Fun with Octonions in C++

John R. Berryhill

Abstract

Historically as well as mathematically, the octonions were derived from the quaternions, and the quaternions from the complex numbers. A proper C++ implementation of these numerical types should reflect these relationships. This brief note describes how the author's previously published C++ Quaternion class serves as the natural foundation for a C++ Octonion class.

Fun with Octonions in C++

John R. Berryhill

A complex number is simply an ordered pair of ordinary numbers:

```
class complex
{
private:
            double re, im;
```

A quaternion, in turn, is a pair of complex numbers:

```
class quatern
{
private:

            complex re, im;
```

Notice that `re` and `im` are just names for the members of the ordered pair. The next step is the octonion, an ordered pair of quaternions:

```
class octon
{
private:

            quatern re, im;
```

One advantage of this way of proceeding is that each subsequent class can rely on the code of the prior ones, as for example, in “real” and “imaginary” parts:

```
inline double real( const complex& z)
{
      return z.re;
}

inline double imag( const complex& z)
{
      return z.im;
}
```

Even though for the quaternions, there are two additional “imaginary” flavors:

```
inline double real( const quatern& q)
{
      return real( q.re );
}

inline double imag( const quatern& q)
{
    return imag( q.re );
}

inline double jmag( const quatern& q)
```

```
{
    return real( q.im );
}

inline double kmag( const quatern& q)
{
      return imag( q.im );
}
```

You get the point. For the octonions, there are 7 “imaginary” flavors, best illustrated by the format for printing out the contents of a given octonion:

```
ostream& operator<<(ostream& s, octon& o)
{
      s <<   "["   << real(o.re) << "," << imag(o.re) << ","
                   << jmag(o.re) << "," << kmag(o.re) << ","
                   << real(o.im) << "," << imag(o.im) << ","
                   << jmag(o.im) << "," << kmag(o.im) << "]\n";
      return s;
}
```

This works because `o.re` and `o.im` are quaternions.

All three classes contain a conjugate function, which simply negates the “imaginary” parts. The scheme is:

```
inline complex conj( const complex& z)
{
      return complex( z.re, -z.im );
}

inline quatern conj( const quatern& q)
{
      return quatern( real(q), -imag(q), -jmag(q), -kmag(q));
}

inline octon conj( const octon& o)
{
      return octon( real(o.re), -imag(o.re), -jmag(o.re), -kmag(o.re),
                   -real(o.im), -imag(o.im), -jmag(o.im), -kmag(o.im));
}
```

All three classes contain a norm function, which is the product of a number with its conjugate; but we implement it as a sum of squares:

```
inline double norm( const complex& z)
{
      return z.re*z.re + z.im*z.im;
}

inline double norm( const quatern& q)
{
      return norm( q.re ) + norm( q.im );
}
```

```
inline double norm( const octon& o)
{
      return norm( o.re ) + norm( o.im );
}
```

Some authors define the norm as the square-root of the function shown here; but then they need to square it in order to use it. Consider, for example, the inverse of a quaternion:

```
inline quatern inv( const quatern& q)
{
      return (conj(q) /= norm(q));
}
```

and an octonion:

```
inline octon inv(const octon& o)
{
      return (conj(o) /= norm(o));
}
```

We stop at octonions because the higher-dimensioned sedonions (16) and trigintaduonions (32) do not possess inverses.

The formula for multiplying two complex numbers is familiar:

```
inline complex operator*( const complex& z1, const complex& z2)
{
      return complex( z1.re*z2.re - z1.im*z2.im,
                              z1.re*z2.im + z1.im*z2.re );
}
```

Quaternion multiplication is more complicated but exploits the fact that a quaternion is a pair of complex numbers:

```
inline quatern operator*( const quatern& q1, const quatern& q2)
{
      return quatern( q1.re*q2.re - q1.im*conj(q2.im),
                              q1.re*q2.im + q1.im*conj(q2.re));
}
```

You will have guessed that octonion multiplication depends in turn upon quaternion multiplication:

```
inline octon operator*( const octon& o1, const octon& o2)
{
      return octon( o1.re*o2.re - conj(o2.im)*o1.im,
                           o2.im*o1.re + o1.im*conj(o2.re));
}
```

Multiplication of quaternions does not commute: in general, the product depends upon the order of the two factors. Octonion multiplication likewise does not commute, and furthermore it is nonassociative: $a\times(b\times c)\neq(a\times b)\times c$.

A type of multiplication called a cross-product can be defined for octonions. We denote this operation

with a caret (^) symbol:

```
inline octon operator^(const octon& o1, const octon& o2)
{
      return ( o1*o2 - o2*o1 ) *= 0.5;
}
```

This simple expression makes it obvious that reversing the order of the operands reverses the sign of the result.

As a check on all this machinery, I computed the cross-product multiplication table shown below. The nonzero entries are identical to the corresponding entries in the octonionic multiplication table shown in Ref. 1. The table below is, of course, perfectly antisymmetrical.

Cross-Product Multiplication Table

$T = L \wedge R$

$\downarrow L \backslash R \rightarrow$	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7
e_0	0	0	0	0	0	0	0	0
e_1	0	0	e_3	$-e_2$	e_5	$-e_4$	$-e_7$	e_6
e_2	0	$-e_3$	0	e_1	e_6	e_7	$-e_4$	$-e_5$
e_3	0	e_2	$-e_1$	0	e_7	$-e_6$	e_5	$-e_4$
e_4	0	$-e_5$	$-e_6$	$-e_7$	0	e_1	e_2	e_3
e_5	0	e_4	$-e_7$	e_6	$-e_1$	0	$-e_3$	e_2
e_6	0	e_7	e_4	$-e_5$	$-e_2$	e_3	0	$-e_1$
e_7	0	$-e_6$	e_5	e_4	$-e_3$	$-e_2$	e_1	0

An octonion with zero real part represents a vector in seven-dimensional space. The cross-product of two such octonions represents a vector perpendicular to both of them and proportional to the sine of the angle between them. So for example, this code:

```
double ang = atan2(4.,3.);
octon R(0.,0.,0.,0.,0.,sin(ang),cos(ang),0.);  cout << R;
octon L(0.,0.,0.,0.,0.,cos(ang),-sin(ang),0.); cout << L;
octon T = L^R;                                 cout << T;
```

yields:

```
[0,0,0,0,0,0.8,0.6,0]
[0,0,0,0,0,0.6,-0.8,0]
[0,0,0,-1,0,0,0,0]
```

R and L represent vectors of length 1 perpendicular to each other in the (e_5, e_6) plane. The result T is a vector of length 1 in the $-e_3$ direction. The reader can verify this result algebraically, referring to the above Table.

Bibliography

1. Anonymous, https://en.wikipedia.org/wiki/Octonion

2. Berryhill, J. R.., C++ Scientific Programming, Wiley-Interscience, 2001.

3. Conway, J. H., & Smith, D. A., On Quaternions and Octonions, CRC Press, 2003.

License