FUZZY RESIDUE

Eugene McDonnell
I. P. Sharp Associates
220 California Avenue, Suite 201
Palo Alto CA, USA 94306
415-327-1700

Abstract

Certain pairs of arguments to the residue function, as implemented on many APL systems, give results which make it seem as if the ordinary decimal relationships we remember from grade school no longer hold. As far as we can tell, it looks as if a given modulus should divide the right argument, but the implementation tells us it doesn't. A definition for a fuzzed residue function is proposed which resolves the difficulties users have complained of. However, certain points of continuing difficulty remain, where the limitations of machine arithmetic continue to defeat the attempt to model the real number system. The representation function is defined in terms of the residue function, and so is affected by the change in residue. The nature of this effect is also discussed in this paper.

The problem

Several authors [1, 2, 3, 4] have given examples where the results provided by APL's residue function, as implemented on many computers compatible with IBM's 370 architecture, are counterintuitive. They complain that sometimes the residue function fails to give zero as result when the modulus (left argument) clearly divides the right argument. Instead it gives either a number essentially equal to the modulus, or a very small number. For example:

```
      ⎕PP←16
      ⎕CT←1E¯13
      .2|1.4 1.6
0.1999999999999999 1.110223024625157E¯16
```


This result comes about as follows. The machine implementation of the residue function closely follows the definition (for non-zero left arguments):

$$RES{:}\omega-\alpha\times\lfloor\omega\div\alpha \qquad (1)$$

However, although we use numbers expressed in a decimal representation when we key in APL statements, on machines which follow the floating-point architecture of the IBM System/370, APL interpreters typically convert these decimal representations into hexadecimal representations. In other words, the numbers are represented in base 16 rather than base 10. The only decimal fractions which can be represented exactly are those which are also exact hexadecimal fractions. Thus .25, .5, and .75 can be represented exactly (since they may be represented as 4, 8, and 12 divided by 16), but .2, .4, and .6 cannot. Using the inexact hexadecimal representations of these decimal numbers gives rise to the counterintuitive results shown in the example above. 1.4÷.2 is slightly less than 7, in the inexact hexadecimal representation, so that taking the exact floor of this gives 6. Consequently 1.4-.2×6 gives a result just slightly less than .2; if only ten digits of precision are asked for (as is the default on many IBM-based APL's), this prints as 0.2. On the other hand, 1.6÷.2 is slightly more than 8, its floor is 8, and then .2×8 is very slightly less than 1.6, so 1.6-.2×8 is quite a small number. Both results "should" be zero, of course, since .2 divides both 1.4 and 1.6 exactly. But closest-possible hexadecimal arithmetic says otherwise.

Not all the blame must be put on the base-sixteen representation, however. Even if the machine had base ten, although every decimal number with few enough digits and with exponent confined to a suitable range could be exactly represented, there would still be many arithmetic operations whose results could not be accurately represented. For example, any time there was a division by a number having a factor relatively prime to ten the result would be inexact. Thus dividing by three would be inexact. The result of 3×÷3 would be not 1, but .99999 ... .

Fuzzed functions

In most APL systems all of the relational functions, and also membership, inverse indexing, floor, and ceiling tolerate inexactness in their arguments by a small amount, called by the first implementers "fuzz". This has now become a formal part of APL in the system variable representing "comparison tolerance". APL's fuzz attempts to conceal from the user the difference between the ideal real-number system and the practical, finite approximation to this found in all computer systems. APL's use of fuzz smooths over some of the rough edges of the number-approximating scheme, but it does so at the cost of losing certain identities. For example, if two numbers are equal, their difference should be zero. With fuzzy equality we may very well have two numbers equal, but their difference could be either a small negative or a small positive quantity instead.

Fuzz is used in two different ways. In the first place, it defines an interval about the larger in magnitude of the two arguments to a function, and if the other argument falls within that interval, the two arguments are said to be equal. This is the behavior for the relational functions, membership, and inverse indexing. For example,:

```
      ⎕CT←1E¯13
      A←2-1E¯14
      2=A
1
      A<2
0
      (2,A)⍳A
1
      1 2 3∈A
0 1 0
```

Secondly, it is used to determine whether a given floating-point number is close enough to an integer, where "close enough" means that the given number would compare equal to the nearest integer. This is done in functions requiring Boolean or integral values as arguments, as for example indexing, or the left arguments for the circular function or reshape. In these cases, reference is not made to comparison tolerance, but rather to a built-in fuzz which is nowhere explicit. It is also done in the case of floor and ceiling, using comparison tolerance. For example,

```
      ⌊1.99999999999999
2
```

The number, though less than 2, is "close enough" to 2. One identity we lose as a consequence of this is that when we take the difference of a number and its floor, we may get a negative number. The definition of fuzzy floor has had a long history in APL, and is still being discussed [5, 6, 7, 8, 9].

The relationship between floor and residue

It does not appear possible to define both the floor function and the residue function in closed form without circularity. That is, the floor function is defined as:

$$FLOOR:\omega-1|\omega \qquad (2)$$

and the residue function, as in definition (1), depends on the floor function. The expression $1|\omega$ is the fractional-part function, of which more later.

The definition (2) can be re-expressed to give a definition of a number as the sum of its integer part and its fractional part:

$$Z=(\lfloor Z)+(1|Z)$$

This identity can be generalized to arbitrary moduli:

$$Z=(M\times\lfloor Z\div M)+(M|Z) \qquad (3)$$

as shown in [10]. Users expect (perhaps naively) that a number can be decomposed accurately using the floor and residue functions. They are told, in fact, in standard APL texts that they will be able to do so [11]. It is desirable that these expectations not be thwarted.

In most APL's that run on computers having IBM's floating-point architecture, there is a lack of harmony between the floor and the residue function, arising from the fact that, while the floor function is fuzzed, the residue function is not. To show the discordance, consider the following:

```
      ⎕CT←1E¯13
      ⎕PP←16
      Z←4-1E¯14
      Z
3.99999999999999
      ⌊Z
4
      1|Z
0.99999999999999
      (⌊Z)+(1|Z)
4.99999999999999
```

The problem is that in the implementation of the residue function in accordance with definition (1), the floor of the quotient is computed exactly. In order to make the residue function harmonious with the floor function either the floor function should be computed exactly (without fuzz), or the residue function should be computed fuzzily. This paper discusses the latter alternative.

An important residue inequality and its complex extension

It might be useful at this point to give briefly the history of the residue function in APL, since this bears on one key point in the discussion of fuzzy residue.

The original documentation [12] for APL conflicted with the first implementation in regard to residue (and many other things, for that matter). It gave the familiar definition (1) for residue, but the implementation chose instead to use the magnitude of the modulus instead of the signed value. The second major APL document [13] reflected this use of the magnitude. This definition had several defects, but these did not become apparent until late in 1968 when I began studying the extension of APL to complex-number arguments. In fact, the question I was asked, by Larry Breed, was "How shall the residue function be extended to complex numbers?"

The defects in the definition at that time were first, the result was always non-negative; it was as if half the range of the function were denied. Second, the modulus zero was not a left identity element, even though it was claimed to be [12,13], since $0|X$ for negative X gave a domain error. Third, the useful inequality:

$$(0\leq(A|B)\div A) \quad \wedge \quad ((A|B)\div A)<1 \qquad (4)$$

as well as identity (3) was lost for negative A. Fourth, the function as defined couldn't be extended to the complex domain, for two reasons. There was no definition for the floor function of complex numbers, and the use of the magnitude function in the complex plane kept the residue of $W|Z$ for complex W and Z from being a Gaussian integer, destroying the notion of complete systems of residues.

I remedied the first defect by creating a definition for a complex floor function [14], and the second by proposing that the definition and the implementation of residue be changed to reflect the original definition (which had never been implemented). I proposed the change to residue in 1968, and by 1973 (when APLSV was announced) the new definition was made available to customers.

Two years after this, in 1975, the third major IBM APL language document [15] was written, and it stated, "if $A\neq 0$, then $A|B$ lies between A and zero (being permitted to equal zero but not A) and is equal to $B-N\times A$ for some integer N." It is this statement which several readers of early drafts of this paper said had to be given up if APL were extended to include complex numbers as data types, and thus couldn't be used in making arguments for a fuzzy residue. These readers were wrong, however, as I shall explain.

Let's take a look at the inequality (4) which lies behind the statement in [15]. Recall that it says $(A|B)\div A$ is greater than or equal to zero and less than one. This is another way of saying that the result is in the range of the fractional-part function (see Figure 1).

Figure 1. Fractional part interval

The definition I gave to complex floor was chosen to be compatible with the real floor definition. In particular, the fractional-part function $1|\omega$ extends faithfully (see Figure 2).

Just as, in the real case, we can say that the result of $1|\omega$ lies in the half-open interval $(0\ 1]$, in the complex case we can say that its result lies in the half-open area delimited as shown in Figure 2.

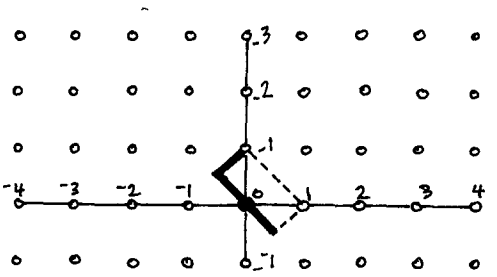

Figure 2. Fractional part area

We can go on to say that the result of $A|B$ lies in the half-open interval $(0\ A]$ formed by multiplying the fractional-part range by A (Figure 3). We shall call this the residue interval. Similarly, in the complex case the result of $A|B$ lies in the half-open rectangle formed by the multiplication of the fractional-part rectangle by A (Figure 4). We shall call this the residue area. Thus, in the real case, for positive A, the result is found in a residue interval extending to the right from zero, and for negative A, the result is found in a residue interval extending to the left from zero. This is the basis for the statement in [15]. When an APL language manual is written which accommodates complex numbers, the discussion of residue must be modified, but the basic idea remains the same.

What I wish you to retain from this discussion is the fact that the result of $A|B$ always lies in the residue interval for A, is never equal to A, and thus its magnitude is less than the magnitude of A.

Figure 3a. Residue interval for modulus 3

Figure 3b. Residue interval for modulus ¯3

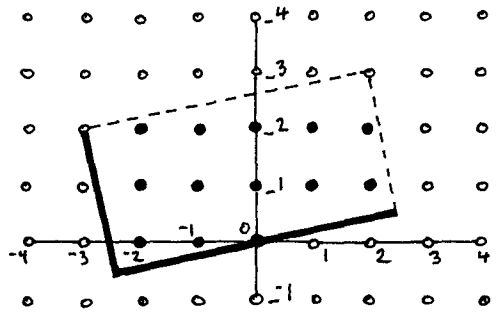

Figure 4. Residue area for modulus 2_3

A proposal for fuzzy residue

One possible definition for fuzzy residue is simply to use (1) with fuzzy rather than exact floor. This definition would, indeed, prevent results being equal to the modulus. For example, ⌊1.4÷2 would be 7 rather than 6, and thus .2|1.4 would be zero. However it would leave the result of .2|1.6 unchanged: this would still be a small number. Thus, if one were content to get a small number rather than zero as the result, this definition would satisfy. But since the purpose of this paper is to show how to make APL arithmetic correspond more closely to our school arithmetic, definition (1), with fuzzy floor, must be regarded as unsatisfactory.

Another candidate is:

```
FR2:ω-α×⌊S:S=⌊S←ω÷α:0
```

In this definition, the floor function and the equals function are fuzzy. It says that the standard definition should be used unless ω÷α is essentially an integer, in which case the result is zero. This is an appealing definition, and covers most cases, but fails if ω÷α is small enough (that is, if the modulus α is sufficiently much larger than ω), so that ⌊ω÷α is zero. In this case, S=⌊S will fail, because comparisons with zero are exact, and thus instead of the result of α|ω being zero, it will be ω. This may not seem wrong to you, but if the signs of α and ω are different, the result will have a different sign from the modulus, and this is not permitted: the result must lie in the residue interval for α. Presently, when α is very much larger than and opposite in sign to ω, the result is exactly α, since the exact floor of ω÷α is ¯1, and ω is lost by cancellation in performing ω-α×¯1, leaving us with α. It's hard to judge which is worse, the present situation or that which would occur with *FR2*.

The next candidate, suggested by Doug Forkes, is

```
FR3:ω-α×⌊S:(⌈S)=⌊S←ω÷α:0
```

Here the comparison is between the ceiling of the quotient and the floor of the quotient. If these are equal, the result will be zero, and if the quotient is very small, the ceiling and the floor will both be zero, the comparison will be true, and the result of α|ω will be zero. The identity that this definition preserves is that if *A*|*B* is zero, then so is *A*|-*B*, since *B* differs from -*B* only by a unit, and thus the numbers are associates, and associates are divisible by the same numbers. It appears to be the case that we have to accept zero for the result of, say, (16*15)|¯1 in order to keep (16*15)|¯1 from being outside the residue interval of 16*15.

Our final definition will be *FR3* extended to accommodate the case of zero as a modulus:

```
FR:ω-α×⌊S
  :(α=0)∨(⌈S)=⌊S←ω÷α+α=0
  :ω×α=0
```

This definition will revert to the present definition when ⎕CT is zero. Definition *FR*, together with the existing fuzzy floor definition, insures that a number will be the sum of its integer and fractional parts except in cases where the number is a small negative number such as -1E¯10+1E¯20. In this case, the floor of the number is ¯1, the fractional part is 1-1E¯10 (losing the 1E¯20 by cancellation), and the sum of the floor and the fractional part will not compare equal to the original number. In this case we accept defeat at the hands of the machine.

Encode

Since encode is defined in terms of residue, changing the definition of residue will have an effect on the way encode works. The definition of encode remains the same, but notice the difference between the present encode, which uses an unfuzzed residue, and the function FE, which uses the fuzzy residue function:

```
FE:((¯1↑α)FE⍉(ω-X)DIV ¯1↑α),X←(¯1↑α)FRω
  :0=ρα:⍳0

      10 10 10⊤99.999999999999
0 9 9.999999999999

      10 10 10 FE 99.999999999999
1 0 0
```

The DIV function is used instead of the primitive divide function in order to give the quotient zero for zero divided by zero, thus "stopping" the encode, as it should, at a result element position corresponding to a zero element in the left argument:

```
DIV:α÷ω:α=0:0
```

giving further reason for wanting the quotient of 0÷0 to be changed from the way

present APL systems compute it (which give one as the result) [16].

Acknowledgments

This work has benefitted from extensive discussions with Mike Jenkins of Queens University, Kingston, Ontario; Jim Brown and Larry Breed, of IBM; Rick Petkiewicz, of Northern Arizona University, Flagstaff, Arizona; Bob Bernecky, Doug Forkes, and Leigh Clayton, of I. P. Sharp Associates; and Paul Penfield, of MIT.

References

[1] "Third SEAS Working Committee's Meeting, Nogorduyk June 8 and 9, 1971", APL Quote Quad, 3, 2&3, p. 10, Oct. 1971

[2] Buscher, David J. and William M. Piper, letter to the editor, APL Quote Quad, 5, 1/3, Spring 1974

[3] Singer, David, letter to the editor, APL Quote Quad, 5, 4, Winter, 1974

[4] Watson, Don, "Question: 'Why does APL/360/370 Give the Following Results?'", APL Quote Quad, 5, 4, Winter, 1974

[5] Seeds, G. M., "Fuzzy floor and ceiling", APL Quote Quad 5, 4, Winter 1974

[6] Lathwell, R. H., "Comparison tolerance in APL", APL76 Conference Proceedings, ACM, New York, 1976

[7] Breed, L. M., "Definitions for fuzzy floor and ceiling", APL Quote Quad, 8, 3 (March 1978) 16-23

[8] Bernecky, R., and D. L. Forkes, "Comparison Tolerance", Sharp APL Technical Notes, SATN 23, I. P. Sharp Associates, Toronto, 1978

[9] Hagerty, P. E., "More on fuzzy floor and ceiling", APL Quote Quad, 8, 4 (June 1978) 20-24

[10] Iverson, K. E., A Programming Language, Wiley, N. Y., 1962, p. 12

[11] Gilman, L. E. and Allen J. Rose, APL/360, an Interactive Approach, Wiley, N. Y., 1970, p.23

[12] Falkoff, A. D. and K. E. Iverson, The APL Terminal System: Instructions for Operation, IBM Corp., Yorktown Heights, 1966

[13] Falkoff, A. D. and K. E. Iverson, APL\360: User's Manual, IBM Corp., Yorktown Heights, 1968

[14] McDonnell, E. E., "Complex Floor", APL Congress 73, North Holland Publishing Co., Amsterdam, 1973

[15] APL Language, publication GC26-3847, IBM Corporation, 1978

[16] McDonnell, E. E., "Zero divided by zero", APL76, ed. G. Truman Hunter, Association for Computing Machinery, Ottawa, 1976