1 A Note on Dividing the Floating-Point Values

Written by Prof. Bogusław Cyganek © April 2020

Let's now ask the question - when is it safe to divide the floating point values? As we will see soon, the problem is not only to avoid divisions by the zero divider. The thing is to assure that the result of a division fits into the chosen floating point representation. In other words – that the result is as much as possible accurate. To fulfill this requirement we must care for BOTH: THE DIVISOR AND THE DIVIDENT, as we will show. More details and references can be found in the book [1].

1.1 Problem Analysis

Let's start with a definition of the division operation in the domain of real numbers:

$$c = a / b \quad \Leftrightarrow \quad a = c \cdot b \,, \tag{1-1}$$

where $a, b, c \in \Re$. Certainly, the above is not fulfilled if b is 0, that is $b \neq 0$. The question we try to answer in this note is what other conditions need to be superimposed on the values of a and b in the floating-point (FP) representation when computing the division of a by b.

Moving to the floating-point representation of the objects a, b, and c[1], their values need to be limited to the allowable range of the FP representation, such as the single precision IEEE 754 format. More precisely, except for the zero and denormals, in the IEEE 754 FP representation the modules of the a, b, and c need to fulfill the following condition:

$$k_{\min} \leq |a|, |b|, |c| \leq k_{\max} \,, \tag{1-2}$$

where $k_{\min}$ denotes the minimal positive value (not a denormal), while $k_{\max}$ is the maximal positive FP value, respectively. It is interesting to consider values of the limits $k_{\min}$ and $k_{\max}$ in the IEEE 754 standard. Let's recall that an FP value is expressed as follows

$$D = (-1)^S \cdot M \cdot B^E = (-1)^S \cdot (d_o \bullet d_1 \ldots d_{p-1}) \cdot B^E \,, \tag{1-3}$$

where $M = d_0 + d_1 \cdot B^{-1} + \cdots + d_{p-1} \cdot B^{-(p-1)}$ denotes an unsigned significand (mantissa), B=2 is a base, and E denotes the exponent (its value is shifted, e.g. for the `float` by 127). Moreover, d_0 in the normalized representation is always set to 1, and because of this it is not stored (this is the so called hidden bit). Hence, a minimal positive value in the single precision IEEE 754 format, which we named $k_{\min}$, is encoded as follows

```
0 00000001 00000000000000000000000
```

The exponent is 1 since all 0s is reserved for a zero value. Hence, a value of $k_{\min}$ is

$$k_{\min} = (1 \bullet 0 \ldots 0) \cdot 2^{1-127} = 2^{-126} \,. \tag{1-4}$$

On the other hand, the maximal value $k_{\max}$ can be encoded as follows

```
0 11111110 11111111111111111111111
```

[1] More precisely we should distinguish the real numbers from their FP approximations by writing $\hat{a} = FP(a)$. However, to avoid too many symbols we use a, b, and c for both domains.

This is so because exponent with all 1s is also reserved, so in this case it is composed of all 1s except the last bit. Hence, a value of k_{max} is as follows

$$k_{max} = (1.1\ldots1)\cdot 2^{254-127} = \left(\underbrace{1 \bullet 1\ldots 1}_{1+2^{-1}+\ldots+2^{-23}}\right)\cdot 2^{127} = \left(2^{24}-1\right)\cdot 2^{-23}\cdot 2^{127} = \left(2^{24}-1\right)\cdot 2^{104}. \tag{1-5}$$

It is interesting to analyze their product, that is[2]

$$u = k_{min}\cdot k_{max} = \left(2^{24}-1\right)\cdot 2^{104}\cdot 2^{-126} = \left(2^{24}-1\right)\cdot 2^{-22} = 4 - 2^{-22}. \tag{1-6}$$

However, we don't need to compute the FP limits by hand. It is not a surprise that the SL provides us with constants defined for each FP representation. For example, for the double precision FP we can write

```
constexpr auto kMin { numeric_limits< double >::min() };
constexpr auto kMax { numeric_limits< double >::max() };
```

So, let's return to the problem (1-1) of dividing the two *positive* FP values. We know that c cannot be more than k_{max}. So, if b can be as small as k_{min}, then a cannot exceed their product given by (1-6). On the other hand, if we allow b to be slightly larger by a factor, say f, then a can be also larger by f. This gives us the clue: the equations (1-1) and (1-2) can be upgraded to the following one

$$c = a/b \quad \Leftrightarrow \quad a \le \left(k_{min}\cdot k_{max}\right)\cdot f \quad and \quad b \ge k_{min}\cdot f\,. \tag{1-7}$$

The choice of the factor f gives us the necessary freedom to adjust the boundaries based on the assumed dynamical range of the processed values. For example, if we can superimpose the upper bound a_{max} upon the value of a, then the above yields

$$c = a/b \quad \Leftrightarrow \quad a \le a_{max} \quad and \quad b \ge a_{max}/k_{max}\,. \tag{1-8}$$

In the other scenario we can set an upper bound on b, such as b_{min}, and then we have

$$c = a/b \quad \Leftrightarrow \quad a \le b_{min}\cdot k_{max} \quad and \quad b \ge b_{min}\,. \tag{1-9}$$

Equation (1-9) is especially useful if we don't have a clear clue on a_{max} but still wish to control the divisor. In such a case, a practical choice is to set b_{min} to some *a priori* threshold. For example **this can be a value of the machine ε**, although in some applications this can be a strong constraint since ε is many orders of magnitude larger than k_{min} (that is: $\varepsilon \gg k_{min}$)[3]. In the above derivations we assumed only nonnegative values; Hence, in practice we need to extract the modules, e.g. by calling the `std::fabs`.

1.2 Practical Code Examples

Let's express the above conditions in the code. In line [1] of Listing 1-1 the floating-point representation is chosen. These can be `float`, as chosen here, or `double`, or `long double`. Then on lines [3-4] the two most important constant values, for the minimal and maximal value, are obtained respectively. Additionally, on lines [6] and [8], we read the value of ε and compute the product (1-6), respectively.

[2] Compute c for the `double` type (p=53, q=11).
[3] For the `float` ε≈1.19e-07, for the `double` ε≈2.22e-16.

Listing 1-1. Checked divisions of the floating point values (in *CppBookCode*, *FloatingPoint.cpp*).

```
1    using FP = float;
2
3    constexpr auto kMin { std::numeric_limits< FP >::min() };
4    constexpr auto kMax { std::numeric_limits< FP >::max() };
5
6    constexpr auto kEpsilon { std::numeric_limits< FP >::epsilon() };
7
8    constexpr auto kCapacity { kMin * kMax };
9
10   // These variables are used in the division.
11   FP a {}, b {}, c {};
```

Finally, the three FP objects a, b, and c are created and zero-initialized on line [11].

On lines [13-15] we can observe three divisions, out of which the last one results in the infinite result. In the IEEE 754 standard this is represented with the special encoding (*inf*).

```
12   // Certainly we must care for BOTH: THE DIVISOR AND THE DIVIDENT !
13   c = kMax / kMax;    // ok
14   c = kMin / kMin;    // ok
15   c = kMax / kMin;    // this will produce "inf"!
```

On lines [18-19] the values of a and b are set to check condition (1-7), with *f* set to 1. The condition is encoded on line [21]. If it is fulfilled, then the division can be performed on [23]. Otherwise we have a special condition which should be handled on [29]. This depends on the type of the given computations.

```
16   // CASE 0:
17   // Check the extreme values
18   a = kCapacity;
19   b = kMin;
20
21   if( a <= kCapacity && b >= kMin ) // The problem is that kCapacity is less than 4.0
22   {
23         c = a / b;
24         auto a_bis = c * b;
25         assert( std::fabs( a - a_bis ) <= ( a * kEpsilon ) );
26   }
27   else
28   {
29      ;     // what to do? throw?
30   }
```

The second case is when the value of a_{max} is known, as set at lines [35-36]. In this case the value of b_{min} can be computed in accordance with (1-8), as shown on line [40]. Let's recall that in either case, b_{min} cannot be less than k_{min}, such as 0. Some test values of a and b are set on lines [44-45] and the condition allowing for division is checked on line [48].

```
31   // CASE 1:
32   // But in practice ( a <= kCapacity ) does NOT hold (a is larger)
33   // In such a case, let's assume we know a_max <== this is the only
34   // one assumption we need!
35   const FP a_max { 1e20f };    // Must be positive since this is the magnitude
36   assert( a_max < kMax );      // certainly we should not exceed the maximum capacity
37                                // of the representation
38
39   // Now we are able to compute b_min, as follows:
40   const FP b_min { std::max( kMin, a_max / kMax ) };
41   // the second conditions is the same as: ( kMin * a_max ) / kCapacity
42
43   // Let's assume we have some real measurements ...
```

```
44    a = 2e5f;
45    b = 2e-15f;
46
47    // If the following condition is true, then we can divide
48    if( std::fabs( a ) <= a_max && std::fabs( b ) >= b_min )
49    {
50       c = a / b;
51       auto a_bis = c * b;
52       assert( std::fabs( a - a_bis ) <= ( a * kEpsilon ) );
53    }
54    else
55    {
56       ;     // what to do? throw?
57    }
```

The case shown by equation (1-9) is investigated in the following code snippet. In this scenario, b_{min} is checked against the machine ε, represented by the `kEpsilon`. Therefore a_{max} can be set relatively to ε, as shown on line [64].

```
58    // CASE 2:
59    // But what to do if we have no idea on the a_max?
60    // We can choose b_min relatively to e.g. the machine epsilon.
61    // But remember that: kEpsilon < kMin
62    // so usually there will be room on max of a, i.e. a_max_eps_rel < a_max
63    // Here we set b_min = kEpsilon but it can be any other value >= kMin
64    const FP a_max_eps_rel { kEpsilon * kMax };
65    // the same as: ( kEpsilon * kCapacity ) / kMin
66
67    a = 2e5f;
68    b = 2e-7f;
69
70    // Now the condition for the save division is as follows
71    if( std::fabs( a ) <= a_max_eps_rel && std::fabs( b ) >= kEpsilon )
72    {
73       c = a / b;
74       auto a_bis = c * b;
75       assert( std::fabs( a - a_bis ) <= ( a * kEpsilon ) );
76    }
77    else
78    {
79       ;     // what to do? throw?
80    }
```

Frequently we can simplify the computations and put the guarding condition(s) into the assertions. A variant of this idea is shown starting from line [84]; Only the divisor is run-time checked on line [85].

```
81    // CASE 2b:
82    // In practice, if we know our domain well,
83    // the first condition can go into the assert
84    assert( std::fabs( a ) <= a_max_eps_rel );
85    if( std::fabs( b ) >= kEpsilon )
86    {
87       c = a / b;
88       auto a_bis = c * b;
89       assert( std::fabs( a - a_bis ) <= ( a * kEpsilon ) );
90    }
91    else
92    {
93       ;     // what to do? throw?
94    }
```

Have fun and don’t overrun!

References

[1] Cyganek B.: *Introduction to Programming with C++ for Engineers*. Wiley, 2020.