Complex Floor Boundaries in APL

Bob Smith
Sudley Place Software

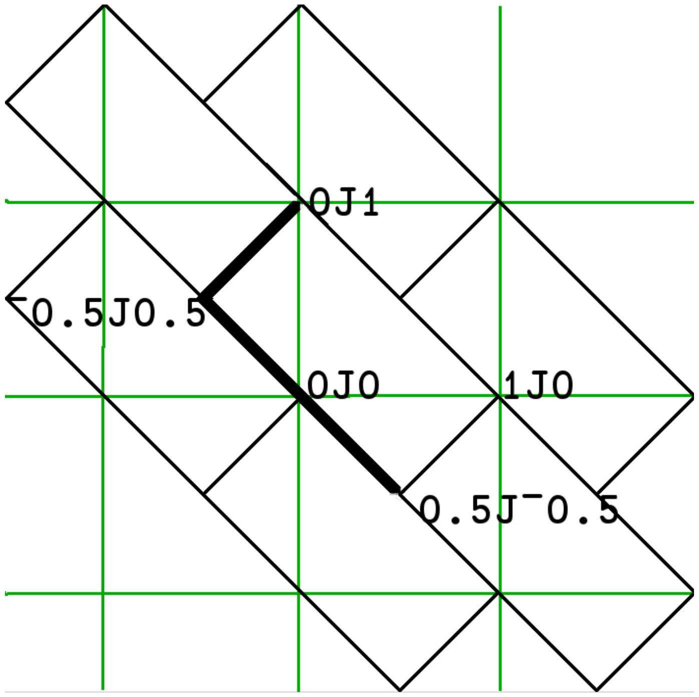
Originally Written
4 Nov 2015
Updated
19 Nov 2015

Introduction

Of the several definitions of Complex Floor, there is one from Gene McDonnell for which there are two different implementations in APL. This is of interest primarily because the two algorithms produce different results in certain edge conditions.

In order to compare the two versions of McDonnell's complex floor algorithm, one each from McDonnell[1] and Forkes[2] this paper provides the gory details of both an interval and precision analysis of the two in order to determine why they produce different results.

The prototypical element in the domain for McDonnell's definition of values whose Complex Floor is `0J0` is a rectangle between the points `0J1`, `1J0`, `0.5J¯0.5`, and `¯0.5J0.5`. This yields a shape of sides `√2` by `√÷2` at an angle of `45°` with the two short lines at the upper left and lower right and the two long lines at the upper right and lower left where the middle point of the lower left long line is at the origin `0J0`. This rectangle then tessellates the entire complex plane in the manner shown in the adjacent diagram.



McDonnell's definition is such that the Complex Floor of all of the **interior** points is `0J0`. The **boundary** points are divided up such that the values on the upper short line [`¯0.5J0.5`, `0J1`) and the lower long line [`¯0.5J0.5`, `0.5J¯0.5`) (shown in bold) all have a Complex Floor of `0J0`. The Complex Floor of all of the other boundary points is other than `0J0`. Thus, of the four corners, only `¯0.5J0.5` has a Complex Floor of `0J0`.

As McDonnell's definition satisfies the property of "Integer Translation", we need choose only one rectangle to analyze: the one whose floor is `0J0`. Because the floor of all of the interior values is 0J0, what remains to analyze is the boundary of the rectangle.

In the following interval analysis, the four boundary lines are parameterized by the variable t ranging from 0 to 1. These limiting values represent the starting and ending points of the line, respectively. Depending upon which of the four lines is being analyzed, the interval for t may include one or both of the endpoints. Regardless of this choice, all of the boundary points are included and analyzed.

McDonnell's Floor as per Forkes

```
    ∇ Z←MF R;A;B;A1;B1;T
[1]   ⍝ McDonnell's Complex Floor by Doug Forkes
[2]   A←9○R   ⍝ Real part
[3]   B←11○R  ⍝ Imaginary part
[4]   T←⌊A+B
[5]   B1←⌈0.5×T-1+A-B
[6]   A1←T-B1
[7]   Z←A1+0J1×B1
    ∇
```

	Upper Short Line t∊[0,1)	**Lower Short Line t∊[0,1)**
Parametrized Line	[¯0.5,0.5] + t × [0.5,0.5]	[0.5,¯0.5] + t × [0.5,0.5]
A	¯0.5 + t × 0.5	0.5 + t × 0.5
B	0.5 + t × 0.5	¯0.5 + t × 0.5
A+B	t	t
T←⌊A+B	0	0
A-B	¯1	1
1+A-B	0	2
T-1+A-B	0	¯2
0.5×T-1+A-B	0	¯1
Im←⌈0.5×T-1+A-B	0	¯1
Re←T-Im	0	1
Re+0J1×Im	0J0	1J¯1

	Upper Long Line t∊[0,1]	**Lower Long Line t∊[0,1)**
Parametrized Line	[0,1] + t × [1,¯1]	[¯0.5,0.5] + t × [1,¯1]
A	t	¯0.5 + t
B	1 - t	0.5 - t

A+B	1	0
T←⌊A+B	1	0
A-B	¯1 + 2 × t	¯1 + 2 × t
1+A-B	2 × t	2 × t
T-1+A-B	1 - 2 × t	¯2 × t
0.5×T-1+A-B	0.5 - t	-t
Im←⌈0.5×T-1+A-B	1 if t∊[0,0.5) 0 if t∊[0.5,1]	0
Re←T-Im	0 if t∊[0,0.5) 1 if t∊[0.5,1]	0
Re+0J1×Im	0J1 if t∊[0,0.5) 1J0 if t∊[0.5,1]	0J0

McDonnell's Floor as per McDonnell

```
    ∇ Z←mf R;r;i;b;x;y
      ⍝ McDonnell's Complex Floor by EEM
      r←9○R   ⍝ Real part
      i←11○R  ⍝ Imaginary part
      b←(⌊r)+0J1×⌊i
      x←1|r
      y←1|i
      :if 1>x+y
        Z←b
      :else
        :if x≥y
          Z←b+1
        :else
          Z←b+0J1
        :end
      :end
    ∇
```

The following APL code summarizes the logic at the end of McDonnell's version of his algorithm so as to be easier to read the following two tables.

1>x+y	**x≥y**	**Result**
FALSE	FALSE	b+0J1
FALSE	TRUE	b+1
TRUE	FALSE	b
TRUE	TRUE	b

	Upper Short Line t∊[0,1)	**Lower Short Line t∊[0,1)**
Parametrized Line	[¯0.5,0.5] + t × [0.5,0.5]	[0.5,¯0.5] + t × [0.5,0.5]

r	¯0.5 + t × 0.5	0.5 + t × 0.5
⌊r	¯1	0
i	0.5 + t × 0.5	¯0.5 + t × 0.5
⌊i	0	¯1
0J1×⌊i	0J0	0J¯1
b←(⌊r)+0J1×⌊i	¯1J0	0J¯1
x←1|r	[0.5,1)	[0.5,1)
y←1|i	[0.5,1)	[0.5,1)
x+y	[1,2)	[1,2)
1>x+y	FALSE	FALSE
x≥y	TRUE	TRUE
(FALSE, TRUE)	b+1 ↔ 0J0	b+1 ↔ 1J¯1

	Upper Long Line t∊[0,1]	**Lower Long Line t∊[0,1)**
Parametrized Line	[0,1] + t × [1,¯1]	[¯0.5,0.5] + t × [1,¯1]
r	t	¯0.5 + t
⌊r	0 if t=0 0 if t∊(0,1) 1 if t=1	¯1 if t∊[0,0.5) 0 if t=0.5 0 if t∊(0.5,1)
i	1 - t	0.5 - t
⌊i	1 if t=0 0 if t∊(0,1) 0 if t=1	0 if t∊[0,0.5) 0 if t=0.5 ¯1 if t∊(0.5,1)
0J1×⌊i	0J1 if t=0 0J0 if t∊(0,1) 0J0 if t=1	0J0 if t∊[0,0.5) 0J0 if t=0.5 0J¯1 if t∊(0.5,1)
b←(⌊r)+0J1×⌊i	0J1 if t=0 0J0 if t∊(0,1) 1J0 if t=1	¯1J0 if t∊[0,0.5) 0J0 if t=0.5 0J¯1 if t∊(0.5,1)
x←1|r	0 if t=0 t if t∊(0,1) 0 if t=1	[0.5,1) if t∊[0,0.5) 0 if t=0.5 (0,0.5) if t∊(0.5,1)
y←1|i	0 if t=0 1-t if t∊(0,1) 0 if t=1	[0.5,0) if t∊[0,0.5) 0 if t=0.5 (1,0.5) if t∊(0.5,1)
x+y	0 if t=0 1 if t∊(0,1)	1 if t∊[0,0.5) 0 if t=0.5

	0 if t=1	1 if t∊(0.5,1)
1>x+y	TRUE if t=0 FALSE if t∊(0,1) TRUE if t=1	FALSE if t∊[0,0.5) TRUE if t=0.5 FALSE if t∊(0.5,1)
x≥y	TRUE if t=0 FALSE if t∊(0,0.5) TRUE if t∊[0.5,1) TRUE if t=1	TRUE if t∊[0,0.5) TRUE if t=0.5 FALSE if t∊(0.5,1)
(TRUE , TRUE) (FALSE, FALSE) (FALSE, TRUE) (TRUE , TRUE) (TRUE , FALSE)	b ↔ 0J1 if t=0 b+0J1 ↔ 0J1 if t∊(0,0.5) b+1 ↔ 1J0 if t∊[0.5,1) b ↔ 1J0 if t=1	b ↔ 0J0 if t=0.5 b+1 ↔ 0J0 if t∊[0,0.5) b+0J1 ↔ 0J0 if t∊(0.5,1)

Conclusion in Theory

Based upon the above interval analysis, **in theory**, both algorithms should produce identical results.

Theory vs. Practice

Perhaps surprisingly, in practice, there are many points where the two algorithms differ. To find such points, define a grid of points in the 2 by 2 square centered at the origin. For example the following function takes a `step` value, produces a 2 by 2 square grid of that granularity, compares the result of the two algorithms, and returns the points where they disagree:

```
    ∇ EX←test4 step;⎕IO ⎕CT XX ZZ a
[1]   ⎕IO←⎕CT←0
[2]
[3]   XX←¯1+step×⍳⌊1+2÷step
[4]   ZZ←⍉XX∘.+⌽0J1×XX
[5]
[6]   a←,(mf¨ZZ)≠MF ZZ
[7]   EX←ZZ[⊂[0] (⍴ZZ)⊤a/⍳×/⍴ZZ]
    ∇
      ⍴EX2←test4 0.1
2
      mf¨EX2
¯1J1 ¯1J1
      MF EX2
0J0 0J0
```

Note that we need the each operator to apply `mf` to multiple values as it is not written as a purely scalar function, whereas `MF` is.

Reducing the step value to `0.05`, `0.01`, or lower produces many more points of disagreement.

For a step value of `0.1`, the points where the two algorithms differ are

```
¯0.09999999999999998J0.9000000000000001
¯0.2999999999999993J0.7000000000000002
```

As per the algorithm in Wikipedia[3] for computing the distance from a line to a point (a version of which is listed in the Appendix), both of the above points are just above the upper short line (and thus outside the bounding rectangle) and so the Complex Floor of neither point should be `0J0`, but that's what `MF` says it is.

This discrepancy can be illuminated on the NARS2000[4] system by converting the input to multiple-precision floating point (as provided by the MPFR library already built into NARS2000) and then varying the precision of the calculations. In particular, adding `0v` to a number converts it to MPFR format and the value of the system variable `⎕FPC` controls the precision.

```
      ⎕FPC←53
      mf¨EX2+0v
¯1J1 ¯1J1
      MF EX2+0v
0J0 0J0
```

Adding `0v` to the input doesn't change its precision (still `53` bits), but it does change the precision of the **calculations** on these numbers to that of `⎕FPC`. When we increase the value of `⎕FPC` by just one bit, that provides enough additional precision needed by the calculations to obtain the correct answer.

```
      ⎕FPC←54
      mf¨EX2+0v
 ¯1J1 ¯1J1
      MF EX2+0v
¯1J1 ¯1J1
```

More careful analysis shows that the problem in `MF` is in the calculation of `A-B` on line `5`. Stopping execution there shows as follows:

```
      5 ⎕STOP 'MF'
      MF EX2[0]
MF[5] *
      A
¯0.09999999999999998
      B
0.9000000000000001
      A-B
```

```
¯1
       →⎕LC
0J0
       ⍬ ⎕STOP 'MF'
```

Clearly, there is not enough precision in the calculation of `A-B` to get an accurate result. There is a way out which is to scale the input so that the scaled value is in a constant range but this trick must be applied very carefully. Here's the above `A-B` scaled by `10`:

```
       ((10×A)-10×B)÷10
¯1.0000000000000002
```

The trick is knowing when and how to use it, a skill I have not mastered.

Note that the lack of precision in the calculation of `A-B` is true regardless of how precise the input is. Here's the same example at `512` bits of precision:

```
       ⎕FPC←512
       ⎕PP←200
       ⍴EX3←test4 0.1v
3
       mf¨EX3
¯1J1 ¯1J1 0J0
       MF EX3
0J0 0J0 1J¯1
       5 ⎕STOP 'MF'
       MF EX3[0]
MF[5] *
       A
¯0.19999999999999999999999999999999999999999999999999999999999999
       99999999999999999999999999999999999999999999999999999999999999
       9999999999999999999999999999999985
       B
0.800000000000000000000000000000000000000000000000000000000000000
       00000000000000000000000000000000000000000000000000000000000000
       00000000000000000000000000000009
       A-B
¯1
       ((10×A)-10×B)÷10
¯1.00000000000000000000000000000000000000000000000000000000000000
       00000000000000000000000000000000000000000000000000000000000000
       000000000000000000000000000000015
       →⎕LC
0J0
```

Again, just one more bit in the calculations makes all the difference:

```
       ⎕FPC←513
       MF EX3[0]
```

```
MF[5] *
      A
¯0.19999999999999999999999999999999999999999999999999999999999999
      999999999999999999999999999999999999999999999999999999999999
      999999999999999999999999999999985
      B
0.800000000000000000000000000000000000000000000000000000000000000
      000000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000009
      A-B
¯1.000000000000000000000000000000000000000000000000000000000000000
      000000000000000000000000000000000000000000000000000000000000
      00000000000000000000000000000007
      →⎕LC
¯1J1
      ⍝ ⎕STOP 'MF'
```

Moreover, watch what happens if we change `MF` and `mf` to allow all calculations as well as the result to be done in the same datatype and precision as the argument, be it fixed precision floating point (IEEE-754), multiple precision floating point (MPFR), or multiple precision rational (MPIR). This involves calculating the real and imaginary parts more carefully so as to retain the datatype of the coefficients and avoiding floating point constants.

In particular, change `MF` lines `2` and `3` from

```
MF[2]   A←9○R   ⍝ Real part
MF[3]   B←11○R  ⍝ Imaginary part
```

to

```
MF[2]  (A B)←⊂[⍳⍴⍴R] ⎕DC R  ⍝ Real and imaginary parts
MF[3]
```

In this instance, the system function `⎕DC` (Data Conversion) splits out the real and imaginary coefficients of the two-dimensional (Complex) number `R` to a one-dimensional array by appending a length `2` dimension on the right of the shape vector of `R` while retaining the datatype of the coefficients. The enclose converts the result of `⎕DC` to a two-element vector suitable for strand assignment into `A` and `B`.

We also need to eliminate the floating point constant in line 5 by changing it from

```
MF[5]  B1←⌈0.5×T-1+A-B
```

to

```
MF[5]  B1←⌈(T-1+A-B)÷2
```

Similarly, `mf` needs the same treatment of how it calculates its real and imaginary coefficients by changing lines 2 and 3 from

```
mf[2]  r←9○R   ⍝ Real part
mf[3]  i←11○R  ⍝ Imaginary part
```

to

```
mf[2]  (r i)←⊂[⍳⍴⍴R] ⎕DC R  ⍝ Real and imaginary parts
mf[3]
```

None of these changes has any effect on the theoretical accuracy of the calculations. However, now we can call the `test4` function with `step` as a Rational number, and all calculations in `test4` as well as in `mf` and `MF` are done in infinite precision Rationals at which point the two algorithms produce identical results:

```
      ⍴test4 1r10
0
```

Conclusion in Practice

While `MF` has several desirable properties (shorter, more readable, and a scalar algorithm) vs. `mf`, unfortunately it is sensitive to the precision of its **calculations**. In particular its calculations need at least one more bit of precision (a guard bit, as it were) than in its input to produce correct results in certain edge conditions.

Theoretically these two algorithms produce identical results, and also do so in practice if the calculations are done in infinite precision. It is only in the finite precision case **regardless of the precision** that the Achilles' heel of floating point arithmetic appears.

We all know that floating point arithmetic is tricky, but we rarely know when we're being tricked. Here's a concrete example of such trickery that should cause us all to re-examine our floating point code.

Legend

In the notation below, `a` may be larger than, the same as, or smaller than `b`.

`(a,b)` is the open interval between `a` and `b` excluding both endpoints
`(a,b]` includes the righthand endpoint only
`[a,b)` includes the lefthand endpoint only
`[a,b]` includes both endpoints

All calculations done with `⎕CT←0`.

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. You may also want to read up on “HyperComplex Numbers in APL” as well as “Rational & Variable-precision Floating Point Numbers” by going to the above link and looking for those titles.

Alpha Version

The latest (unreleased) Alpha version of the NARS2000 software may be found in http://www.nars2000.org/download/binaries/alpha/ in either the w32/ or w64/ directory depending upon your OS.

References

1. http://www.jsoftware.com/papers/eem/complexfloor1.htm
2. http://dl.acm.org/citation.cfm?id=805343
3. https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line
4. http://www.nars2000.org/download/binaries/alpha/

Appendix

The following function was coded from the Wikipedia article on calculating the distance from a line to a point. This shortened version returns the signum (`1`, `0`, `¯1`) of the distance.

```
    ∇ Z←P0 DirectionLnPt (P1 P2)
[1]   ⍝ Direction (1, 0, ¯1) from a line defined
[2]   ⍝   by its endpoints P1 P2 to a point P0.
[3]   ⍝ Looking from P1 to P2, the direction is
[4]   ⍝    1 if P0 is on the right,
[5]   ⍝    0 ...             line,
[6]   ⍝   ¯1 ...             left.
[7]   ⍝ https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line
[8]
[9]   ⍝ Convert Complex numbers to their coefficients
[10]  :if 2==P0 ⋄ P0←⎕DC P0 ⋄ :end
[11]  :if 2==P1 ⋄ P1←⎕DC P1 ⋄ :end
[12]  :if 2==P2 ⋄ P2←⎕DC P2 ⋄ :end
[13]
[14]  Z←×(-/P0×⌽P2-P1)+-/P2×⌽P0
    ∇
```

Using this function, we can show that the points `EX2` and `EX3` are all outside the rectangle whose floor is `0J0`. For example, both points in `EX2` and the first two points in `EX3` are above and to the left of the upper short line:

```
      EX2[0] DirectionLnPt ¯0.5J0.5 1J0
¯1
      EX2[1] DirectionLnPt ¯0.5J0.5 1J0
¯1
      EX3[0] DirectionLnPt ¯0.5J0.5 1J0
¯1
      EX3[1] DirectionLnPt ¯0.5J0.5 1J0
¯1
```

The third point in `EX3` is below and to the right of the lower short line:

```
      EX3[2] DirectionLnPt 0.5J¯0.5 0J1
1
```