Introduction to Quaternions

Philip McLeod and Damon Simpson

10 April 2002

1 Introduction

Quaternions were discovered by Irishman, Sir William Rowan Hamilton (1805-1865). The problem of how to multiply triplets (eg. 3D points and vectors) together had pondered William for a over decade. One day he was walking across the Brougham Bridge in Dublin, when the solution popped into his head and he carved the basic formula into the bridge stone [8].

Sir William Rowan Hamilton (1805-1865) [15].

By the mid 1900s quaternions had lost favour to other forms of maths including vector and tensor analysis, and had all but disappeared [3]. In recent years quaternions have proved to be very useful in some fields of science including computer science. It's powerful treatment of 3D rotations has been applied successfully in Vision and Graphics.

Quaternions are an extension to mathematics in the field of complex numbers. Any operations that can be preformed in standard maths (eg. differentiation) can be done on quaternions, but they often have their own complicated methods for doing so. Quaternions can be applied to many areas of science, but most quaternion maths is very complicated and will not be covered in this tutorial [1][2].

This tutorial gives an introduction to basic quaternion mathematics. Firstly, the various notations of describing quaternions, and their special properties are introduced. Then we show how to preform basic mathematical manipulations on them, including addition, subtraction, multiplication, division and 3D rotations. Examples of each manipulation are also provided. Armed with this knowledge we then look at why quaternions are useful. Finally we look at what fields they are currently being used in. This tutorial also supplies a C++ definition of a quaternion class with many useful operations.

2 General Description of Quaternions

A quaternion has the form

$$\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \tag{1}$$

where w is a real number and x, y and z are real coefficients, $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are all square roots of -1. $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are all orthogonal to each other (like the x, y and z axis in euclidean space).

$$\begin{aligned} \mathbf{i} * \mathbf{i} &= \text{-1} \\ \mathbf{j} * \mathbf{j} &= \text{-1} \\ \mathbf{k} * \mathbf{k} &= \text{-1} \end{aligned}$$

However they have some properties that make them behave similarly to cross products *(Appendix 8.2)* of unit basis vectors.

$$\begin{aligned} \mathbf{i} * \mathbf{j} = \text{-}\mathbf{j} * \mathbf{i} &= \mathbf{k} \\ \mathbf{j} * \mathbf{k} = \text{-}\mathbf{k} * \mathbf{j} &= \mathbf{i} \\ \mathbf{k} * \mathbf{i} = \text{-}\mathbf{i} * \mathbf{k} &= \mathbf{j} \end{aligned}$$

From this we get the property

$$\mathbf{i} * \mathbf{j} * \mathbf{k} = \text{-1}$$

Quaternions are also represented in other forms. As a vector

$$\mathbf{q} = [x\ y\ z\ w] \tag{2}$$

where the imaginary parts are listed first and the real (scaler) part last, or as a list.

$$\mathbf{q} = (s,\ \mathbf{v}) \tag{3}$$

where s is the real scaler, w, and $\mathbf{v}$ is a vector of the imaginary parts of the form $[x\ y\ z]$.

2.1 Properties of Quaternions

The magnitude of a quaternion is given by

$$|\mathbf{q}| = \sqrt{w^2 + x^2 + y^2 + z^2} \tag{4}$$

A unit quaternion has a magnitude of 1.
Quaternions are associative ie. $(\mathbf{q}_1 * \mathbf{q}_2) * \mathbf{q}_3 = \mathbf{q}_1 * (\mathbf{q}_2 * \mathbf{q}_2)$.
Quaternions are not commutative ie. $\mathbf{q}_1 * \mathbf{q}_2 \neq \mathbf{q}_2 * \mathbf{q}_1$.

3 Basic Manipulation of Quaternions

All of the following mathematical operations have been implemented in C++ code for your convenience and can be found in *Appendix 8.4*.

3.1 Addition and Subtraction of Quaternions

The addition and subtraction of quaternions is trivial. The constituent parts of the quaternions are simply added (or subtracted) together.

$$\begin{aligned}
\mathbf{q}_1 &= w_1 + x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k} \\
\mathbf{q}_2 &= w_2 + x_2\mathbf{i} + y_2\mathbf{j} + z_2\mathbf{k} \\
\mathbf{q}_1 + \mathbf{q}_2 &= (w_1 + w_2) + (x_1 + x_2)\mathbf{i} + (y_1 + y_2)\mathbf{j} + (z_1 + z_2)\mathbf{k} \qquad (5)
\end{aligned}$$

For example

$$\begin{aligned}
\mathbf{q}_1 &= 6 + 4\mathbf{i} + 1\mathbf{j} - 3\mathbf{k} \\
\mathbf{q}_2 &= 11 + 2.5\mathbf{i} + 7\mathbf{j} + 3\mathbf{k} \\
\mathbf{q}_1 + \mathbf{q}_2 &= 17 + 6.5\mathbf{i} + 8\mathbf{j} + 0\mathbf{k} \\
&= 17 + 6.5\mathbf{i} + 8\mathbf{j}
\end{aligned}$$

3.2 Multiplication of Quaternions

The multiplication of two quaternions is a bit more complicated. It is done in the following way.

$$\begin{aligned}
\mathbf{q}_1 &= (s_1, \mathbf{v}_1) \\
\mathbf{q}_2 &= (s_2, \mathbf{v}_2) \\
\mathbf{q}_1 * \mathbf{q}_2 &= (s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2\ ,\ s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2) \qquad (6)
\end{aligned}$$

For example

$$\begin{aligned}
\mathbf{q}_1 &= (2\ ,\ [5\ \ 3\ \ \text{-}4]) \\
\mathbf{q}_2 &= (0.5\ ,\ [2\ \ \text{-}4\ \ 0]) \\
\mathbf{q}_1 * \mathbf{q}_2 &= \Big((2 * 0.5) - ([5\ \ 3\ \ \text{-}4] \cdot [2\ \ \text{-}4\ \ 0])\ , \\
&\quad (2 * [2\ \ \text{-}4\ \ 0]) + (0.5 * [5\ \ 3\ \ \text{-}4]) + ([5\ \ 3\ \ \text{-}4] \times [2\ \ \text{-}4\ \ 0])\Big) \\
&= (3\ ,\ [\text{-}9.5\ \ \text{-}14.5\ \ \text{-}28])
\end{aligned}$$

A more detailed explanation of the dot product and the cross product can be found in *Appendix 8.1* and *8.2*. We remind the reader that quaternions are not commutative.

$$\mathbf{q}_1 * \mathbf{q}_2 \neq \mathbf{q}_2 * \mathbf{q}_1$$

3.3 Division of Quaternions

Due to the complex nature of quaternions, there is no direct method of division. Instead we use the multiplication of the inverse *(3.5)* in a similar fashion to complex numbers.

$$\frac{\mathbf{q}_1}{\mathbf{q}_2} \implies \mathbf{q}_1 * \mathbf{q}_2^{-1} \tag{7}$$

3.4 Quaternion conjugates

The conjugate of any quaternion is just the quaternion with negated imaginary parts. As for complex numbers the conjugate is written $\mathbf{q}'$.

$$\begin{aligned} \mathbf{q} &= w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \\ \mathbf{q}' &= w - x\mathbf{i} - y\mathbf{j} - z\mathbf{k} \end{aligned} \tag{8}$$

For example

$$\begin{aligned} \mathbf{q} &= 8 + 3\mathbf{i} - 5\mathbf{j} + 2.2\mathbf{k} \\ \mathbf{q}' &= 8 - 3\mathbf{i} + 5\mathbf{j} - 2.2\mathbf{k} \end{aligned}$$

3.5 Quaternion inverses

The quaternion inverse, ie. the multiplicative inverse, can be found by multiplying the top and bottom lines of $\frac{1}{\mathbf{q}}$ by the quaternion conjugate.

$$\mathbf{q}^{-1} = \frac{1}{\mathbf{q}} = \frac{\mathbf{q}'}{\mathbf{q} * \mathbf{q}'} \tag{9}$$

This is done to remove the complex parts from the bottom line *(Appendix 8.3)*, in the same manner as complex number division. For example

$$\begin{aligned} \mathbf{q} &= (1\ ,\ [2\ \ 1\ \ \text{-}2]) \\ \mathbf{q}' &= (1\ ,\ [\text{-}2\ \ \text{-}1\ \ 2]) \\ \mathbf{q}^{-1} &= \frac{\mathbf{q}'}{\mathbf{q} * \mathbf{q}'} \\ &= \frac{(1\ ,\ [\text{-}2\ \ \text{-}1\ \ 2])}{(1\ ,\ [2\ \ 1\ \ \text{-}2]) * (1\ ,\ [\text{-}2\ \ \text{-}1\ \ 2])} \\ &= \frac{(1\ ,\ [\text{-}2\ \ \text{-}1\ \ 2])}{10} \\ &= (0.1\ ,\ [0.2\ \ 0.1\ \ \text{-}0.2]) \end{aligned}$$

In the special case of a unit quaternion, its conjugate is its inverse.

$$|\mathbf{q}| = 1 \implies \mathbf{q}^{-1} = \mathbf{q}' \tag{10}$$

3.6 Rotations Using Quaternions

A quaternion can be used to rotate any 3D point in space, around an arbitrary axis [9]. Given a point $\mathbf{p} = [x\ y\ z]$, we build a quaternion $\mathbf{P} = (0,\ \mathbf{p})$. Then given a unit vector $\mathbf{u}$, which is the desired axis of rotation, and θ, the desired angle of rotation, we build a quaternion

$$\begin{aligned} \mathbf{q} &= (s, \mathbf{v}) && (11) \\ s &= \cos\frac{\theta}{2} && (12) \\ \mathbf{v} &= \mathbf{u}\sin\frac{\theta}{2} && (13) \end{aligned}$$

To perform the rotation, we multiply $\mathbf{q}$ on the left of $\mathbf{P}$ and $\mathbf{q}^{-1}$ on the right.

$$\begin{aligned} \mathbf{P} &= (0, \mathbf{p}) && (14) \\ \mathbf{P}_{\text{rotated}} &= \mathbf{q} * \mathbf{P} * \mathbf{q}^{-1} && (15) \end{aligned}$$

For example

1. Rotating a point $\mathbf{p}$ on the x axis, around the z axis, by $90°$ (anti-clockwise)

$$\begin{aligned} \mathbf{p} &= [2\ \ 0\ \ 0] \\ \mathbf{u} &= [0\ \ 0\ \ 1] \\ \theta &= 90° \end{aligned}$$

From this we build the following quaternions,

$$\begin{aligned} \mathbf{P} &= (0\ ,\ [2\ \ 0\ \ 0]) \\ \mathbf{q} &= (\cos\left(\frac{90}{2}\right)\ ,\ [0\ \ 0\ \ 1]\ *\ \sin\left(\frac{90}{2}\right)) \\ &= \left(\frac{\sqrt{2}}{2}\ ,\ [0\ \ 0\ \ \frac{\sqrt{2}}{2}]\right) \\ \mathbf{q}^{-1} &= \left(\frac{\sqrt{2}}{2}\ ,\ [0\ \ 0\ \ \frac{-\sqrt{2}}{2}]\right)\quad (\ =\ \mathbf{q}'\ \ eqn.(10)) \end{aligned}$$

Now we perform the rotation,

$$\begin{aligned} \mathbf{P}_{\text{rotated}} &= \mathbf{q} * \mathbf{P} * \mathbf{q}^{-1} \\ &= (0\ ,\ [0\ \ 2\ \ 0]) \end{aligned}$$

Clearly, this has returned the expected result $\mathbf{p}_{\text{rotated}} = [0\ \ 2\ \ 0]$.

2. Rotating a point $\mathbf{p}$, around $\mathbf{u}$, by 180°

$$\begin{aligned} \mathbf{p} &= [3 \;\; 4 \;\; 5] \\ \mathbf{u} &= [\frac{4}{5} \;\; \frac{\text{-}3}{5} \;\; 0] \\ \theta &= 180^\circ \end{aligned}$$

From this we build the following quaternions,

$$\begin{aligned} \mathbf{P} &= (0 \; , \; [3 \;\; 4 \;\; 5]) \\ \mathbf{q} &= (\cos\left(\frac{180}{2}\right) \; , \; [\frac{4}{5} \;\; \frac{\text{-}3}{5} \;\; 0] \; * \; \sin\left(\frac{180}{2}\right)) \\ &= \left(0 \; , \; [\frac{4}{5} \;\; \frac{\text{-}3}{5} \;\; 0]\right) \\ \mathbf{q}^{-1} &= \left(0 \; , \; [\frac{\text{-}4}{5} \;\; \frac{3}{5} \;\; 0]\right) \;\; (\; = \; \mathbf{q}' \;\; eqn.(10)) \end{aligned}$$

Now we perform the rotation,

$$\begin{aligned} \mathbf{P}_{\text{rotated}} &= \mathbf{q} * \mathbf{P} * \mathbf{q}^{-1} \\ &= (0 \; , \; [\text{-}3 \;\; \text{-}4 \;\; \text{-}5]) \end{aligned}$$

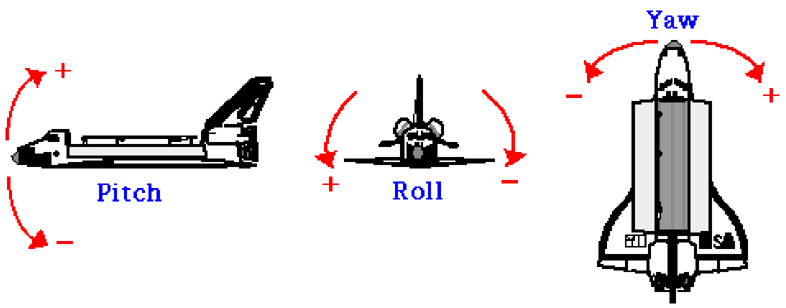
It happens that our axis of rotation is perpendicular to the point vector $\mathbf{p}$, so the returned result of $\mathbf{p}_{\text{rotated}} = [\text{-}3 \;\; \text{-}4 \;\; \text{-}5]$ is expected.

Any combination of rotations applied to a point can be encapsulated by one quaternion rotation. For example

$$\begin{aligned} \mathbf{P}_{\text{rotated}} &= (\mathbf{q}_3 * \mathbf{q}_2 * \mathbf{q}_1) * \mathbf{P} * (\mathbf{q}_1^{-1} * \mathbf{q}_2^{-1} * \mathbf{q}_3^{-1}) \\ &= \mathbf{q}_{final} * \mathbf{P} * \mathbf{q}_{final}^{-1} \end{aligned}$$

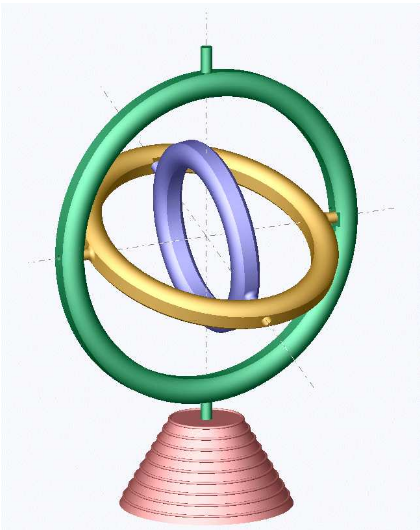
4 Why use Quaternions

Matrices have long been associated with the scaling, translation and rotation of 3D points. However their treatment of rotations is poor, cumbersome at best. A desired rotation must be performed by careful application of yaw, roll and pitch (the aeronautical terms for rotation using the euclidean coordinate system). For example, imagine you are viewing a space shuttle from above and from directly behind. The z axis is lined up with the tail and nose of the space shuttle. The x axis runs from the tip of the left wing to the tip of the right wing. The y axis points straight up from the ground. Pitch then becomes rotation in the x axis, yaw becomes rotation in the y axis and roll becomes rotation in the z axis. These limitations can cause problems.



Yaw, roll and pitch, the aeronautical terms for rotation using the euclidean coordinate system [13].

Gimbal lock is the name given to a problem that occurs with the use of the euclidean coordinate system. Because the final rotation matrix depends on the order of combining these three rotations, it is sometimes the case that the rotation in one axis will be mapped onto another rotation axis. Even worse, it may become impossible to rotate an object in a desired axis. This is called Gimbal lock. For example, assume that an object is being rotated in the order z, y, x and that the rotation in the y axis is 90 degrees. In this case, rotation in the z axis is performed first and therefore correctly. The y axis is also rotated correctly. However, after rotation in the y axis, the x axis is rotated onto the z axis. Thus, any rotation in the x axis actually rotates the object in the z axis. The only solution to this problem is to make use of quaternions.



With the graphic above you can think of Gimbal lock occurring whenever any two rings share a common plane [12].

Quaternions have the advantage over matrices for doing 3D rotations, that they don't suffer from Gimbal lock [4]. Quaternions extend the concept of rotation in three dimensions to rotation in four dimensions and allows for the implementation of smooth and continuous rotation.

Rotations are not the only useful application of quaternions. In the field of physics, they have been used for other favourable qualities. Namely, they prove useful for showing relationships between equations that are not always apparent [8]. This helps to generalise patterns in physical phenomenon in order to gain a deeper understanding of our universe.

5 Converting between Quaternions and Matrices

With the wide use of matrices in computer science, it is often necessary to have a quaternion rotation in to matrix form. In this case it is useful to have a formula for converting a quaternion in to a matrix. Here we provide methods for conversion in both directions.

5.1 Converting Quaternions to Matrices

A Quaternion can converted in to a rotation matrix in the the following way. Given a quaternion

$$\mathbf{q} = \begin{bmatrix} x & y & z & w \end{bmatrix}$$

then the rotation matrix is

$$\begin{bmatrix} 1-2y^2-2z^2 & 2xy-2zw & 2xz+2yw & 0 \\ 2xy+2zw & 1-2x^2-2z^2 & 2yz-2xw & 0 \\ 2xz-2yw & 2yz+2xw & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.2 Converting Matrices to Quaternion

A rotation matrix is a bit harder to convert in to a quaternion, but can be done in the following way. Given a rotation matrix

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & 0 \\ x_{21} & x_{22} & x_{23} & 0 \\ x_{31} & x_{32} & x_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

First, the trace must be calculated, by summing the diagonal

$$T = x_{11} + x_{22} + x_{33} + 1$$

If T is greater than zero then

$$\mathbf{q} = \begin{bmatrix} \frac{(x_{32}-x_{23})}{2\sqrt{T}} & \frac{(x_{13}-x_{31})}{2\sqrt{T}} & \frac{(x_{21}-x_{12})}{2\sqrt{T}} & \frac{\sqrt{T}}{2} \end{bmatrix}$$

If T is less than or equal to zero then identify which major diagonal element (x_{11}, x_{22} or x_{33}) has the greatest value. Depending on this, calculate the following.

If x_{11} is greatest:

$$\begin{aligned} s &= 2 * \sqrt{1.0 + x_{11} - x_{22} - x_{33}} \\ \mathbf{q} &= \left[\begin{array}{cccc} \frac{1}{2s} & s(x_{32} - x_{23}) & s(x_{13} - x_{31}) & s(x_{12} - x_{21}) \end{array} \right] \end{aligned}$$

If x_{22} is greatest:

$$\begin{aligned} s &= 2 * \sqrt{1.0 + x_{22} - x_{11} - x_{33}} \\ \mathbf{q} &= \left[\begin{array}{cccc} s(x_{12} - x_{21}) & \frac{1}{2s} & s(x_{23} - x_{32}) & s(x_{13} - x_{31}) \end{array} \right] \end{aligned}$$

If x_{33} is greatest:

$$\begin{aligned} s &= 2 * \sqrt{1.0 + x_{33} - x_{11} - x_{22}} \\ \mathbf{q} &= \left[\begin{array}{cccc} s(x_{13} - x_{31}) & s(x_{23} - x_{32}) & \frac{1}{2s} & s(x_{12} - x_{21}) \end{array} \right] \end{aligned}$$

6 Recent Applications of Quaternions

In working with simultaneous localisation and map-building (SLAM) for mobile robots, quaternions have been found to be useful [6]. In real-world environments, where robots are exposed to undulating ground, it is important to know both the position and orientation of the robot. This is necessary in the interpretation of the robot vision, to be able to extrapolate objects from a scene and produce an accurate mapping of the environment. It is in the recording of the robots orientation that quaternions have been applied in preference to other methods because of its mathematical convenience.

Colour images are being passed through Fourier transforms that use 2D quaternion (hypercomplex) numbers. This transform makes possible the handling of colour images in the frequency domain in a holistic manner, without separate handling of the colour components, and it thus makes possible very wide generalisation of monochrome Frequency domain techniques to colour images [14].

Neural networks are using quaternions to good effect in dealing with multidimensional signals. Learning algorithms using quaternion-valued multilayer perceptrons (QMLP) have been built which allow functions of three or four variables to be interpolated with a smaller number of connections with respect to the corresponding real-valued multi-layer perceptron [5].

The Computer Graphics world has perhaps benefited the most from quaternions. The predictable nature of the smooth and continuous rotations that result when using quaternions, are now used in such areas as computer games and in modelling human movement. They have also aided the field animation greatly.

Quaternions have recently found a home in physics. In furthering the understanding of quantum mechanics, they have studied the right eigenvalue equation for quaternionic and complex linear matrix operators defined in n-dimensional quaternionic vector spaces [7]. They are also being used to help find new relationships between physics equations, and may one day help in combining quantum theory and relativity in to the unified theory [8].

7 Conclusion

Quaternions are a fairly old branch of mathematics that has recently found many useful applications in computer science and physics. In particular its powerful treatment of 3D rotations has proved invaluable to the fields of Vision and Graphics. This extension to complex numbers holds the interesting mathematical properties of being associative but not commutative. Quaternions can be found in many different forms which may seem confusing at first. They often have their own complicated manipulations to perform normal mathematical operations. Unlike matrices, with their cumbersome rotations, quaternions offer smooth and continuous control without any of the problems associated with Gimbal lock. Still, it is useful to convert quaternion rotations in to matrix form because they can then be easily applied to other transformations. In conclusion, the use of quaternions can not be overlooked, Sir William could not have envisaged the far reaching applications of his quaternions as they are used today.

8 Appendix

8.1 Dot Product

The dot product takes two vectors and gives a real number [11].

$$
\begin{aligned}
\mathbf{a} &= [a_1 \;\; a_2 \;\; a_3] \\
\mathbf{b} &= [b_1 \;\; b_2 \;\; b_3] \\
\mathbf{a} \cdot \mathbf{b} &= (a_1 b_1 + a_2 b_2 + a_3 b_3)
\end{aligned}
$$

8.2 Cross Product

The cross product takes two vectors and gives a new vector which is perpendicular to both [10].

$$
\begin{aligned}
\mathbf{a} &= [a_1 \;\; a_2 \;\; a_3] \\
\mathbf{b} &= [b_1 \;\; b_2 \;\; b_3] \\
\mathbf{a} \times \mathbf{b} &= [(a_2 b_3 - a_3 b_2) \;\; (a_3 b_1 - a_1 b_3) \;\; (a_1 b_2 - a_2 b_1)]
\end{aligned}
$$

8.3 Proof that Multiplying $\mathbf{q} * \mathbf{q}'$ gives a real number

$$
\begin{aligned}
\mathbf{q} &= (s, [x \;\; y \;\; z]) \\
\mathbf{q}' &= (s, [\text{-}x \;\; \text{-}y \;\; \text{-}z]) \\
\mathbf{q} * \mathbf{q}' &= \Big(s * s - [x \;\; y \;\; z] \cdot [\text{-}x \;\; \text{-}y \;\; \text{-}z] \;, \\
&\qquad s[\text{-}x \;\; \text{-}y \;\; \text{-}z] + s[x \;\; y \;\; z] + [x \;\; y \;\; z] \times [\text{-}x \;\; \text{-}y \;\; \text{-}z] \Big) \\
&= \Big(s^2 - (x * \text{-}x + y * \text{-}y + z * \text{-}z) \;, \\
&\qquad [\text{-}sx \;\; \text{-}sy \;\; \text{-}sz] + [sx \;\; sy \;\; sz] + \\
&\qquad [(y * \text{-}z - z * \text{-}y) \;\; (z * \text{-}x - x * \text{-}z) \;\; (x * \text{-}y - y * \text{-}x)] \Big) \\
&= \Big(s^2 + x^2 + y^2 + z^2 \;, \\
&\qquad [(\text{-}sx + sx) \;\; (\text{-}sy + sy) \;\; (\text{-}sz + sz)] + \\
&\qquad [(\text{-}yz + yz) \;\; (\text{-}xz + xz) \;\; (\text{-}xy + xy)] \Big) \\
&= \Big(s^2 + x^2 + y^2 + z^2 \;, \; [0 \;\; 0 \;\; 0] \Big)
\end{aligned}
$$

8.4 Example C++ code for quaternions

```
//--------------------------quaternion.h----------------------------------
#ifndef QUATERNION_H
#define QUATERNION_H

//The Quaternion class encapsulates the basic manipulations that can be
//preformed on quaternions including addition, subtraction, multiplication,
//division, inverse, magnitude, conjugate and finding the unit.
```

```
//This class is used by the Vector3d class in order to preform rotations.
class Quaternion
{
public:
  double w, x, y, z;

  Quaternion() {}
  Quaternion(double w1, double x1, double y1, double z1) { w=w1; x=x1; y=y1; z=z1; }
  void setValue(double w1, double x1, double y1, double z1) { w=w1; x=x1; y=y1; z=z1; }
  double magnitude();
  Quaternion conjugate(); //returns q'
  Quaternion inverse(); //return 1/q
  Quaternion unit(); //returns a unit quaternion, of the same direction
  Quaternion operator+(const Quaternion &q);
  Quaternion operator-(const Quaternion &q);
  Quaternion operator*(const Quaternion &q);
  Quaternion operator*(double s); //multiply by a scaler
  Quaternion operator/(Quaternion q);
  Quaternion operator/(double s); //divide by a scaler
  void write(); //in the form w + xi + yj + zk
  void writeln();
  void write2(); //in the form (w, [x y z])
  void writeln2();
  void write3(); //in the form [x y z w]
  void writeln3();
};
#endif

//--------------------------quaternion.cc----------------------------------
#include "quaternion.h"
#include <math.h>
#include <stdio.h>

//returns the length of the quaternion
double Quaternion::magnitude()
{
  return sqrt(w*w + x*x + y*y + z*z);
}

//returns the conjugate of the quaternion
Quaternion Quaternion::conjugate()
{
  return Quaternion(w, -x, -y, -z);
}

//returns the inverse of the quaternion
//q^-1 = q'/(q*q')
//note: q*q' only has a real (scaler) part, so no point working out the rest
//note: (w*w) - (x*-x + y*-y + z*-z) == w*w + x*x + y*y + z*z
Quaternion Quaternion::inverse()
{
  double scaler = w*w + x*x + y*y + z*z;
  Quaternion result = conjugate()/scaler;
  return result;
}

//returns a unit quaternion, of the same direction
Quaternion Quaternion::unit()
{
  return (*this)/magnitude();
}
```

```
//adds q
Quaternion Quaternion::operator+(const Quaternion &q)
{
  return Quaternion(w + q.w, x + q.x, y + q.y, z + q.z);
}

//subrtacts q
Quaternion Quaternion::operator-(const Quaternion &q)
{
  return Quaternion(w - q.w, x - q.x, y - q.y, z - q.z);
}

//multiplies q on the right
Quaternion Quaternion::operator*(const Quaternion &q)
{
  Quaternion result;
  result.w = (w * q.w) - (x*q.x + y*q.y + z*q.z);
  result.x = w*q.x + q.w*x + (y*q.z - z*q.y);
  result.y = w*q.y + q.w*y + (z*q.x - x*q.z);
  result.z = w*q.z + q.w*z + (x*q.y - y*q.x);
  return result;
}

//multiply by a scaler
Quaternion Quaternion::operator*(double s)
{
  return Quaternion(w*s, x*s, y*s, z*s);
}

//division in done by multipling by the inverse
Quaternion Quaternion::operator/(Quaternion q)
{
  return (*this) * q.inverse();
}

//divide by a scaler
Quaternion Quaternion::operator/(double s)
{
  return Quaternion(w/s, x/s, y/s, z/s);
}

//writes the quaternion in the form w + xi + yj + zk
void Quaternion::write()
{
  printf("%lf + %lfx + %lfy + %lfz", w, x, y, z);
}

//same a write, but put an end of line character
void Quaternion::writeln()
{
  printf("%lf + %lfx + %lfy + %lfz\n", w, x, y, z);
}

//writes the quaternion in the form (w, [x y z])
void Quaternion::write2()
{
  printf("(%lf, [%lf %lf %lf])", w, x, y, z);
}

//same a write2, but put an end of line character
void Quaternion::writeln2()
{
```

```
  printf("(%lf, [%lf %lf %lf])\n", w, x, y, z);
}

//writes the quaternion in the form [x y z w]
void Quaternion::write3()
{
  printf("[%lf %lf %lf %lf]", x, y, z, w);
}

//same a write3, but put an end of line character
void Quaternion::writeln3()
{
  printf("[%lf %lf %lf %lf]\n", x, y, z, w);
}

//----------------------------vector.h------------------------------------
#ifndef VECTOR3D_H
#define VECTOR3D_H

#ifndef PI
#define PI 3.1415926536
#endif

//The Vector3d class can be used for representing 3d points and vectors
//It provides a rotate method which uses the qauternion class to preform
//a rotation around any axis by a specified angle.
//A rotate_points method is also supplied to rotate an array of points
//in one go, to be more efficient.
//These methods can be copied into your own vector class if desired.
class Vector3d
{
public:
  double x, y, z;

  //rotates an array of points around any vector axis by the specified angle (in radians)
  static void rotate_points(Vector3d *points, int num, double angle, Vector3d axis);

  Vector3d() {}
  Vector3d(double x1, double y1, double z1) { x=x1; y=y1; z=z1; }
  void setValue(double x1, double y1, double z1) { x=x1; y=y1; z=z1; }
  double magnitude();
  Vector3d unit(); //returns a unit vector, of the same direction
  //rotates a point around any vector axis by the specified angle (in radians)
  Vector3d rotate(double angle, Vector3d axis);
  Vector3d operator*(double s); //multiply by a scaler
  Vector3d operator/(double s); //divide by a scaler
  void write(); //in the form [x y z]
  void writeln();
};
#endif

//----------------------------vector.cc-----------------------------------
#include "vector3d.h"
#include "quaternion.h"
#include <math.h>
#include <stdio.h>

//returns the length of the vector.
double Vector3d::magnitude()
{
  return sqrt(x*x + y*y + z*z);
}
```

```
//returns a vector of length 1, in the same direction.
Vector3d Vector3d::unit()
{
  return (*this)/magnitude();
}

//use quaternions to do a rotation around axis by angle.
//note: angle in radians.
//note: axis doesn't have to be a unit Vector, it does this for you.
Vector3d Vector3d::rotate(double angle, Vector3d axis)
{
  Quaternion p(0, x, y, z);
  Vector3d unit_axis = axis.unit();
  double s = sin(angle/2);
  Quaternion q(cos(angle/2), unit_axis.x*s, unit_axis.y*s, unit_axis.z*s);
  //q is already a unit quaternion because the axis was made a unit vector
  Quaternion r = q * p * q.conjugate();
  Vector3d result(r.x, r.y, r.z);
  return result;
}

//rotates_points works like rotate above but you pass it a pointer to an array of
//points. It will preform the rotation on these points.
//num = the number of points in the array.
//angle in radians.
void Vector3d::rotate_points(Vector3d *points, int num, double angle, Vector3d axis)
{
  Quaternion p, r;
  Vector3d unit_axis = axis.unit();
  double s = sin(angle/2);
  Quaternion q(cos(angle/2), unit_axis.x*s, unit_axis.y*s, unit_axis.z*s);
  for(int j=0; j<num; j++) {
    p.setValue(0, points[j].x, points[j].y, points[j].z);
    r = q * p * q.conjugate();
    points[j].setValue(r.x, r.y, r.z);
  }
}

//multiply by a scaler
Vector3d Vector3d::operator*(double s)
{
  return Vector3d(x*s, y*s, z*s);
}

//divide by a scaler
Vector3d Vector3d::operator/(double s)
{
  return Vector3d(x/s, y/s, z/s);
}

//writes the vector out in the form (x, y, z)
void Vector3d::write()
{
  printf("(%lf, %lf, %lf)", x, y, z);
}

//same as write, but put a end of line character
void Vector3d::writeln()
{
  printf("(%lf, %lf, %lf)\n", x, y, z);
}
```

```
//-----------------------------main.cc------------------------------------
#include "quaternion.h"
#include "vector3d.h"
#include <stdio.h>

//Here is a simple example program which uses the Quaternion
//and Vector3d classes.
void main()
{
  int j;
  Quaternion q1(2, 5, 3, -4);
  Quaternion q2(0.5, 2, -4, 0);

  printf("q1 = "); q1.writeln();
  printf("q2 = "); q2.writeln();
  printf("q1 + q2 = "); (q1+q2).writeln();
  printf("q1 * q2 = "); (q1*q2).writeln();
  printf("The inverse of q1 is "); q1.inverse().writeln();
  printf("The unit quaternion of q1 is "); q1.unit().writeln();
  printf("The magnutude of q1 is %lf\n", q1.magnitude());

  Vector3d points[5];
  points[0].setValue(1, 0, 0);
  points[1].setValue(0, 1, 0);
  points[2].setValue(0, 0, 1);
  points[3].setValue(4, 5, 0.5);
  points[4].setValue(6,-7, 4);
  printf("\nSome points before undergoing a rotation\n\n");
  for(j=0; j<5; j++) { printf("points[%d] = ", j); points[j].writeln(); }
  printf("\nRotate the points 90 degrees (anti-clockwise) about the axis ");
  printf("[0, 0, 1] (the z axis).\n\n");
  //rotate 90' about z
  Vector3d::rotate_points(points, 5, 90*PI/180, Vector3d(0, 0, 1));
  for(j=0; j<5; j++) { printf("points[%d] = ", j); points[j].writeln(); }
}

/* Output
q1 = 2.000000 + 5.000000x + 3.000000y + -4.000000z
q2 = 0.500000 + 2.000000x + -4.000000y + 0.000000z
q1 + q2 = 2.500000 + 7.000000x + -1.000000y + -4.000000z
q1 * q2 = 3.000000 + -9.500000x + -14.500000y + -28.000000z
The inverse of q1 is 0.037037 + -0.092593x + -0.055556y + 0.074074z
The unit quaternion of q1 is 0.272166 + 0.680414x + 0.408248y + -0.544331z
The magnutude of q1 is 7.348469

Some points before undergoing a rotation

points[0] = (1.000000, 0.000000, 0.000000)
points[1] = (0.000000, 1.000000, 0.000000)
points[2] = (0.000000, 0.000000, 1.000000)
points[3] = (4.000000, 5.000000, 0.500000)
points[4] = (6.000000, -7.000000, 4.000000)

Rotate the points 90 degrees (anti-clockwise) about the axis [0, 0, 1] (the z axis).

points[0] = (-0.000000, 1.000000, 0.000000)
points[1] = (-1.000000, -0.000000, 0.000000)
points[2] = (0.000000, 0.000000, 1.000000)
points[3] = (-5.000000, 4.000000, 0.500000)
points[4] = (7.000000, 6.000000, 4.000000)
*/
```

References

[1] *The Outlines of Quaternions.* London : Longmans, Green, and CO., 1894.

[2] *Introduction to Quaternions.* London : Macmillan and CO., 1904.

[3] elibrary, Accessed 28 March 2002. URL http://www.encyclopedia.com/html/h/hamiltwr.asp.

[4] andreas@(no spam)cs.ualberta.ca. The matrix and quaternions faq, 2000. URL http://www.cs.ualberta.ca/ãndreas/math/matrfaq_1.06.html.

[5] P. Arena, L. Fortuna, and M. Xibilia. Neural networks in multidimensional domains. *JOURNAL OF SYSTEMS ENGINEERING*, 6(1):1 – 11, FEB 1996. ISSN 0938-7706.

[6] A. J. Davison and N. Kita. 3d simultaneous localisation and map-building using active vision for a robot moving on undulating terrain, 2001. URL http://www.frc.ri.cmu.edu/j̃mishler/data/cvpr/CVPR-3DSLAM.pdf.

[7] S. De Leo and G. Scolarici. Right eigenvalue equation in quaternionic quantum mechanics. *JOURNAL OF PHYSICS A-MATHEMATICAL AND GENERAL*, 33(15):2971 – 2995, APR 21 2000. ISSN 0305-4470.

[8] doug sweetser@TheWorld.com. Physics with quaternions, 1997. URL http://world.std.com/~sweetser/quaternions/qindex/qindex.html.

[9] L. K. J.C Hart, G.K Francis. Visualizing quaternion rotation. *ACM Trans. Graph.*, 1994.

[10] P. Kirby. The cross procuct, Accessed 7 April. URL http://members.tripod.com/~Paul_Kirby/vector/Vcrossproduct.html.

[11] P. Kirby. The dot procuct of two vectors, Accessed 7 April. URL http://members.tripod.com/~Paul_Kirby/vector/Vdotproduct.html.

[12] Mathworks. Gimbal (simmechanics), 2001. URL http://www.mathworks.com/access/helpdesk/help/toolbox/physmod/msb/gimbal.shtml.

[13] C. Rorden. 3d controls, Accessed 5 April. URL http://www.psychology.nottingham.ac.uk/staff/cr1/3d.html.

[14] S. Sangwine. Fourier transforms of colour images using quaternion or hypercomplex, numbers. *ELECTRONICS LETTERS*, 32(21):1979 – 1980, OCT 10 1996. ISSN 0013-5194.

[15] L. S. Taylor. Optics highlights! an anecdotal history of optics. URL http://rf.rfglobalnet.com/library/tutorials/articles/optics5.htm.