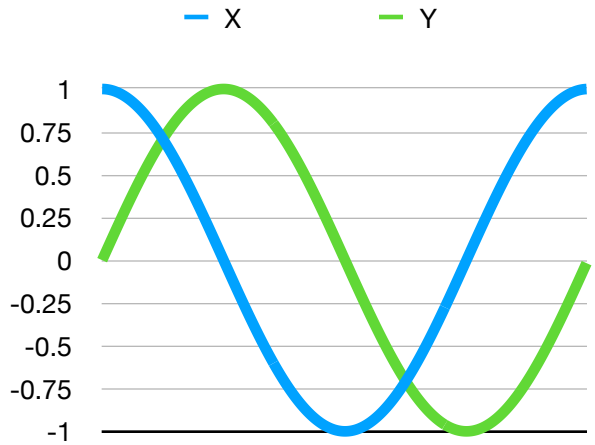


Vertical axis

$$x = cos(\theta)$$
$$y = sin(\theta)$$

Horizantal axis

1 *degree per tick*

Short Study on Quaternions

The study is focused on the math and properties of quaternions removing any obscurity it has received as being a difficult subject to grasp. The goal is to uncover the correctness in its use and its implementation when applied to rotations. The author admits the difficulty in its correct use as well. The free and open papers used for the study are few in number when discussing quaternions. The paper used in this study titled, *Quaternions, Interpolation and Animation* by Erik B. Dam, Martin Koch, and Martin Lillholm under the Technical Report DIKU-TR-98/5 dated July 17, 1998. In section 3.3, it covers quaternions and its properties. Many textbooks and other papers sadly covered little information on the properties. The authors may not fully grasp as well although speculatively, it might be why these papers were incompletely written without understanding the math and its properties. The selected paper, therefore, was chosen for the important information, the properties of quaternions. Furthermore this study also includes vectors and matrices in relation to quaternions and more so in relation to the active project for the development of the Algebra Library named Numerics kept at MageMCU Github.

• The *definition* of quaternions is the set of quaternions denoted by the Hamiltonian ***H***.

Quaternions can be identified as a vector with four-components.

$$\mathbf{q} = (w, x, y, z)$$

It can be identified as a complex number in three dimensions where the w is the real part and the $x\mathbf{i}, y\mathbf{j}, z\mathbf{k}$ are the imaginary parts.

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

The quaternion components (w, x, y, z) can be further identified to be a scalar w and a 3-component vector **v** with the components (*x, y, z*).

$$\mathbf{q} = w + \mathbf{q_v}$$

• Quaternion Dot (*Inner*) Product is a scalar.

$$\mathbf{p} \cdot \mathbf{q} = (p_w, p_x, p_y, p_z) \cdot (q_w, q_x, q_y, q_z) = p_w q_w + p_x q_x + p_y q_y + p_z q_z = p_w q_w + \mathbf{p_v} \cdot \mathbf{q_v}$$

• The *norm-squared* of the quaternion.

$$||q||^2 = w^2 + x^2 + y^2 + z^2 = w^2 + \mathbf{q_v} \bullet \mathbf{q_v}$$

• The *norm* of the quaternion, also known as the length or magnitude.

$$||q|| = \sqrt{q \bullet q} = \sqrt{w^2 + x^2 + y^2 + z^2}$$

• The *unit-quaternion*.

$$\hat{\mathbf{q}} = \frac{\mathbf{q}}{||q||} = \left(\frac{w}{||q||}, \frac{\mathbf{v}}{||q||} \right) = \left(\frac{w}{||q||}, \frac{x}{||q||}, \frac{y}{||q||}, \frac{z}{||q||} \right)$$

• The length of unit-quaternions.

$$\hat{\mathbf{q}} \cdot \hat{\mathbf{q}} = 1$$

• The *identity* $\mathbf{I}$ of the quaternion.

$$\mathbf{qI} = \mathbf{Iq} = \mathbf{q}$$

• The *conjugate* $\mathbf{q}^*$ of the quaternion

$$\mathbf{q}^* = (w, \mathbf{v})^* = (w, -\mathbf{v}) = (w, -x, -y, -z)$$

$$\mathbf{q}\mathbf{q}^* = (w^2, \mathbf{0}), \textit{where quaternion multiplication is discussed}.$$

• The *inverse* of the quaternion.

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

- *Quaternion multiplication* in algebraic form. This involves the vector dot product and the vector cross product. The resultant is a rotational unit-quaternion.

$$\hat{\mathbf{p}}\hat{\mathbf{q}} = (p_w q_w - \hat{\mathbf{p}}_\mathbf{v} \cdot \hat{\mathbf{q}}_\mathbf{v} \, , \, p_w \hat{\mathbf{q}}_\mathbf{v} + q_w \hat{\mathbf{p}}_\mathbf{v} + \hat{\mathbf{p}}_\mathbf{v} \times \hat{\mathbf{q}}_\mathbf{v}) = \hat{\mathbf{r}}$$

Or quaternion multiplication by components giving a four-component column-quaternion.

$$\hat{\mathbf{p}}\hat{\mathbf{q}} = \begin{bmatrix} \hat{r}_w \\ \hat{r}_x \\ \hat{r}_y \\ \hat{r}_z \end{bmatrix} = \begin{bmatrix} \hat{p}_w - \hat{p}_x - \hat{p}_y - \hat{p}_z \\ \hat{p}_x + \hat{p}_w - \hat{p}_z + \hat{p}_y \\ \hat{p}_y + \hat{p}_z + \hat{p}_w - \hat{p}_x \\ \hat{p}_z - \hat{p}_y + \hat{p}_x + \hat{p}_w \end{bmatrix} \begin{bmatrix} \hat{q}_w \\ \hat{q}_x \\ \hat{q}_y \\ \hat{q}_z \end{bmatrix} = \begin{bmatrix} \hat{p}_w \hat{q}_w - \hat{p}_x \hat{q}_x - \hat{p}_y \hat{q}_y - \hat{p}_z \hat{q}_z \\ \hat{p}_x \hat{q}_w + \hat{p}_w \hat{q}_x - \hat{p}_z \hat{q}_y + \hat{p}_y \hat{q}_z \\ \hat{p}_y \hat{q}_w + \hat{p}_z \hat{q}_x + \hat{p}_w \hat{q}_y - \hat{p}_x \hat{q}_z \\ \hat{p}_z \hat{q}_w - \hat{p}_y \hat{q}_x + \hat{p}_x \hat{q}_y + \hat{p}_w \hat{q}_z \end{bmatrix}$$

$$= \begin{bmatrix} \hat{q}_w - \hat{q}_x - \hat{q}_y - \hat{q}_z \\ \hat{q}_x + \hat{q}_w + \hat{q}_z - \hat{q}_y \\ \hat{q}_y - \hat{q}_z + \hat{q}_w + \hat{q}_x \\ \hat{q}_z + \hat{q}_y - \hat{q}_x + \hat{q}_w \end{bmatrix} \begin{bmatrix} \hat{p}_w \\ \hat{p}_x \\ \hat{p}_y \\ \hat{p}_z \end{bmatrix} = \begin{bmatrix} \hat{p}_w \hat{q}_w - \hat{p}_x \hat{q}_x - \hat{p}_y \hat{q}_y - \hat{p}_z \hat{q}_z \\ \hat{p}_w \hat{q}_x + \hat{p}_x \hat{q}_w + \hat{p}_y \hat{q}_z - \hat{p}_z \hat{q}_y \\ \hat{p}_w \hat{q}_y - \hat{p}_x \hat{q}_z + \hat{p}_y \hat{q}_w + \hat{p}_z \hat{q}_x \\ \hat{p}_w \hat{q}_z + \hat{p}_x \hat{q}_y - \hat{p}_y \hat{q}_x + \hat{p}_z \hat{q}_w \end{bmatrix}$$

Source: *Multiplication by components*:
Jack B. Kuipers, ***Quaternions and Rotation Sequences*** (Published by Princeton University, 1999) page156.

Oddly these two matrix multiplications are equal. See the quaternion code. Quaternion multiplication is not commutative where $\hat{\mathbf{p}}\hat{\mathbf{q}} \neq \hat{\mathbf{q}}\hat{\mathbf{p}}$.

- *The axis-angle to quaternion* $\hat{\mathbf{q}}(\hat{\mathbf{a}}, \theta)$ can be created with the axis of rotation $\hat{\mathbf{a}}$ a 3-component unit-vector, and with the angle θ an arbitrary angle θ oriented around the rotational-axis $\hat{\mathbf{a}}$.

$$\hat{\mathbf{q}}(\theta, \hat{\mathbf{a}}) = w(\theta) + \mathbf{v}(\theta \, , \, \hat{\mathbf{a}}) = \left(cos\left(\frac{\theta}{2}\right) \, , \, \hat{\mathbf{a}} \, sin\left(\frac{\theta}{2}\right) \right)$$

$$= \left(cos\left(\frac{\theta}{2}\right) \, , \, \hat{\mathbf{a}}_\mathbf{x} sin\left(\frac{\theta}{2}\right), \hat{\mathbf{a}}_\mathbf{y} sin\left(\frac{\theta}{2}\right), \hat{\mathbf{a}}_\mathbf{z} sin\left(\frac{\theta}{2}\right) \right)$$

- *The quaternion to axis-angle* requires obtaining different values from the single quaternion. For me personally, this was not easy to interpret. Let's consider the inverse cosine function defined as:

$$y = cos^{-1}(x)\ if\ cos(y) = x,\ where\ -1 \leq x \leq 1\ and\ 0 \leq y \leq \pi,$$

the quaternion identity $\mathbf{I} = (w, \mathbf{0})$ where $w = 1$, therefore conditionally the angle θ *is calculated:*

$$If\ w \leq 1,\ then\ angle\ \theta = 2\,cos^{-1}(w).$$

The axis $\hat{\mathbf{a}}$ conditionally is calculated:

$$If\ w = 1,\ then\ \hat{\mathbf{a}} = \mathbf{0}.$$

$$If\ w < 1,\ then\ \hat{\mathbf{a}} = \frac{(x, y, z)}{\sqrt{1 - w^2}}.$$

Source: *The axis-angle to quaternion - The quaternion to axis-angle*:
Philip J. Schneider, David H. Eberly, ***Geometric Tools for Computer Graphics*** (Morgan Kaufmann Publishers, 2003) page 860.

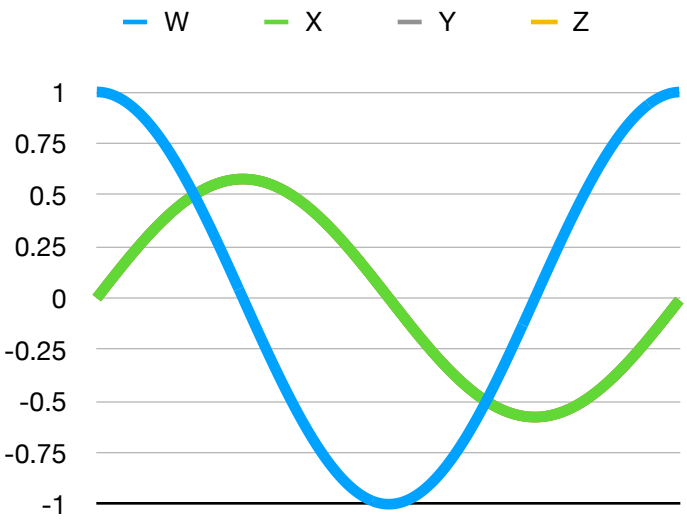
The quaternion to axis-angle has yet to be implemented hence the TESTS reflecting its *nearly completed* notice…

Programmer's Notes

As a mental frame of reference, the ***first plot*** above the title of this article is a spreadsheet line chart of a ***simple trigonometric cosine-sine plot*** programmed iteratively to the csv-file of the data giving about 360 data points (x, y). The quaternions have a similar analog in quaternion multiplication. ***Note the cosine-sine relationships among all the plots including the Arduino.***

The equation used to plot the quaternion multiplication is $\hat{\mathbf{q}}' = \hat{\mathbf{q}}\hat{\mathbf{c}}$ where $\hat{\mathbf{q}}$ is the unit-quaternion variable initially set to the identity $\mathbf{I}$ as $\hat{\mathbf{q}} = \mathbf{I}$. Each iteration or after each quaternion multiplication, $\hat{\mathbf{q}}$ is re-assigned as $\hat{\mathbf{q}} = \hat{\mathbf{q}}'$ which is actually the same variable. The unit-quaternion $\hat{\mathbf{c}}$ is a rotational constant assigned a ***rotational axis of (1, 1, 1***) then again with ***(1, 2, 3***) for two separate plots. The angle for each plot remains at a constant of 1-degree which in radian measure is 0.01745329…, yet the quaternion constructor reduces the radian measure by one-half thereby giving about 720 data points (w, x, y, z).

In this experiment, the unit-quaternion $\hat{\mathbf{c}}$ is set as the rotational constant assigned a ***rotational axis of (1, 1, 1***) and the angle is a constant of 1-degree which is converted to radian measure of 0.01745329 where the radian measure is then reduced by one-half by the constructor. The curves for Y and Z sit behind the green curve for X where W is the blue curve.



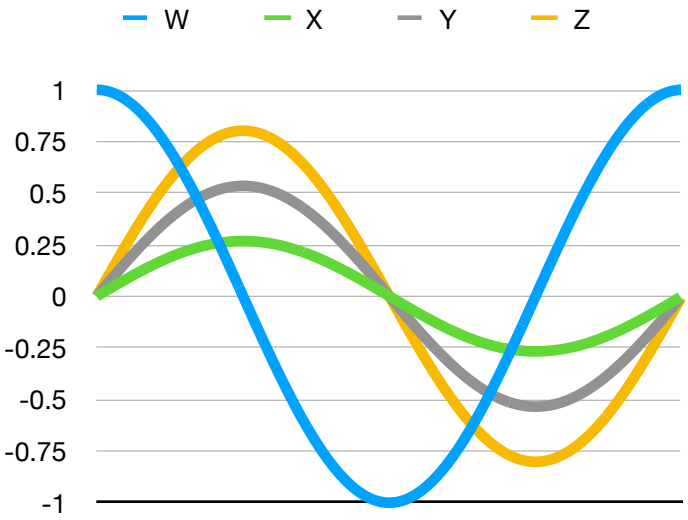
Vertical axis

w
x
y
z

Horizantal axis

$\frac{1}{2}$ *degree per tick*

In this next experiment, the unit-quaternion $\hat{\mathbf{c}}$ is set as the rotational constant assigned a ***rotational axis of (1, 2, 3***) and the angle is a constant of 1-degree which is converted to radian measure of 0.01745329 where the radian measure is then reduced by one-half by the constructor.



Vertical axis

w
x
y
z

Horizantal axis

$\frac{1}{2}$ *degree per tick*

I had fun designing this experiment… I leave it to the reader to interpret all 3 plots especially their similarities and differences. Do the experiment yourself to verify whether you can reproduce the experiment. The Numerics Library can be located at MageMCU at Github. The csv-files can be located there as well. Match the curves along with the data as well. Enjoy!

Simulations

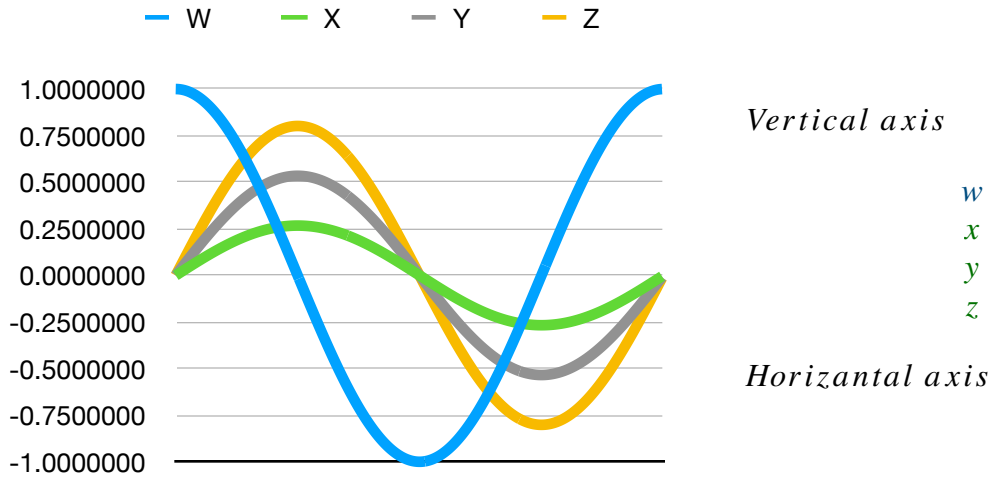
The simulations were initially done using the Unity Game Engine. The reason for the simulations was to elucidate to me personally how quaternions would behave under different conditions and to demonstrate how to use them correctly.

Experimental Demonstration

Once the simulations were understood, I designed the experiment to demonstrate an analogy between simple cosine-sine functions and quaternions. Computer simulations produced a clear image versus trying to visualize math-proofs. In the first plot, for example, the angle θ was incremented 1-degree giving about 320 data points. It was used as a comparative image placed against that of the math of quaternions. In the quaternion plots instead of using addition, each iteration was multiplied by a constant to a quaternion variable giving about 720 data points. The analogy between the two maths can be easily recognized. The fact was not to add quaternions but to multiply quaternions producing similar results as in the trigonometric plot. This simple difference between addition and multiplication was elusive. All I get from math-proofs, well perspectively, what I cannot see. There will be further study when time permits. The quaternion class has the basic foundation to further develop other methods.

Arduino Experiment

Once satisfied with the simulations, the Numerics program for the Arduino Uno using VS-Code and PlatformIO IDE was tested with the updated quaternion class in C++ code. Arduino produced the exact results as the second quaternion experiment above…



There was an issue when using Serial.print() and its counterpart Serial.println(). It would only print two decimal places. In the TESTS folder, is the Common.h file where it has the function *printQuaternion()* with a single parameter. There, it discusses Arduino's float data-type and String

object-type along with its sources. The code statement, *Serial.print(String(q.Element(i), 7)),* will print 7 decimal places as seen in the vertical axis of the chart. Review the code for the details and again, try to reproduce the data yourself.

Code Snippets

The snippets of the quaternion code below are part of the C++ Library called Numerics at MageMCU at Github.

The Constructor:

```
122     template <typename real>
123     Quaternion<real>::Quaternion(Vector3<real> axis, real angleRadian)
124     {
125         m_size = 4;
126         Vector3<real> normalized = axis.Normalize();
127         real s = (real)sin((double)angleRadian / (double)2);
128         real w = (real)cos((double)angleRadian / (double)2);
129         q_tuples[0] = w;
130         q_tuples[1] = normalized.x() * s;
131         q_tuples[2] = normalized.y() * s;
132         q_tuples[3] = normalized.z() * s;
133     }
```

The Quaternion Multiplication Components:

```
267     // Although different, the result is the same...
268     template <typename real>
269     Quaternion<real> Quaternion<real>::Multiply(Quaternion<real> c)
270     {
271         // Quaternion Multiplication (qc) - Unit Quaternions
272         Quaternion<real> q = ToQuaternion();
273         real w = q.w() * c.w() - q.x() * c.x() - q.y() * c.y() - q.z() * c.z();
274         real x = q.x() * c.w() + q.w() * c.x() - q.z() * c.y() + q.y() * c.z();
275         real y = q.y() * c.w() + q.z() * c.x() + q.w() * c.y() - q.x() * c.z();
276         real z = q.z() * c.w() - q.y() * c.x() + q.x() * c.y() + q.w() * c.z();
277         // Quaternion
278         Quaternion<real> quat(w, x, y, z);
279         // Is there a norm-squared  deviation
280         if (NormDeviation(quat.NormSquared()))
281         {
282             // Unit Quaternion
283             // Must normalize quaterion afterwards otherwise the components
284             // 'might' converge to zero due to the floating point rounding
285             // errors that diviates a unit-quaternion from its norm.
286             Quaternion<real> uQuat = quat.UnitQuaternion();
287             return uQuat;
288         }
289         //
290         return quat;
291     }
```

The Quaternion Multiplication Algebraic:

```
295     template <typename real>
296     Quaternion<real> Quaternion<real>::operator*(Quaternion<real> c)
297     {
298         // Quaternion Multiplication - Unit Quaternions
299         Quaternion<real> q = ToQuaternion();
300         Vector3<real> qV = q.ToVector();
301         Vector3<real> cV = c.ToVector();
302         // Quaternion Product
303         real wS = q.w() * c.w(); // Scalar Multiplication
304         wS -= qV * cV;           // Vector Dot Product
305         // Vector-Scalar Products & Vector-Cross Product (^)
306         // includes vector additions...
307         Vector3<real> v = (qV * c.w()) + (cV * q.w()) + (qV ^ cV);
308         // Quaternion
309         Quaternion<real> quat(wS, v.x(), v.y(), v.z());
310         // Is there a norm-squared  deviation
311         if (NormDeviation(quat.NormSquared()))
312         {
313             // Unit Quaternion
314             // Must normalize quaterion afterwards otherwise the components
315             // 'might' converge to zero due to the floating point rounding
316             // errors that diviates a unit-quaternion from its norm.
317             Quaternion<real> uQuat = quat.UnitQuaternion();
318             return uQuat;
319         }
320         //
321         return quat;
322     }
```

The Test:

```
31  void Quaternion_T8_QuaternionMultiplication()
32  {
33      printTitle("Quaternion T8 Quaternion Multiplication");
34      // This prints out a stream of 4-columns of data
35      // it could take a minute or two to finish...
36
37      // Intialization
38      // This becomes a half-angle in
39      // quaternion multiplication...
40      float radian = 1.0 * DEG_TO_RAD;
41      nmr::Vector3<float> axis(1, 2, 3);
42      // Constant
43      nmr::Quaternion<float> c(axis, radian);
44      // Multiplicative
45      nmr::Quaternion<float> q(axis, 0.0);
46
```

```
62      // Game Loop 2 with second quaternion multiplication
63      // int i = 0;
64      do
65      {
66          // Prints four columns with 7 decimal digits
67          // data for a csv-file
68          printQuaternion(q);
69          // Quaternion Multiplication --- SECOND method
70          q = q.Multiply(c);
71          // Counter
72          i++;
73      } while (i < 721);
74  }
```

Final Note:

Matrix multiplications and additions for a MCU like the Atmega328P is a little bit too much processing especially the memory usage although the quaternions are a little faster than matrix computations. Yet even quaternions can be studied further to reduce these calculations. For example, in some cases using the directional unit-vector when used for robotic rotations, the number of calculations is surprisingly at a bare-minimum. This technique will be discussed in up-coming articles.

Carpenter Software
Article 1005-Math-Quaternions-005 20221207.
Revision 1005-Math-Quaternions-010 20221224.
by Jesse Carpenter