints on a dog model. The scatterplot is the transformed quaternion data and the ellipsoid shows the Mahalanobis distance 1.0 isocontour of the estimated density from the data.

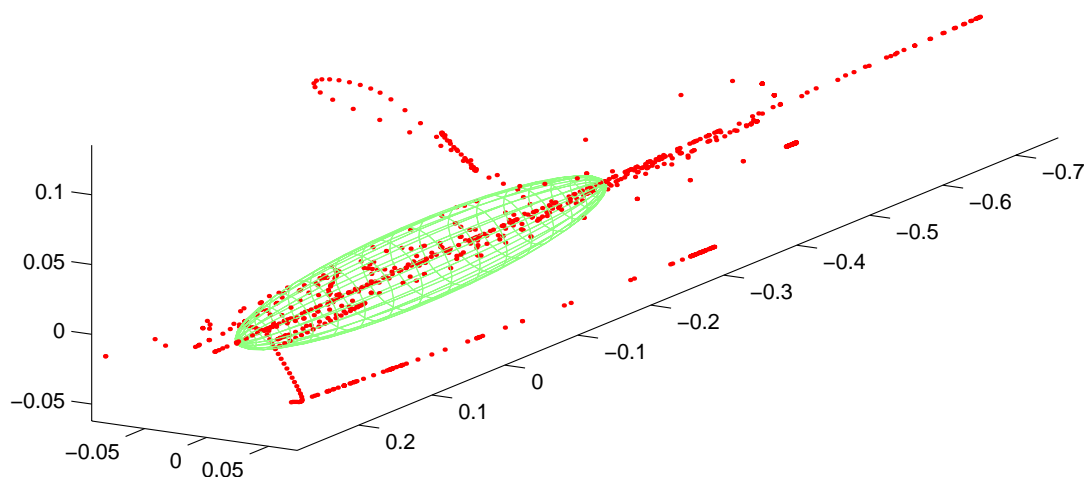


Figure 10-3: TEM plot of just the elbow joint of our dog. Notice the structure contains more than one degree of freedom, although it tends to lie in a particular direction.

---

verb (see Chapter 7, such as walking left or right. In these cases, the motion tends to be similar in variation, but with a different mean. It is expected that as more animations with different variations are added, the data will fill the ellipsoidal area more, although more experiments need to be done.

Another interesting result from these data is that the elbow and knee joints do not seem to be entirely one degree of freedom, as we expected, but tend to vary a little in other directions as well (in particular, see Figure 10-3). In particular, although *individual animations* tend to be one degree of freedom (the estimation process finds only one significant variance direction), when taken together, the different means for each tend to add extra directions. Our hypothesis is that since the bone animations actually are deforming a mesh *skin*, the animator used these extra degrees of freedom to get around problems with the skinning algorithm. We need to investigate this further.

## 10.2 Synthesis of New Motion from the QuTEM

To test our pose synthesis algorithm described in Chapter 7, we created a simple method for generating new animations similar to an example (or multiple example) animation. We did this using the following approach:

1. Learn a QuTEM from one or more example animations.

2. Generate two new postures  $\hat{P}_1$  and  $\hat{P}_2$  with the QuTEM.
3. Generate a squad cubic spline from the current posture  $\hat{P}_0$  which goes through  $\hat{P}_1$ , then  $\hat{P}_2$  then returns to  $\hat{P}_0$ .
4. Choose a time function  $f(t)$  for the squad interpolation parameter to play out the new animation at some desired speed.

Using this simple approach and a constant interpolation speed, we learned a QuTEM from a walk cycle and then generated animations from it. The results were as expected — the dog appeared to be swimming randomly. This test was useful for validating the QuTEM models. We omit the results here.

The time function could also theoretically be learned from data using a temporal frequency analysis of the animation in addition to spatial analysis we perform in the QuTEM.

## 10.3 Sasquatch Experiments

This section will describe some simple experiments on the Sasquatch algorithm to investigate its behavior.

First, we ran Monte-Carlo simulations by sampling random systems of springs and points in order to calculate average convergence rates and to calculate an (empirically) optimal  $\Delta t$ . Second, we ran experiments to see statistically how bad of an approximation the naive renormalized Euclidean weighted average was compared to our Sasquatch solution. Third, we made sure the system empirically reduces to Shoemake’s slerp function in the boundary case of two examples.

### 10.3.1 Monte-Carlo Convergence Trials

The first experiment we performed was to get a feeling for the convergence rate of the algorithm with respect to the timestep choice. This experiment also compared the two integration techniques. One trial of the experiment went as follows:

1. Generate  $M$  uniformly distributed random quaternions as the data (nails).
2. Enforce local hemisphere constraint on the examples.
3. Generate  $M$  non-negative random spring constants which sum to unity.
4. Run the algorithm (either intrinsic or embedding) over uniformly sampled  $\Delta t \in [0, 2]$  and record the steps to converge to  $\epsilon$  for that  $\Delta t$ .

**Uniform Quaternions** First, we must generate random quaternion examples. We chose to do this over the uniform distribution on the unit hypersphere. This is done by sampling from a zero-mean, unit-variance 4-dimensional Gaussian and normalizing the result to the sphere. Shoemake describes this correct algorithm in [75], along with caveats for other naive methods.

**Hemispherization** Next, we enforce the local hemispherical constraint. We refer the reader to Section 6.2.2 for details on this important problem.

**Spring Gains** To generate the spring constants, we simply choose  $M$  constants in the interval  $[0, 1]$ , then divide each by the sum of all to enforce the unity summation constraint.

**Sampling  $\Delta t$**  Next, we select a uniform sampling of timesteps,  $\Delta t$ , over the interval  $(0, 2]$ . If we choose a spacing of  $\gamma$  between samples, then we run the system over the timesteps  $\Delta t = i\gamma$ , where  $i$  is an integer and goes from 1 up to  $\lceil \frac{2}{\gamma} \rceil$  such that  $0 < \Delta t \leq 2.0$ .

**Integration** Finally, we run each of the integration schemes on the trial for each of the values of the timestep and record statistics on how many steps were required to converge for all the various  $\Delta t$  for each of the integration schemes.

**Results** In order to get a feeling for how the system behaves, we ran 100 trials of this experiment on 5 points chosen as above and collected statistics on the number of steps taken to converge for the samples of  $\Delta t$ . These statistics give us a feeling for how the system converges on average and how the convergence rate depends on  $\Delta t$ . Figure 10-4 illustrates the results of these trials for the intrinsic numerical integration technique given in Appendix C (results were practically identical for the embedding integration, so we omit them). The middle curve on the graph shows the mean number of steps to convergence for each  $\Delta t$ . The curves above and below show one standard deviation away from the mean. Convergence for  $\Delta t < 0.25$  became huge quickly, so we truncate the graph to see the detail in the minimum.

We can deduce from this graph that the algorithm is robust in its convergence behavior since the standard deviations are small in the middle area of the graph. Also, we see that our guess of a timestep of 1 was actually reasonable, although on average the system converges slightly more quickly at  $\Delta t \approx 1.175$ . The reason for this number is not obvious, but we expect that it could be found if the system were analyzed in terms of a linear update, where the eigenvalues of the update matrix define convergence behavior, although the nonlinearity of the solution makes this analysis beyond the scope of this paper.

Another slightly surprising result was that the intrinsic and embedding integration techniques both converged at virtually identical rates, though the actual trajectories were slightly different. Therefore, we can choose either in practice, depending on which runs faster in real-time, though we have not done these timing experiments yet. We expect that the embedding integration approach will be faster since it does not use any trigonometric function calls.

### 10.3.2 Reduction to slerp

We also tested whether the algorithm as described reduces to the same result as Shoemake's slerp in the case of two examples since we desire it to be a multiple example extension to slerp. In this case, the slerp blend parameter, call it  $\alpha$ , can be considered the weight on the

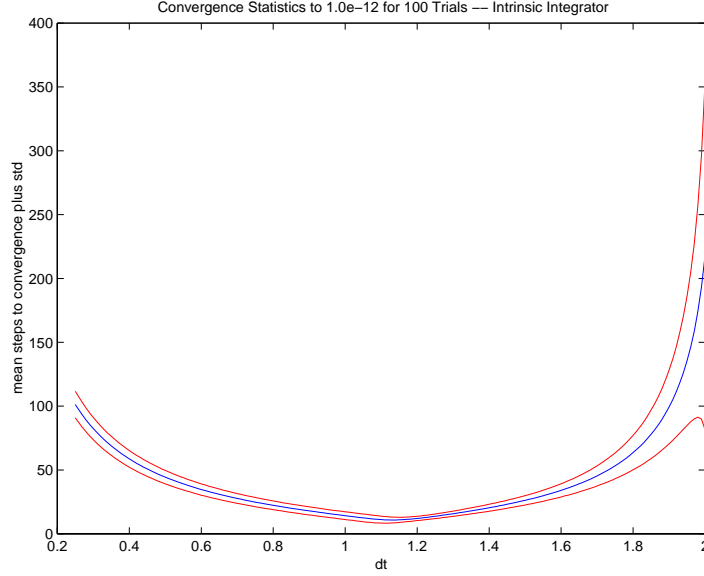


Figure 10-4: Convergence statistics for a 5 point system using intrinsic integration over 100 random trials. Here,  $\epsilon = 1.0e - 12$ . The middle curve is the mean and those bracketing it are one standard deviation.

second example, forcing the second weight to be  $1 - \alpha$  due to unity constraint. Therefore, we must have:

$$\forall \alpha \in [0, 1], \text{sasquatch}((\alpha, \hat{A}), (1 - \alpha, \hat{B})) = \text{slerp}(\hat{A}, \hat{B}, 1 - \alpha)$$

where slerp can be defined exponentially as:

$$\text{slerp}(\hat{A}, \hat{b}, \alpha) = \hat{a} e^{\alpha \ln(\hat{A}^* \hat{B})} \quad (10.1)$$

To test this, we also used a simple Monte-Carlo simulation. We created a random set of two quaternions chosen uniformly, then sampled sasquatch and slerp for evenly sampled choices of  $\alpha$  and calculated the angular distance between the results. For all trials, we got only roundoff errors on the order of  $1.0e-8$ . This was the case whether we started from a purely uniformly chosen initial location or the Euclidean average of the two points, which also shows the algorithm is robust and will converge properly from many choices of initial configuration, even for initial choices not on the one-parameter subgroup between the two examples.

### 10.3.3 Attractor Trajectories

We can view some of the attractors for our ODE by choosing a fixed system of examples and weights, then plotting the trajectories for various choices of initial conditions. This allows us to illustrate that the attractor is stable over the sphere, even though we will be choosing initial conditions more intelligently than at random.

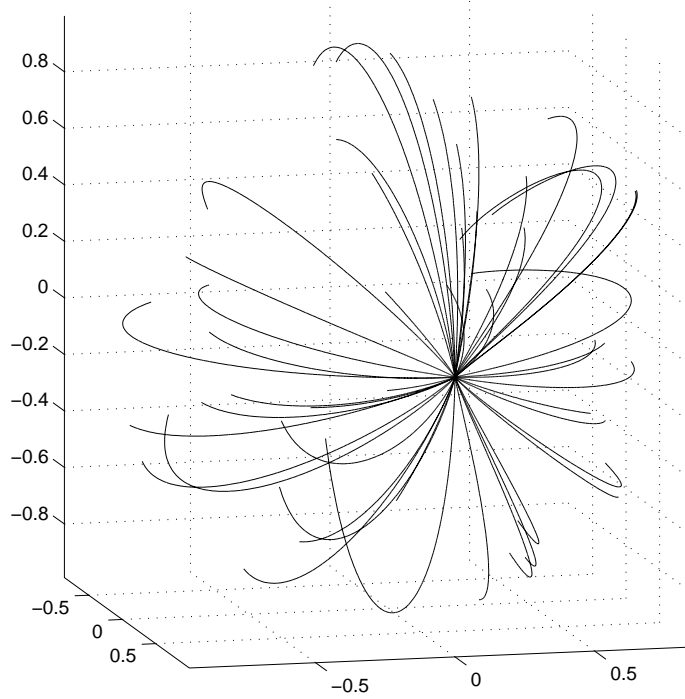


Figure 10-5: An illustration of the stable attractor which is the steady state solution to Sasquatch. Here we choose 50 random initial points not too close to each other then integrate the system with  $dt = .01$  and plot the resultant trajectories. Here we choose two examples, whose attractor (steady state) is the identity quaternion, 1. Since the data live in  $S^3$ , we project out the  $z$  component for this plot.

**Experimental Design** We chose to plot a simple attractor based on two examples. Since the solution lives in a one-parameter subgroup of the quaternion group, we can see how solutions arrive at the solution from points off the subgroup. In particular, we chose the simple system of the quaternions

$$\hat{Q}_{1,2} = e^{\pm \alpha \hat{n}}$$

with equal weights on each. Hence, the solution will be at the identity,  $\hat{1}$ . We then choose random initial  $\hat{Q}_0$  for solving the ODE, but making sure that no example started off too close to another, so the trajectories could be seen. If a new  $\hat{Q}_0$  is within some chosen  $\epsilon$  distance to another example.

**Results** The results for  $\alpha = 1.0$  and  $\hat{n} = [100]^T$  with equal weights is shown in Figure 10-5. In this figure, we project out the  $z$  component to make a 3D plot. More visually appealing is a plot of the logspace of the quaternion, which is in  $\mathbb{R}^3$ . We choose to use the logmap at the solution (identity),  $\hat{1}$ , such that the attractor is at the origin. The results of this plot on the same trajectories can be seen in Figure 10-6.



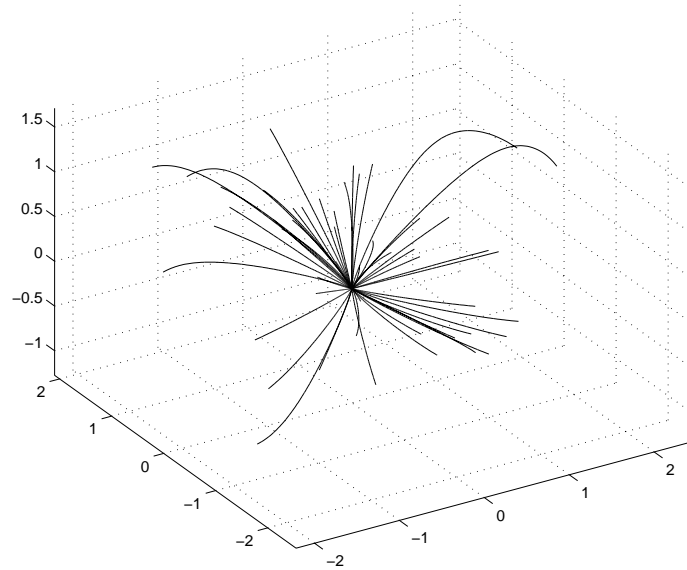


Figure 10-6: A plot of the trajectories for the attractor taken as the log at the attractor location (the identity), which is the tangent space  $\mathbb{R}^3$ . The attractor in the log is located at the origin.

---

## 10.4 Slime Results

We have used the basic slime blending algorithm as the basis for pose blending in several successful projects. This section will give a quick synopsis of each project and some basic lessons we learned for each.

### 10.4.1 *Swamped!*

The *Swamped!* project was the first to use slime posture-blending in 1998 and was based on the author's SCOOT motor system (unpublished), which was based on Perlin's and Blumberg's motor systems.

#### Overview

The first project to use one of my quaternion blending algorithms was the *Swamped!* project [10] shown at the interactive exhibition at SIGGRAPH 1998. The story scenario is simple: an autonomous raccoon tries to steal the eggs of the semi-autonomous chicken, controlled by the interactor using a wireless sensed plush toy (see Figure 10-7) which we coined a *sympathetic interface* [46].



Figure 10-7: The *Swamped!* project, shown at SIGGRAPH 1998. The interactor directs the chicken character using natural gestures of the plush toy (*sympathetic interface*). The raccoon is autonomous and uses an early slime-based RBF system based on Rose’s Verbs and Adverbs work.

---

### Technology Used

The raccoon (Figure 10-8) was a fully autonomous character with several simple emotional states — happy, tired, angry — which changed based on interactions with the chicken. These emotional states were expressed continually in the animation using an early version of our Slime RBF algorithms based on Rose’s Verbs and Adverbs research, as implemented in our SCOOT motor system. The RBF’s converted the normalized emotional values, such as happiness, into weights on the animations which were blended with the slime algorithm around the identity pose. The animator was forced to use a particular basis for the identity pose.

### Lessons Learned

The *Swamped!* project successfully demonstrated how to blend animations represented as quaternion efficiently, and how to extend Rose’s Verbs and Adverbs work into a direct quaternion representation rather than his Euler angle factorization.

This project taught me about the problems with always using the identity as the reference pose for the blending and forcing the animator to work from a certain structure rather than designing their own model. These showed up as “glitches” in the animation as the rotations for certain widely varying joints, in particular the shoulders, hit the mathematical singularity in this global linearization. Also, animators tended to dislike being forced to keep resetting geometry transforms every time they made a change. This motivated the need for the *mean* of the animations and began the investigation into the QuTEM model.



Figure 10-8: The raccoon character is autonomous. He can blend between several emotional states based on his interactions with the chicken. These states are expressed through the motion with pose-blending.

---

### 10.4.2 (void\*)

The (void\*) project was shown at the interactive installation at SIGGRAPH 1999, as well as within the Media Lab.

#### Overview

Another example of a sympathetic interface was used in the (void\*) project. Here, a pair of wireless sensed bread rolls with forks stuck in them were used to make dance move gestures as if they were legs (inspired by Charly Chaplin’s classic restaurant bit in *The Gold Rush*). Three characters — Earl the beefy trucker, Elliot the nerdy salesman, and Eddy the slick dude — meet in a diner. They get possessed by an interactor and made to dance using the forks and buns as the interface.

Each character had several dance contents based on recognizable gestures on the buns. Some of these were: split, march, leg twirls, and jump. Each of the dances also had stylistic variations based on the character’s “feeling” about being forced to dance. For example, Elliot the nerd was much more inhibited as he started dancing, holding his hands close to his body (Figure 10-9). If the interactor did dance moves that he seemed to like, he would get more happy and his dance style more open. Earl the trucker was very stiff and hated doing certain moves, such as the split, which Eddy the Dude was very proud of (see Figure 10-10).

The dancing scenario was quite interesting, as it led to a fast interaction and forced the motor system to blend quickly between the various dance moves flawlessly. Likewise, while another character performed, the idle characters still had to appear alive and could



Figure 10-9: Elliot the shy nerd starts off dancing very inhibited. Over time, his dance styles becomes more open as he enjoys himself.

---

make reactions to certain moves by the other character, such as giving a thumbs-up if they liked the move.

### Technology Used

(void\*) ran on a version of the SCOOT motor system using similar Slime RBF technology as *Swamped!*, but with a few new features. Blends were done in the same way, but we needed better *layering* support to handle reaction shots, such as giving the dancer a thumbs-up for a cool move. These blends were layered on top of other basic idling animations when needed using a temporal transition blend (the weight smoothly changes from one to the other and then back). Figure 10-11 shows an example of a blend at a particular point in time of one of the characters. Again, a one-dimensional blend like this could be done with *slerp* directly, but we actually had several axes in some of the animations.

Idling characters on the screen also needed to appear alive, even if they were not dancing. Therefore, we introduced a simple Perlin noise source (see [63] as a *blend weight between idling postures*). In other words, each character had several idling (“sitting around”) animations. To avoid obvious repetition created if these were played in a loop, we simply blended between the examples using a random walk through the weight-space.

### Lessons Learned

The high energy dancing scenario, with all the possible transitions, was interesting from a motor system standpoint. An early lesson was that we needed to maintain angular velocity continuity, since velocity glitches were very noticable in dance moves. Therefore, we switched from *slerp* to *squad* for blending our animations in time.



Figure 10-10: Eddy the Dude shows off the range of motion of his hip joints with his split move.

---

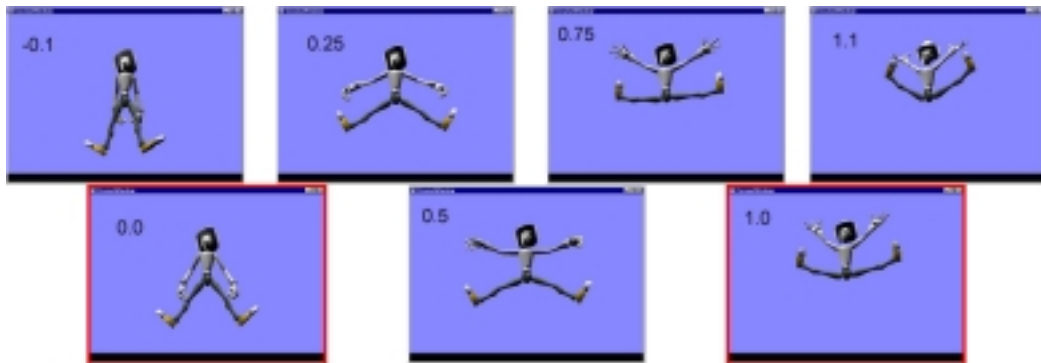


Figure 10-11: Blend of two animations, sampled at the same time  $t$  but at different blend weights, from -0.1 to 1.1. The examples (red boxes) are the original animations, so the algorithm can extrapolate as well as interpolate.

---



Figure 10-12: Rufus, a simple articulated robot dog head with a camera in its eye. Rufus was the first example of using out pose-blending slime algorithm on a physical robot.

---

As mentioned above, we found that adding layered partial animations (like waves and other gestures) could add expressivity. This worked fairly well.

Finally, the slime based pose blending performing quite well except for a few instances of widely-varying joint ranges, such as hips and shoulders. These were evidenced as “glitches” where the blend would go through the singular shell since we were not using the mean as a reference.

### 10.4.3 Rufus

Rufus (see Figure 10-12) was a class project by Burke, Eaton, and Stiehl of the *Synthetic Characters* Group in the fall of 1999. It was the first to hook up our pose-blending system to a robot output. Rufus had a camera for tracking objects and could express several emotional states with just his ears and tongue.

#### Technology Used

Rufus consisted of simple 1 DOF Futaba airplane servos, which meant animations for Rufus actually lived on a one-parameter subgroup of the quaternions. The fact that physical



Figure 10-13: Duncan and the Shepherd. This project was one of the first to begin to look at clicker training the animal. Both the shepherd and dog used an early slime-based blend.

---

robots still used several 1 DOF joints to produce higher DOF joints became clear from this project.

Rufus ran a version of the SCOOT motor system that was close to the one used for (void\*).

### **Lessons Learned**

The fact that many interactive physical robots still used several 1 DOF joints to produce higher DOF joints became clear from this project. It is unfortunate that need for such servos forces the mathematical coordinate singularity to be expressed physically. On the other hand, these joints are usually built to avoid the singularity in the operating region.

We realized in this project that we could use the same motor system for both physical and graphical creatures. The only difference was in the “render” stage when the quaternions needed to be converted to Euler angles (for a robot) or homogenous matrices (graphics render).

#### **10.4.4 Duncan the Highland Terrier**

The Year of the Dog began in 2000, where the *Synthetic Characters* Group tried to build a virtual dog with similar perceptive, cognitive, emotional and learning abilities exhibited by real dogs. One of the first projects from this was Duncan the Highland Terrier.

### **Technology Used**

Duncan was one of the first projects where I did not have an active role in building the actual motor system. The motor system used was an early port of my SCOOT system by Marc Downie.





Figure 10-14: Sheep—Dog used an acoustic pattern recognition system to direct the dog to herd sheep into a pen using traditional dog-training lingo.

---

### Lessons Learned

This being one of the first projects with dogs, I quickly noticed a “roll-over” animation was required. Unfortunately, a rollover caused the root joint to perform a complete revolution, going right through the exponential map singularity in the slime algorithm and causing a glitch. This led me to begin looking for a local method which would not have this singularity and would allow weighted blends over the entire group. The result was the sasquatch algorithm.

### 10.4.5 Sheep|Dog: Trial by Eire

Sheep|Dog was a project in 2000, shown at the Media Lab Europe opening in Dublin, Ireland. It was one of the first “Year of the Dog” projects aimed at achieving dog-level perception, cognition and expression. Figure 10-14 shows the dog herding several sheep. Sheep|Dog was also shown at the  $E^3$  electronics exposition show as part of academic trends in videogames.

### Technology Used

Sheep|Dog was one of the first systems to stop using an RBF function approximator with slime to perform blends and to just use direct weights which were calculated by the programmer. This motor system was a home-grown, simple system developed by Blumberg and Downie based on my slime algorithm.

### Lessons Learned

Again, one of the earliest lessons learned in this exhibit was the singularities introduced by the slime algorithm when used on the root joint during a roll-over.

### 10.4.6 $\alpha$ -Wolf

$\alpha$ -Wolf, Tomlinson’s Phd research project, was presented at SIGGRAPH 2001 and consisted of simple social interactions (a dominance hierarchy) amongst a virtual wolf pack [90].



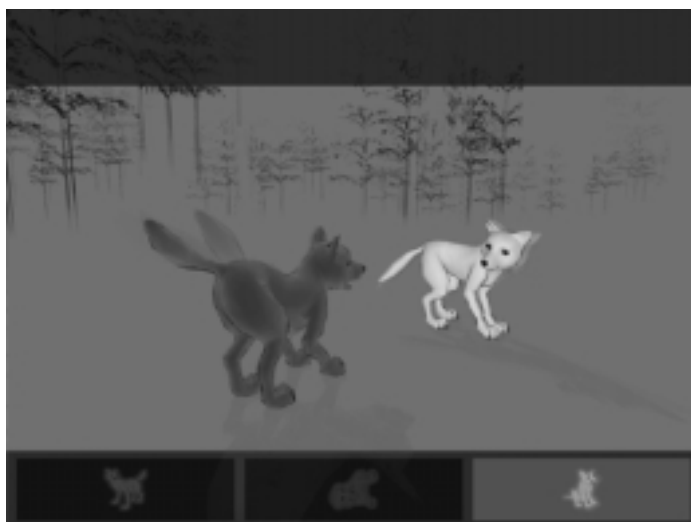


Figure 10-15: A shot of the  $\alpha$ -Wolf installation

---



Figure 10-16: A blend along one of the emotional adverb axes. Picture credit to Bill Tomlinson.

---

Tomlinson showed that emotional perception, memory and expressivity are required for proper social relationships.  $\alpha$ -Wolf had three wolf pups and a mother wolf, which was the largest number of complex motor systems we had running at once.

### Technology Used

$\alpha$ -Wolf was one of the first systems to use Downie's *pose-graph* motor system [20]. Pose-graphs extend the simple verb graph ideas which Perlin, Blumberg, Rose and I used into a graph structure which explicitly models the motion manifold in terms of animations and transition animations between them. Transitions became heuristic (A\*) searches through this graph according to some distance metric, such as minimizing acceleration.

The quaternion pose-blending technology used in the pose-graph is essentially slime on a set of weights inside the convex hull of examples. Thus, no extrapolation is possible in Downie's current pose-graph. Transitions all happened explicitly through the graph whereas the SCOOT system generated a transition qith squad if one did not exist.

Figure 10-16 shows samples along one of the emotional adverb axes for the wolf pup.

## Lessons Learned

$\alpha$ -Wolf basically hit the current limit on using purely blend-based methods<sup>2</sup> Essentially, the number of transitions required to achieve graph closure, as well as the number of required examples to parameterize the many different actions the wolves could take, were at about the limits of an animator's ability. This project made explicit the need to make the more general procedural methods such as IK more expressive.

Another issue that became clear is that we needed to handle the problems of "close contact" that showed up in the dominance interactions, such as biting each other and wrestling. These close, collision-ridden postures of several characters are very difficult to solve with a purely pose-blend solution, and require the more flexible abilities of and collision response algorithms. This is the subject of future research.

### 10.4.7 Slime Results Summary

The basic slime algorithm has proven to be very robust and efficient for performing a weighted blend of multiple unit quaternions. It has been the core animation blending technology in all the *Synthetic Characters* Group motor systems since *Swamped!* in 1998.

## 10.5 Expressive IK Results

A variant on this approach was used on the Anenome robot pictured in Figures 10-17 and 10-18<sup>3</sup>. The Public Anenome was shown at the SIGGRAPH 2002 interactive exhibit. The Anenome lived in a small stage environment with a waterfall and plants. An expressive IK algorithm was needed to orient the "head" of the Anenome towards things in its environment. For example, a stereo-vision algorithm was used to find people in the environment and track them as they moved.

In the Anenome, the degrees of freedom near the base were calculated using a slime-based pose-blending and the degrees of freedom nearer the effector were calculated exclusively using CCD. This allowed the gross movement of the base to be specified by the animator through examples that get the effector nearest the solution but also look natural. The top DOFs could then be controlled by the more general CCD algorithm to "finish the job." This approach is useful since producing a pose-blending space that spans the entire possible IK space of the robot is time-consuming since many examples are potentially needed. By combining approaches, we argue that we are better able to leverage the animator's ability by reducing the number of examples.

In order to "render" the animation, the quaternions are converted to the Euler angles which are fed to a feedback motor controller which tries to track the given kinematic trajectory. This is simply the analogue of converting the quaternions to a homogenous matrix which is usually required by graphics hardware to render polygons. By holding off on this conversion until "just-in-time," we can maintain the efficiency of the quaternion represen-

---

<sup>2</sup>Downie, personal communication.

<sup>3</sup>Very special thanks to Jesse Gray and Matt Berlin who implemented this approach on the Anenome.

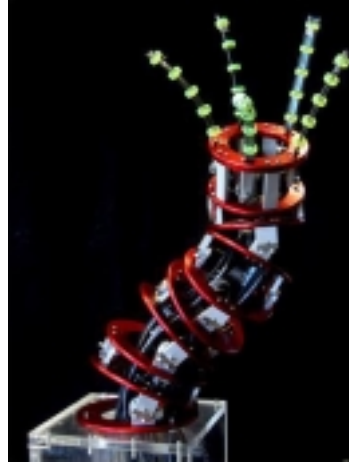


Figure 10-17: The insides of the “Public Anenome” robot by the Robotic Life Group at the MIT Media Lab that was shown at SIGGRAPH 2002.

---

tation as long as possible and avoid problems with gimbal lock not associated with the physical servos on the robot.

Also, we can use the same motor system for both virtual and physical creatures. The main issue with this approach is that the robot is physical and dynamics come into play even with the feedback controllers. Adding dynamics is the subject on ongoing future work on our group.

## 10.6 Results Summary

This chapter:

- Presented results of joint motion analysis experiments with our QuTEM model and described how to visualize it.
- Described initial experiments at pose and animation synthesis from the QuTEM.
- Presented experimental results of the sasquatch algorithm’s convergence behavior.
- Described chronologically the many motor systems which have used the slime primitive as the core blending technology, lessons learned for each and motivations for the other building blocks we discovered along the way.
- Presented initial results on a hybrid of pose-blending and CCD IK on a physical robot.

The next chapter will present and discuss related and influential work.



Figure 10-18: The Anenome with its skin on.

---

# Chapter 11

## Related Work

This chapter will present and discuss some of the more directly related work than is spread throughout the thesis. We break it into several separate sections:

**Section 11.1** describes other research in real-time expressive character motion engines. Some of these influenced our approach and others were done independently.

**Section 11.2** describes several related methods for blending poses either with an Euler angle description or quaternions.

**Section 11.3** described several other approaches to using statistical analysis and synthesis on joint rotation data. We present those which use quaternions or which follow a related approach.

**Section 11.4** describes several other approach to quaternion joint limits, none of which existed when we began our work.

**Section 11.5** discusses research which is related to our work on expressive IK. In particular, we look at example-based IK methods or those that use a quaternion representation.

### 11.1 Animation Engines

There have been several other researchers who have looked into expressive motion engines for interactive characters. We discuss each of these in this section from a high level, and discuss particular subsets of the work in more detail below.

#### 11.1.1 Perlin

Perlin’s Improv system served as a primary influence in the beginning of our work [63]. Perlin shows how to use his noise functions for generating organic textures can also be applied to joint animations. Perlin describes how a programmer can use the blending of sinusoidals, noise functions and inverse kinematics solutions to create a real-time character animation engine with personality. He also gives heuristics for handling the classic “foot

sliding” problem with these methods and introduces the *buffered action* method of avoiding self-intersection, or a transition point to go through when traversing from one skill to another. Our motor system uses many of these ideas.

One problem with Perlin’s work is that the artist must program the textures on each joint (and correlations between them) by twiddling with parameters and several constructive primitives. In order to make this problem intuitive for the programmer, he uses an Euler angle representation, avoiding the problems with generating band-limited quaternion noise or multi-dimensional blending. Also, there is no way to *learn* these functions given desired data.

### 11.1.2 Blumberg

Blumberg’s “Silas the Dog” character in his PhD research [11] uses an interactive motor system for controlling the dog motion. Here, he explicitly uses a DOF-locking system so that different hand-programmed procedural *motor skills* can only run simultaneously if they do not touch the same DOFs. Also, he used a real-time inverse kinematics solution for the placement of the dog’s paws in the creation of a procedural walk cycle, although this used an Euler representation. To handle emotional parameters, motor skills which did not receive a lock could make suggestions to the winners for how they would do the motion, so that the winning skill could layer or add the expressive skill if it did not overlap with other DOFs.

Although this system ran in real-time, the orthogonality of motor skills was a limitation. Also, expressivity was limited since hand-programmed procedures are difficult to make expressive, as we discussed earlier. He suggests interpolation of these skills as future work.

### 11.1.3 Rose

Our pose-blending work is similar to Rose’s PhD work on Verbs and Adverbs in several ways [69, 44]. Rose’s motion formalism for combining motor skills according to a resource prioritized locking mechanism is practically identical to the one in the author’s SCOOT motor system which was used in *Swamped!* and (void\*), which was in turn based on Perlin and Blumberg’s systems.

Rose is one of the first to look at multi-dimensional blending of animations. He chose to use Radial Basis Functions to approximate the blending functions from an adverb parameter (like happiness) to a weighted blend of Euler angles in the animation. Rose does a good job of covering the high level issues with using pose-blending practically. For example, he shows ways to handle the *time-warping* problem of matching up the structure of examples. He describes what good examples should consist of. He shows how to use a quick, local IK solver to “fix” the classic foot-sliding problem with blending techniques. Finally, he shows how to learn inverse kinematics as an RBF from the Cartesian effector space to the space of joint angles, and looks at errors.

One major problem that Rose discusses is that he found the extrapolation behavior to be poor, so that animators needed to generate the convex hull of animations, rather than just a point along each axis. In other words, if there was a happy axis and a drunk axis, he would need examples of a happy walk, drunk walk, normal walk, and the drunk-happy

walk. We feel that his poor extrapolation behavior (and perhaps some of the issues in the error in his IK learning) is due to his choice of an Euler angle representation as the basis, which causes these sorts of problems. Rose avoids gimbal lock by a pre-processing step that puts the gimbal lock point away from the data and reinterprets the data in that Euler set.

Rose hints at quaternions, mentioning that he uses squad to transition between two animations, which leads some readers to believe that he used a quaternion representation for the blending and IK (such as Grassia), which is not true <sup>1</sup>.

Rose’s work was influential in our implementations of quaternion RBF’s for pose-blending, and we followed his lead on the many issues with creating good example animations. Our work complements his by extending it to a quaternion representation.

### 11.1.4 Grassia

Grassia’s PhD work [30] is also quite similar to our research as well, although was done independently. Grassia, like Rose, investigates a clip-based animation authoring system for transforming animations to work in new contexts. He also uses example data in the form of quaternion motion curves and quaternion joint limits. He considers operations for blending and editing known animations to work for new situations.

For example, he also describes his own inverse kinematics solver based on a quaternion representation, but does not discuss his joint limit model, if he even uses one. On the other hand, he focuses on the use of scaleless metrics in the IK algorithm, although rather than removing the scale of the joint motion ranges as we looked into with our QuTEM model and Mahalanobis distance metric, he uses a notion of energy and mass and removes the scale caused by these effects. When he discusses pose distances here, he considers a Euclidean norm between “pose vectors” but does not discuss the details such as hemispherization.

Grassia discusses many of the issues with *transitioning* from one animation to another in detail, and how to handle the timings and angular velocities of joints getting there. We discovered many of the same issues in the implementations of *Swamped!* and (void\*) and refer to Grassia’s work here rather than duplicate it.

Grassia only discusses real-time engines towards the end of his thesis, suggesting that he IK solver is likely the limiting factor, though not enough had been done. Our work focuses on real-time performance as the first design principle so we avoided slow algorithms from the start.

### 11.1.5 Downie: Pose Graph

The last few projects in the Synthetic Characters Group have used Marc Downie’s *pose graph* formulation of the motion manifold for a character. A pose graph is a directed graph of animation frames from examples. The graph needs transitive closure in that *transition* animations between certain verbs (like walk to sit) need to be added explicitly. They can be created by hand, or potentially by some algorithmic method such as spacetime optimization [25].

---

<sup>1</sup>Rose, personal communication

Blended animations are specified in terms of a motor skill with a set of examples that can be blended. He assumes that he weights sum to one, therefore he ignores extrapolation issues. He uses an  $A^*$  search through the graph in order to find shortest paths from the current motor skill to the desired motor skills, according to a pose distance metric that also considers velocity information, though not joint ranges. The addition of velocity is important and our algorithms should be extended to handle it. Since angular velocity is a vector quantity, this should be fairly straightforward.

The pose graph uses our slime blending algorithm at its heart. Rather than use an RBF approach like we did in *Swamped!* and (void\*), however, the programmer now needs to specify the weights on each animation by writing an *AdverbConverter* function. We feel that this does not leverage the animator’s ability maximally, but for certain types of motor skills it has proven to be useful.

## 11.2 Multi-dimensional Quaternion and Pose Blending

This section will cover related work on multi-dimensional quaternion blending.

### 11.2.1 Grassia: Nested Slerps

Grassia uses a quaternion representation for his joints in his PhD work on example-based motion transformation [30]. He admits that the weighted blend of  $n$  quaternion values for joint blending is an outstanding research problem and chooses to use nested slerp’s to construct his blends for “simplicity.” For example, to throw a ball in a certain direction, he would blend between the examples of throwing high/low and left/right:

$$\text{slerp}(\text{slerp}(Q_{\text{left}}, Q_{\text{right}}, w_{\text{leftright}}), Q_{\text{up}}, w_{\text{up}})$$

Unfortunately, this process produces *different* blend results depending on the order the slerps are done in due to the non-commutivity of rotations. For example, if he chose to blend up/down first, then left/right, he would get a different result. He also admits that the complexity gets deep quickly as the number of blend axes increases. We feel that having the animator and programmer have to deal with the non-commutivity directly in the specification of blends is a bad idea. Likewise, we do not want our character to have to learn this non-commutivity by having it choose the slerp values that will produce the desired behavior or have it have to deal with the ordering of its motions.

Grassia argues that he made this choice since there were no other alternatives at the time. He suggests that one can instead use a “mathematically sophisticated non-Euclidean blending function” to blend the quaternions “such as the RBF’s used by Rose.” Grassia is not quite correct here, however, since as we said Rose used an Euler angle representation for his RBF’s and did not handle the many issues with creating a useful quaternion RBF. Proper use of a radial basis function (or other kernel technique) on quaternions requires dealing with the group theory of rotations directly. Instead, we argue that our slime and sasquatch algorithms solve this problem better than nested slerps for two main reasons:

- Nested slerps do not extrapolate since they blend inside the convex hull of examples.



- Nested slerp scale poorly as the number of examples increases.

Grassia also discusses the quaternion exponential map for joint representation [29]. He concludes that it is a poor representation to use directly due to singularities and interpolation errors, but does not really go into enough detail here. Therefore, he uses quaternion joint models in his thesis. We argue that the exponential mapping is a useful computational tool for generating local coordinate systems and invariants and that it can be used properly in conjunction with a quaternion joint model in the development of algorithms and does not need to be used as the representation explicitly.

### 11.2.2 Buss and Filmore: Spherical Weighted Averages

Buss and Filmore sought the same type of pseudo-linear spherical blending function that extends Shoemake’s slerp to  $N$  points and for spheres of arbitrary dimension, arguing that a proper technique that respects the spherical metric has not been presented before in the literature [15]. They show how the spherical blend can be used to generate B-splines without using one of the nested slerp geometric spline constructions.

Like our sasquatch, they also start from a Euclidean analogy, namely the minimization of a quadratic error function in the spherical distance metric for  $S^d$ :

$$f(Q) = \frac{1}{2} \sum_{i=1}^N w_i \text{dist}_{S^d}(Q, P_i)^2$$

They show how this can be solved using the exponential mapping of the sphere, which works for any dimension sphere. To solve the system, they calculate derivatives of the system and solve for a critical point. To find this point, they describe a first order and second order algorithm which amount to gradient descent without and with a Hessian matrix, respectively. They argue that the first order algorithm runs faster and therefore linear convergence is on par with quadratic in compute time due to the calculation of the Hessian. They also give uniqueness and other proofs.

The first order algorithm they present is practically identical to our sasquatch algorithm, which was derived as a physical system. Their timestep is effectively unity. This is not surprising given that the error function they are minimizing can be considered as the potential energy in our “springs.” They do not exploit the physical analogy any further, however, which means their proofs get mathematically technical, whereas we can often argue from physical analogy directly. We also show that on average a higher time-step than unity (which they use) leads to slightly better convergence.

We feel that our work, independently discovered at about the same time, is complementary to theirs, as well as more intuitive since we start from physical systems analogy and leverage the familiarity most interactive engine designers have with this approach already. We also show how to use the averaging technique to do multidimensional blends directly, rather than creating a unidimensional temporal spline as they do.

### 11.2.3 Lee: Orientation Filters

Jehee Lee’s PhD work is quite similar to ours in that it also uses a pure quaternion joint model, along with the exponential mapping, to build useful primitives for analysis and synthesis of motion. Unlike our work, his statistical analysis is hierarchical in that he computes a Gaussian pyramid of filtered data over time. This gives more power in interpolation given that similar frequency bands can be blended together, rather than all frequencies at once as in our non-hierarchical approach. Unfortunately, the power comes at the expense of complexity in time and space, which is why we focused on simple, fast techniques.

Lee also shows how to build coordinate-invariant temporal filters for orientation data. These filters are useful for producing his filter pyramids as well cleaning up noisy motion capture data. Lee’s temporal filters can also be used for blending multiple quaternions together, although Lee does not go into the specifics of real-time blending, arguing that the formulation of the problem for real-time is often different. He also does not go into the details of multi-dimensional interpolation and extrapolation issues. On the other hand, he has some good quaternion algebra proofs on showing that the filters are rotationally-invariant and time-invariant.

## 11.3 Joint Rotation Statistical Synthesis

Statistical analysis and synthesis has become popular for animation recently, though many use an Euler angle representation do use the standard linear algorithms. This section will describe

### 11.3.1 Brand: Style Machines

Matt Brand describes how to synthesize new animation content consistent with the style found in examples using a purely statistical method called Hidden Markov Models [12]. His results are mixed — one issue is that this method needs a lot of data.

One problem with this work is that the representation of rotation and how he generates new samples is not clear in the paper. For example, one difficulty with using HMM’s is that synthesis is not easy to do, so this is important <sup>2</sup>. We expect he used an Euler angle representation since he was not explicit.

### 11.3.2 Pullen and Bregler

Katherine Pullen and Christoph Bregler describe how to use statistical analysis across poses encoded in terms of Euler angles to “texture” a sketch of an animation [66]. They rely on the fact that joint motions are correlated, and that by specifying some of the joint values, others can be generated from motion capture examples. This method is similar to our work on encoding body knowledge in terms of eigenposes in that both will consider the statistics across all joints. Pullen’s method uses tiny clips from the motion capture data and joins

---

<sup>2</sup>Andy Wilson, personal communication

them together in order to preserve the original data. They also have the same problem with blending techniques in that the foot slippage problem shows up here as well.

### 11.3.3 Lee: Hierarchical Analysis and Synthesis

Lee also shows how to use his orientation filters to perform a hierarchical, multi-resolution analysis of quaternion-represented animation data and how to use this for synthesis. His approach is theoretically more powerful than ours since it can blend animations at different frequency bands. We considered using a quaternion wavelet approach for this same reason (see [17]), but we decided that a multi-resolution approach would be too expensive for a real-time engine.

## 11.4 Quaternion Joint Limits

This section will give an overview of related work on joint constraints with quaternions.

### 11.4.1 Grassia

Grassia discusses separating out the swing and twist components of ball-and-socket joints in his paper on the exponential map [29]. He shows how this simple model can be used to handle constraints in terms of an ellipse for swing and another constraint for twist, but does not discuss how to learn these from data, as we do. He argues that quaternion angular joint limits are difficult, but goes no further.

Our work goes into more depth with gleaning joint limit constraints directly from data in terms of our QuTEM model. This allows constraints to be learned from data. We do not explicitly factor out the twist/swing components for efficiency and simplicity — we have found reasonable performance for joint limits using this technique. Neither of these techniques directly handles the fact that the twist constraint often depends on the swing value in humanoid joints, like the shoulder.

### 11.4.2 Lee

Jehee Lee’s PhD work [54, 55] describes how to model several common types of joint limit models on the quaternion sphere. He describes three: *conic*, *axial* and *spherical* and shows how to do inclusion tests in the quaternion algebra. He argues that more complex constraints like a shoulder can be made up from intersections of these; for example, a conic plus an axial limit the swing and twist or a shoulder. He does not show how these can be learned from data, so we assume that they must be generated by hand algorithmically by the programmer and not the animator. Also, it is not clear that an intersection of three axial constraints will give rise to the same ellipsoid boundary on the sphere which we use.

### 11.4.3 Wilhelms and Van Gelder

Recently, Wilhelms and Van Gelder [89] show how to use *joint sinus cones*. A sinus cone is a region on  $S^2$  which limits the range of motion of the swing component of a joint to being within the region. This allows the twist component to be limited separately so it can be made a function of the swing, which is the case for humanoid joints such as shoulders. Wilhelms and Van Gelder describe how these ranges can be created by having the animator specify the ranges by examples. They show how to quickly perform inclusion/exclusion methods on this structure. Finally, they describe how to smooth out the spherical polygons so they appear more like ellipses on the surface of the sphere by using the *stereographic* projection of the sphere. The stereographic projection is very related to the exponential map of  $S^2$ .

### 11.4.4 Herda, Urtason, Fua and Hanson

Recently, Herda *et al* [39, 40] have shown how to learn joint limits from data automatically using *implicit surfaces on quaternion fields*. Here, joint data for a human shoulder is analyzed using a quaternion representation. By ignoring the scalar component<sup>3</sup>, they get a cloud of points in 3D which describe a valid rotation. Given this data cloud, they can find an implicit surface which contains all the joint animation frames for the joint. The problem of constraining a joint then amounts to finding the nearest point on the implicit surface. One advantage of their approach allows them to find the relationship between swing and twist limits, which is why they focus on the shoulder joint. Unfortunately, they notice the problem of holes in the data and discuss some early ways to handle this.

One issue is that although more general, their primitives are not as computationally efficient as ours, since we focused on real-time performance inside an inverse kinematics algorithm. Also, our work has not looked into the holes in the data directly — some of the holes depend on the configuration of other body parts (in order to avoid self-penetration, for example), so in some sense a local joint constraint model will not be able to understand these holes. This is an interesting area for future work.

## 11.5 Expressive IK

Much of the work on *motion retargeting* uses an inverse kinematic solver to warp (or displacement map) existing animations to meet certain kinematic constraints, such as keeping a support foot fixed on the ground. Almost all of these are designed for “interactive editing” of motion capture data. In general, we focus on those that use a quaternion representation or are designed for real-time performance by multiple interactive characters, rather than as an animation creation tool.

---

<sup>3</sup>I believe the direct use of the exponential mapping is probably equivalent to this, though they do not describe that here.

### 11.5.1 Blow: Quaternion CCD IK with Joint Limits

Jonathan Blow's recent work presents some insights with using a quaternion joint model with CCD and adding joint limits [9]. He also is one of the few developers who focuses on speed, and even argues that an arcosine and a sine are expensive. He never mentions the exponential mapping, but it seems clear that he is referring to it here, since Grassia's decomposition is basically the same as this [29].

He encountered the same problem we did with adding joint constraints as a separate computation between each CCD step — very poor convergence, or lack of convergence, for certain starting configurations. To deal with this, he selects several poses which seem to converge to a large set of solutions. When he gets stuck, he can then choose a different pose to iterate from. He shows how these can be chosen by hand with visual inspection, or by calculating clusters experimentally. It is not clear which solution this will find, however — usually, we expect to iterate from the current pose forward in time by a little bit. In general, however, this seems like a very good idea.

Blow also describes a joint model with quaternions which is similar to Grassia's. It factors the rotation into two terms: a factor  $\hat{R}$  which rotates the bone to align its axis in the correct direction and a factor  $\hat{S}$  next which then twists around that axis, resulting in the composite rotation  $\hat{S}\hat{R}$ . In this way, swing and twist ranges can be limited separately. He shows how this can be done by using a similar 2D polygon inclusion test that Wilhelms uses.

### 11.5.2 Lee: Quaternion IK with Constraints Using Conjugate Gradient

Lee presents a quaternion-based IK algorithm which uses his joint limits in his Phd work (also SIGGRAPH 99) [54, 55]. He uses a quaternion representation of rotation and shows how the inverse kinematics problem amounts to a minimization which respects the constraint functions. He shows how to use the exponential mapping to do this minimization without adding an explicit unity constraint and resulting Lagrange multiplier. He solves the minimization problem with a conjugate gradient solver, which effectively chooses the directions to take steps in, rather than being forced to take steps in each decoupled joint direction as in CCD. Ideally, this should increase convergence rates for the case of an axial constraint. Unfortunately, use of a conjugate gradient technique increases the computational burden quite a bit, especially since a matrix representation of quaternions is almost unavoidable here. Lee's experiments showed that the IK algorithm is slow in practice. To help this, he reduces some of the IK chains to fewer coordinates by encoding humanoid heuristics (in particular the *elbow circle*) into his representation. Effectively, this reduces some of the 1 DOF joints to a 1 DOF search in the optimization rather than 4. One problem is that the elbow circle is a known redundancy in the human arm (and legs), but might not be the case for some arbitrary alien creature which a videogame animator might create.

### 11.5.3 Grassia: Quaternion IK for Motion Transformation

Grassia's PhD research [30] also considers a quaternion version of an IK algorithm similar to Lee's for transforming motion capture data with a displacement map. The most interesting part of this work is that he describes how to make the IK algorithm scale-invariant with respect to a description of mass and energy. A problem with this approach is that some notion of dynamics is needed, which can get expensive. He admits that he expects the IK algorithm to be the slowest part of his motion transformation algorithm, which Lee seems to agree with. In our case, the IK algorithm is the most expensive as well since it is iterative. Since we avoid the full conjugate gradient matrix approach, however, and use geometric methods, we expect that we should ultimately be able to get better performance than these methods although more experiments need to be done.

### 11.5.4 Hecker: Advanced CCD Extensions

Chris Hecker describes his investigation of using CCD on a human figure in a rock-climbing simulation at the GDC 2002 [38]. He uses an Euler angle representation. He describes ways to add constraints at each iteration by clamping the joint angle ranges and describes how this slows the solution and produces different results than are expected often. He also describes how to handle multiple branching points in the IK chain, which Welman only hints at and no one else addresses. He suggests performing the CCD on each chain at a branch point separately, then blending the resulting answers. For the case of more than two branches, this amounts to blending more than two poses. He does not discuss interpolation much here, but we expect he will have the classic Euler angle problems. Our blending primitives will allow this CCD extension to be incorporated directly into our QuCCD algorithm.

Finally, he touches on the fundamental problem of expressive IK: CCD chooses a pose which is not the pose the character (according to an animator) would have chosen. He mentions early uses of physics to "relax" these poses under gravity and other effects, but does not present further. He describes future work of a "Body Knowledge" engine that would encode valid poses by looking at all animated poses and "somehow" push the answer near there. As we argued, we expect that a statistical analysis of posture such as our Eigenposes will be a useful approach to creating such a body knowledge engine and handling this problem.

### 11.5.5 Fod, Mataric and Jenkins: Eigen-movements

Recently, Fod *et al* [22] used a Principal Component Analysis (PCA) on movement examples in order to find useful "movement primitives." The goal is to use these primitives as a lower-dimensional subspace to help the recognition of movement. They use an IK solver to convert their 3D point positions of joints tracked on a performer into an Euler angle representation. They then perform PCA and K-means clustering to find a lower dimensional set of "eigen-movements." They compare reconstruction results and use a simple servo to move a robot with combinations of these primitives.

One issue they have is that the IK solver needs to handle gimbal lock since they are using Euler angles. It is not clear how they handle this since they do not go into the details of the IK algorithm. An advantage of their work is that by looking at full movements some of the dynamics might be captured by the PCA. We have begun to look at PCA on posture plus its derivative to hopefully discover more structure in the data.

### 11.5.6 D’Souza, Vijayakumar, Schaal and Atkeson: Locally Weighted Projection Regression for Learning Inverse Kinematics

D’Souza *et al* describe a method for learning inverse kinematics solutions a humanoid robot [21]. They use locally weighted projection regression (LWPR) algorithm to learn the nonlinear inverse kinematics solution. In order to make the algorithm local to the posture, they must include it into the learning input. Therefore, they learn a mapping from:

$$(\theta, \dot{\mathbf{x}}) \rightarrow \dot{\theta}$$

where  $\theta$  is the vector of Euler angles that describes the current posture,  $\dot{\mathbf{x}}$  is the desired velocity at the end effector (for the next timestep) and  $\dot{\theta}$  is the required angular velocity of the posture.

They use Gaussian “receptive fields” (RF) as a source of local blending functions. They use these local RF’s as the source of blend weights on locally-linear models, with each kernel calculating how far it is from the query. The output is simply the weighted mean of the local models, using the Gaussians as the weight.

Finally, they create a cost function to provide a learning update. To help resolve kinematic redundancies, they add a cost for the posture from some “optimal” posture, which essentially allows the chain to move in the nullspace towards this optimal once it has found a solution. This is the equilibrium point that resolved rate IK systems use and we described in Chapter 9.

One interesting method they use to train the system is called “motor babbling.” Here, the robot chooses a mean for each joint, then “wiggles” around it to learn the local behavior.

These locally-weighted mixtures of linear models are becoming popular these days (see Gershenfeld’s book for a good overview of these mixture model techniques, including his own Cluster-Weighted Modeling [24]). One issue is that they implicitly assume a vector space, but since the model is local, this is often close to true. We considered using Cluster-Weighted Modeling to approach this problem, but decided to try a geometric approach (CCD) first.

The equations they use should be fairly simply to convert into a quaternion representation. The Gaussian receptive fields simply require a proper distance metric on postures, which we have discussed, and should be able to be implemented with our QuTEM model. The unity sum weight blend of local models on quaternions can be done using our spherical blending primitives. Furthermore, our QuTEM can be used to perform “motor babbling” by sampling new orientations.

This work is quite promising, and we feel that a quaternion version should perform even better for a virtual character. Minimally, we expect that less linear models will be necessary and the system will perform better with less data since the quaternion metrics are

more appropriate, but this needs to be tested empirically.

## **11.6 Summary and Recommended Reading**

We recommend that the reader interested in the problems of example-based expressive interactive character animation read Rose's PhD on using Radial Basis Functions for animation blending [44], Grassia's PhD on motion transformation for editing motion capture data [30], and Jehee Lee's PhD on multi-resolution statistical analysis and synthesis of motion [54]. Their work is complementary to ours and to each other and between our work and theirs much of the problems with creating expressive animation from examples are addressed.



# Chapter 12

## Discussion, Future Work and Conclusions

We have come a long way, from a high level description of expressive interactive characters, down to some most likely unfamiliar quaternion mathematics, and back out to a set of new algorithms for solving the problems encountered in designing a real-time motion engine. So what lessons can we take away?

**Section 12.1** discusses several points about using quaternions in a real-time, expressive interactive character engine and evaluates the success of the approach.

**Section 12.2** describes future directions we think will be fruitful.

**Section 12.3** collects the conclusions drawn throughout the work.

**Section 12.4** summarizes the main contributions of this research.

### 12.1 Discussion

Both myself and others have successfully used the algorithms and ideas in this dissertation in the design of real-time animation engines and motor systems for expressive interactive characters. Along the way, we learned quite a bit about the theory and application of quaternions for use in real-time skeletal character animation. This section will try to collect some of the more useful ideas and intuitions for readers who need to implement their own such system. We break the section into the main related work areas.

#### 12.1.1 Pose Metrics

We use the geodesic metric based on the exponential map throughout this work. We made this choice for two reasons:

- It is closely related to Euler's theorem and therefore models the magnitude of the physical action of rotation mathematically as directly as possible.

- It is valid over points “far” apart on  $S^3$

One point that should be made is that for two points  $\hat{P}, \hat{Q} \in S^3$  that are “close” together, the length of the *chord* in the embedding space  $\mathbb{R}^4$  (denoted  $\hat{\mathbf{q}}$  and  $\hat{\mathbf{p}}$ ) between the points

$$d = \|\hat{\mathbf{q}} - \hat{\mathbf{p}}\|$$

is a good approximation to the geodesic metric since the local structure of the sphere is flat.

Metrics are often used to estimate the angular speed between the two rotations described by the points if they are samples  $\Delta t$  apart in time. On the other hand, the geodesic metric will give a much better approximation to the angular speed for points that are farther away (with respect to the spherical metric) since it respects the group metric. Explicitly,

$$\dot{\theta} \approx \frac{1}{\Delta t} \|\ln(\hat{Q}^* \hat{P})\|$$

is a better estimate of the angular speed than

$$\dot{\theta} \approx \frac{1}{\Delta t} \|\hat{\mathbf{p}} - \hat{\mathbf{q}}\|$$

as the samples get farther apart since the geodesic metric respects the rotation group metric of Euler’s theorem. In other words, for uniformly sampled points in time, estimates of the derivative will be better behaved as the sample period ( $\Delta t$ ) is increased or the angular velocity of the curves is larger. In these cases, the distance between two sample points (on the sphere) will be larger. Explicitly, an estimate using the chord-length between two quaternions will asymptotically under-estimate the true angular velocity as compared to the spherical metric.

### 12.1.2 Multi-variate Unit Quaternion Blending

We described two new algorithms based on the invertible exponential mapping of the unit quaternions to and from a tangent space, *slime* and *sasquatch*. We argued for the use of the faster, approximate *slime* algorithm with the mean over the corpus of animation data as the reference. This both puts the singular shell as far from where the data lives as possible as well as giving the best approximation behavior there most of the data lives. We also suggested that *slime* be used on internal character joints and *sasquatch* on the root node.

These are not the only possibilities for blending quaternions, however. We discuss several other briefly.

#### Renormalized Embedding Space Blending

If the examples are hemispherized, then the renormalized Euclidean blend will also produce reasonable solutions, as we showed when we used it as the initial value for our *sasquatch* algorithm. The problem with this method is that the embedding renormalization does not respect the group metric, so a constant change of the weights will not necessarily produce a constant angular velocity curve. This parametric variation is undesirable and was the

motivation for Shoemake’s slerp in the first place. We also require it in the case of more than two examples. We have not done the actual analytic derivatives of neither slime nor sasquatch, but expect the desired behavior from sasquatch and approximate behavior for slime (when examples lie on a great circle through the reference).

### Nested slerps

Grassia used nested slerp’s which he calculated the weights of by hand for each of his problems. This is undesirable if we wish to just use the blending primitive as a “black box” like that shown in Figure 7-1. Furthermore, the order the slerps are applied in matters due to the non-commutivity of rotation. For example, if we have three examples with weights on each, applying a slerp from  $\hat{A}$  to  $\hat{B}$  first then a slerp of the result to  $\hat{C}$ , we will in general get a *different blend result for the same weight vector* if we perform the slerps in a different order, such as from  $\hat{B}$  to  $\hat{C}$  and then slerp the result to  $\hat{A}$ . Additionally, the extrapolation behavior, which we argued leverages the animator’s skill, is not clear using this method.

### Barycentric Coordinates

*Barycentric coordinates* (see Hanson’s Gem [35]) allow interpolation of points inside a simplex in terms of affine transformations between the simplex vertices. Since we can use a spherical triangle on  $S^3$  as a simplex and the geodesic interpolator slerp to perform affine combinations on them, we could use barycentric coordinates to describe a point inside the spherical triangle. We did not consider this approach in our work and leave it to future work to compute the barycentric coordinates on  $S^3$  of an interpolated point. One issue with using this method, however, is that it defines points *inside* the simplex, and therefore will not extrapolate which as we argued leverages the animator through the requirement of less examples and the ability to make caricatures of an animation. On the other hand, our sasquatch algorithm also only interpolates. It would be interesting to work out the relationship between these approaches.

## 12.1.3 Quaternion Statistics

Our approach to quaternion statistics was to use the logarithmic mapping at the largest value eigenvector of the sample covariance matrix of the data as embedded in  $\mathbb{R}^4$  to map from the non-Euclidean surface of  $S^3$  into a tangent space  $\mathbb{R}^3$  where we could use standard vector-space Gaussian densities. This approach was motivated to be an analogue of the vector Gaussian density — subtract of the mean, create the sample covariance matrix, find its principal axes with an eigenvector or SVD algorithm, and use the resulting Mahalanobis distance formula in a scalar Gaussian function to find the unnormalized density value.

The Bingham distribution, on the other hand, estimates a singular Gaussian in the embedding space  $\mathbb{R}^4$ . It is singular since the data lie on a sphere, removing a degree of freedom. In practice, this implies a constraint on the eigenvalues and therefore a convention is used to choose the actual parameters used. These parameters do not have a direct physical interpretation in terms of the rotations we are modeling, however, which this convention just underscores. On the other hand, the Bingham distribution really *is* just a Gaussian

density in  $\mathbb{R}^4$ , which means that sampling from it would not involve the exponential map and its resulting trigonometric functions. Since the QuTEM distribution and Bingham distribution seem very related mathematically by the exponential mapping, and especially since they both calculate the eigenvectors of the sample covariance matrix as the Maximum Likelihood Estimate of the covariance rotational factor, we feel that it could be possible to estimate the Bingham parameters as a function of our estimated variances. The paper by Prentice [64] also seems to imply this, but more investigation is required.

## 12.2 Future Work

This section offers some future work directions we feel are promising.

### 12.2.1 Dynamics

This work assumed only a first order approach — kinematics. Many systems these days require dynamics, such as physical robotic systems, so these issues need to be addressed. Grimes in our group has begun on implementing non-linear force fields using the QuTEM model for solving some inverse dynamics (joint torque calculation) problems.

### 12.2.2 Joint Limits

We use an isoprobability contour of the QuTEM density to model a hard joint limit model learnable from data. The main problem with this model is its simplicity. In reality, a human shoulder joint, for example, has different ranges for the twist around the upper arm depending on which way it is facing due to the internal organic joint structure. Other methods such as Herda *et al* allow for more general joint constraints and will therefore have better results. We plan to implement their method and investigate its properties on our dog data.

### 12.2.3 QuCCD

Our extension to CCD is fast, but still has problems when a joint constraint model is applied after each step in the case of a 1 DOF joint. This is because the algorithm is unaware of the constraints when it takes its step. It should be possible to compute an analytic solution to the joint subproblem using the Mahalanobis distance to “make the algorithm aware of the limits.”

Chris Hecker showed an advanced CCD IK system at the GDC '02, showing how to handle branching chains and more complex constraints. He uses an Euler approach, but again, we can drop our primitives into his general framework and should gain immediate benefits.

### 12.2.4 Orientation Statistics

As we argued above, the Bingham distribution can also model unit quaternion densities. Although we chose not to use it for our work, the recent PhD by Antone [1] shows its efficacy for application to a computer vision problem. It would be interesting to test this approach on our data as well. One main advantage of the Bingham approach is that it stays in the embedding space, where standard familiar vector calculus techniques can be used. In certain cases this may simplify the mathematics, but more investigation is required. Also, the mathematical link between our QuTEM tangent-space approach and the Bingham distribution should prove elucidating.

### 12.2.5 Posture Statistics

Since Principal Component Analysis (PCA) finds a linear subspace of the posture, it cannot capture curved manifolds. If the motion-space of a particular character lives on a curved manifold, then PCA will miss this structure. Recently, global manifold unfolding methods have begun to be used in computer vision to solve this problem of PCA and should be applicable here. In particular, we did an initial characterization of some of our animation data with the new, successful Isomap algorithm [82], which finds a global Euclidean coordinate system of minimum dimension that tries to capture the intrinsic degrees of freedom of a manifold given only a pair-wise distance metric between examples. We used the geodesic metric for this. When applied to a walk cycle of a dog, the algorithm found a 2D Euclidean space that mapped the points on the walk cycle into a circle, which is not surprising.

We think it would be interesting to try Isomap on the entire corpus of animation data and then perform Euclidean blends on the resulting Euclidean mapping of the data to see what the inherent degrees of freedom of the motion manifold correspond to. Much more investigation needs to be done here.

### 12.2.6 Expressive IK

We made it only part of the way in our approach to expressive IK, but the early results seem promising. Even though the Eigenpostures did not give as good results as we expect, it would still be interesting to try and use them to keep a procedural IK solution from drifting too far away from the character's motion subspace as expressed by the examples. Future work by the Robotic Life group led by Breazeal will attempt to apply some of the algorithms and ideas to the more complex humanoid robot from Stan Winston Studios shown on the right in Figure 1-1.

### 12.2.7 Translational Joints

Sometime translational joints *are* desired by an animator to create expressive animation. For example, cartoons characters sometime have their eyes “pop out” of their head to show surprise. Since we ignore translational effects in joints, we cannot handle these types of prismatic joints yet.

The exciting field of *Geometric* (or *Clifford*) Algebra (see for example [17, 31, 68]), however, models the entire Euclidean transformation group (rotation and translation) in terms of the algebra generated by formal sums of a vector and a scalar rather than starting from a hypercomplex viewpoint. It generalizes the notion of the vector cross product to  $N$  dimensions by introducing the *geometric product* of two vectors as a sum of an inner and outer product of these elements, similarly to the quaternions. This creates a *multilinear* algebra which can describe rotations in  $N$  dimensions in a unified, principled framework. The geometric algebra for  $\mathbb{R}^3$  can be factored into a *dual quaternion* representation (see [59]). One of the unit quaternions represents the rotation effect and the other quaternion of arbitrarily-large radius<sup>1</sup> can represent translation since rotations of a large sphere look like translations up close. McCarthy uses dual quaternions in some of his robotics work [59, 17]. Jüttler also shows how to create rational splines (Bezier and B-spline) using proportional dual quaternions in [48] uses them in [49].

We also have attempted to use the geometric algebra to calculate a Bayesian solution to Inverse Kinematics problem using the QuTEM model as a prior on the joint's mobility, but got stuck in the derivation. The problem seems to lead to a generalized eigenvector problem, but more work needs to be done.

## 12.3 Conclusions

This thesis presented several new mathematical and computational building blocks for the design of real-time expressive interactive quaternions by exploiting the quaternion representation of spatial rotation for modeling joint rotation of a skeletal articulated figure.

Our goal was to create a set of building blocks that:

- Are computationally efficient
- Are mathematically robust
- Leverage the animator's skill in the form of examples

We now present conclusions from using quaternions and their exponential mappings to design these computational building blocks.

**Pose metrics** The pose metrics we provide are far superior to a Euclidean norm on Euler angle triples as examples are farther apart. We conclude that they are more mathematically robust.

**slime** The slime algorithm was the first algorithm we developed and therefore has gotten the most use. Its successful use in our early motor system for *Swamped!* and (void\*) as well as the subsequent successful incorporation into Downie's pose-graph algorithm [20] demonstrate its robustness. The fact that we can use it to blend multiple characters on the screen at one time in a subset of CPU cycles demonstrates its efficiency. On the other hand, it can only be used on internal joints without modification.

---

<sup>1</sup>Hanson calls this the "Giant Beach Ball."

Since the majority of joints in a character are internal, this is not a limitation for its use in figure animation blending. We conclude that it is a good building block for pseudo-linear unit quaternion blending. Since it affords excellent extrapolation behavior due to its use of the Lie algebra of quaternions, it also allows us to leverage the animator by requiring fewer examples and allowing us to generate caricatures of example animations.

**sasquatch** The iterative sasquatch algorithm has not been used in a full engine yet, but trial experiments have shown that it produces desirable results, as we depicted in Figure 7-7 and Figure 7-8. Although it is iterative, it is linear and robust in convergence. This taken with the fact that it need only be used on root nodes implies that it can be used efficiently in an engine.

**QuRBF** We used our quaternion extension to Radial Basis Functions using the slime primitive to implement Rose’s Verbs and Adverbs [44] seminal work in real-time animation blending. This implementation was used successfully in *Swamped!* and (void\*). This demonstrates both the usefulness of the slime blending algorithm as well as the usefulness of being able to approximate unit quaternion-valued functions. Rose’s work was also motivated by a need to leverage an animator’s skill, but he found poor extrapolation behavior, which we did not. We argue that the quaternion group affords this behavior as we argued above since our slime algorithm has much better extrapolation as well as interpolation properties than the Euler angle representation which Rose used. Our work complements his in that we show how to increase both performance and behavior of his framework by using quaternions while still being able to use many of his techniques and design recommendations for creating appropriate pose-blending examples.

**QuTEM** The QuTEM statistical model for learning a model of joint motion from example data was shown to be useful for choosing an appropriate tangent space for slime, calculating fast joint limits for QuCCD and quickly mapping large amounts of quaternion data to a local hemisphere of  $S^3$  for handling antipodal symmetry when required. Furthermore, it can be used as a quaternion kernel function in other machine learning algorithms. Furthermore, early results at using it to synthesize new animations similar to another animation suggest that it will be useful for leveraging the animator’s skill. We conclude that it is a very useful building block for designing many expressive interactive character algorithms.

**QuCCD** Our quaternion extension to CCD demonstrates that by exploiting a quaternion representation of joints, we can not only simplify the mathematics of an IK algorithm, we can also increase its computational efficiency. On the other hand, the use of a joint constraint model as a projection operation slows or stops convergence of the CCD algorithm. On the positive side, we have described how it should be possible to extend this to take constraints into account in a more principled way. The CCD algorithm also does not need to deal with ill-conditioning caused by coordinate singularities (singular Jacobian matrix) and is much, much faster in general. Full

optimization techniques or Jacobian methods are still in general too slow to be viable for multiple characters with multiple IK chains. We therefore conclude that the QuCCD is a useful building block.

**Joint Limits** Our model of joint limits is learnable from data, so allows us to leverage the animator’s ability. In addition, it is convex, which affords mathematical robustness by keeping an IK solver from getting stuck on a corner and requiring the need for a complicated non-linear programming algorithm such as described in Badler [3]. Furthermore, we describe how the limits can be computed simply with an exponential mapping, rotation and non-uniform scale, followed by a vector magnitude comparison. These operations are fairly efficient and therefore we conclude that the joint limit model is computationally efficient.

**Eigenpostures** Our early experiments with eigenpostures suggests that they offer some computational efficiency advantage by potentially allowing for animation compression. We cannot make claims about the robustness until more work is done. Since they potentially offer the ability to directly learn a model of the subspace of motion of a character from a corpus of animation data, they should allow us to leverage the animator. Our results to date do not support nor invalidate this claim. More work needs to be done.

**Expressive IK** Our initial results in combining our building blocks to approach the problem of Expressive IK have shown that a hybrid example-based pose-blending algorithm coupled with a numerical IK procedure can leverage the animator by reducing the number of example animations required. The fact that pose-blending can be used to “get the solution close” before an IK algorithm is applied demonstrated that a hybrid can also speed up the calculation of an appropriate IK solution, which is often the slowest building block in any character engine. We cannot make robustness claims yet as there are still several issues with CCD in general. Hecker’s recent findings might help alleviate some of these problems, however, as he demonstrated that it can be made fairly robust with a few extensions.

**Exponential Map** In our work, we discovered that the exponential mapping of the quaternions is an extremely useful primitive for performing a robust local-linearization of the quaternion group. We used this fact in the design of all of our algorithms, which shows that it is an important primitive that is often ignored. By coupling it with a unit quaternion representation instead of using it as a parameterization of rotation itself, we can avoid some of the problems encountered by Grassia in his work [29]. By using a unit quaternion joint representation, we gain the benefit of a straightforward calculus and an algebra of pose. Specifically, we avoid the need to calculate the exponential and logarithmic maps in order to compose two rotations, which is a very common operation in forward kinematics as we showed in Chapter 5. Since these maps use trigonometric functions, they can be expensive computational on some platforms. The quaternion composition of rotations involves only multiplications and additions, making it much more efficient.



**Physical Robots** Our quaternion pose-blending primitives were also applied to the motion of the Public Anenome robot (Figures 10-17 and 10-18) along with a CCD algorithm.

From these conclusions we make the final conclusion:

The quaternion representation of joint rotation, along with its exponential mapping and Lie algebra, allow us to create building blocks for real-time expressive interactive character engines that are computationally efficient, mathematically robust and leverage the ability of an animator.

## 12.4 Summary of Contributions

We summarize the main contributions of this research:

- Appropriate posture metrics for use in example-based algorithms that require domain-specific metrics for robustness and performance
- The fast slime algorithm to give the optimal average approximate blending performance and good extrapolation performance for internal (non-root) joints.
- The iterative sasquatch algorithm for an exact weighted blend of  $n$  quaternions for handling root nodes and other joints where slime is not enough.
- How to use these two blend primitives for non-linear blending and a specific description of using them with Radial Basis Functions.
- The QuTEM statistical model for learning a model of joint motion from example data, synthesizing new joint orientations similar to example data, choosing an appropriate tangent space for slime, calculating fast joint limits for QuCCD and quickly mapping large amounts of quaternion data to a local hemisphere of  $S^3$  for handling antipodal symmetry when required.
- QuCCD : a quaternion version of the heuristic Cyclic Coordinate Descent (CCD) IK algorithm.
- A simple, intuitive and fast way to estimate joint rotation limits and inherent degrees of freedom using the QuTEM for use in blending and IK.
- Eigenpostures : How to use Principal Component Analysis (PCA) on pose data for finding an “expressive” subspace of the full motion-space.
- An initial investigation of Expressive Inverse Kinematics using all of the primitives in conjunction.



# Appendix A

## Hermitian, Skew-Hermitian and Unitary Matrices

This appendix gives a quick mathematical background in unitary matrices. When we refer to the group  $SU(2)$ , we refer to the group of special (determinant one) unitary (rotation) 2 by 2 matrices consisting of complex entries.

For complex matrices, we introduce the *hermitian transpose* operator on complex matrices as the analog of the transpose for real matrices.

**Definition 6** The **hermitian transpose**  $A^\dagger$  of a matrix  $A$  with complex entries is the **conjugate transpose** of the matrix,  $(A^*)^T$ , or the matrix with each entry replaced by its complex conjugate ( $z \rightarrow z^*$ ) and then transposed ( $A_{ij} \rightarrow A_{ji}$ ):

$$A^\dagger = (A^*)^T$$

Immediately we see that if  $A$  has real entries, the hermitian transpose is the same as the real transpose.

Using this definition, we extend the inner product of vectors in  $\mathbb{R}^n$  to vectors in  $\mathbb{C}^n$ .

**Definition 7** The **inner product** of  $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$  is

$$\mathbf{x}^\dagger \mathbf{y} = \sum_i^n x_i^* y_i$$

**Definition 8** The **magnitude** of length of a complex vector  $\mathbf{x} \in \mathbb{C}^n$  is the sum of the squared moduli of the components:

$$\|\mathbf{x}\| \stackrel{\text{def}}{=} \mathbf{x}^\dagger \mathbf{x} = \sum_i^n |x_i|^2$$

The analog of the symmetric real matrices are the complex *Hermitian* matrices.

**Definition 9** A complex matrix  $A$  is called **Hermitian** if it equals its Hermitian transpose:

$$\text{If } A = A^\dagger \text{ then } A \text{ is Hermitian}$$

The analog of an orthogonal real matrix is the complex unitary matrix.

**Definition 10** *A complex matrix is **unitary** if*

$$\mathbf{A}^\dagger \mathbf{A} = \mathbf{I}$$

Clearly for unitary  $\mathbf{A}$ ,  $\mathbf{A} = \mathbf{A}^{-1}$ , as was the case with orthogonal matrices. Finally, we present the complex analog of a real skew-symmetric matrix.

**Definition 11** *A complex matrix  $\mathbf{A}$  is **skew-Hermitian** if*

$$\mathbf{A}^\dagger = -\mathbf{A}$$

**Corollary 1** *If complex  $\mathbf{A}$  is Hermitian, then  $(i\mathbf{A})$  is skew-Hermitian.*

Likewise, the complex matrix exponential work similarly, though we will not go into much detail (the interested reader should see the wonderful introduction to theoretical algebra by Artin [2]).

**Property 3** *The exponential of a skew-Hermitian matrix of trace zero is unitary.*

This is the complex analog of the fact that the matrix exponential of a skew-symmetric trace-zero matrix in the real 3x3 matrices is special orthogonal, or  $\mathbf{SO}(3)$ . It can be shown with the power series form of the exponential.

These skew-symmetric (hermitian) matrices are very similar to each other and form the Lie algebra elements of the Lie groups  $\mathbf{SO}(3)$  and  $\mathbf{SU}(2)$ . In fact, it can be shown that the Lie algebras of  $\mathbf{SO}(3)$  and  $\mathbf{SU}(2)$  are in fact isomorphic to a proportionality constant (since the group  $\mathbf{SU}(2)$  double-covers  $\mathbf{SO}(3)$  and we get half-angles). In other words, infinitesimally (or locally) the groups  $\mathbf{SU}(2)$  and  $\mathbf{SO}(3)$  are isomorphic since they have the same infinitesimal generators — these skew-symmetric matrices which in effect encode the axis and angle of the two rotation groups (the classical rotation group  $\mathbf{SO}(3)$  and the quantum spin group  $\mathbf{Spin}(2) = \mathbf{SU}(2)$ ). Globally, however, we know the structures are different since one double-covers the other.

# Appendix B

## Multi-variate (Vector) Gaussian Distributions

This appendix gives a brief introduction to vector Gaussian probability distributions.

### B.1 Definitions

The multi-variate (vector) Gaussian probability density function (p.d.f.) for a vector  $\mathbf{x} \in \mathbb{R}^n$  has the form

$$p_{\mathbf{x}}(\mathbf{x}) = \frac{1}{(2\pi)^{n/2} |\mathbf{K}|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T \mathbf{K}^{-1}(\mathbf{x}-\mathbf{m})}$$

where the scalar factor in front of the exponential is the normalizing constant (so the p.d.f. integrates to one over its domain),  $\mathbf{K}$  is called the *covariance matrix* and  $\mathbf{m}$  is the mean (see, for example, [83] for an introduction to multi-variate Gaussian densities). We use a subscript on the density to make it clear which variable it is the density of, as we will use more than one at a time often.

Compare this to the scalar version of the normal distribution, which is likely more familiar:

$$p_x(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-m)^2}{2\sigma^2}}.$$

The covariance matrix is usually decomposed into a *principal axis* description as:

$$\mathbf{K} = \mathbf{U}\mathbf{D}\mathbf{U}^T$$

where  $\mathbf{U}$  is an orthogonal (rotation) matrix and  $\mathbf{D}$  is diagonal. This is sometimes called an eigenvector decomposition since the columns of  $\mathbf{U}$  are the set of eigenvectors of the covariance matrix and the corresponding eigenvalues are on the diagonal of  $\mathbf{D}$ . Specifically, recall that an eigenvector of a matrix  $\mathbf{A}$  satisfies the equation:

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

where  $\mathbf{x}$  is a vector of appropriate dimension called an *eigenvector* of  $\mathbf{A}$  and  $\lambda$  is a scalar associated with  $\mathbf{x}$  called the *eigenvalue* of the eigenvector  $\mathbf{x}$ . An eigenvector is merely scaled by some value when the matrix is applied to it, so the eigenvectors describe the *invariant directions* or *principal axes* of the matrix. In general, a *real* matrix  $\mathbf{A}$  can have *complex* eigenvectors and eigenvalues! Luckily, we state without proof here that symmetric real matrices which are positive semi-definite (having determinant greater than or equal to zero) will always have real, non-negative eigenvalues and real eigenvectors. Covariance matrices fall into this class, so we need only handle real eigenvectors and eigenvalues. A numerical algorithm (which we use extensively) for finding all the eigenvectors and eigenvalues of a matrix can be found in *Numerical Recipes* [65].

The reasoning for the decomposition is that the eigenvector matrix  $\mathbf{U}$  of the covariance can be used to *diagonalize* the covariance matrix. In our case, the eigenvectors in  $\mathbf{U}$  will form a new basis in which the components are uncorrelated. Using the eigenvector matrix to change basis:

$$\mathbf{U}^T \mathbf{K} \mathbf{U} = \mathbf{D}$$

we see explicitly that the matrix  $\mathbf{U}$  can turn the covariance matrix into a diagonal matrix. Mathematically, this is showing how any covariance matrix (which we use to model Gaussians) can be diagonalized by finding its eigenvectors.

Using this decomposition, the inverse of  $\mathbf{K}$  can also then be found using the simpler inverse formulas for these decomposition factors:

$$\mathbf{K}^{-1} = (\mathbf{U} \mathbf{D} \mathbf{U}^T)^{-1} = \mathbf{U}^T \mathbf{D}^{-1} \mathbf{U}$$

since the inverse of a rotation matrix is its transpose

$$\mathbf{U}^{-1} = \mathbf{U}^T$$

and, the inverse of the diagonal matrix is just the diagonal matrix with the reciprocals in place of the original entry. Notice that the inverse will only exist if the diagonal matrix is non-singular (has all positive entries). If any variances are zero, the density is singular. We are forced to handle this by either using the pseudoinverse or by forcing a lower bound on the variance (we shall use both ideas in our work, depending on what we need to do). In general, practitioners force the variances to a small lower value to avoid singular data issues.

## B.2 Isoprobability Contours and Ellipsoids

In order to simplify the use and discussion of Gaussian densities, we will usually write them in a more compact form which will make the comparison between the QuTEM density and the vector space Gaussian more clear, as well as show how Gaussians relate to ellipsoids.

First, consider the isocontours of the Gaussian p.d.f. found by setting the density to some constant value, call it  $c$ :

$$p_{\mathbf{x}}(\mathbf{x}) = \frac{1}{(2\pi)^{n/2} |\mathbf{K}|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T \mathbf{K}^{-1}(\mathbf{x}-\mathbf{m})} = c$$

Since the value is constant, we can take the logarithm of both sides to get

$$(\mathbf{x} - \mathbf{m})^T \mathbf{K}^{-1}(\mathbf{x} - \mathbf{m}) = r^2 \quad (\text{B.1})$$

where we have collected the constant scalar terms into a new scalar,  $r^2$  in the obvious way. Writing the new constant as  $r^2$  is suggestive; in fact, it can be considered as the radius of an ellipsoid. This is made explicit by using the diagonalization technique described above to change the basis of  $\mathbf{x} - \mathbf{m}$  into the principal axes of the covariance matrix:

$$\mathbf{x}' - \mathbf{m}' = \mathbf{U}^T(\mathbf{x} - \mathbf{m})$$

where the prime ( $\prime$ ) denotes that the vector is represented in the principal basis. This gives us the simpler formula for the log of the density in the principal basis:

$$(\mathbf{x}' - \mathbf{m}')^T \mathbf{D}^{-1}(\mathbf{x}' - \mathbf{m}') = r^2 \quad (\text{B.2})$$

which immediately can be multiplied out (since  $\mathbf{D}$  is diagonal) in order to give

$$\frac{(x'_1 - m'_1)^2}{d_1} + \frac{(x'_2 - m'_2)^2}{d_2} + \dots + \frac{(x'_n - m'_n)^2}{d_n} = r^2 \quad (\text{B.3})$$

which should be familiar as the equation of an ellipsoid in  $n$ -dimensions with a “radius”  $r$  (the two dimensional case is simple to visualize). The principal axes are the eigenvectors, as we have said, and have length  $2r\sqrt{d_i}$ . We call this  $r$  the *radius* to be consistent with the spherical case, where all the  $d_i = 1$ .

The other thing to notice about Equation B.3 is that it is a log density. If we exponentiate it back into a p.d.f., we notice that we have a sum of scalars in the exponent. Therefore, we can factor this into a product of separate, uncorrelated scalar Gaussians, each one aligned along one of the principal directions and with variance the same as the variance in that principal direction:

$$p_{\mathbf{x}}(\mathbf{x}) = c e^{\frac{(x'_1 - m'_1)^2}{d_1}} e^{\frac{(x'_2 - m'_2)^2}{d_2}} \dots e^{\frac{(x'_n - m'_n)^2}{d_n}} \quad (\text{B.4})$$

This factoring capability is another reason why Gaussians are often used in practice. This factored form is mostly useful in our case for generating samples, as we describe below.

## B.3 Mahalanobis distance

Notice that the original logdensity equation (Equation B.1) is a simple quadratic expression in the input vector with respect to the mean. In general, quadratic vector functions of the form:

$$\mathbf{y}^T \mathbf{M} \mathbf{y}$$

can be considered as a *distance function* on the vector  $\mathbf{x}$  with the *metric tensor*  $\mathbf{M}$  if  $\mathbf{M}$  is symmetric positive definite. Notice that if  $\mathbf{M}$  is diagonal, this just weights different dimensions (components) differently. If  $\mathbf{M}$  is the inverse of the covariance matrix, as in the logdensity ( $\mathbf{M} = \mathbf{K}^{-1}$ ), the distance function is known as the *Mahalanobis distance*.

Specifically, we can use Equation B.3 as a distance metric. It effectively divides through the principal variance directions (eigenvectors) by their variance (eigenvalues), therefore “sphere-izing” the space by making the variance “units” the same for all components. The Mahalanobis distance can therefore correctly be thought of as the distance of a point from the mean taking into account unequal variances, and is in units of “standard deviations.” In the scalar Gaussian case, most students are familiar with their grades defined in terms of standard deviations from the mean — this is just the Mahalanobis distance of their grade from the mean.

We have also shown how this quadratic form encodes the equation of an ellipsoid, where the particular ellipsoid is defined by the radius,  $r$  and particular variances  $\mathbf{a}$ . This fact lets us say that the *locus of all points at a constant Mahalanobis distance from the mean is an ellipsoid centered around the mean*. For example, the ellipsoid with Mahalanobis distance (radius) 1.0 is the set of all points one “standard deviation” from the mean.

Computationally, we define the Mahalanobis distance of a vector  $\mathbf{x}$  from the mean  $\mathbf{m}$  to be

$$d_{mahalanobis}^2(\mathbf{x}) = (\mathbf{x} - \mathbf{m})^T \mathbf{K}^{-1} (\mathbf{x} - \mathbf{m}) \quad (\text{B.5})$$

We can use the Mahalanobis distance to simplify the appearance of the Gaussian density function:

$$p_{\mathbf{x}}(\mathbf{x}) = ce^{-\frac{1}{2}d_{mahalanobis}^2(\mathbf{x})}$$

where we have now made the mean and covariance implicit parameters of the Mahalanobis distance function and labelled our normalization constant as  $c$ . Given the mean and covariance, we can simply find the distance of any point by using the quadratic form above.

Following in this vein, we can write the Mahalanobis distance simply as the magnitude of the vector transformed into the principal basis, which will prove useful for other purposes:

$$\begin{aligned} d_{mahalanobis}^2(\mathbf{x}) &= (\mathbf{x} - \mathbf{m})^T \mathbf{K}^{-1} (\mathbf{x} - \mathbf{m}) \\ &= \left[ \mathbf{y}^T \mathbf{U} \mathbf{D}^{-1/2} \right] \left[ \mathbf{D}^{-1/2} \mathbf{U}^T \mathbf{y} \right] \\ &= \mathbf{s}^T \mathbf{s} \\ &= \|\mathbf{s}\|^2 \end{aligned} \quad (\text{B.6})$$

where we have noticed that the quadratic product is just the magnitude squared of the input vector translated to the mean, rotated into the principal axes of the Gaussian, and then sphere-ized by dividing through the standard deviations (square root of the variance). Specifically,

$$\mathbf{s} = \mathbf{D}^{-1/2} \mathbf{U}^T \mathbf{x} \quad (\text{B.7})$$



where we use the  $\mathbf{s}$  to stand for the vector scaled to unit variance, and  $\mathbf{y}$  to represent  $\mathbf{x}$  in the coordinate system with the origin translated to the mean  $\mathbf{m}$ . This is the form of the Mahalanobis distance, and therefore the Gaussian density, that we shall use computationally. The transformation from an arbitrary point into a point aligned with the principal axes and scaled to have unit variance is a simple rotation followed by a non-uniform scale. The magnitude of this vector, again, is our Mahalanobis distance:

$$d_{mahalanobis} = \|\mathbf{s}\| = \|\mathbf{D}^{-1/2}\mathbf{U}^T\mathbf{y}\|$$

## B.4 Sampling a Multi-Variate Gaussian Density

To generate random vectors according to a multi-variate Gaussian density, we use the standard technique of generating the vector in the principal (uncorrelated, diagonalized) basis of the covariance matrix by using Box-Muller scalar sampling algorithm in *Numerical Recipes* [65] to generate the uncorrelated components, then rotating the uncorrelated sample into the basis of the actual vector. This section details how this is done and can be skipped by those familiar with the technique.

Recall that we can diagonalize the covariance into principal axes using an eigenvector decomposition. In this basis, the density factors, as we saw, and can be considered as  $n$  separate scalar Gaussian densities, one for each component. Put another way, each component of the random vector in this basis can be thought of as a separate scalar Gaussian density. Therefore, to generate a *vector* sample, we need only generate a sample in the principal basis using and then rotate it into the basis of our actual random vector. If we have factored the covariance matrix  $\mathbf{K}$  into  $\mathbf{U}\mathbf{A}\mathbf{U}^T$ , then our component densities will simply be zero mean densities with variances  $a_i$ , the diagonal entries of  $\mathbf{A}$ , or entries in our QuTEM's variance vector  $\mathbf{a}$ .

Mathematically, in the principal basis (denoted by the  $\prime$ ) our random vector's components are distributed according to the uncorrelated scalar densities:

$$x'_i \leftarrow N(0, a_i)$$

where we denote sampling from a density with an arrow and the scalar Gaussian (Normal) density is written in the compact  $N(m, v)$  form with mean  $m$  and variance  $v$ .

To generate the scalar samples, we use the Box-Muller algorithm (as described in *Numerical Recipes* [65]) to generate a sample according to a zero-mean scalar Gaussian with variance  $a_i$ <sup>1</sup>. This gives us a sample vector,  $\mathbf{x}'$ , in the principal basis with components  $x'_i$ . Next we rotate this vector back into our standard basis in the tangent space by rotating by the  $\mathbf{U}$  matrix:

$$\mathbf{x} = \mathbf{U}\mathbf{x}'$$

As a side note, in our case of a 3-dimensional density, we mention that we actually will use the quaternion representation of  $\mathbf{U}$ ,  $\mathbf{u}$ , to perform this rotation for efficiency:

---

<sup>1</sup>Box-Muller cannot handle zero variances, but sampling from a zero variance density is simple — return the mean itself for that component, which is just zero for our zero-mean data.

$$\mathbf{x} = u\mathbf{x}'u^* .$$

## B.5 Recommended Reading

The treatment in this section was compiled from several sources. The author recommends [8, 65, 83] for a more in-depth treatment of probability theory from an engineering and computational stand-point.

# Appendix C

## Quaternion Numerical Calculus

This appendix introduces quaternion ordinary differential equations (ODE's) and how to solve them numerically using both the embedding space or intrinsically in the group using the exponential map. Since unit quaternions live on a sphere, care must be taken when integrating. We refer to the problem of finding the steady state for the sasquatch quaternion blending algorithm in Chapter 7 which motivated the need for solving unit quaternion ODE's.

We will change notation for unit quaternions slightly in this appendix for readability of some of the differential equations. We will be explicit about what group each of the terms belongs to if it is not clear from context.

### C.1 Solving Quaternion ODE's

The quaternion ODE in Equation 7.12 is non-linear since it lives on the sphere, so an analytic closed-form solutions seems unlikely, even for steady state. Several numerical integration techniques can be used, however. We will discuss two of these, both versions of the first order forward Euler integration scheme, which is the most simple and common, and its flaws are well-known. These methods are:

- Renormalized Embedding Space Euler Integration
- Intrinsic Euler Integration

We argue that the last of these, which is by far the least familiar, is the best for solving such systems. It is also the most elegant in that it does integration intrinsically within the group rather than relying on the calculus in the embedding space,  $\mathbb{R}^4$ , and renormalization. Since we are solving for a steady state solution where speed (number of iterations) is an issue, we would like the largest timesteps possible for convergence. We mention that both methods will assume that the angular velocity over a timestep is constant. This fact is not true, as the exponential map is location-dependent on the sphere, but for reasonable steps and a simple system like that we are trying to solve for steady-state, this is less of a worry.

## C.2 Embedding Euler Integration with Renormalization

The standard practice for solving a simple ODE in a vector space is to use *forward Euler integration*, which is based on a first order Taylor series expansion. Specifically, given the derivative at a point, Euler integration takes a step in the derivative direction scaled by the timestep:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \dot{\mathbf{x}}(t) \quad (\text{C.1})$$

This type of integration has many issues with error accumulation, but for our purposes it will be sufficient and fast.<sup>1</sup> Most importantly, it is simple.

If we apply this formula to the quaternion derivative, we always will step off the sphere, since all derivatives are tangent to the sphere. Worse, the larger the timestep, the further we step away from the group. In order to solve this, standard practice is simply to renormalize the quaternion, essentially projecting it back onto the sphere. The fact that we can use embedding space integration is a consequence of spheres looking flat locally, or like a vector space. If the steps are small enough, we get close to the actual solution, but otherwise we only get an approximation of the true integral. Formally, if we desire to integrate forward by a finite step  $\delta t$

$$\int_t^{t+\Delta t} \dot{Q} ds \approx \frac{t + \Delta t \dot{Q}}{\|t + \Delta t \dot{Q}\|} \quad (\text{C.2})$$

This formula leads to similar warping of the space as a standard Euclidean weighted average caused by stepping in the tangent plane and then renormalizing. A step of magnitude  $s$  in the tangent space does not lead to a step of magnitude  $s$  on the sphere. For larger timesteps, we get worse behavior of the integration step because the step moves further from the sphere. Therefore, this method usually requires small timesteps to be effective. We offer an alternative below.

## C.3 Intrinsic Euler Integration

We can also integrate Equation 7.12 intrinsically — that is, by staying on the surface of the sphere. Since

$$\dot{Q} = \omega q = q \omega' \quad (\text{C.3})$$

we can choose to integrate the local or inertial (global) angular velocities, even though we formulated the derivative locally.

To simplify the calculation, we will use the isomorphism between the quaternions and  $\text{SU}(2)$ , the group of complex unitary 2x2 matrices with determinant 1, called *Special Unitary*. These matrices are the complex analog of the special orthogonal real matrices. Formally, we have a mapping from a unit quaternion represented as a unit 4-vector ( $\mathbf{q} \in S^3$ ) to a matrix in  $\text{SU}(2)$ .

---

<sup>1</sup>For a great discussion of numerical integration techniques and problems, see Gershendfeld's book [24].

It is important to note that since the  $\omega$  actually represent vectors in  $\mathbb{R}^3$ , or pure quaternions that are *not* unit, these are also mapped into complex 2x2 matrices, though they will not have determinant 1. In fact, it can be shown that  $\omega$  will transform to a skew-hermitian matrix (that is,  $\Omega = -\Omega^*$ , where  $*$  denotes complex conjugation of the matrix. Furthermore,  $e^\Omega \in \text{SU}(2)$  if  $\Omega$  is skew-Hermitian with trace 0. The multiplication of the complex matrices is equivalent to the quaternion product due to the isomorphism. Therefore, we can map all our quaternions into 2x2 complex matrices and calculate the derivatives there using the familiar calculus of complex matrices. We will not prove this isomorphism, though the reader is referred to Artin [2] for more details or can work it out themselves as an exercise.

Formally, the isomorphism of Equation C.3 results in the complex matrix differential equation:

$$\dot{\mathbf{Q}}(t) = \mathbf{Q}(t)\Omega' \quad (\text{C.4})$$

which we must solve for  $\mathbf{Q}(t)$ . This equation can be solved formally with the *matrix exponential*, which converges absolutely for all complex matrices ([2]). To prove this fact, we will assume an *ansatz* solution, or guess, and then show that this is truly a solution by plugging it back in and solving for unknowns that make it true. Since we wish to solve the system in local coordinates, we assume the *ansatz* solution

$$\mathbf{Q} = \mathbf{A}e^{\mathbf{S}t} \quad (\text{C.5})$$

where  $\mathbf{A}$  is a constant matrix in  $\text{SU}(2)$  which effectively defines the initial condition in terms of the inertial frame. Also,  $\mathbf{S}$  is an unknown constant matrix which we know must be skew-Hermitian, since  $\mathbf{Q}$ ,  $\mathbf{A}$  and  $e^{\mathbf{S}t}$  all must lie in  $\text{SU}(2)$ . This fact implies that  $\mathbf{S}$  must be skew-Hermitian, or equivalently, must lie in the Lie algebra of  $\text{su}(2)$ , denoted as  $\text{su}(2)$ <sup>2</sup>. It is well-known in differential equation theory that if we find any solution to the differential equation, it must be unique (given conditions which hold in our case, but we ignore the details). Therefore, we must plug our *ansatz* into Equation C.4 and see if it is a valid solution. If so, we have found the solution. Differentiating Equation C.5 gives us

$$\dot{\mathbf{Q}} = \mathbf{A}\mathbf{S}e^{\mathbf{S}t} \quad (\text{C.6})$$

since the derivative of the matrix exponential is

$$e^{\mathbf{B}t} = \mathbf{B}e^{\mathbf{B}t}.$$

This follows from the formal power series for the exponential:

$$e^{\mathbf{A}} = \mathbf{I} + \mathbf{A} + \frac{\mathbf{A}^2}{2!} + \frac{\mathbf{A}^3}{3!} + \dots$$

and can be worked out by the reader. Plugging our *ansatz* and its derivative back into the original equation gives us

---

<sup>2</sup>The reader does not need the details of Lie theory here, but many of these calculations can be expressed in terms of Lie group theory. We refer the reader to Artin [2] for an introduction to Lie theory, or [71] for an introduction focusing on physics calculations.

$$\mathbf{A} \mathbf{S} e^{\mathbf{S}t} = \mathbf{A} e^{\mathbf{S}t} \mathbf{\Omega}' . \quad (\text{C.7})$$

We must solve this equation for  $\mathbf{A}$  and  $\mathbf{S}$ . First, we use the fact that  $\mathbf{S}$  and  $e^{\mathbf{S}}$  commute (since  $\mathbf{S}$  can be thought of as an infinitesimal rotation, which commutes). Explicitly,

$$\mathbf{S} e^{\mathbf{S}} = e^{\mathbf{S}} \mathbf{S} \quad (\text{C.8})$$

which can be found directly from the power series expansion of the exponential. This property lets us write the equation as

$$\mathbf{A} e^{\mathbf{S}t} \mathbf{S} = \mathbf{A} e^{\mathbf{S}t} \mathbf{\Omega}' \quad (\text{C.9})$$

which we can obviously simplify. First, we cancel  $\mathbf{A}$  since its inverse exists. Also, the inverse of the matrix exponential exists, and we can left-cancel it as well, leading to the simple *characteristic polynomial* of the equation

$$\mathbf{S} = \mathbf{\Omega}' \quad (\text{C.10})$$

which makes sense since both matrices are known to be skew-Hermitian. Now we need to solve for the unknown constant matrix coefficient  $\mathbf{A}$  using the constraint that we know the boundary condition at the beginning of the timestep. Assuming without loss of generality that  $t = 0$  at the start of the timestep (we can change the time variable if not), let the value of  $\mathbf{Q}$  at  $t = 0$  be  $\mathbf{Q}_0$ . Plugging this into our solution at  $t = 0$  gives us:

$$\mathbf{Q}(0) = \mathbf{Q}_0 = \mathbf{A} e^{\mathbf{\Omega}'0} \quad (\text{C.11})$$

which immediately implies that  $\mathbf{A} = \mathbf{Q}_0$ . Therefore, our solution for ome timestep, assuming  $t = 0$  at the start

of timestep and that  $\mathbf{S}$  is constant over the integration interval (which is true for small  $\Delta t$ ), we get the following forward Euler integration formula for the timestep:

$$\mathbf{Q}(t + \Delta t) \approx \mathbf{Q}(t) e^{\mathbf{\Omega}'(t)\Delta t} . \quad (\text{C.12})$$

Due to our isomorphism, we can convert this directly back into the quaternion algebra, giving us the quaternion equation

$$q(t + \Delta t) \approx q(t) e^{\omega'(t)\Delta t} . \quad (\text{C.13})$$

By integrating each timestep assuming the angular velocity is constant at each position, we get a trajectory of the system of the form:

$$q(N\Delta t) = q(0) e^{\omega'(0)\Delta t} e^{\omega'(\Delta t)} e^{\omega'(2\Delta t)} \dots e^{\omega'(N\Delta t)} . \quad (\text{C.14})$$

We note here that we can also solve the differential equation in terms of the global angular velocity, which leads to a solution of the form

$$q(t + \Delta t) \approx e^{\omega(t)t} q(t) \quad (\text{C.15})$$

which is clearly of similar form but with all multiplication orders reversed and the global angular velocity used. We also mention here that this type of trajectory is given for describing quaternion curves with simple higher derivatives in [52].

An intuitive way to think about this numerical integral is that transform the quaternion derivative to a vector quantity (either the local or global angular velocity vector), take a step in this direction, then transform the answer back onto the sphere using the exponential map. Note that this is an approximate answer to the integral, but for our simple first order system we have found this to be valid. Also, since we seek the steady state solution, the approximate answer and actual answer should be identical if the system converges.





# Appendix D

## Quaternions: Group Theory, Algebra, Topology, Lie Algebra

This appendix is a more formal algebraic treatment of quaternions and will serve as a reference for the more mathematically inclined, such as those who already know quantum physics, or who would like to learn it. The interested reader is also directed to the recent book by Gallier [23] or the edited collection on geometric algebra [17].

### D.1 Vector Space

Most algorithms are designed to work in a vector space rather than on a curved manifold such as a sphere. We will see below that unit quaternions do not form a vector space. What is a vector space?

**Definition 12** A real vector space is a set  $\mathcal{V}$  together with the two composition laws of:

**Addition**  $\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$ , written  $\mathbf{x} + \mathbf{y} = \mathbf{z}$

**Scalar multiplication**  $\mathbb{R} \times \mathcal{V} \rightarrow \mathcal{V}$ , written  $a\mathbf{x}$ .

Furthermore, the following axioms must hold for all elements:

1. Addition with  $\mathcal{V}$  forms an abelian (commutative) group.
2. Scalar multiplication is associative with multiplication by real numbers.
3. Scalar multiplication by the real number 1 is the identity operation.
4. Two distributive laws hold:

$$(a + b)\mathbf{x} = a\mathbf{x} + b\mathbf{x} \quad a(b\mathbf{x}) = (ab)\mathbf{x}$$

Specifically, the unit quaternions are not a vector space since addition is not closed. Put simply, adding two unit vectors does not produce another unit vector. Therefore, all Euclidean operations or algorithms on unit quaternions must take this into account.

## D.2 The Rotation Group in $\mathbb{R}^3$

Much of this dissertation discusses the rotation group of  $\mathbb{R}^3$ , three-dimensional Euclidean space, and various ways to parameterize, represent and compute within it. This group of rotations is called  $\text{SO}(3)$ , which stands for *special orthogonal*  $3 \times 3$  matrices. Recall that an orthogonal matrix consists of orthogonal column vectors which are of unit magnitude — in other words, the columns for an *orthonormal basis* for the space. Of these matrices, called  $\mathbb{O}(3)$ , there are two subsets: those with  $\det = +1$  and  $\det = -1$ , where  $\det$  is the matrix determinant. The subset of  $\mathbb{O}(3)$  with  $\det = +1$  are called the *special orthogonal* matrices.

A rotation transformation must preserve magnitude of vectors and must preserve orientation of coordinate systems in the space. In other words,  $\mathbf{R} \in \text{SO}(3)$  must preserve the inner product of two vectors:

$$(\mathbf{R}\mathbf{x})^T(\mathbf{R}\mathbf{y}) = \mathbf{x}^T\mathbf{y}.$$

It follows that

$$\mathbf{x}^T\mathbf{R}^T\mathbf{R}\mathbf{y} = \mathbf{x}^T\mathbf{y}$$

which is only true for the constraint

$$\mathbf{R}^T\mathbf{R} = \mathbf{I}$$

where  $\mathbf{I}$  is the identity matrix. Clearly

$$\mathbf{R}^T = \mathbf{R}^{-1}$$

which describes the set of orthogonal matrices as defined above. Unfortunately, the constraint holds for  $\det \mathbf{R} = \pm 1$ . We need to choose the matrices with  $\det \mathbf{R} = 1$  since to preserve the orientation of space, i.e.:

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = \hat{\mathbf{z}}$$

we must have

$$(\mathbf{R}\hat{\mathbf{x}}) \times (\mathbf{R}\hat{\mathbf{y}}) = \mathbf{R}\hat{\mathbf{z}}$$

where  $\times$  denotes the cross product of two vectors. Consider the matrix  $-\mathbf{I}$  which clearly has  $\det = -1$  and is orthogonal. Plugging into the constraint gives

$$(-\hat{\mathbf{x}}) \times (-\hat{\mathbf{y}}) = \hat{\mathbf{x}} \times \hat{\mathbf{y}} = -\hat{\mathbf{z}}$$

which is a contradiction. The negative determinant matrices change the orientation of space, and therefore are *inversions* of the space rather than rotations.

## Axis-Angle Representation

According to a famous theorem of Euler, every rotation in  $\mathbb{R}^3$  can be parameterized by some axis and some angle. This representation is called **axis-angle** notation.

**Theorem 1** (Euler) *Every rotation  $\mathbf{R} \in \mathbb{R}^3$  can be represented as a rotation around some axis  $\hat{\mathbf{n}}$  by some  $\theta$ , which we denote as  $\mathbf{R}(\theta, \hat{\mathbf{n}})$ .*

We do not prove the theorem here, but offer a few corollaries which follow immediately.

**Corollary 2** *Every composition of rotations  $\mathbf{R}_1 \circ \mathbf{R}_2$  can be written as a single rotation around some axis and angle.*

This result follows immediately from the closure of rotations.

Another way to visualize axis-angle notation is as a point in the solid ball in  $\mathbb{R}^3$  of radius  $\pi$  (notice  $\pi$  is enough since a rotation of more than  $\pi$  can be represented as a rotation less than  $\pi$  around  $-\hat{\mathbf{n}}$ ). Then we can represent the vector  $\theta\hat{\mathbf{n}}$  in the ball as the rotation  $\mathbf{R}(\theta, \hat{\mathbf{n}})$  by separating the magnitude and direction of the point. Clearly the center of the ball is the identity rotation. Also, notice that opposite points on the *surface* of the ball are identified since

$$\mathbf{R}(\pi, \hat{\mathbf{n}}) = \mathbf{R}(\pi, -\hat{\mathbf{n}}).$$

## D.3 Quaternion Theory

A *quaternion* is a hypercomplex number with one real and three imaginary components discovered by Sir William Rowan Hamilton in 1866 [33].

### D.3.1 Hypercomplex Representation

This section describes the original quaternion formulation as described by Hamilton as an extension of the complex numbers  $\mathbb{C}$  to four dimensions (there does not exist a three-dimensional, or any odd-dimension, extension to complex numbers).

**Definition 13** *A quaternion is a hypercomplex number which can be written in the form*

$$q = w + xi + yj + zk$$

where  $w, x, y, z \in \mathbb{R}$  and  $i, j, k$  are each distinct imaginary numbers such that

$$i^2 = j^2 = k^2 = ijk = -1$$

and pairs multiply similarly to a cross product in a right-handed manner

$$ij = -ji = k$$

$$jk = -kj = i$$

$$ki = -ik = j$$

Hamilton called the real part a *scalar* and the 3-component imaginary part a *vector*, which are the precursors of the modern definitions of vector analysis which were derived from quaternions. In a strange quirk of fate, quaternions, which spawned vector analysis, were then reinterpreted in terms of vector analysis which (in the author's opinion) obscures the beauty of the original description. We describe the vector representation in Chapter 3.

We will denote the set of all quaternions as  $\mathbb{H}$  after Hamilton. A few special subsets of  $\mathbb{H}$  deserve mention. First, the set of quaternions of the form  $(w + 0i + 0j + 0k)$  with only a scalar term are called *pure scalars*. It should be obvious that this subset is in one-to-one correspondence with the real numbers  $\mathbb{R}$ . The subset of quaternions with 0 scalar  $(0 + xi + yj + zk)$  are called *pure quaternions* or *pure vectors*. Those familiar with vector analysis will notice that the set of pure vectors can be represented as a vector in  $\mathbb{R}^3$ . We discuss this further below. Finally, we denote the set of unit-magnitude quaternions (magnitude will be defined formally below) as the set  $\hat{\mathbb{H}}$ .

Perhaps more elegantly, the properties of quaternions can be derived similarly to complex numbers by interpreting addition and multiplication as quadronomials and reducing combinations of imaginary terms according to the rules above.

**Definition 14** Let  $q_1, q_2 \in \mathbb{H}$ . The **addition operator**  $+$  is defined as

$$\begin{aligned} q_1 + q_2 &= (w_1 + x_1i + y_1j + z_1k) + (w_2 + x_2i + y_2j + z_2k) \\ &= ((w_1 + w_2) + (x_1 + x_2)i + (y_1 + y_2)j + (z_1 + z_2)k) \end{aligned}$$

Quaternions add component-wise like complex numbers.

**Theorem 2** Quaternions form an abelian (commutative) group  $\{\mathbb{H}, +\}$  under addition.

PROOF We need to show the four group properties plus commutivity hold:

1. *Closure*: If  $p, q \in \mathbb{H}$  then  $p + q \in \mathbb{H}$ .
2. *Associativity*:  $(p + q) + r = p + (q + r) \quad \forall p, q, r \in \mathbb{H}$
3. *Identity*:  $\exists 0 \in \mathbb{H}$  such that  $p + 0 = 0 + p = p \quad \forall p \in \mathbb{H}$ .
4. *Inverse*: For any  $p \in \mathbb{H}$ ,  $\exists (-p) \in \mathbb{H}$  such that  $p + (-p) = (-p) + p = 0$ .
5. *Commutivity*:  $p + q = q + p \quad \forall p, q \in \mathbb{H}$ .

All these properties can be proven trivially by using the group properties of the reals for each real component of the quaternion 4-tuple and the definition of quaternion addition. ♣

We can also define a multiplication operator on the quaternions by using normal polynomial multiplication with  $i, j, k$  as the variables. Combinations of  $i, j, k$  are then reduced using the quaternion rules.

**Definition 15** Let  $q_1, q_2 \in \mathbb{H}$ . Then the **quaternion product** is defined as:

$$\begin{aligned} q_1 q_2 = & ((w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) + \\ & (y_1 z_2 - y_2 z_1 + w_1 x_2 + w_2 x_1) i + \\ & (x_2 z_1 - x_1 z_2 + w_1 y_2 + w_2 y_1) j + \\ & (x_1 y_2 - x_2 y_1 + w_1 z_2 + w_2 z_1) k) \end{aligned}$$

Before we explore the properties of multiplication, we offer a few more definitions to help the discussion.

**Definition 16** The **conjugate**  $q^*$  of a quaternion  $q$  is created by negating the vector part:

$$q^* = (w - xi - yj - zk)$$

The quaternion conjugate has similar properties to the complex conjugate:

$$(q^*)^* = q \tag{D.1}$$

$$(pq)^* = q^* p^* \tag{D.2}$$

$$(p + q)^* = p^* + q^* \tag{D.3}$$

$$qq^* = w^2 + x^2 + y^2 + z^2 \tag{D.4}$$

These properties can be proven by some algebra on the definition of multiplication and conjugate. Notice that multiplying a quaternion by its conjugate results in a real number, just as with complex numbers. Analogously, we can define the absolute value, or modulus, of a quaternion.

**Definition 17** The **modulus** (also called *absolute value* or *magnitude*) of a quaternion is

$$|q| = \sqrt{w^2 + x^2 + y^2 + z^2} = \sqrt{qq^*}$$

Several properties of the quaternion subsets described above can be found directly using these properties.

**Theorem 3** Let  $s = (s)$  be a pure scalar quaternion. Let  $q = (w + xi + yj + zk)$  be an arbitrary quaternion. Then the product commutes:

$$sq = qs = (sw + sxi + syj + szk) .$$

Multiplication of a quaternion by a scalar is commutative and involves scaling each component by the scalar. The proof is trivial from the definition of multiplication. It also follows that

$$|sq| = s|q|$$

where  $s$  is a scalar and  $q$  is a quaternion. These properties can simplify calculations. We can also show that

$$|q^*| = |q| \quad (\text{D.5})$$

$$|pq| = |p| |q| \quad (\text{D.6})$$

through application of the definition of modulus and multiplication.

**Theorem 4** *Quaternion multiplication over  $\mathbb{H}$  forms a non-commutative group.*

PROOF Again, we need to show the group properties for quaternion multiplication:

1. *Closure*: If  $p, q \in \mathbb{H}$  then  $pq \in \mathbb{H}$
2. *Associativity*:  $(pq)r = p(qr) \quad \forall p, q, r, \in \mathbb{H}$ .
3. *Identity* There exists an element  $1 \in \mathbb{H}$  such that  $1p = p1 = p \quad \forall p \in \mathbb{H}$ .
4. *Inverse*: For any  $p \in \mathbb{H}$ , there exists an element  $p^{-1}$  such that  $pp^{-1} = p^{-1}p = 1$ .

Closure follows immediately from the definition of multiplication. Associativity involves lengthy algebraic manipulation of the equation gotten by substituting the definition of multiplication into the definition of associativity. We omit it here.

The identity quaternion for multiplication, 1, is obviously the pure real quaternion (1) since

$$(1)(w + xi + yj + zk) = (w + xi + yj + zk)$$

when the definition of scalar multiplication is applied.

The inverse is a bit harder. Let  $q$  be an arbitrary quaternion in  $\mathbb{H}$ . We seek a quaternion  $q^{-1}$  such that  $qq^{-1} = q^{-1}q = 1$ . Assume  $q^{-1}$  is defined as

$$\frac{1}{w + xi + yj + zk}$$

where 1 is the identity quaternion. We can then multiply the numerator and denominator by the complex conjugate of the denominator to make the denominator real:

$$\begin{aligned} \frac{1}{q} \frac{q^*}{q^*} &= \frac{1}{w + xi + yj + zk} \left( \frac{w - xi - yj - zk}{w - xi - yj - zk} \right) \\ &= \frac{w - xi - yj - zk}{w^2 + x^2 + y^2 + z^2} \\ &= \frac{q^*}{|q|^2} \\ &= \frac{1}{|q|^2} q^* \end{aligned}$$

First, we check left-multiplication:

$$\begin{aligned}
\frac{1}{|q|^2} q^* q &= \frac{1}{|q|^2} (w - xi - yj - zk)(w + xi + yj + zk) \\
&= \frac{1}{|q|^2} ((w^2 + x^2 + y^2 + z^2) + \\
&\quad (-yz + yz + wx - wx)i + \\
&\quad (-xz + xz + wy - wy)j + \\
&\quad (-xy + xy + wz - wz)k) \\
&= \frac{1}{|q|^2} (|q|^2) \\
&= 1.
\end{aligned}$$

We get the identity element, 1, as we chose to prove. A similar calculation or symmetry argument can confirm that it also holds true for right multiplication by the inverse. ♣

A useful property of the inverse follows:

$$(pq)^{-1} = q^{-1}p^{-1}$$

Finally, the quaternion product and addition operators with  $\mathbb{H}$  constitute a **ring**.

**Definition 18** A **ring** is a set  $\mathcal{R}$  over which there are defined two binary operations,  $+$  and  $\times$ , which satisfy the following properties:

1.  $\{\mathcal{R}, +\}$  is a commutative group.
2.  $\{\mathcal{R}, \times\}$  has closure, associativity and identity properties.
3. Distributive (bilinear): For all  $a, b, c \in \mathcal{R}$ ,

$$a \times (b + c) = (a \times b) + (a \times c) \quad (\text{D.7})$$

$$(a + b) \times c = (a \times c) + (b \times c) \quad (\text{D.8})$$

$$(\text{D.9})$$

Taking  $(\times)$  to be the quaternion product (we will usually suppress the  $\times$  symbol and denote  $a \times b = ab$  as above), and  $(+)$  as the quaternion summation, we can show that  $\{\mathbb{H}, +, \times\}$  is a ring.

**Theorem 5**  $\{\mathbb{H}, +, \times\}$  forms a ring.

**PROOF** We have already proven that  $\{\mathbb{H}, +\}$  is a commutative group and that  $\{\mathbb{H}, \times\}$  is a group, which satisfy the first two ring properties. We must show that it is distributive as well. Again, this involves substituting the operator definitions into the two properties and showing they are true. We omit the details here, but intuitively we know that

component-wise the elements are reals and that reals with real multiplication and addition are distributive. By applying the real distributive rules to each component and noticing that the products stay on the left or right sides (since multiplication is non-cummutative), the quaternion distributive property holds. ♣

### D.3.2 Vector Space Interpretation of Quaternions

Some readers may have noticed the similarity of the definition of a quaternion to a vector in  $\mathbb{R}^4$  with the component directions being the real axis and the three imaginary axes. A quaternion can be represented as such a 4-vector for convenience (as we shall do later), but we still use the symbol  $\mathbb{H}$  rather than  $\mathbb{R}^4$  to represent the set of quaternions to avoid confusion. Another useful representation of a quaternion is as a sum of a real number and a vector in  $\mathbb{R}^3$ :

$$q = (w + \mathbf{v}) \stackrel{\text{def}}{=} (w, \mathbf{v})$$

where  $\mathbf{v}$  is the column vector of reals

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

interpreted in the basis defined by the imaginary components  $i, j, k$ . In other words, we could also define the vector part as a pure quaternion as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}^T \begin{bmatrix} i \\ j \\ k \end{bmatrix} = xi + yj + zk$$

where  $i, j, k$  are the imaginary numbers described above the transpose gives the inner product of the components with the imaginary basis. We will alternate between the sum and pair notation for a quaternion as needed.

Given this description, we can consider any real number  $w \in \mathbb{R}$  to be interpreted as a quaternion  $(w, \mathbf{0})$  as well as a real scalar. The distinction will be obvious in context. Also, a vector  $\mathbf{v} \in \mathbb{R}^3$  will be simultaneously interpreted as a true vector and also as the pure quaternion  $(0, \mathbf{v})$ , depending on context. We can reinterpret some of the previous definitions and theorems using this new notation. We simply present them without proof for the interested reader.

**Definition 19** Given two quaternions  $q_1 = (w_1, \mathbf{v}_1)$ ,  $q_2 = (w_2, \mathbf{v}_2)$ , the product

$$pq = (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, \mathbf{v}_1 \times \mathbf{v}_2 + w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1)$$

where  $(\cdot)$  represents the dot product and  $(\times)$  the cross product of vectors in  $\mathbb{R}^3$ .

Consider the pure quaternions  $\mathbf{v}_1$  and  $\mathbf{v}_2$ . The product is

$$\mathbf{v}_1 \mathbf{v}_2 = (-\mathbf{v}_1 \cdot \mathbf{v}_2, \mathbf{v}_1 \times \mathbf{v}_2)$$



So pure vectors multiply with a simultaneous cross product and a dot product term. The definitions of these products in vector analysis came out of Hamilton's quaternion multiplication originally. Given this definition, we can easily find the set of commutative subgroups of  $\mathbb{H}$ .

**Theorem 6** *Given two quaternions  $q_1, q_2 \in \mathbb{H}$ ,  $q_1 q_2 = q_2 q_1 \Leftrightarrow \mathbf{v}_1 = \mathbf{v}_2$ .*

PROOF Consider the vector representation of the quaternions  $q_1 = (w_1, \mathbf{v}_1)$ ,  $q_2 = (w_2, \mathbf{v}_2)$ . Using the multiplication formula we need to prove

$$\begin{aligned} q_1 q_2 &= q_2 q_1 = (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, \mathbf{v}_1 \times \mathbf{v}_2 + w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1) \\ &= (w_2 w_1 - \mathbf{v}_2 \cdot \mathbf{v}_1, \mathbf{v}_2 \times \mathbf{v}_1 + w_2 \mathbf{v}_1 + w_1 \mathbf{v}_2) \end{aligned}$$

Since scalar multiplication and the dot product are commutative, the constraint is

$$\mathbf{v}_1 \times \mathbf{v}_2 = \mathbf{v}_2 \times \mathbf{v}_1$$

but this is only true for  $\mathbf{v}_1 = \mathbf{v}_2$ , when the cross product is zero. ♣

Commutative subgroups of  $\mathbb{H}$  will be useful later in the discussion.

Since  $q \in \mathbb{H}$  can be represented as a vector  $\mathbf{q} \in \mathbb{R}^4$ , it inherits the dot product operator. Specifically,

**Definition 20**

$$\text{For } q_1, q_2 \in \mathbb{H}, \quad q_1 \cdot q_2 = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2$$

For  $p = q$ , we get

$$q q^* = q^* q = |q|^2$$

as with the familiar dot product.

We also extend the functions which return the real and imaginary part of a complex number to quaternions.

**Definition 21** *The **real part** (or scalar part) of  $q = (w + xi + yj + zk)$  can be calculated with*

$$\Re(q) = \frac{q + q^*}{2} = w.$$

*The **imaginary part** can be extracted with*

$$\Im(q) = q - \Re(q) = \frac{q - q^*}{2}.$$

It can also be shown that

$$\Re(p^* q) = \Re(q^* p) = p \cdot q.$$

Another useful identity for pure quaternions can be used to extract the cross product:

$$\mathbf{x} \times \mathbf{y} = \frac{\mathbf{xy} - \mathbf{yx}}{2}$$

### Unit Quaternions and Polar Form

A very important (as we shall see later) subset of  $\mathbb{H}$  is the set of *unit magnitude* quaternions, which we shall denote as  $\hat{\mathbb{H}}$  to distinguish it from the arbitrary magnitude quaternions  $\mathbb{H}$ . Formally,

$$\hat{\mathbb{H}} \stackrel{\text{def}}{=} \{q \in \mathbb{H} : |q| = 1\}.$$

Clearly  $\hat{\mathbb{H}}$  is not a subgroup of  $\mathbb{H}$  with respect to addition. It can be shown that it *is* a subgroup with respect to quaternion multiplication, however.

**Theorem 7** *The subset  $\hat{\mathbb{H}} \subset \mathbb{H}$  is a subgroup of  $\mathbb{H}$  with respect to quaternion multiplication.*

PROOF The identity element is inherited from  $\mathbb{H}$  directly. Associativity also follows directly. We need to prove closure and that the inverse exists in  $\hat{\mathbb{H}}$ . Closure follows from the property that  $|pq| = |p| |q|$ . Assume the inverse is the same as the inverse inherited from  $\mathbb{H}$ ,  $\frac{1}{|q|^2} q^*$ . Since  $|q|^2 = 1$ , the inverse of  $q$  is simply its conjugate  $q^*$ . Conjugation clearly does not change the magnitude of a quaternion, so if  $|q| = 1$  then  $|q^*| = 1$ . ♣

If we interpret unit quaternions in vector notation, we can extend some useful theorems of the complex numbers. First, consider the pure unit quaternion  $\hat{\mathbf{n}}$ , where the hat denotes a unit vector. We choose  $\mathbf{n}$  rather than  $\mathbf{v}$  to reinforce this distinction. It is easy to show that

$$\hat{\mathbf{n}}^2 = -1$$

by using the definition of multiplication. This result implies that the pure imaginary unit vector  $\hat{\mathbf{n}}$  has a *correspondence* with the pure imaginary complex number  $i$ , since  $i^2 = -1$  as well. Using this result, we can extend Euler's theorem and DeMoivre's theorem of powers of complex numbers to quaternions.

### Theorem 8

$$e^{\theta \hat{\mathbf{n}}} = (\cos \theta, \hat{\mathbf{n}} \sin \theta).$$

PROOF Expand the exponential in a power series:

$$e^{\theta \hat{\mathbf{n}}} = 1 + \theta \hat{\mathbf{n}} + \frac{(\theta \hat{\mathbf{n}})^2}{2!} + \frac{(\theta \hat{\mathbf{n}})^3}{3!} + \frac{(\theta \hat{\mathbf{n}})^4}{4!} + \frac{(\theta \hat{\mathbf{n}})^5}{5!} + \dots$$

Notice that we can reduce the powers of  $\hat{\mathbf{n}}$  using the replacement  $\hat{\mathbf{n}}^2 = -1$  in order to get

$$e^{\theta \hat{\mathbf{n}}} = 1 + \theta \hat{\mathbf{n}} - \frac{\theta^2}{2!} - \frac{\theta^3 \hat{\mathbf{n}}}{3!} + \frac{\theta^4}{4!} + \frac{\theta^5 \hat{\mathbf{n}}}{5!} + \dots$$

Grouping real terms together and imaginary terms together produces the suggestive

$$e^{\theta \hat{\mathbf{n}}} = \theta \hat{\mathbf{n}} - \frac{\theta^3}{3!} \hat{\mathbf{n}} + \frac{\theta^5}{5!} \hat{\mathbf{n}} + \dots +$$

$$1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} + \dots$$

The even powers form the power series of  $\cos(\theta)$ . Factoring out the  $\hat{\mathbf{n}}$  from the odd powers leaves the power series of  $\sin(\theta)$ . In other words,

$$e^{\theta \hat{\mathbf{n}}} = \cos(\theta) + \sin(\theta) \hat{\mathbf{n}} = (\cos \theta, \hat{\mathbf{n}} \sin \theta)$$

as we chose to prove. A simple check shows that

$$|e^{\theta \hat{\mathbf{n}}}| = 1.$$



It follows that any unit quaternion  $q \in \hat{\mathbb{H}}$  can be written in the form  $(\cos \theta, \hat{\mathbf{n}} \sin \theta)$  for some  $\theta$  and some  $\hat{\mathbf{n}}$ . Additionally, as with complex numbers, this theorem implies that an arbitrary quaternion  $q \in \mathbb{H}$  can be represented in polar form

$$q = |q| e^{\theta \hat{\mathbf{n}}}$$

by pulling out the magnitude and representing the unit quaternion in exponential form.

One must be very careful about applying the familiar rules of products of exponentials. These rules apply *only* to commutative subgroups of  $\mathbb{H}$ , which were defined above. For example, consider unit quaternions  $p, q \in \hat{\mathbb{H}}$ . Then we have

$$pq = e^{\theta_1 \hat{\mathbf{n}}_1} e^{\theta_2 \hat{\mathbf{n}}_2}$$

It is very tempting to collect the exponents to

$$e^{\theta_1 \hat{\mathbf{n}}_1} e^{\theta_2 \hat{\mathbf{n}}_2} = e^{\theta_1 \hat{\mathbf{n}}_1 + \theta_2 \hat{\mathbf{n}}_2} \quad \textbf{Wrong!}$$

but this is not true. Since it follows that

$$e^{\theta_1 \hat{\mathbf{n}}_1 + \theta_2 \hat{\mathbf{n}}_2} = e^{\theta_2 \hat{\mathbf{n}}_2 + \theta_1 \hat{\mathbf{n}}_1}$$

by commutivity of addition, which then leads to

$$e^{\theta_2 \hat{\mathbf{n}}_2 + \theta_1 \hat{\mathbf{n}}_1} = e^{\theta_2 \hat{\mathbf{n}}_2} e^{\theta_1 \hat{\mathbf{n}}_1} = qp$$

which is a contradiction since in general  $pq \neq qp$  for arbitrary quaternions.

The polar form is useful for many calculations, as we will demonstrate. We can also define the inverse of exponentiation, the natural logarithm of a unit quaternion.

**Definition 22** *The natural logarithm of a unit quaternion is defined as*

$$\ln q = \ln(\cos\theta, \hat{\mathbf{n}} \sin\theta) = \theta \hat{\mathbf{n}}$$

As with exponentials, familiar rules for reducing sums of logarithms can only be used if the involved quaternions commute. In general,  $\ln p + \ln q \neq \ln pq$ .

Notice that exponentiation only works on pure quaternions. Conversely,  $\ln q$  always produces a pure quaternion. These functions lead to several useful identities:

$$e^{\ln q} = q \quad \forall |q| = 1 \tag{D.10}$$

$$e^0 = 1 \tag{D.11}$$

$$(|q|e^{\theta \hat{\mathbf{n}}})^* = |q|e^{-\theta \hat{\mathbf{n}}} \tag{D.12}$$

$$(|q|e^{\theta \hat{\mathbf{n}}})^{-1} = \frac{1}{|q|}e^{-\theta \hat{\mathbf{n}}} \quad \forall q \in \mathbb{H}, q \neq 0. \tag{D.13}$$

$$\tag{D.14}$$

In addition we can now raise unit quaternions to arbitrary real powers  $t \in \mathbb{R}$  by

$$q^t = e^{\ln q^t} = e^{t \ln q}.$$

We can use the power rule for logarithms since  $t$  is a real scalar, which commutes with any quaternion. We can also raise an arbitrary quaternion to a power with

$$q^t = |q|^t e^{t \ln(q/|q|)}$$

which follows immediately.

For the remainder of the discussion, we will mostly concern ourselves with the subgroup of unit quaternions,  $\hat{\mathbb{H}}$ . The reason for this will become clear in the next section as we show that a unit quaternion represents a rotation of  $\mathbb{R}^3$  in the same way that a unit complex number represents a rotation of the plane  $\mathbb{R}^2$ , a quite beautiful result.

## Rotations in $\mathbb{R}^3$

The previous description of the properties of quaternions can be used to discover a useful property of quaternions — they can represent a rotation in  $\mathbb{R}^3$ . This result is beautiful since the complex numbers can be used to represent rotations in the plane ( $\text{so}(2)$ ). We show that using quaternions to parameterize rotations leads to some useful properties and avoids the problems of an Euler angle representation, such as singularities and lack of rotational invariance.

Euler proved that any arbitrary rotation (or composition of rotations) in  $\mathbb{R}^3$  can be written as a single rotation by some angle  $\theta$  around some axis  $\hat{\mathbf{n}}$ . This parameterization is called *axis-angle* notation. We now show that a quaternion  $q = |q|e^{\theta \hat{\mathbf{n}}}$  can be used to rotate a pure vector  $\mathbf{x}$  by  $2\theta$  degrees around the axis  $\hat{\mathbf{n}}$ .

**Theorem 9** *Let  $q \in \mathbb{H}$  and  $\mathbf{x}$  be a pure quaternion (zero scalar component). The transformation  $T_q(\mathbf{x}) = q\mathbf{x}q^{-1}$  rotates  $\mathbf{x}$  around axis  $\hat{\mathbf{n}}$  by  $2\theta$ .*

**PROOF** Clearly for the transformation to be a rotation it must preserve the magnitude of the vector, which it does.

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

We also need to show that the transformation does not affect the scalar component, which must remain 0 for a pure vector. Recall

$$\text{For all } p \in \mathbb{H}, 2\Re(p) = p + p^*.$$

First, we will assume that  $|q| = 1$  so that we can replace the inverse with the conjugate. We will show that this is a reasonable thing later. For now, we motivate by considering the polar form of  $q = |q|e^{\theta\hat{\mathbf{n}}}$ . Since scalar multiplication commutes, we see that the magnitude of  $q = |q|$  and the magnitude of  $q^{-1} = \frac{1}{|q|}$  can be pulled out, cancelling each other and leaving unit quaternions  $\hat{q}$  and  $\hat{q}^{-1} = \hat{q}^*$ .

The effect of  $T_q$  on the scalar part of  $p$  (assuming  $|q| = 1$  as motivated above) is

$$\begin{aligned} 2\Re(T_q(p)) &= 2\Re(qpq^*) \\ &= qpq + (qpq^*)^* \\ &= qpq + qp^*q^* \\ &= q(p + p^*)q^* \quad (\text{by bilinearity}) \\ &= q(2\Re(p))q^* \\ &= 2\Re(p)qq^* \quad (\text{scalar multiplication commutes}) \\ &= 2\Re(p) \end{aligned}$$

So  $\Re(p)$  is invariant with respect to  $T_q$ .

Since  $T_q$  preserves magnitude and the scalar component, it must describe a rotation (notice that it cannot be a reflection since this would affect the scalar part of an arbitrary quaternion).

Since it is a rotation, it must have at least one fixed point  $\mathbf{f} \in \mathbb{R}^3$  such that

$$q\mathbf{f}q^{-1} = \mathbf{f}$$

Using polar form,

$$|q|e^{\theta\hat{\mathbf{n}}}\mathbf{f}\frac{1}{|q|}e^{-\theta\hat{\mathbf{n}}} = \mathbf{f}$$

The magnitudes of  $q$  and  $q^{-1}$  cancel for any  $q \in \mathbb{H}$ . Therefore, we can consider only unit quaternions without loss of generality. This simplification lets us replace  $q^{-1}$  with  $q^*$ . Expanding  $\mathbf{f}$  into polar form gives

$$e^{\theta \hat{\mathbf{n}}} |\mathbf{f}| e^{\omega \hat{\mathbf{v}}} e^{-\theta \hat{\mathbf{n}}} = |\mathbf{f}| e^{\omega \hat{\mathbf{v}}}$$

Cancelling the magnitudes again

$$e^{\theta \hat{\mathbf{n}}} e^{\omega \hat{\mathbf{v}}} e^{-\theta \hat{\mathbf{n}}} = e^{\omega \hat{\mathbf{v}}}$$

Premultiplying by  $e^{-\theta \hat{\mathbf{n}}}$  gives

$$e^{\omega \hat{\mathbf{v}}} e^{-\theta \hat{\mathbf{n}}} = e^{-\theta \hat{\mathbf{n}}} e^{\omega \hat{\mathbf{v}}}$$

since it the inverse. This equation says that to be a fixed point, the two quaternions  $e^{-\theta \hat{\mathbf{n}}}$  and  $e^{\omega \hat{\mathbf{v}}}$  must commute. Clearly  $e^{-\theta \hat{\mathbf{n}}}$  commutes with  $e^{\theta \hat{\mathbf{n}}}$ . This property only holds if  $\hat{\mathbf{n}} = \hat{\mathbf{v}}$ , as we saw above. Commutivity allows us to collect exponents which leaves

$$e^{\theta \hat{\mathbf{n}} + \omega \hat{\mathbf{n}} - \theta \hat{\mathbf{n}}} = e^{\omega \hat{\mathbf{n}}} = e^{\omega \hat{\mathbf{v}}}$$

So the set of fixed points under this transformation is the set of vectors along the axis  $\hat{\mathbf{n}}$ , as is expected of a rotation. We have shown that the transformation preserves length and leaves points along  $\hat{\mathbf{n}}$  fixed, which is sufficient to prove that the transformation is a rotation around axis  $\hat{\mathbf{n}}$ . We now need to show that the rotation is by  $2\theta$ .

Since points in the  $\hat{\mathbf{n}}$  direction are fixed, we can remove the component from  $\mathbf{x}$  in the  $\hat{\mathbf{n}}$  direction since it passes through unchanged. In other words, write  $\mathbf{x}$  as

$$\mathbf{x} = (\mathbf{x} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} + (\mathbf{x} - (\mathbf{x} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}})$$

which breaks it into components parallel to  $\hat{\mathbf{n}}$  and perpendicular to  $\hat{\mathbf{n}}$ . Making this explicit,

$$\mathbf{x} = \mathbf{x}_{\parallel} + \mathbf{x}_{\perp}$$

By the bilinearity of quaternions,

$$\begin{aligned} T_q(\mathbf{x}) &= q \mathbf{x} q^* \\ &= q(\mathbf{x}_{\parallel} + \mathbf{x}_{\perp}) q^* \\ &= q \mathbf{x}_{\parallel} q^* + q \mathbf{x}_{\perp} q^* \end{aligned}$$

Since  $T_q$  leaves points parallel to  $\hat{\mathbf{n}}$  fixed

$$q \mathbf{x}_{\parallel} q^* + q \mathbf{x}_{\perp} q^* = \mathbf{x}_{\parallel} + q \mathbf{x}_{\perp} q^* = \mathbf{x}_{\parallel} + T_q(\mathbf{x}_{\perp})$$

So we need only look at the action of  $T_q$  on the components in the plane perpendicular to  $\hat{\mathbf{n}}$ , or  $\mathbf{x}_{\perp}$ . We will do this in halves. First, we look at the action of left-multiplication by  $q$ , then we look at the action of right-multiplication by  $q^*$ . Since  $|q| = 1$ , we use the vector form  $q = (\cos \theta, \hat{\mathbf{n}} \sin \theta)$  Then we have

$$\begin{aligned} q \mathbf{x}_{\perp} &= (\cos \theta, \hat{\mathbf{n}} \sin \theta) \mathbf{x}_{\perp} \\ &= (-\sin \theta \hat{\mathbf{n}} \cdot \mathbf{x}_{\perp}, \sin \theta \hat{\mathbf{n}} \times \mathbf{x}_{\perp} + \cos \theta \mathbf{x}_{\perp}) \end{aligned}$$

by applying the vector form of the quaternion product. But since  $\hat{\mathbf{n}} \cdot \mathbf{x}_\perp = 0$ , this reduces to

$$q\mathbf{x}_\perp = (0, \cos \theta \mathbf{x}_\perp + \sin \theta \hat{\mathbf{n}} \times \mathbf{x}_\perp)$$

Notice that  $\hat{\mathbf{n}} \times \mathbf{x}_\perp$  is orthogonal to both  $\hat{\mathbf{n}}$  and  $\mathbf{x}_\perp$ . Therefore it also lies in the plane perpendicular to  $\hat{\mathbf{n}}$ . Since it is also orthogonal to  $\mathbf{x}_\perp$ , we can write any vector in the perpendicular plane in terms of the orthogonal basis defined by  $\mathbf{x}_\perp$  and  $\hat{\mathbf{n}} \times \mathbf{x}_\perp$ . We make this explicit by introducing the unit vectors

$$\hat{\mathbf{u}} = \frac{\mathbf{x}_\perp}{\|\mathbf{x}_\perp\|}$$

and

$$\hat{\mathbf{v}} = \frac{\hat{\mathbf{n}} \times \mathbf{x}_\perp}{\|\hat{\mathbf{n}} \times \mathbf{x}_\perp\|}.$$

Clearly in this coordinate system  $\mathbf{x}_\perp = \|\mathbf{x}_\perp\| \hat{\mathbf{u}}$ . This change of variables results in

$$\begin{aligned} q\mathbf{x}_\perp &= q\|\mathbf{x}_\perp\| \hat{\mathbf{u}} \\ &= \|\mathbf{x}_\perp\| q\hat{\mathbf{u}} \\ &= \|\mathbf{x}_\perp\| (\cos \theta \hat{\mathbf{u}} + \sin \theta \hat{\mathbf{v}}) \end{aligned}$$

which should be familiar as a rotation of a point in the plane defined by  $\hat{\mathbf{u}}$  and  $\hat{\mathbf{v}}$  clockwise by  $\theta$ . We define this action as the linear transformation  $R(\theta, \hat{\mathbf{n}})$  which rotates vectors in the plane perpendicular to  $\hat{\mathbf{n}}$  by  $\theta$ . Left-multiplication of a vector in the plane perpendicular to the axis by a unit quaternion  $q$  therefore results in that vector rotating by  $\theta$  around the plane. Explicitly,

$$q\mathbf{x}_\perp = R(\theta, \hat{\mathbf{n}})\mathbf{x}_\perp.$$

Now we find the action of right-multiplication by  $q^*$ . Similarly to left multiplication, we can find that

$$\begin{aligned} \mathbf{x}_\perp q^* &= (0, \mathbf{x}_\perp \times (-\sin \theta \hat{\mathbf{n}}) + \cos \theta \mathbf{x}_\perp) \\ &= (0, \cos \theta \mathbf{x}_\perp - \sin \theta \mathbf{x}_\perp \times \hat{\mathbf{n}}) \\ &= (0, \cos \theta \mathbf{x}_\perp - \sin \theta (-\hat{\mathbf{n}} \times \mathbf{x}_\perp)) \\ &= (0, \cos \theta \mathbf{x}_\perp + \sin \theta \hat{\mathbf{n}} \times \mathbf{x}_\perp) \end{aligned}$$

which is the same as left-multiplying by  $q$ ! So we get

$$\begin{aligned} \mathbf{x}_\perp q^* &= \|\mathbf{x}_\perp\| (0, \cos \theta \hat{\mathbf{u}} + \sin \theta \hat{\mathbf{v}}) \\ &= R(\theta, \hat{\mathbf{n}})\mathbf{x}_\perp \\ &= q\mathbf{x}_\perp \end{aligned}$$

We now write

$$R(\theta, \hat{\mathbf{n}})\mathbf{x}_\perp = \mathbf{y}_\perp = \mathbf{x}_\perp q^*$$

which leads to

$$q^* = \mathbf{x}_\perp^{-1} \mathbf{y}_\perp$$

Now we can substitute this into form of  $T_q$  above

$$\begin{aligned} T_q(\mathbf{x}) &= \mathbf{x}_\parallel + q\mathbf{x}_\perp q^* \\ &= \mathbf{x}_\parallel + q\mathbf{x}_\perp \mathbf{x}_\perp^{-1} \mathbf{y}_\perp \\ &= \mathbf{x}_\parallel + q\mathbf{y}_\perp \\ &= \mathbf{x}_\parallel + R(\theta, \hat{\mathbf{n}})\mathbf{y}_\perp \\ &= \mathbf{x}_\parallel + R(\theta, \hat{\mathbf{n}})(R(\theta, \hat{\mathbf{n}})\mathbf{x}_\perp) \end{aligned}$$

So we see that  $T_q$  rotates the perpendicular part by  $\theta$  around  $\hat{\mathbf{n}}$ , then rotates the newly rotated vector around  $\hat{\mathbf{n}}$  by  $\theta$  again. Since rotations in the same plane commute, we have

$$R(\theta_1, \hat{\mathbf{n}}) \circ R(\theta_2, \hat{\mathbf{n}}) = R(\theta_1 + \theta_2, \hat{\mathbf{n}})$$

where  $\circ$  denotes composition of functions. It follows that

$$T_q(\mathbf{x}) = \mathbf{x}_\parallel + R(2\theta, \hat{\mathbf{n}})\mathbf{x}_\perp$$

which is clearly a rotation of  $\mathbf{x}$  around axis  $\hat{\mathbf{n}}$  by  $2\theta$ , as we were to prove. ♣

The interested reader can also consider the inverse of  $T_q(\mathbf{x}) = q^* \mathbf{x} q$ . It can be shown that left multiplication by  $q^*$  results in a rotation by  $\theta$  and then reflection through  $\hat{\mathbf{v}}$ . The triple product, since it reflects twice, is again a rotation by  $2\theta$ .

Finally, we can show that  $T_q = T_{-q}$ . In other words,  $q$  and  $-q$  represent the *same* rotation in  $\mathbb{R}^3$ .  $-q$  represents the action  $R(-\theta, -\hat{\mathbf{n}})$  which is clearly the same as  $R(\theta, \hat{\mathbf{n}})$ . Thus, the unit quaternion group  $\hat{\mathbb{H}}$  has a 2-1 correspondence with the rotation group in  $\mathbb{R}^3$ . Explicitly,

$$R(\theta, \hat{\mathbf{n}}) = \left( \cos \frac{\theta}{2}, \hat{\mathbf{n}} \sin \frac{\theta}{2} \right)$$

where  $R(\theta, \hat{\mathbf{n}})$  is the rotation transformation in  $\mathbb{R}^3$  by  $\theta$  around  $\hat{\mathbf{n}}$ . To make this explicit, we will usually write a unit quaternion in terms of the half-angle as

$$q = \left( \cos \frac{\theta}{2}, \hat{\mathbf{n}} \sin \frac{\theta}{2} \right).$$



## Unit Quaternion Group Manifold

The multiplicative subgroup of unit quaternions  $\hat{\mathbb{H}}$  lies on the surface of the three-dimensional hypersphere in 4-space:  $S^3 \subset \mathbb{R}^4$ . This geometry can be easily verified by considering the components of a quaternion  $q$  as a point (vector)  $\mathbf{q} \in \mathbb{R}^4$ . The constraint  $|\mathbf{q}| = 1$  keeps the vector on the surface of the unit sphere. As we mentioned above,  $-q$  and  $q$  both represent the same rotation. So antipodal points on  $S^3$  refer to the same rotation. This double mapping can be tricky when we use quaternions to represent rotations, and we must constantly be careful when defining metrics and functions.

Using the hyperspherical description of a quaternion, we can easily find a quaternion which will rotate one pure quaternion (vector in  $\mathbb{R}^3$ ) into another pure quaternion, which results in a change of coordinate system. Consider two pure quaternions  $\mathbf{x}, \mathbf{y}$  represented as vectors in  $\mathbb{R}^4$ . From linear algebra, we know that the dot product of two unit vectors is the angle  $\alpha$  between the two vectors:

$$\mathbf{x} \cdot \mathbf{y} = |\mathbf{x}||\mathbf{y}| \cos \alpha$$

We are looking for a unit quaternion  $r = (\cos \theta, \hat{\mathbf{n}} \sin \theta)$  such that

$$r\mathbf{x}r^* = \mathbf{y}$$

From above, we know that unit quaternion will rotate a vector by  $2\theta$  around the axis  $\hat{\mathbf{n}}$ . From the geometry, we see that we desire a rotation of  $\alpha$  around the axis formed by the perpendicular to both vectors (in other words, the two vectors form a plane of rotation). This axis is perpendicular to both  $\mathbf{x}$  and  $\mathbf{y}$ , so clearly must be  $\mathbf{x} \times \mathbf{y}$ . It follows that the desired quaternion  $r$  is

$$r = \left( \cos \frac{\alpha}{2}, (\mathbf{x} \times \mathbf{y}) \sin \frac{\alpha}{2} \right)$$

Notice that the axis vanishes as  $\mathbf{x} \rightarrow \mathbf{y}$ . The limit clearly exists, however, since we know that the identity quaternion  $1$  would leave the vector fixed.

Since they lie on  $S^3$ , we can talk about quaternion curves, or paths, that lie on the hypersphere. Representing quaternions geometrically as points on a hypersphere allows us to use the results of spherical geometry to simplify proofs and to visualize results, as we did above.

### D.3.3 Quaternion Curves

We now consider the set of curves on the hypersphere,  $q(t) \in S^3$ . In vector form a curve can be expressed as

$$q(t) = \left( \cos \frac{\theta}{2}(t), \hat{\mathbf{n}}(t) \sin \frac{\theta}{2}(t) \right)$$

or exponentially as

$$q(t) = e^{\hat{\mathbf{n}}(t) \frac{\theta}{2}(t)}.$$

First, we look at the set of curves with a fixed axis,  $\hat{\mathbf{n}}(t) = \hat{\mathbf{n}}$ . We get

$$q(t) = (\cos \frac{\theta}{2}(t), \hat{\mathbf{n}} \sin \frac{\theta}{2}(t))$$

By the geometry of the manifold, it is clear that the curve lies in the subgroup of  $S^3$  consisting of the great circle with axis  $\hat{\mathbf{n}}$ . Recall that a great circle of a sphere is a circle on the sphere whose embedding plane passes through the center of the sphere. On the Earth, for example, the lines of longitude and the equator are all great circles, but the other lines of latitude are not.

We will soon consider the constant speed curve with fixed axis  $\hat{\mathbf{n}}$ . We have not yet defined the speed of a point on a curve, however, since we have not yet introduced quaternion calculus. Since the quaternions lie on a non-Euclidean manifold, the normal rules of Euclidean calculus do not hold. For the one-parameter subgroup of  $\hat{\mathbb{H}}$  we have described here, however, we *can* use the familiar vector calculus formulae. Specifically, for the single-parameter set of curves with fixed axis  $q_{\hat{\mathbf{n}}}(t)$ , which we denote

$$\begin{aligned} \frac{d}{dt}q_{\hat{\mathbf{n}}}(t) &= \frac{d}{dt}(\cos \frac{\theta}{2}(t), \hat{\mathbf{n}} \sin \frac{\theta}{2}(t)) \\ &= (\frac{\dot{\theta}}{2} - \sin \frac{\theta}{2}, \hat{\mathbf{n}} \frac{\dot{\theta}}{2} \cos \frac{\theta}{2}) \\ &= \frac{\dot{\theta}}{2}(-\sin \frac{\theta}{2}, \hat{\mathbf{n}} \cos \frac{\theta}{2}) \\ &= \frac{\dot{\theta}}{2}\hat{\mathbf{n}}^2(\sin \frac{\theta}{2}, -\hat{\mathbf{n}} \cos \frac{\theta}{2}) \\ &= \frac{\dot{\theta}}{2}\hat{\mathbf{n}}\hat{\mathbf{n}}(\sin \frac{\theta}{2}, -\hat{\mathbf{n}} \cos \frac{\theta}{2}) \\ &= (\frac{\dot{\theta}}{2}\hat{\mathbf{n}})(-(\hat{\mathbf{n}} \cdot (-\hat{\mathbf{n}} \cos \frac{\theta}{2})), \hat{\mathbf{n}} \times (-\hat{\mathbf{n}}) + \sin \frac{\theta}{2}\hat{\mathbf{n}}) \\ &= (\frac{\dot{\theta}}{2}\hat{\mathbf{n}})(\cos \frac{\theta}{2}, \hat{\mathbf{n}} \sin \frac{\theta}{2}) \\ &= \frac{\dot{\theta}}{2}\hat{\mathbf{n}}q_{\hat{\mathbf{n}}} \\ &= \frac{\dot{\theta}}{2}(t)\hat{\mathbf{n}}e^{\frac{\theta}{2}(t)\hat{\mathbf{n}}} \end{aligned}$$

We could also get to this result by using the exponential form and the familiar rules for single parameter exponential derivation:

$$\begin{aligned} \frac{d}{dt}e^{\frac{\theta}{2}(t)\hat{\mathbf{n}}} &= (\frac{d}{dt}\frac{\theta}{2}(t)\hat{\mathbf{n}})e^{\frac{\theta}{2}(t)\hat{\mathbf{n}}} \\ &= \frac{\dot{\theta}}{2}(t)\hat{\mathbf{n}}e^{\frac{\theta}{2}(t)\hat{\mathbf{n}}} \end{aligned}$$

The form of the derivative should be suggestive. We have only derived it for the one-parameter subgroup, but it appears to be of the form

$$\frac{d}{dt}q = \frac{d \ln(q)}{dt}q = \frac{1}{2}\omega(t)q(t)$$

where  $\omega$  is a pure vector function in  $\mathbb{R}^3$  (or equivalently a pure quaternion). We prove this formally later. For now, this is enough to explore some other properties of the quaternions. Finally, notice that

$$\left| \frac{d}{dt}q \right| = \left| \frac{\dot{\theta}}{2} \right|$$

since  $\hat{n}$  and  $q$  are both unit length. This fact implies an important property — the derivative of a unit quaternion is not a unit quaternion. It is a general quaternion in  $\mathbb{H}$ . Such a derivative quaternion  $\dot{q} = \frac{1}{2}\omega q$  can, as we will also prove formally later, be used to represent the derivative at  $q$  as an instantaneous rotation around axis  $\hat{\omega}$  with angular speed  $\frac{1}{2}|\omega|$ . Thus, the vector  $\omega$  describes (locally) an instantaneous rotation around  $\hat{\omega}$  with speed  $|\frac{1}{2}\omega|$ .

Now we can consider the set of fixed axis curves of constant speed. The derivative for a constant speed  $\dot{\theta}$  curve around  $\hat{n}$  is

$$\dot{q} = \frac{\dot{\theta}}{2}\hat{n}q$$

which we will write as

$$\dot{q} = \frac{1}{2}\omega q$$

Assuming we can separate variables normally for this case

$$dq q^{-1} = \frac{1}{2}\omega dt$$

whose solution is

$$\ln q = \frac{t\omega}{2} + c$$

where  $c$  is a constant of integration. Exponentiating, we get

$$q = e^{\frac{t\omega}{2} + c}$$

$c$  is obviously  $\ln q(0)$  since

$$q(0) = e^c$$

and so the constant speed curves are those of the form

$$q(t) = e^{\frac{t\omega}{2} + \ln q(0)}$$

Of course, this is not rigorous, but we can also use the antiderivative of the derivative

we described above since

$$\begin{aligned}\dot{q} &= \frac{1}{2}\omega q \\ &= \ln(q)q\end{aligned}$$

Then the antiderivative must have

$$q = e^{t \ln q}$$

since  $\ln q$  is constant. This gives us

$$q = e^{\frac{t\omega}{2}}$$

which can clearly be augmented for  $q(0)$  by adding the same term as above.

For now, we will just consider the curves which start at the identity, in other words  $q(t)$  such that  $q(0) = \mathbf{1}$ . This reduces the form to

$$q(t) = e^{\frac{t\omega}{2}} = \left( \cos \frac{t\|\omega\|}{2}, \hat{\omega} \sin \frac{t\|\omega\|}{2} \right)$$

which by the geometry of the manifold also describes a constant speed rotation around  $\hat{\mathbf{n}}$ , as we desired. So the one-parameter derivative described above is valid. We use it to look at the tangent space of the quaternion group.

### Tangent Space of $\hat{\mathbb{H}}$

We find it useful to consider the tangent space of the unit quaternion group  $\hat{\mathbb{H}}$ . First, we define the tangent space.

**Definition 23** *The tangent space anchored at the identity  $\mathbf{1}$  in a continuous group  $\mathbb{G}$  is defined as the space of tangent vectors of all curves  $g(t)$  passing through  $\mathbf{1}$  at  $t = 0$  evaluated at  $t = 0$ . In other words,*

$$\text{Tan } \mathbb{G} = \left. \frac{d}{dt}g(t) \right|_{t=0} = \dot{g}(0) \quad \forall g(t) \in \mathbb{G} \text{ such that } g(0) = \mathbf{1}.$$

For the quaternion group (we will now only consider unit quaternions and will only be explicit if we use a non-unit quaternion),  $\hat{\mathbb{H}}$ , we consider the curves  $q(t)$  through the quaternion identity  $\mathbf{1}$  such that  $q(0) = \mathbf{1}$ . As we saw above,

$$\dot{q} = \ln(q(t))q(t) = \frac{\theta}{2}(t)\hat{\mathbf{n}}(t)q(t) .$$

Evaluating this at  $t = 0$ , we get

$$\dot{q}(0) = \frac{\theta}{2}(0)\hat{\mathbf{n}}(0)\mathbf{1} = \frac{\theta}{2}(0)\hat{\mathbf{n}}(0)$$

which clearly describes some arbitrary vector in  $\mathbb{R}^3$ . Therefore, the tangent space (set of tangent vectors) of  $\hat{\mathbb{H}}$  can be considered as  $\ln \hat{\mathbb{H}}$  (we will see this is called the Lie algebra of the  $\hat{\mathbb{H}}$  Lie group).

This result is important.  $\mathbb{R}^3$  is a Euclidean vector space, whereas  $\hat{\mathbb{H}}$  is not. Therefore the (locally invertible) mapping

$$\rho : \hat{\mathbb{H}} \rightarrow \mathbb{R}^3 = \ln q$$

takes our non-Euclidean quaternion  $q$  into a three-dimensional vector space, which is likely more familiar to us. The inverse of this mapping exists locally:

$$\rho^{-1}\left(\frac{\theta}{2}\hat{\mathbf{n}}\right) = e^{\frac{\theta}{2}\hat{\mathbf{n}}}$$

For  $\frac{\theta}{2} < 2\pi$  the inverse is unique (single-valued) and therefore  $\rho$  is one-to-one.

## D.4 SU(2)

Another classical group investigated by physicists and mathematicians is the group of Special Unitary  $2 \times 2$  matrices,  $\text{SU}(2)$  (see Artin [2] for an excellent introduction). Special unitary matrices are the extension of special orthogonal matrices (with real entries) to those matrices with complex entries. The unfamiliar reader can review complex matrices in Appendix A.

An element  $\mathbf{U} \in \text{SU}(2)$  is usually written in terms of two complex entries:

$$\mathbf{U} = \begin{bmatrix} \alpha & \beta \\ -\beta^* & \alpha^* \end{bmatrix}, \quad \alpha, \beta \in \mathbb{C}, \quad |\alpha|^2 + |\beta|^2 = 1.$$

Since an  $\text{SU}(2)$  element has four real components, we can also express it as a vector in  $\mathbb{R}^4$ . Additionally, the magnitude constraint implies that the magnitude of the 4-vector is also unity. Therefore, the group manifold of  $\text{SU}(2)$  is the hypersphere  $S^3 \in \mathbb{R}^4$ . This manifold is the same as the unit quaternion group  $\hat{\mathbb{H}}$ ! This similarity suggests that there exists an isomorphism between  $\hat{\mathbb{H}}$  and  $\text{SU}(2)$ , and therefore a 2-1 mapping of  $\text{SU}(2)$  into  $\text{SO}(3)$ . Indeed, such a mapping exists, as we now prove.

### D.4.1 Isomorphism

Consider the representation of a unit quaternion as a unit vector  $\hat{\mathbf{q}} \in S^3 \subset \mathbb{R}^4$ . Let  $\alpha = q_0 + iq_1$  and  $\beta = q_2 + iq_3$ . Let  $h(\mathbf{q})$  be the bijection from a unit quaternion into a matrix in  $\text{SU}(2)$  by plugging these two complex values into the components of the  $\text{SU}(2)$ . We now show that  $\hat{\mathbb{H}}$  is isomorphic to  $\text{SU}(2)$  (which we write as  $\hat{\mathbb{H}} \approx \text{SU}(2)$ ).

**Theorem 10** *The group of unit quaternions  $\hat{\mathbb{H}}$  is isomorphic to the group of special unitary  $2 \times 2$  matrices  $\text{SU}(2)$ .*

**PROOF** To prove group isomorphism, we must show that the bijection (invertible one-to-one mapping) we defined above respects the multiplication operator in both groups. Formally,

$$h(\mathbf{pq}) = h(\mathbf{p}) h(\mathbf{q}) \quad \forall \mathbf{p}, \mathbf{q} \in \hat{\mathbb{H}}.$$

The proof simply involves multiplying out the matrices on the RHS of the above equation and showing that it is the same as the LHS.

$$\begin{aligned} h(\mathbf{p}) h(\mathbf{q}) &= \begin{bmatrix} \alpha & \beta \\ -\beta^* & \alpha^* \end{bmatrix} \begin{bmatrix} \gamma & \delta \\ -\delta^* & \gamma^* \end{bmatrix} \\ &= \begin{bmatrix} \alpha\gamma - \beta\delta^* & \alpha\delta + \beta\gamma^* \\ -\beta^*\gamma - \alpha^*\gamma^* & -\beta^*\delta + \alpha^*\gamma^* \end{bmatrix} \end{aligned}$$

The matrix is clearly also an element of  $\mathbf{SU}(2)$ . Therefore, we can pull out the two complex degrees of freedom from the matrix (written in a complex vector for convenience):


$$\begin{bmatrix} \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} \alpha\gamma - \beta\delta^* \\ \alpha\delta + \beta\gamma^* \end{bmatrix}.$$

Now we calculate  $\lambda$  and  $\mu$  using Hamilton's multiplication rules. Let  $\mathbf{p} = (p_0, p_1, p_2, p_3)$  and  $\mathbf{q} = (q_0, q_1, q_2, q_3)$ .

$$\begin{aligned} \lambda &= (p_0 + ip_1)(q_0 + iq_1) - (p_2 + ip_3)(q_2 - iq_3) \\ &= (p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3) + i(p_0q_1 + p_1q_0 + p_2q_3 - p_3q_2) \end{aligned}$$

and

$$\begin{aligned} \mu &= (p_0 + ip_1)(q_2 + iq_3) + (p_2 + ip_3)(q_0 - iq_1) \\ &= (p_0q_2 - p_1q_3 + p_2q_0 + p_3q_1) + i(p_0q_3 + p_3q_0 + p_1q_2 - p_2q_1) \end{aligned}$$

The equality follows from the definition of the quaternion product in Definition 15 since  $h(\mathbf{pq})$  clearly produces the  $\mathbf{SU}(2)$  matrix with the two complex degrees of freedom  $\lambda$  and  $\mu$  above. Clearly, the real components of  $\lambda$  and  $\mu$  map directly to the components of the product of  $\mathbf{p}$  and  $\mathbf{q}$ . Therefore,  $\hat{\mathbb{H}} \approx \mathbf{SU}(2)$ . 

Given this isomorphism, we can now convert any  $\mathbf{SU}(2)$  element into a quaternion and then into a rotation, so the group  $\mathbf{SU}(2)$  also maps 2-1 onto the rotations in  $\mathbb{R}^3$ .

## D.4.2 Pauli Spin Matrices

Consider the following set of matrices (we break our normal style convention here of using capital bold for matrices and use lower case bold to agree more closely with the physics literature):

$$\sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

These Hermitian matrices are called the *Pauli spin matrices* and can be used to generate elements of  $\text{SU}(2)$  [71, 60].

First, we notice the Pauli spin matrices form a basis for some the set of traceless (trace zero) Hermitian matrices (see Appendix A for a refresher on Hermitian matrices). Now consider the mapping from a vector  $\mathbf{x} \in \mathbb{R}^3$  into a complex  $2 \times 2$  matrix spanned by the Pauli spin basis:

$$\mathbf{X} = \mathbf{x} \cdot \vec{\sigma} = \sum_{j=1}^3 x_j \sigma_j.$$

In other words, we project a real unit vector in  $\mathbb{R}^3$  into the space defined by the basis of the Pauli spin matrices. We can also invert this transformation easily using the relation

$$x_i = \frac{1}{2} \text{Tr}(\mathbf{X} \sigma_i)$$

where  $x_i$  is the  $i$ th component of  $\mathbf{x}$  and  $\mathbf{X}$  is the matrix form  $\mathbf{x} \cdot \vec{\sigma}$ .

It is useful to note that the Pauli spin matrices square to unity:

$$\sigma_j^2 = \mathbf{I}$$

which is useful since they form an orthonormal basis, and multiply cyclically

$$\sigma_j \sigma_k = \delta_{jk} \mathbf{I} + \epsilon_{jkl} \sigma_l$$

or equivalently

$$\sigma_j \sigma_k + \sigma_k \sigma_j = 2\delta_{jk} . \quad (\text{D.15})$$

Next consider the matrix

$$\mathbf{U}(\theta, \hat{\mathbf{n}}) = \exp\left(-\frac{i\theta}{2} \hat{\mathbf{n}} \cdot \vec{\sigma}\right)$$

This matrix is very similar to the polar form of the quaternions. It can be shown that it is a unitary matrix by carrying out the normal power series expansion of the exponential.

**Theorem 11** *The matrix*

$$\mathbf{U}(\theta, \hat{\mathbf{n}}) = \exp\left(-\frac{i\theta}{2} (\hat{\mathbf{n}} \cdot \vec{\sigma})\right)$$

*can be written as*

$$\mathbf{I} \cos \frac{\theta}{2} - i(\hat{\mathbf{n}} \cdot \vec{\sigma}) \sin \frac{\theta}{2}$$

**PROOF** The proof proceeds by expanding the matrix exponential formally. Let  $\alpha = -\theta/2$  and  $\mathbf{N} = \hat{\mathbf{n}} \cdot \vec{\sigma}$ . Then we have

$$e^{i\alpha \mathbf{N}} = \mathbf{I} + i\alpha \mathbf{N} + \frac{i^2 \alpha^2 \mathbf{N}^2}{2!} + \frac{i^3 \alpha^3 \mathbf{N}^3}{3!} + \frac{i^4 \alpha^4 \mathbf{N}^4}{4!} + \dots .$$

First we note that:

$$\mathbf{N}^2 = \mathbf{I}$$

which follows by writing out  $\mathbf{N}$  as a multinomial with the components of  $\hat{\mathbf{n}}$  as components in terms of the Pauli basis:

$$\mathbf{N}^2 = (n_1\sigma_1 + n_2\sigma_2 + n_3\sigma_3)^2$$

and using the skew-symmetric properties of the algebra as shown in Equation D.15 to remove the zero cross-terms

$$(n_1\sigma_1 + n_2\sigma_2 + n_3\sigma_3)^2 = n_1^2\sigma_1^2 + n_2^2\sigma_2^2 + n_3^2\sigma_3^2$$

since all the  $\sigma_j\sigma_k + \sigma_k\sigma_j$  terms vanish for  $j \neq k$ . Noting that each of the  $\sigma_j^2$  terms is the identity, we get simply:

$$\mathbf{N}^2 = (n_1^2 + n_2^2 + n_3^2)\mathbf{I}$$

which is simply the identity  $\mathbf{I}$  since  $\hat{\mathbf{n}}$  is a unit vector.

After this simplification, collecting the even and odd terms and reducing powers of the imaginary unit  $i$  gives the standard formula for the complex exponential in terms of the Pauli matrices:

$$\mathbf{I} \cos \alpha + i(\hat{\mathbf{n}} \cdot \vec{\sigma}) \sin \alpha$$

which reduces to:

$$\mathbf{I} \cos \frac{\theta}{2} - i(\hat{\mathbf{n}} \cdot \vec{\sigma}) \sin \frac{\theta}{2}$$

by replacing  $\alpha$  with  $-\frac{\theta}{2}$  and using trigonometric rules

$$\cos -\theta = \cos \theta$$

$$\sin -\theta = -\sin \theta$$

as we desired to prove. ♣

This theorem implies that the matrix  $\mathbf{U}(\theta, \hat{\mathbf{n}})$  rotates the space spanned by the Pauli matrices by  $\theta$  around  $\hat{\mathbf{n}} \cdot \vec{\sigma}$ . (Again, the half-angle is implicit in the transformation, as we are about to see).

**Theorem 12** *The transformation*

$$\mathbf{X} \rightarrow \mathbf{X}' = e^{-i(\theta/2)\hat{\mathbf{n}} \cdot \vec{\sigma}} \mathbf{X} e^{i(\theta/2)\hat{\mathbf{n}} \cdot \vec{\sigma}}$$

*rotates the vector  $\mathbf{x}$  represented as  $\mathbf{X} = \mathbf{x} \cdot \vec{\sigma}$  by  $\theta$  around  $\hat{\mathbf{n}}$ . In other words,*



$$\mathbf{X}' = \mathbf{x}' \cdot \vec{\sigma} = (\mathbf{R}(\theta, \hat{\mathbf{n}})\mathbf{x}) \cdot \vec{\sigma}$$

We can extract the real components from the resulting matrix with the inverse of the change of basis as described above. We will not prove this theorem, but note that the proof is very similar to the other rotation proofs due to the trigonometric representation of the exponential.

Thus, we can create the unitary matrix  $\mathbf{U}$  as above from the axis and angle description in terms of the exponential matrices and use the similar quadratic product to rotate a vector expressed in the Pauli matrix basis much like a quaternion representation:

$$\mathbf{Y} = \mathbf{U}\mathbf{X}\mathbf{U}^*$$

## D.5 Lie Groups and Lie Algebras

This section provides a quick introduction to the theory and application of Lie groups and Lie algebras, which are important for theoretical physics. They can be used to simplify calculus on manifolds. We will not give a formal definition of a Lie group just yet, but instead offer the simple description given by Sattinger and Weaver [71]. A Lie group is a continuous group which is also a topological manifold (concepts such as connectedness and continuity apply) on which the group operations are analytic. We have already seen several examples of Lie groups — the unit quaternions  $\hat{\mathbb{H}}$  and the special orthogonal  $3 \times 3$  matrices. We now offer a set of definitions and explanations about Lie groups and how to get to the Lie algebra of a group. The theory of Lie algebras will lead to some other proofs about quaternions leading to rotations.

**Definition 24** *The Lie algebra of a Lie group is the tangent space at the group identity element, which can be found for a linear group by differentiating all curves that pass through the identity element  $\mathbf{1}$  at  $t = 0$  and evaluating at  $t = 0$ .*

We have already seen that for the unit quaternion group  $\hat{\mathbb{H}}$  the tangent space is  $\mathbb{R}^3$ . We will denote the Lie algebra of an arbitrary group  $\mathfrak{G}$  or  $\mathbb{G}$  as the lowercase  $\mathfrak{g}$ , following Sattinger.

A Lie algebra is a vector space over some field  $F$  (usually  $\mathbb{R}$  or  $\mathbb{C}$ ), which implies it is linear:

$$\alpha(\mathbf{X} + \mathbf{Y}) = \alpha\mathbf{X} + \alpha\mathbf{Y}$$

It also has a product operator called the *Lie bracket*, denoted as  $[\ , \ ]$ , with the following properties:

1. *Closure*:  $\mathbf{X}, \mathbf{Y} \in \mathfrak{g}$  implies  $[\mathbf{X}, \mathbf{Y}] \in \mathfrak{g}$ .
2. *Distributive*:  $[\mathbf{X}, \alpha\mathbf{Y} + \beta\mathbf{Z}] = \alpha[\mathbf{X}, \mathbf{Y}] + \beta[\mathbf{X}, \mathbf{Z}] \quad \forall \alpha, \beta \in F, \mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathfrak{g}$ .
3. *Skew symmetry*:  $[\mathbf{X}, \mathbf{Y}] = -[\mathbf{Y}, \mathbf{X}]$ .

4. *Jacobi identity*:  $[\mathbf{X}, [\mathbf{Y}, \mathbf{Z}]] + [\mathbf{Y}, [\mathbf{Z}, \mathbf{X}]] + [\mathbf{Z}, [\mathbf{X}, \mathbf{Y}]] = 0$

For a matrix Lie algebra (elements are represented as matrices), the Lie product can be chosen to be the *commutator* of the two matrices, which gives an idea of how “non-commutative” two elements are. The commutator of  $\mathbf{X}$  and  $\mathbf{Y} = \mathbf{X}\mathbf{Y} - \mathbf{Y}\mathbf{X} = [\mathbf{X}, \mathbf{Y}]$ .

If we imagine vectors in  $\mathbb{R}^3$  to be pure quaternions, then as the commutator (in terms of quaternion multiplication) of two elements is  $\mathbf{x}\mathbf{y} - \mathbf{y}\mathbf{x}$ . As we saw above, this is  $2(\mathbf{x} \times \mathbf{y})$ . Therefore, we can use the cross product operator in  $\mathbb{R}^3$  as the Lie bracket of the quaternion algebra without using a matrix representation! It is easily shown that the cross product satisfies the four properties of a Lie bracket as defined above.

We define the *structure constants* of the Lie algebra as the set of constants  $C_{ijk}$  such that

$$[\mathbf{E}_i, \mathbf{E}_j] = \sum_k C_{ijk} \mathbf{E}_k$$

where the  $\{\mathbf{E}_i\}$  form a basis for the algebra.

Again using our quaternion example, we use the normal basis for  $\mathbb{R}^3$  of  $\{\mathbf{e}_i\}$ , consisting of the unit vectors  $(1, 0, 0)$ ,  $(0, 1, 0)$  and  $(0, 0, 1)$ . Then the structure constants must have the property that

$$\mathbf{e}_i \times \mathbf{e}_j = \sum_k C_{ijk} \mathbf{e}_k$$

Then clearly  $C_{ijk}$  must be 1 if  $(i, j, k)$  is a cyclic permutation of  $(1, 2, 3)$ , -1 if  $(i, j, k)$  is an anti-cyclic permutation of  $(1, 2, 3)$  and 0 otherwise. In physics this is often described by the completely antisymmetric tensor  $\varepsilon_{ijk}$  defined as

$$\varepsilon_{ijk} = \begin{cases} 1 & \text{if } ijk \text{ are a cyclic permutation of } 123 \\ -1 & \text{if } ijk \text{ is an anticyclic permutation of } 123 \\ 0 & \text{otherwise} \end{cases}$$

The structure constants for the cross product are therefore defined by  $\varepsilon_{ijk}$ .

**Definition 25** The adjoint operator of an element  $\mathbf{X} \in \mathfrak{G}$  (denoted as  $\text{ad } \mathbf{X}$ ) is the matrix which maps  $\mathbf{Y}$  into  $[\mathbf{X}, \mathbf{Y}]$ .

For our vector space  $\mathbb{R}^3$  and the cross product Lie bracket, the adjoint representation needs to define a matrix  $\mathbf{X}$  based on the vector  $\mathbf{x}$  such that  $\mathbf{X}\mathbf{y} = \mathbf{x} \times \mathbf{y}$ . The following mapping of  $\mathbf{x}$  into a  $3 \times 3$  skew-symmetric matrix accomplishes this:

$$\mathbf{x} \mapsto \tilde{\mathbf{X}} \stackrel{\text{def}}{=} \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}$$

We will sometimes use the  $\sim$  character over matrices which we have created in this way for clarity.

So the skew-symmetric matrix made from  $\mathbf{x}$  is the adjoint representation of  $\mathbf{x}$ . In other words,  $\text{ad } \mathbf{x} = \tilde{\mathbf{X}}$ .

## D.6 Recommended Reading

The treatment in this section has been derived from several sources. A good introduction to vector spaces, linear transformations and group theory can be found in Artin [2] and also the less theoretical book by Strang [81]. The quaternion hypercomplex algebra extension to complex numbers was re-derived from the original Hamilton equations by the author, but a fairly good introduction can be found in [32] although it gets dense very quickly. The section on group theory and the group manifold was mostly collected from McCarthy's kinematics book [59] and portions of Sattinger and Weaver's book on Lie groups [71], which unfortunately uses mostly examples from quantum theory. Recently, the book by Gallier [23] collects many of these concepts into a fairly comprehensive book with a computational focus.



# Bibliography

- [1] Matthew E. Antone. *Robust Camera Pose Recovery Using Stochastic Geometry*. PhD thesis, Massachusetts Institute of Technology, 2001.
- [2] Michael Artin. *Algebra*. Prentice-Hall, 1991.
- [3] Norman I. Badler, Cary B. Phillips, and Bonnie Lynn Webber. *Simulating Humans: Computer Graphics Animation and Control*. Oxford University Press, 1993.
- [4] A. Barr, B. Currin, S. Gabriel, and J. Hughes. Smooth interpolation of orientations with angular velocity constraints using quaternions. In *Computer Graphics*, pages 313–320. ACM Press, 1992.
- [5] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to Splines for use in Computer Graphics and Geometric Modeling*. Morgan Kaufman, 1987.
- [6] E. T. Bell. *Men of Mathematics*. Simon and Schuster, 1937.
- [7] Christopher Bingham. An antipodally symmetric distribution on the sphere. *Annals of Statistics*, 2(6):1201–1225, 1974.
- [8] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.
- [9] Jonathan Blow. Inverse kinematics with joint limits. *Game Developer Magazine*, April 2002.
- [10] Bruce Blumberg. Swampy! Synthetic Characters Group, MIT Media Lab. Appeared at SIGGRAPH '98 Interactive Exhibition, 1998.
- [11] Bruce Mitchell Blumberg. *New Dogs, Old Tricks: Ethology and Interactive Characters*. PhD thesis, The Media Lab, Massachusetts Institute of Technology, 1997.
- [12] Matthew Brand and Aaron Hertzmann. Style machines. *Computer Graphics (Proceedings of SIGGRAPH 2000)*, 2000.
- [13] Cynthia L. Breazeal. *Designing Sociable Robotics*. The MIT Press, 2002.
- [14] William L. Burke. *Applied Differential Geometry*. Cambridge University Press, 1985.

- [15] Samuel R. Buss and Jay P. Fillmore. Spherical averages and applications to spherical splines and interpolation. *ACM Transactions on Graphics*, 20(2), April 2001.
- [16] Robert T. Collins. *Model Acquisition Using Stochastic Projective Geometry*. PhD thesis, University of Massachusetts, 1993.
- [17] Eduardo Baryo Corrochano and Garret Sobczyk, editors. *Geometric Algebra with Applications in Science and Engineering*. Birhauser, 2001.
- [18] John J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley Publishing Company, Inc., second edition edition, 1989.
- [19] Erik B. Dam, Martin Koch, and Martin Lillholm. Quaternions, interpolation and animation. Technical Report DIKU-TR-98/5, University of Copenhagen, July 1998.
- [20] Marc Downie. behavior, animation, music: the music and movement of synthetic characters. Master's thesis, Massachusetts Institute of Technology, 2000.
- [21] Aaron D'Souza, Sethu Vijayakumar, and Stefan Schaal. Learning inverse kinematics. *International Conference on Intelligence in Robotics and Autonomous Systems (IROS 2001)*, 2001.
- [22] Ajo Fod, Mya J. Mataric, and Odest Chadwicke Jenkins. Automated derivation of primitives for movement classification. *IEEE-RAS International Conference on Humanoid Robotics (Humanoids-2000)*, 2000.
- [23] Jean Gallier. *Geometric Methods and Applications for Computer Science and Engineering*. Springer, 2001.
- [24] Neil Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge University Press, 1999.
- [25] Michael Gleicher. Motion editing with spacetime constraints. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 139–ff. ACM Press, 1997.
- [26] Michael Gleicher and Andrew Witkin. Through the lens camera control. In *Computer Graphics (Proceedings of SIGGRAPH '92)*, volume 26, pages 331–340, 1992.
- [27] Ron Goldman. Cross product in four dimensions and beyond. In David Kirk, editor, *Graphic Gems III*, pages 84–88. AP Professional, 1992.
- [28] Ronald H. Goldman. Transformations as exponentials. In James Arvo, editor, *Graphics Gems II*, pages 332–337. Academic Press, Inc., 1991. useful intro to matrix exponentials.
- [29] F. S. Grassia. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3):29–48, 1998.
- [30] F. Sebastian Grassia. *Believable Automatically Synthesized Motion by Knowledge-Enhanced Motion Transformation*. PhD thesis, Carnegie Mellon University, 2000.

- [31] S.F. Gull, A.N. Lasenby, and C.J.L. Doran. Imaginary numbers are not real — the geometric algebra of spacetime. *Found. Phys.*, 23(9), 1993.
- [32] Klaus Gürlebeck and Wolfgang Sprössig. *Quaternionic and Clifford Calculus for Physicists and Engineers*. John Wiley and Sons, 1997.
- [33] William Rowan Hamilton. *Elements of Quaternions*. Longmans, 1866.
- [34] Andrew Hanson. The rolling ball. In David Kirk, editor, *Graphic Gems III*, pages 51–60. AP Professional, 1992.
- [35] Andrew Hanson. Geometry for n-dimensional graphics. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 149–170. AP Professional, 1994.
- [36] Andrew Hanson. Visualizing quaternions. In *SIGGRAPH 2000 Course Notes*, number 9. ACM SIGGRAPH, 2000.
- [37] Andrew J. Hanson. Rotations for n-dimensional graphics. In Alan W. Paeth, editor, *Graphics Gems V*, pages 55–64, 1995.
- [38] Chris Hecker. Game developer conference (gdc 2002) talk. (unpublished), 2002.
- [39] L. Herda, R. Urtasun, P. Fua, and A. Hanson. Automatic determination of shoulder joint limits using quaternion field boundaries. In *Proceedings of the 5th International Conference on Automatic Face and Gesture Recognition*, pages 95–100. IEEE Computer Society, 2002.
- [40] L. Herda, R. Urtasun, P. Fua, and A. Hanson. Automatic determination of shoulder joint limits using experimentally determined quaternion field boundaries. *International Journal on Robotics Research*, 2002. In press.
- [41] David Hestenes and Garret Sobczyk. *Clifford Algebra to Geometric Calculus*. Kluwer Academic Publishers, 1984.
- [42] David Hoag. Apollo guidance and navigation considerations of apollo imu gimbal lock. Technical report, Massachusetts Institute of Technology, 1963. MIT Instrumentation Laboratory Document E-1344 (available online at <http://www.hq.nasa.gov/office/pao/History/alsj/e-1344.htm>).
- [43] Berthold Klaus Paul Horn. *Robot Vision*. MIT Press, 1986. good appendix for vector algebra and calculus of variations.
- [44] Charles F. Rose III. *Verbs and Adverbs: Multidimensional Motion Interpolation Using Radial Basis Functions*. PhD thesis, Princeton University, 1999.
- [45] Michael Patrick Johnson. Multi-dimensional quaternion interpolation. In *SIGGRAPH '99 Conference Abstracts and Applications*, page 258, 1999.

- [46] Michael Patrick Johnson, Andrew Wilson, Bruce Blumber, Chris Kline, and Aaron Bobick. Sympathetic interfaces: Using a plush toy to direct synthetic characters. In *Proceedings of SIGCHI 1999*, 1999.
- [47] P.E. Jupp and K.V. Mardia. Maximum likelihood estimators for the matrix von mises-fisher and bingham distributions. *Annals of Statistics*, 7(3):599–606, May 1979.
- [48] B. Jüttler. Visualization of moving objects using dual quaternion curves. *Computers and Graphics*, 18(3):315–326, 1994.
- [49] B. Jüttler and M.G. Wagner. Computer-aided design with spatial rational B-spline motions. *Journal of Mechanical Design*, 118:193–201, June 1996.
- [50] John T. Kent. Asymptotic expansion for the bingham distribution. *Applied Statistics*, 36(2):139–144, 1987.
- [51] John T. Kent. The complex bingham distribution and shape analysis. *Journal of the Royal Statistical Society Series B (Methodological)*, 56(2):285–299, 1994.
- [52] Myoung-Jun Kim, Myung-Soo Kim, and Sung Yong Shin. A general construction scheme for unit quaternion curves with simple high order derivatives. In *Computer Graphics (Proceedings of SIGGRAPH '95)*, pages 369–376, 1995.
- [53] Jack B. Kuipers. *Quaternions and Rotation Sequences*. Princeton University Press, 1999.
- [54] Jehee Lee. *A Hierarchical Approach to Motion Analysis and Synthesis for Articulated Figures*. PhD thesis, Korea Advanced Institute of Science and Technology, 2000.
- [55] Jehee Lee and Sung Yong Shin. A hierarchical approach to interactive motion editing for human-like figures. *Computer Graphics, Proceedings of SIGGRAPH '99*, 1999.
- [56] Patrick-Gilles Mailliot. Using quaternions for coding 3d transformations. In Andrew S. Glassner, editor, *Graphics Gems*, pages 498–515. Academic Press Ltd., 1990. early ref to quats in CG.
- [57] Kanti V. Mardia and Peter E. Jupp. *Directional Statistics*. John Wiley and Sons, Ltd., 2000.
- [58] K.V. Mardia. Statistics of directional data. *Journal of the Royal Statistical Society. Series B (Methodological)*, 37(3):349–393, 1975.
- [59] J. M. McCarthy. *Introduction to Theoretical Mechanics*. MIT Press, 1990.
- [60] Charles W. Misner, Kip S. Thorne, and John Archibald Wheeler. *Gravitation*. W. H. Freeman and Company, 1970.
- [61] Jack Morrison. Quaternion interpolation with extra spins. In David Kirk, editor, *Graphic Gems III*, pages 96–97. AP Professional, 1992.



- [62] Parviz E. Nikravesh. *Computer-Aided Analysis of Mechanical Systems*. Prentice-Hall, Inc., 1988.
- [63] Ken Perlin. Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1), 1995.
- [64] Michael J. Prentice. Orientation statistics without parametric assumptions. *Journal of the Royal Statistical Society, Series B (Methodolo)*, 8:2:214–222, 1986. [www.jstor.org](http://www.jstor.org).
- [65] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.
- [66] Katherine Pullen and Christoph Bregler. Motion capture assisted animation: Texturing and synthesis. *Computer Graphics. Proceeding of SIGGRAPH 2002*.
- [67] R. Ramamoorthi and A.H. Barr. Fast construction of accurate quaternion splines. In *Computer Graphics (Proceedings of SIGGRAPH 1997)*, pages 287–292, 1997.
- [68] Alyn Rockwood. Geometric algebras: New foundations, new insights. In *SIGGRAPH 2000 Course Notes*, number 31, 2000.
- [69] C. Rose, B. Bodenheimer, and M. Cohen. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications*, Sep/Oct 1998.
- [70] E. B. Saff and A. D. Snider. *Fundamentals of Complex Analysis for Mathematics, Science and Engineering*. Prentice-Hall, Inc., 1976.
- [71] D. H. Sattinger and O. L. Weaver. *Lie Groups and Algebras with Applications to Physics, Geometry and Mechanics*. Springer-Verlag, 1986.
- [72] John Schlag. Using geometric constructions to interpolate orientation with quaternions. In James Arvo, editor, *Graphic Gems II*, pages 377–380. Academic Press, Inc., 1991.
- [73] Ken Shoemake. Quaternion calculus for animation. *SIGGRAPH '89 Course Notes*, 23, 1989. Math for SIGGRAPH.
- [74] Ken Shoemake. Quaternions and 4 x 4 matrices. In James Arvo, editor, *Graphic Gems II*, pages 351–354. Academic Press, Inc., 1991.
- [75] Ken Shoemake. Uniform random rotations. In David Kirk, editor, *Graphic Gems III*, pages 124–132. AP Professional, 1992.
- [76] Ken Shoemake. Arcball rotation control. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 175–192. AP Professional, 1994.
- [77] Ken Shoemake. Euler angle conversion. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 222–229. AP Professional, 1994.

- [78] Ken Shoemake. Fiber bundle twist reduction. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 230–236. AP Professional, 1994.
- [79] Karl Sims. Locomotion of jointed figures over complex terrain. Master’s thesis, Massachusetts Institute of Technology, 1987.
- [80] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.
- [81] Gilbert Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, 1988.
- [82] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, pages 2319–2323, Dec 22 2000.
- [83] Charles W. Therrien. *Decision Estimation and Classification: An Introduction to Pattern Recognition and Related Topics*. John Wiley and Sons, 1989.
- [84] Frank Thomas and Ollie Johnston. *The Illusion of Life: Disney Animation*. Walt Disney Productions, 1981.
- [85] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1), 1991.
- [86] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, 1992.
- [87] Chris Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. Master’s thesis, Simon Fraser University, 1993.
- [88] Hermann Weyl. *The Classical Groups: Their Invariants and Representations*. Princeton University Press, 1939.
- [89] Jane Wilhelms and Allen Van Gelder. Fast and easy reach-cone joint limits. *Journal of Graphics Tools*, 6(2):27–41, 2001.
- [90] Jr. William M. Tomlinson. *Synthetic Social Relationships for Computational Entities*. PhD thesis, Massachusetts Institute of Technology, 2002.
- [91] Robert C. Wrede. *Introduction to Vector and Tensor Analysis*. Dover Publications, Inc., 1963.