Hypercomplex Implementation in APL

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

Introduction

In this paper I describe the nitty gritty details of implementing Hypercomplex numbers in APL, typically a topic of interest to developers only.

Prerequisite Reading

The paper “Hypercomplex Notation in APL”[6] provides a summary of the notation used in this paper for Hypercomplex numbers, and the paper “Hypercomplex Numbers in APL”[7] provides an overview of Hypercomplex numbers in general.

Nomenclature

CHO: This acronym references the collection of Complex (ℂ), Quaternion (ℍ), and Octonion (𝕆) numbers. You might expect it to be abbreviated CQO, but by the time Quaternions came along (1843[1]), 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.

Algorithms

Algorithms on Real and Complex numbers are plentiful, and when coupled with various open source libraries yields:

- Fixed precision Real and Complex floating point support from GSL
- Multiple precision Complex support from MPC
- Multiple precision integer/rational support from MPIR
- Multiple precision floating point support from MPFR

Algorithms on Quaternions and Octonions are extremely rare. Yet, to my great surprise, I found a cache of Quaternion algorithms in Dave Barber's excellent Quaternion Calculator[2] written in Javascript. Not only were these algorithms easily translated into C, but they were trivially easy to generalize to Octonions – so trivial that in every case, changing a 4 to an 8 was all that was necessary. Moreover, if I needed a Complex number version of one of those algorithms, all I had to do was change a 4 to a 2.

For the irrational and transcendental functions we need fixed and multiple precision floating point algorithms which can then be parametrized with not only the CHO argument, but also the CHO dimension. In other words, for the irrational non-Real cases, we need only two versions of these algorithms: one using fixed precision floating point functions and one using multiple precision floating point functions.

These algorithms filled in all of the non-trivial circle functions along with exponentiation, power, square root, and log both natural and based for fixed precision floating point support for HO and multiple precision support for CHO. In practice, once you have powers of ***e*** and natural logarithms, almost all of the trigonometric functions can be defined in those terms, but having the algorithms already coded in Javascript makes the process much simpler.

Eventually, I found another excellent source, good old invaluable A&S (Abramowitz & Stegun[4]) sections 4.3.55-57, 4.4.37-39, 4.5.49-51 for the formulae "... in Terms of Real and Imaginary Parts" of the trigonometric functions. Little known facts: A&S was a consequence of the Mathematical Tables Project[3] of the WPA of President Franklin Roosevelt. A&S is considered by some to be the most cited book in the mathematical literature[5] with an estimated 40,000 citations.

Interestingly, algorithms on Complex numbers are not very general – those on Quaternions are best because the role of the imaginary coefficients is exposed much more so than it is in the corresponding Complex number algorithm. Usually, an algorithm for Complex numbers has had all of the generality boiled and optimized out of it so that it is no longer obvious how to generalize it to a higher dimensional number.

For example, here's an algorithm for natural log on Quaternions from Barber's Quaternion Calculator:

```
function logH (h, i, j, k)
{
    g = i*i + j*j + k*k;
    m = r_sqrt (h*h + g);
    g = r_sqrt (g);
    u = r_log (m);
    v = (g == 0) ? 0 : r_atan2 (g, h) / g;

    if ((h < 0) && (g == 0))
        r = Math.PI;
    else
        r = v*i;

    return (u, r, v*j, v*k);
}
```

As you can see, the `log` function takes in four numbers and returns four numbers. Also, it's trivial to see how to extend this algorithm to Octonions as well as contract it for Complex numbers because the role of the imaginary coefficients is made clear. The corresponding Complex number-only algorithm would have been simplified to the point where you would have no idea of how to extend it.

Here is an APL function based on the above algorithm to calculate natural logs on CHO numbers:

```
Z←log R;c g h m v
⍝ Log for Hypercomplex numbers
c←↕R              ⍝ Expand into its coefficients
h←1↑c             ⍝ The real coefficient
m←√+/c*2          ⍝ Magnitude of R
g←√+/(1↓c)*2      ⍝ Magnitude of imaginary parts
Z←⍟m              ⍝ Real part is always ⍟m
:if g=0           ⍝ Imaginary parts = 0
  :if h<0         ⍝ Real part < 0
    Z←Z,1p1       ⍝ Angle is +180 degrees
  :end
:else             ⍝ Imaginary parts ≠ 0
  v←(12○h+g×0j1)÷g ⍝ 12○ = phase = atan2
  Z←Z,v×1↓c
:end
Z←↑Z
```

The argument to `log` can be a scalar Real or CHO number with any of the four possible types of coefficient. Note that CHO numbers are handled as just a longer vector. The imaginary parts have equal weight and are mixed together in an obvious way. The `:if` … `:else` part of the control structure is there solely to handle principal values as they are normally exceptional cases.

Euler's Identity

An interesting implementation challenge was confirming Euler's Identity: `*○1j` ↔ `¯1`. As soon as I had exponentiation working, I tried that marvelous and elegant piece of code, as simple to state in APL as it is in mathematical notation. I typed it and was greeted with

something like `¯1i¯2.6123769112521 65E¯19` – not the elegant answer I was expecting.

The reason why took a while to ferret out. Eventually after checking my C code carefully, I went back to APL and typed the equivalent power series for exponentiation knowing that integer powers as a special case were already implemented:

```
      N←0..40
      R←(○1i)*N
      Q←R÷!N
      +/Q
¯1
```

This was even more puzzling – I was delighted to see that result, however, how could the APL code get the right answer, but the C code could only get close? After scratching my head for a while, I remembered an unpleasant and counter-intuitive truth about numerical analysis – **on floating point numbers, addition is not associative**. That is,

`a+(b+c) ←/→ (a+b)+c.`

My C code calculated the above quotient the same way as the APL code, but the sum was calculated differently – in particular, in a different order. My C code took the naïve approach and calculated each term from low-order exponents on up, accumulating it per iteration with no need for lengthy temporary storage. In other words, the values in `Q` were added iteratively left to right as `Q[1]`, `Q[1]+Q[2]`, `(Q[1]+Q[2])+Q[3]`, etc. However, the APL sum was calculated from right to left starting with `Q[39]+Q[40]`. In effect, the APL reduction added the high-order small value terms first, eventually working its way down to the low-order (but larger magnitude) terms.

The C code did just the opposite. The effect was that in the C code, the small value terms were ignored as they were overwhelmed by the large value terms by the time they were considered. That is, the C code actually was calculating

```
      +/ϕR÷!N
¯1i¯2.612376911252165E¯19
```

Once I changed the C code to store all of the intermediate results and then add them together from right to left, the expected answer from Euler's Identity finally displayed

```
      *○1i
¯1
```

Internal Functions

In order to simplify coding of the CHO algorithms, I defined numerous internal functions which could be used “inline” as it were – these proved to be a great time saver. For example, the fixed precision floating point CHO number implementation for `0○R ↔ √1-R*2` looks like this (after a bit of editing for display purposes):

```
/* 1 - R * 2 */
Tmp = SubHC##N##F_RE (HC##N##Con1,
                      MulHC##N##F_RE (R.HC##N##F,
                                      R.HC##N##F));
/* sqrt (1 - R * 2) */
lpMemRes = SqrtHCxF_RE (Tmp, N)##suf##;
```

This code appears in a macro which implements this algorithm for all CHO dimensions. As such, the ##N## represents the CHO dimension which is inserted as text during pre-processing of the macro. For

example, the name `MulHC##N##F_RE` expands to `MulHC2F_RE` for the Complex number version. The call to

```
SqrtHCxF_RE (Tmp, N)##suf##
```

is an example of where the dimension is an argument to the function rather than there being separate 2-, 4-, and 8- dimensional versions as is the case with `MulHC##N##F_RE`. Because that function handles all dimensions, it can't return a result specific to the input dimension, so it returns a result cast to the highest dimension (but filling in only as many dimensions as needed) and the caller then pares it down to the expected dimension via the suffix `##suf##` which may expand to `.partsLo.partsLo` (for Complex numbers) or `.partsLo` (for Quaternions) or empty (for Octonions) as a technique to cast the result to the correct dimension type.

Promote/Demote/Free Tables

One implementation technique I found to be invaluable was to put in the non-trivial effort to construct large tables (in effect, 21 by 21) of entries of functions and then to code each function. These tables are indexed row and column by the 21 possible storage datatypes used to promote as well as demote data from one type to another and are used throughout the CHO code. This not only made it easier to code functions sensitive to CHO numbers, but it simplified existing code which had been more verbose than necessary when doing type promotion, demotion, and free on non-CHO data.

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. https://en.wikipedia.org/wiki/Quaternion
2. http://tamivox.org/redbear/qtrn_calc/index.html
3. https://en.wikipedia.org/wiki/Mathematical_Tables_Project
4. http://www.cs.bham.ac.uk/~aps/research/projects/as/resources/AandS-letter-v1-2.pdf
5. https://archive.org/details/handbookofmathem1964abra
6. "Hypercomplex Notation in APL" http://www.sudleyplace.com/APL/HyperComplex Notation in APL.pdf
7. "Hypercomplex Numbers in APL" http://www.sudleyplace.com/APL/HyperComplex Numbers in APL.pdf