On adding quaternions to Scheme

Download

Dorai Sitaram

The numbers supported by the Scheme programming language form a tower, where the set of numbers at each level is a subset of the set at the level above. The levels are, respectively: integers, rationals, reals,[1] and complex numbers. This article investigates the potential extension of Scheme's numeric tower upward to include *quaternions* [3], a set of hypercomplex numbers that properly includes the complex numbers, and by extension, all the numbers below.

1 Quaternions

A quaternion q is a 4-tuple (w,x,y,z), where w, x, y, and z are real. To give it the aspect of a number, it is written $w + ix + jy + kz$, where i, j, and k are *imaginary units* that satisfy the conditions $ii = jj = kk = -1$, $ij = -ji = k$, $jk = -kj = i$, and $ki = -ik = j$.[2] Clearly, multiplication on quaternions does not commute. w is q's real (or scalar) part, and x, y, and z are its i-, j-, and k-part respectively. $ix + jy + kz$ is q's vector part. A quaternion with zero j- and k-parts is an ordinary complex number, and if, further, its i-part is also zero, it is just a real number. A quaternion with zero real part is called a vector quaternion or, simply, vector.[3]

Notation: We will use q_n (where n is either an integer or absent) interchangeably with $w_n + ix_n + jy_n + kz_n$. q_n's vector part $ix_n + jy_n + kz_n$ is also written v_n and its unit vector (see below) is written u_n. `E => e` means that the Scheme expression E evaluates to a Scheme value e. In order to prevent confusion, we add a subscript q to the name of a Scheme procedure that has been made quaternion-aware. For example, `exp` is the exponential procedure described in R5RS [4]; whereas $\texttt{exp}_q$ is the same procedure that has been extended to accept a quaternion argument.

2 Lexical representation of quaternions

Recall that Scheme complex numbers that are not also real are represented in *rectangular* (or cartesian) notation as `a+bi` or `a-bi`, where a is optional (for numbers that are purely imaginary) or real, and b is a non-negative real. Examples:

```
+i
-i
+2i
-2i
3+4i
```

(The sign is never optional, eg, i must be represented as `+i` rather than `i`, which latter is parsed as an identifier.) We can naturally extend this to quaternions: thus q above is written

`w+xi+yj+zk`. The pluses can be minuses, and any part that is zero may be omitted, with the proviso that a vector quaternion must always start with a sign or an explicit zero real part, in order to prevent being mistaken for an identifier. Examples of some Scheme quaternions that are not also complex numbers:

```
+j
+3k
-2+3i-3k
-6i-6k
3-4i+5j+6k
```

3 Polar notation

Scheme also permits users to enter complex numbers in a notation that uses *polar* coordinates, viz, μ`@`θ, where μ and θ are real. A corresponding notation for quaternions is also possible. For example, $q = w + ix + jy + kz$ has the polar (or spherical or curvilinear) coordinates μ, θ, ϕ, ψ, which are related to the rectangular coordinates as follows:[4]

$$\mu = \sqrt{w^2 + x^2 + y^2 + z^2}$$
$$\theta = \tan^{-1}(\sqrt{x^2 + y^2 + z^2}/w)$$
$$\phi = \tan^{-1}(\sqrt{y^2 + z^2}/x)$$
$$\psi = \tan^{-1}(z/y)$$

μ, θ, ϕ, ψ are respectively the *magnitude*, *angle*, *colatitude*, and *longitude* of q.[5] Extending Scheme's notation for complex numbers, q may be represented as μ `@` θ `%` ϕ `&` ψ. Any zero angle may be omitted along with its prefix. If the colatitude and longitude are both zero, the number is complex.

As for complex numbers, polar notation is used only to *enter* quaternions. Scheme outputs quaternion values exclusively in rectangular. Thus,

```
-3@4%-5&6
=>
1.960930862590836+0.6440287493492101i+2.0904336369493484-0.6083291310311992k
```

4 Building quaternions and taking them apart

The predicate `quaternion?` checks if its argument is a quaternion. Because quaternions now form the pinnacle of Scheme's numeric tower, `quaternion?` returns true (`#t`) for any number, and thus agrees with Scheme's `number?` predicate.

The Scheme complex-number procedures `real-part`, `imag-part`, `magnitude`, and `angle` are extended to accept quaternion arguments. They return, respectively, the real part, the *i*-part, the magnitude, and the angle, of their argument. To these are added four new related procedures: `jmag-part`, `kmag-part`, `colatitude`, and `longitude` return, respectively, the *j*-part, *k*-part, colatitude, and longitude of their argument.

The procedure `vector-part` returns the vector part of its argument.

The procedures `make-rectangular` and `make-polar` are extended to take two extra arguments in order to produce quaternions that aren't just complex numbers.

```
(make-rectangular q 3 4 5 6)
```

```
=>  3+4i+5j+6k

(make-polar_q -3 4 -5 6)
=>
1.960930862590836+0.6440287493492101i+2.0904336369493484-0.6083291310311992k
```

Note that the result *printed* by Scheme is the rectangular equivalent of the polar coordinates specified.

Every quaternion q has a special vector quaternion u called its *unit vector*, which is defined as

$$\frac{ix + jy + kz}{\sqrt{x^2 + y^2 + z^2}}$$

In Scheme,

```
(define unit-vector
  (lambda (q)
    (*_q (vector-part q)
       (/ (magnitude_q q)))))
```

The magnitude of a unit vector is of course unity.

5 Arithmetic

Scheme numerical procedures that work on complex numbers can be naturally extended to accept quaternion arguments.[6]

The procedures `=`, `+`, `-` extend in the expected manner for quaternion arguments -- they all operate on the parts severally.

```
(=_q 3+4j 3+4j)       => #t
(=_q 3+4j 3+4j 3+4k) => #f
(+_q 3+4j 3+4j)       => 6+8j
(+_q 3+4j 3+4j 3+4k) => 9+8j+4k
(-_q 4+5j 2-3k)       => 2+8k
(-_q 2+3j-4k)         => -2-3j+4k
```

The multiplication procedure $*_q$ (using the identities $ii = jj = kk = ijk = -1$) can be implemented as follows:

$$\begin{aligned} q_1 q_2 = \quad & (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) \\ & + (w_1 x_2 + x_1 w_2 + y_1 z_2 - y_2 z_1) i \\ & + (w_1 y_2 + y_1 w_2 + z_1 x_2 - z_2 x_1) j \\ & + (w_1 z_2 + z_1 w_2 + x_1 y_2 - x_2 y_1) k \end{aligned}$$

Example:

```
(*_q 4-3i+2j+6k 4+5i-7j+9k)
=>  -9+68i+37j+71k
```

Generally, multiplication is not commutative. However if two quaternions have the same *unit vector* disregarding sign, they do commute on multiplication, and furthermore, their product has the same unit vector. The square of a unit vector (its product with itself) is -1.

If q_1, q_2 are vector quaternions, ie, $w_1 = w_2 = 0$, then the real part of $q_1\ q_2$ is the negative of the *dot product* $q_1 \cdot q_2$ and the vector part of $q_1\ q_2$ is the *cross product* $q_1 \times q_2$.

```
(define dot-product
  (lambda (q1 q2)
    (if (not (and (= (real-part_q q1) 0)
                  (= (real-part_q q2) 0)))
        (error 'dot-product "arguments must be vector quaternions")
        (- (real-part_q (*_q q1 q2))))))

(define cross-product
  (lambda (q1 q2)
    (if (not (and (= (real-part_q q1) 0)
                  (= (real-part_q q2) 0)))
        (error 'cross-product "arguments must be vector quaternions")
        (vector-part (*_q q1 q2)))))
```

If the imaginary parts of q_1 and q_2 are severally the negatives of each other, then the product $q_1 q_2$ is real. Such q_1, q_2 are termed *conjugates* of each other. They have the same magnitude, and their product is the square of that magnitude.

```
(define conjugate
  (lambda (q)
    (make-rectangular_q
      (real-part_q q)
      (- (imag-part_q q))
      (- (jmag-part q))
      (- (kmag-part q)))))
```

We can use fact (1) to implement the *reciprocal* of a quaternion. Multiply both numerator and denominator of $1/q$ by q's conjugate. Unary `/` is the Scheme procedure for reciprocals. Example:

```
(/_q 8+3i+4j+5k)
=>
0.07017543859649122-0.02631578947368421i-0.03508771929824561j-0.043859649122807015k
```

Multiplication by the reciprocal leads to a definition of division. Dividing q_1 by q_2 is viewed as multiplying q_1 by q_2^{-1}. The noncommutativity of multiplication allows a choice between putting the reciprocal before or after q_1. Thus division from the left and from the right are different.

In keeping with the visual layout of Scheme's `/` procedure, we can define `(/_q q1 q2 ...)` with `(*_q q1 (/_q q2) ...)`, but to prevent confusion, it is best to always use explicit reciprocals (unary `/_q`) and multiplication (`*_q`).

6 Exponential and logarithmic functions

As for complex numbers, we use the Maclaurin series to define quaternion extensions of the transcendental functions. Thus, for quaternion $q = w + uv$, where uq is its unit vector and w and v are real, we can derive

$$e^q = e^{w+uv} = e^w(\cos v + u \sin v)$$

u functions much the same as i for complex numbers. The difference from the complex case is that u depends on (ie, varies with) q, whereas i is the same for all complex z.

In Scheme,

```
(define exp_q
```

```
  (lambda (q)
    (let ((w (real-part_q q))
          (u (unit-vector q))
          (v (magnitude_q (vector-part q))))
      (*_q (exp w)
           (+_q (cos v) (*_q u (sn v)))))))
```

Note that the unit vectors of q and e^q are the same, disregarding sign. The inverse of of the exponential function is the natural logarithm,[7] and is given by:

$$\log(w + uv) = \frac{1}{2}\log(w^2 + v^2) + u\tan^{-1}(v/w)$$

(where the log on the right only need be defined for positive reals). To make the log function single-valued, we choose $\tan^{-1}(v/w)$ between $-\pi$ adn π. In Scheme,

```
(define log_q
  (lambda (q)
    (let ((w (real-part_q q))
          (u (unit-vector q))
          (v (magnitude_q (vector-part q))))
      (+_q (*_q 1/2 (log (magnitude_q q)))
           (*_q u (atan v w))))))
```

We can use `log_q` to define `expt_q`, since:

$$q_1^{q_2} = (e^{\log q_1})^{q_2} = e^{(\log q_1)q_2}$$

In Scheme,

```
(define expt_q
  (lambda (q1 q2)
    (exp_q (*_q (log_q q1) q2))))
```

The `sqrt_q` function then is:

```
(define sqrt_q
  (lambda (q)
    (expt_q q 1/2)))
```

7 Trigonometric functions

For sin q and cos q, we have (using the Maclaurin's series for sinh and cosh):

$$\sin(w + uv) = \sin w \cosh v + u\cos w \sinh v$$
$$\cos(w + uv) = \cos w \cosh v - u\sin w \sinh v$$

In Scheme,[8]

```
(define sin_q
  (lambda (q)
    (let ((w (real-part_q q))
          (u (unit-vector q))
          (v (magnitude_q (vector-part q))))
      (+_q (*_q (sin w) (cosh v))
           (*_q u (cos w) (sinh v))))))
```

```
(define cos_q
  (lambda (q)
    (let ((w (real-part_q q))
          (u (unit-vector q))
          (v (magnitude_q (vector-part q))))
      (-_q (*_q (cos w) (cosh v))
         (*_q u (sin w) (sinh v))))))
```

tan q can be defined as (sin q)/(cos q). Since sin q, and cos q have the same unit vector as q, the direction of the division doesn't matter.

8 Inverse trigonometric functions

The following are derived using Maclaurin's and straightforward quadratic solving:

$$\sin^{-1} q = -u \log(uq + \sqrt{1 - q^2})$$
$$\cos^{-1} q = -u \log(q + \sqrt{q^2 - 1})$$
$$\tan^{-1} q = \frac{u}{2} \log \frac{u + q}{u - q}$$

In Scheme,

```
(define asin_q
  (lambda (q)
    (let ((u (unit-vector q)))
      (-_q (*_q u (log_q (+_q (*_q u q)
                              (sqrt_q (-_q 1 (*_q q q))))))))))

(define acos_q
  (lambda (q)
    (let ((u (unit-vector q)))
      (-_q (*_q u (log_q (+_q q
                              (sqrt_q (-_q (*_q q q) 1))))))))

(define atan_q
  (lambda (q)
    (let ((u (unit-vector q)))
      (*_q 1/2 u (log_q (*_q (+_q u q) (/_q (-_q u q))))))))
```

9 Signatures

The following collects the signatures of all the quaternion procedures proposed above. The types *number* and *quaternion* are synonymous. I shall use the name *quaternion* merely to emphasize that a function result can be a quaternion. Arguments enclosed in brackets are optional.

`quaternion?` : *any* → *boolean*

`real-part_q` : *number* → *real*
`imag-part_q` : *number* → *real*
`jmag-part` : *number* → *real*
`kmag-part` : *number* → *real*

`vector-part` : *number* → *quaternion*
`unit-vector` : *number* → *quaternion*

`magnitude`$_q$: *number* → *real*
`angle`$_q$: *number* → *real*
`colatitude` : *number* → *real*
`longitude` : *number* → *real*

`make-rectangular`$_q$: *real* × *real* [× *real* × *real*] → *quaternion*
`make-polar`$_q$: *real* × *real* [× *real* × *real*] → *quaternion*

`+`$_q$: *number** → *quaternion*
`-`$_q$: *number*+ → *quaternion*
`*`$_q$: *number** → *quaternion*
`/`$_q$: *number*+ → *quaternion*

`dot-product` : *number* × *number* → *real*
`cross-product` : *number* × *number* → *quaternion*
`conjugate` : *number* → *quaternion*

`exp`$_q$: *number* → *quaternion*
`log`$_q$: *number* → *quaternion*
`expt`$_q$: *number* × *number* → *quaternion*
`sqrt`$_q$: *number* → *quaternion*

`sin`$_q$: *number* → *quaternion*
`cos`$_q$: *number* → *quaternion*
`tan`$_q$: *number* → *quaternion*

`asin`$_q$: *number* → *quaternion*
`acos`$_q$: *number* → *quaternion*
`atan`$_q$: *number* → *quaternion*
`atan` : *real* × *real* → *real* (two arguments)

10 References

[1] A D Aleksandrov, A N Kolmogorov, and M A Lavrentev, *Mathematics: Its Content, Methods and Meaning*, Dover Publications, 1963.

[2] Ruel V Churchill, *Complex Variables and Applications*, McGraw-Hill, 2nd ed, 1960.

[3] William Rowan Hamilton, *On Quaternions; or on a new System of Imaginaries in Algebra*, Philosophical Magazine, 1844-1850.

[4] Richard Kelsey, William Clinger, and Jonathan Rees (eds), *Revised^5 Report on the Algorithmic Language Scheme (``R5RS'')*, 1998.

[5] Cornelius Lanczos, *The Variational Principles of Mechanics*, Dover Publications, 4th ed, 1986.

[6] Gerald Jay Sussman and Jack Wisdom with Meinhard E Meyer, *Structure and Intepretation of Classical Mechanics*, MIT Press, 2001.

[7] Peter Guthrie Tait, *Quaternions*, Encyclopedia Britannica, 9th ed, 1886, vol XX, pp 160-164.

---

[1] In practice, an implementation does not differentiate between rationals and reals.

[2] Invoking the associativity of multiplication, these conditions can be more succinctly stated as $ii = jj = kk = ijk = -1$.

[3] These are indeed the vectors familiarly used in physics to represent 3-dimensional quantities. They are not be confused with Scheme's one-dimensional array data type called `vector`.

[4] The rectangular coordinates can be deduced from the polar as follows:

$$\begin{aligned} w &= \mu \cos\theta \\ x &= \mu \sin\theta \cos\phi \\ y &= \mu \sin\theta \sin\phi \cos\psi \\ z &= \mu \sin\theta \sin\phi \sin\psi \end{aligned}$$

[5] The more usual terms for *magnitude* and *angle* are *modulus* and *amplitude* respectively, but we wish to remain compatible with Scheme's already established names for complex numbers.

[6] Procedures like `<`, `>`, `<=`, `>=`, `positive?`, `negative?`, `min`, `max`, `floor`, `ceiling`, `truncate`, `round`, `rationalize` two-argument `atan`, `make-rectangular` and `make-polar` are not defined for quaternions, but they are not defined for complex numbers either.

[7] Following Scheme practice, we use log to mean $\log_e$ rather than $\log_{10}$.

[8] `sinh` and `cosh` for reals are not part of the Scheme standard [4] but they can be readily defined using the identities

$$\begin{aligned} \sinh x &= (e^x - e^{-x})/2 \\ \cosh x &= (e^x + e^{-x})/2 \end{aligned}$$

Last modified: Fri, Nov 23, 2001, 1:01 pm US/Eastern
HTML conversion by TeX2page 4p4k6