# Hypercomplex Numbers in APL

**Bob Smith**
**Sudley Place Software**
**Originally Written**
**14 Sep 2015**
**Updated**
**11 Apr 2018**

"There are exactly four normed division algebras: the real numbers ($\mathbb{R}$), complex numbers ($\mathbb{C}$), quaternions ($\mathbb{H}$), and octonions ($\mathbb{O}$). The real numbers are the dependable breadwinner of the family, the complete ordered field we all rely on. The complex numbers are a slightly flashier but still respectable younger brother: not ordered, but algebraically complete. The quaternions, being noncommutative, are the eccentric cousin who is shunned at important family gatherings. But the octonions are the crazy old uncle nobody lets out of the attic: they are *nonassociative*."

John Baez, The Octonions[1]

# Introduction

Three new internal number types are made available to APL users to round out the set of four starting with the Real numbers. Complex numbers have been implemented in APL interpreters for over thirty years, but this is the first time for Quaternions and Octonions in an APL interpreter (but not in an APL compiler[2]). Also as a first, these new number types may have any one of four different types of coefficients as opposed to the typical single coefficient type of fixed

precision floating point.

Overall, this feature adds twelve new datatypes to the language.

Primitive functions are extended in a natural way to accommodate the new numbers. Four new primitive functions, one new system function, and one new system variable are defined.

# Nomenclature

**CHO:** This acronym references the collection of Complex ($\mathbb{C}$), Quaternion ($\mathbb{H}$), and Octonion ($\mathbb{O}$) numbers. You might expect it to be abbreviated CQO, but by the time Quaternions came along ($1843^5$), Q was already used to refer to the Rational numbers. You might think of using R to refer to Rationals, but instead Q was chosen (for Quotients) because R was already used to refer to Real numbers. H was chosen to honor the discoverer of Quaternions, Sir William Rowan Hamilton. I'm glad we cleared that up.

**Dimension vs. dimension:** The term dimension in APL refers to an array coordinate; first dimension, last dimension, etc. That same term is also used by mathematicians concerning CHO numbers to indicate the number of coordinates of the number, Real numbers are 1-dimensional, Complex numbers 2-dimensional, Quaternions 4-dimensional, etc.

I use the same word in different contexts, and I expect that in nearly every case, the context in which the word is used will be obvious and will clarify which meaning is intended.

**Bi-quadrants and bi-octants:** Complex numbers are 2-dimensional and are graphed in the plane with its two axes dividing the plane into four ($=2^2$) quadrants. Quaternions are 4-dimensional and divide space

into 16 (=$2^4$) “corners” which I'm calling bi-quadrants as in $2^4$. Similarly, I'm calling the 256 (=$2^8$) Octonion corners bi-octants.

# Complex Numbers

The input and output notation for Complex numbers uses infix notation with `i` or `J` as a separator. For example, `2i3` and `2J3` represent the same number. Following Iverson's lead in J, two additional input notations (for Complex numbers only) allow you to use polar form as in `1ad90` and `1ar2` which describe the radius and angle in either degrees (`ad`) or radians (`ar`). The display form of either input notation uses either `i` or `J` depending upon a program-wide User Preference. By default, an imaginary part with value zero is omitted on output, but a program-wide User Preference allows it to be shown.

The infix notation shown for CHO numbers is only one way to enter such numbers. Jacob Brickman[9] has suggested an alternative input and output notation for CHO numbers which uses postfix. For example, `0i1` may be written more succinctly as `1i`, and in general, Complex numbers are written as `1s2i`, Quaternions as `1s2i3j4k`, and Octonions as `1s2i3j4k5l6ij7jk8kl`. If any part is zero (e.g., the `0s` in `0s1i`), it may be elided.

The coefficients of a CHO number may be any one of four datatypes, as long as all the coefficients come from one of the following four datatypes:

- Fixed precision (64-bit) integer (a.k.a. Gaussian integers)
- Fixed precision (64-bit) floating point (the typical coefficient)
- Multiple precision integer/rational
- Multiple precision floating point

For example,

| | |
|---|---|
| `1i2` | Complex fixed precision integer |
| `1i2.5` | Complex fixed precision floating point |
| `1i5r2` | Complex multiple precision integer/rational |
| `1i2.5v` | Complex multiple precision floating point |

## Properties

- Addition and subtraction are performed element-wise (using mathematical notation for a moment)
  $p \leftarrow a_1 + b_1 i$
  $q \leftarrow a_2 + b_2 i$
  $p + q \leftrightarrow (a_1 + b_1) + (b_1 + b_2) i$
- Addition is commutative
  `p + q ↔ q + p`
- Addition is associative
  `p + q + r ↔ p + (q + r) ↔ (p + q) + r`
- Multiplication distributes over addition
  `p × q + r ↔ p × (q + r) ↔ (p × q) + (p × r)`
- Multiplication is associative
  `p × q × r ↔ p × (q × r) ↔ (p × q) × r`
- Multiplication is defined by Cayley-Dickson[3] as in if `a`, `b`, `c`, and `d` are Real numbers and `(a,b)` as well as `(c,d)` are treated as the real and imaginary parts of Complex numbers, then
  `(a,b) × (c,d) ↔ (a×c - d×b, d×a + b×c)`
- Multiplication is commutative
  `p × q ↔ q × p`
- `0 ↔ 0i0` is the identity element for addition.
- `1 ↔ 1i0` is the identity element for multiplication.
- The Conjugate of a Complex number negates the imaginary part, as in `+1i2 ↔ 1i¯2`, which means that conjugate is self-

inverse, i.e., `++q ↔ q`. Expanding the above notation for multiplying two Complex numbers as per Cayley-Dickson, it's easy to see that the product of a number with its conjugate (`q×+q`) is a real number.

- The quotient of two Complex numbers `p` and `q` where `q≠0` is achieved by multiplying the numerator and denominator both by the conjugate of the denominator as in `(p×+q)÷q×+q` where `q×+q` is a real number.

# Quaternion Numbers

The input and output notation for Quaternions uses infix notation with `i`, `j`, and `k` as separators just as `i` or `J` is used in complex numbers. For example. `1i2j3k4`. One or more imaginary parts may be elided on input if zero, as in `1j3 ↔ 1i0j3k0`. By default, imaginary parts with value zero are omitted on output, but a program-wide User Preference allows them to be shown.

As with Complex numbers, the coefficients of Quaternions may be any one of the same four datatypes.

## Properties

- Addition and subtraction are performed element-wise (using mathematical notation for a moment)
  $p \leftarrow a_1+b_1 i+c_1 j+d_1 k$
  $q \leftarrow a_2+b_2 i+c_2 j+d_2 k$
  $p + q \leftrightarrow (a_1+b_1) + (b_1+b_2) i + (c_1+c_2) j + (d_1+d_2) k$
- Addition is commutative
  $p + q \leftrightarrow q + p$
- Addition is associative
  $p + q + r \leftrightarrow p + (q + r) \leftrightarrow (p + q) + r$

- Multiplication distributes over addition
  `p × q + r ↔ p × (q + r) ↔ (p × q) + (p × r)`
- Multiplication is associative
  `p × q × r ↔ p × (q × r) ↔ (p × q) × r`
- Multiplication is defined by Cayley-Dickson[3] as in if `a`, `b`, `c`, and `d` are Complex numbers and `(a,b)` as well as `(c,d)` represent Quaternions, then
  `(a,b) × (c,d) ↔ (a×c - d´×b, d×a + b×c´)`
  where `d´` and `c´` are the Conjugates of `d` and `c` respectively.
- **N.B.** in general, multiplication is **NOT** commutative, that is
  `p × q ↚↛ q × p`
- `0 ↔ 0i0j0k0` is the identity element for addition.
- `1 ↔ 1i0j0k0` is the identity element for multiplication.
- The Conjugate of a Quaternion negates all of the imaginary parts, as in `+1i2j3k4 ↔ 1i¯2j¯3k¯4`, which means that conjugate is self-inverse, i.e., `++q ↔ q`. Moreover, like the inverse of the product of matrices, the conjugate of a product is the product of the conjugates in reverse order, i.e., `+p×q ↔ (+q)×+p`. Consequently, the product of a Quaternion with its conjugate (`q×+q`) is a real number because the conjugate of that product `+q×+q ↔ (++q)×+q ↔ q×+q` is itself, which means that the imaginary part must be zero.
- The quotient (a.k.a. right quotient) of two Quaternions `p` and `q` where `q≠0` is achieved by multiplying the numerator and denominator both by the conjugate of the denominator as in `(p×+q)÷q×+q` where `q×+q` is a real number. Because multiplication of Quaternions is not commutative, there is another quotient (a.k.a. left quotient) defined as `((+q)×p)÷q×+q` and is, in general, different from the right quotient. Note that `q×+q`, and `(+q)×q` are the same real numbers as per the discussion above on Complex number conjugates.

## Applications

From the Wikipedia article[5] on quaternions, "..., quaternions are used in computer graphics, computer vision, robotics, control theory, signal processing, attitude control, physics, bioinformatics, molecular dynamics, computer simulations, and orbital mechanics. For example, it is common for the attitude-control systems of spacecraft to be commanded in terms of quaternions. Quaternions have received another boost from number theory because of their relationships with the quadratic forms".

# Octonion Numbers

The notation for Octonions uses infix notation with `i`, `j`, `k`, `l`, `i j`, `j k`, `k l` as separators. For example `1 i 2 j 3 k 4 l 5 i j 6 j k 7 k l 8`. As with Quaternions, one or more imaginary parts may be elided on input if zero, as in `1 j 3 k l 4` ↔ `1 i 0 j 3 k 0 l 0 i j 0 j k 0 k l 4`. By default, imaginary parts with value zero are omitted in output, but a program-wide User Preference allows them to be shown.

As with Complex numbers and Quaternions, the coefficients of Octonions may be any of the same four datatypes.

## Properties

- Addition and subtraction are performed element-wise (using mathematical notation for a moment)
  $p \leftarrow a_1 + b_1 i + c_1 j + d_1 k + e_1 l + f_1 ij + g_1 jk + h_1 kl$
  $q \leftarrow a_2 + b_2 i + c_2 j + d_2 k + e_2 l + f_2 ij + g_2 jk + h_2 kl$
  $p + q \leftrightarrow (a_1+a_2) + (b_1+b_2)i + (c_1+c_2)j + (d_1+d_2)k + (e_1+e_2)l + (f_1+f_2)ij + (g_1+g_2)jk + (h_1+h_2)kl$
- Addition is commutative

`p + q ↔ q + p`

- Addition is associative
  `p + q + r ↔ p + (q + r) ↔ (p + q) + r`
- Multiplication distributes over addition
  `p × q + r ↔ p × (q + r) ↔ (p × q) + (p × r)`
- **N.B.** multiplication is **NOT** associative, that is
  `p × q × r ↔ p × (q × r) ↚↛ (p × q) × r`
- Multiplication is defined by Cayley-Dickson[3] as in if `a`, `b`, `c`, and `d` are Quaternions and `(a,b)` as well as `(c,d)` represent Octonions, then
  `(a,b) × (c,d) ↔ (a×c − d´×b, d×a + b×c´)`
  where `d´` and `c´` are the Conjugates of `d` and `c` respectively.
- **N.B.** multiplication is **NOT** commutative, that is
  `p × q ↚↛ q × p`
- `0 ↔ 0i0j0k0l0ij0jk0kl0` is the identity element for addition.
- `1 ↔ 1i0j0k0l0ij0jk0kl0` is the identity element for multiplication.
- The Conjugate of an Octonion negates all of the imaginary parts, as in `+1i2j3k4l5ij6jk7kl8 ↔ 1i¯2j¯3k¯4l¯5ij¯6jk¯7kl¯8`, which means that conjugate is self-inverse, i.e., `++q ↔ q`. Moreover, like the inverse of the product of matrices, the conjugate of a product is the product of the conjugates in reverse order, i.e., `+p×q ↔ (+q)×+p`. The product of a number with its conjugate (`q×+q`) is a real number.
- The quotient (a.k.a. right quotient) of two Octonions `p` and `q` where `q≠0` is achieved by multiplying the numerator and denominator both by the conjugate of the denominator as in `(p×+q)÷q×+q` where `q×+q` is a real number. Because multiplication of Octonions is not commutative, there is another quotient (a.k.a. left quotient) defined as `((+q)×p)÷q×+q` and is, in general, different from the right quotient. Note that `q×+q`, and `(+q)×q` are the same real numbers as per the discussion above

on Complex number conjugates.

## Applications

From the Wikipedia article[4] on octonions, “Octonions are not as well known as the quaternions and complex numbers, which are much more widely studied and used. Despite this they have some interesting properties and are related to a number of exceptional structures in mathematics, among them the exceptional Lie groups. Additionally, octonions have applications in fields such as string theory, special relativity, and quantum logic”.

# Division Quotients

In order to deal with the choice of left or right quotients when dividing non-commutative numbers (i.e., Quaternions and Octonions), a new system variable ⎕LR is introduced. This variable may assume the value of `'r'` or `'l'` to indicate that the right (resp. left) quotient should be returned from division. The default value is `'r'`.

Correspondingly, the Variant operator has been extended to allow ⎕LR to be specified locally on selected primitive functions as well as all user defined functions and operators, anonymous functions, and derived functions.

For more details on this topic, see the paper “Hypercomplex Quotients in APL”[12].

# Data Conversion

At times it is convenient to be able to manipulate the individual coefficients of a CHO number. Rather than extend the left argument of

the circle function by six more (recall that `9○R` and `11○R` return the real and (first) imaginary part), I decided to define a system function `⎕DC` for that purpose. In its monadic form, the function splits out the coefficients of a CHO number into a new column dimension, preserving the datatype of the coefficients OR it converts a non-CHO argument into the corresponding CHO number using the values along the column coordinate as the coefficients (presuming the number of columns of `R` is 1, 2, 4, or 8). This duality w.r.t. the right argument makes this function self-inverse. For example,

```
      ⎕DC ⍳4
1i2j3k4
      ⎕DC 1i2j3k4
1 2 3 4
```

This conversion function has been superseded by the **Condense** and **Dilate** primitive functions (see below).

The dyadic version of this function converts the coefficients of a Real or CHO right argument to the type specified by the character scalar (`'ifrv'`) left argument. The CHO dimension is preserved from the right argument to the result, just the type of the coefficients is changed. For example,

```
      'r' ⎕DC 1.5i1 2.5i3.5
3r2i1 5r2i7r2
```

# Type Promotion

As is done everywhere in APL, datatypes are fungible, e.g., the reciprocal of an integer produces a fixed precision floating point number without bothering the end user who just want results. Similarly, CHO numbers promote in the same way as do the one-

dimensional numbers, `I → F`, `R → V`, etc. Moreover, the interpreter preserves the argument datatype whenever possible such as with Gaussian integers and rationals:

```
      ÷2i1r3
18r37i¯3r37
      2i1r3*2
35r9i4r3
```

# Dimension Demotion

At the moment, CHO numbers are not subject to type demotion, although, say, CHO float could demote to CHO integer. Instead, the dimension of a CHO number resulting from **certain functions** may be lowered if enough of the trailing imaginary parts are zero. For example, given a CHO floating point `R`, `1○R` returns an angle as a single floating point value. Without Dimension Demotion, it would return a result of the same CHO dimension as `R` with all imaginary coordinates zero. With Dimension Demotion, because the result is actually a real number, the result is demoted from multi-dimensional floating point to one-dimensional floating point, and similarly if `R` is a multi-dimensional multiple precision floating point number, the result of `1○R` is a one-dimensional multiple precision floating point number.

At the moment, the only function whose results are subject to dimension demotion is dyadic Circle (`L○R`).

# Integer Units

An integer unit of one of the CHO number systems is a value whose coefficients are integers and whose norm is `1`. Thus there are four integer units of Complex numbers, eight for Quaternions, and sixteen

for Octonions.

A compact functional form to generate integer units is as follows

```
uf←<∘(⊢,-)∘(∘.=⍨)∘⍳
```

or as an anonymous function without the operator glue

```
uf←{<(⊢,-)∘.=⍨⍳⍵}
      uf 1
1 ¯1
      uf 2
1 0i1 ¯1 0i¯1
      uf 4
1 0i1 0j1 0k1 ¯1 0i¯1 0j¯1 0k¯1
      uf 8
1 0i1 0j1 0k1 0l1 0ij1 0jk1 0kl1 ¯1 0i¯1 0j¯1 0k¯1
      0l¯1 0ij¯1 0jk¯1 0kl¯1
```

These values are useful for rotating a CHO by a multiple of 90º as is needed for GCD.

# Primitive Functions

Generally, the definitions of primitive functions follow the ones for Complex numbers with the obvious extension to 4- and 8-dimensional numbers. For an explanation of the Complex number definitions of the primitive functions below, see Eugene McDonnell's paper “Complex Numbers” SATN 40[6].

## Conjugate

The conjugate primitive `+R` negates all of the imaginary coordinates.

This has the effect of reflecting the number about the real axis, the only coordinate that doesn't change its sign. This means that conjugate is self-inverse, i.e., `++q ↔ q`. Because the conjugate of a real number is itself, a test for realness is `R≡+R`. The effect of multiplying a CHO number by its conjugate is to produce a real number which is the sum of the squares of its coefficients.

## Magnitude

The magnitude (a.k.a. norm) primitive `|R` calculates the Pythagorean distance from the origin to the point regardless of its dimension (`1`, `2`, `4`, or `8`). This means that it computes the square root of the sum of the squares of its coefficients. In other words, `|R ↔ √R×+R ↔ √+/(>R)*2`. For example,

```
|3i4 ↔ √+/3 4*2 ↔ √+/9 16 ↔ √25 ↔ 5
```

## Direction

The direction primitive `×R` divides the number by its magnitude, except for `×0` which is `0`. Essentially, this function maps the number to the point where the line between the origin and the number intersects the unit `1`-, `2`-, `4`-, or `8`-dimensional sphere, resulting in a number with magnitude `0` (for `×0`) or `1` (for all other numbers). In other words, for `R≠0`, `×R ↔ R÷|R`. For example,

```
×3i4 ↔ 3i4÷5 ↔ 0.6i0.8
```

## Floor, Ceiling, Residue and Encode

These functions on Complex Numbers are defined as in the paper “Complex Floor”[11] by Eugene McDonnell. For more details on this topic see the paper “Hypercomplex GCD in APL”[13].

## GCD and LCM

As discussed in Doug Forkes'[10] APL81 paper, the GCD function (`L∨R`) may be defined by the Euclidean algorithm in terms of the residue function and may use either the McDonnell or Hurwitz definition of residue. The LCM function (`L∧R`) is defined in terms of GCD as in `L∧R ↔ (L×R)÷L∨R ↔ L(×÷∨)R`. For more details on this topic see the paper "Hypercomplex GCD in APL"[13].

## Dimension

In order to detect the dimension of a CHO number, one could use `>R` and check the number of columns, but I thought it might be nice to do that more easily and more primitively. Instead, I co-opted the primitive function monadic equal (`=R`) to return the integer scalar (`1`, `2`, `4`, or `8`) which is the dimension of `R`.

## Squared Norm

The squared norm primitive returns the square of the magnitude function, but the squared norm primitive is sometimes useful in itself. To that end, `≠R` returns `R×+R` which is another way of calculating squared norm, a.k.a. the sum of the squares of the coefficients. One value of this primitive is that the datatype of the right argument is preserved, but reduced from the original CHO dimension (`1`, `2`, `4`, or `8`) to `1`. For example,

```
≠3j4 ↔ 25
```

## Dilate and Condense

One way to manage the coefficients of Hypercomplex numbers uses the left and right carets as monadic functions. The **Condense** function (<R) takes an array whose number of columns is 1, 2, 4, or 8 and creates a new array whose shape is ¯1↓⍴R and whose last coordinate is filled with Hypercomplex numbers of the appropriate dimension. For example,

```
      <2 4⍴⍳8
1i2j3k4 5i6j7k8
```

The **Dilate** function (>R) reverses the effect of the Condense function and creates a new array whose shape is (⍴R),=R, where =R returns an integer scalar of the Hypercomplex dimension (1, 2, 4, or 8) of R:

```
      ><2 4⍴⍳8
1 2 3 4
5 6 7 8
```

This feature was suggested by David A. Rabenhorst.

## Shriek Functions

These functions return results on Real, Complex, and Quaternion Fixed-Precision numbers and Real Multiple-Precision numbers only; otherwise, they signal a DOMAIN ERROR on all types of Octonions and a NONCE ERROR on Complex and Quaternion Multiple-Precision numbers.

Extending these functions to the remaining datatypes requires calculating the Complex-valued Eigenvalues and Eigenvectors of certain 2×2 or 4×4 Real non-symmetric matrices which in turn requires writing my own Multiple-Precision routines for calculating Eigenvalues and Eigenvectors.

# Choices

- The fact that Quaternions and Octonions are not commutative means that we must be careful when implementing definitions that include multiplying numbers. For example, the EAS defines `¯11○R ↔ 0J1×R`, which needs to be taken literally because if we switch the multiplication arguments as in `R×0J1` we get a different answer.
- All of the other definitions in the Extended APL Standard that involve multiplying numbers (I'm looking at you, circle functions), should be considered written in stone.
- Also, as multiplication of Octonions isn't associative, the problem there is even stickier. We must also be careful to write definitions that are unambiguous w.r.t. multiplicative associativity where Octonions are concerned.
- In general, many algorithms we take for granted need to be examined closely. While I've implemented these algorithms in a particular way, do not take them as how I think they should be done. More thought needs to go into these choices through discussion with other implementors such as Sam Sirlin whose APLc compiler project[2] already supports Quaternions and Octonions. As an obvious example, how would you write GCD and LCM algorithms if multiplication were neither commutative nor associative? See this paper[13] for one answer.
- As mentioned above, division in Quaternions and Octonions involves a choice: right or left quotient? That is, in the result of `p÷q` is the numerator `p×+q` or `(+q)×p` ? See this paper[12] for one answer.
- Multiplication of Quaternions and Octonions also involves a choice. There are two (isomorphic) choices for multiplication of Quaternions, boiling down to “Is `0i1×0j1` equal to `0k1` or

`0k¯1` ?” Octonions are much worse as there are 480 different (but also all isomorphic) multiplication tables. My approach was to define multiplication for both types to use the Cayley-Dickson[3] construction which made the choice (easy) for me.

# Still To Do

In no particular order,

- Factorial and combinations on Multiple Precision Hypercomplex numbers (perhaps using Lanczos's[7] or Spouge's[8] approximation to the gamma function, or computing the Factorial of certain Multiple Precision floating point matrices)
- Infinities
- `)IN` and `)OUT`

# Online Version

This paper is an ongoing effort and can be out-of-date the next day. To find the most recent version, goto http://sudleyplace.com/APL/ and look for the title of this paper on that page.

# Executable Version

The latest released version of the NARS2000 software may be found in http://www.nars2000.org/download/ in either 32- or 64-bit versions. This software runs natively under Microsoft Windows XP or later as well as any Linux or Mac OS version which supports Wine (32-bit only) which acts as a translation layer.

# References


1. http://math.ucr.edu/home/baez/octonions/octonions.html
2. http://home.earthlink.net/~swsirlin/aplcc.html
3. https://en.wikipedia.org/wiki/Cayley-Dickson_construction
4. https://en.wikipedia.org/wiki/Octonion
5. https://en.wikipedia.org/wiki/Quaternion
6. http://www.jsoftware.com/papers/satn40.htm
7. https://en.wikipedia.org/wiki/Lanczos_approximation
8. https://en.wikipedia.org/wiki/Spouge's_approximation
9. Personal communication, 10 August 2015
10. http://dl.acm.org/citation.cfm?id=805343
11. http://www.jsoftware.com/papers/eem/complexfloor1.htm
12. “Hypercomplex Quotients in APL” http://www.sudleyplace.com/APL/HyperComplex Quotients in APL.pdf
13. “Hypercomplex GCD in APL” http://www.sudleyplace.com/APL/HyperComplex GCD in APL.pdf