Simulation of A Flexible Floating Point Format

Jarmo T. Alander
University of Vaasa, Finland

Abstract

Doing calculations with floating point numbers is the basic tool of any engineering design process. However, the precision of the result may in some cases be surprisingly low, which may further cause serious problems. To increase the quality of numerical software we should consider the basics of computer arithmetics with floating point numbers. Here we present a C++ implementation of flexible format of floating-point numbers closely resembling the format that was introduced by John L. Gustafson as `unums`. The implementation is based on the author's previous implementation of interval arithmetic. The importance of floating-point calculation precision and rounding error monitoring automation is discussed.

Keywords: computational topology, floating-point numbers, FPGA, interval arithmetic, numerical methods, rounding errors, software quality, software verification, unum.

1 Introduction

To most users of computational methods it is known that the results of numerical computations with real numbers are imprecise due to round-off errors. Luckily quite often the errors are not large. However, sometimes the situation is such that the errors are considerable, even totally wrong. This should worry everyone responsible for critical calculations[11]. Different results got using different computer platforms really confuse those trying to solve complex problems such are global climate warming. If the results got using the very same software vary, can we at all trust the computational results in planning and decision making. Due to the internet, including Internet of Things (IoT), more and more data is processed and used in complex modeling. This makes numerical computations even more important but also prone to round-off errors. [30].

The numerical problems encountered in scientific and engineering calculations can be mostly, but not totally, solved by deep understanding of calculus and its application in numerical method implementations. The obvious problem is that we do not simply have enough experts mastering calculus, numerical methods and also the application area to which the numerical model is tailored to. How much experts on numerical methods do we have. [7] The truth seems to be that practically none of the engineers doing numerical computations have a rigorous training in numerical analysis. Some kind of automation with numerical computations is really needed. Could we use computer to do a little more mathematics during numerical computing. It seems that Dr. John L. Gustafson may have given us a model of floating-point calculations that helps us to use the properties of computers to do numerical calculations in a way that makes it more reliable, of higher quality, and easier to test. Let us see how that is done.

Floating-point numbers in modern computers and processors use the IEEE floating-point number standard[21]. Even if practically all computers use it, the results got for the very same software may vary between platforms. This is because the standard allows some freedom in its implementation. Typically the differences are due to rounding: when to round the results and how long words are used in the calculations within the processor. Todays processors can easily have long registers to help to keep high precision, but that is not defined by the standard but let to the designers of processors and also some extend to those who design and implement compilers and numerical software. The result is that the standard does not guarantee identical numerical results. This has also other problems like making use of parallel processing difficult in numerical computations, which is quite depressing fact when we can easily have a lot of parallel processing power. We simply cannot use it efficiently.

Recently John L. Gustafson[12] has presented a new floating-point format he calls `unum`. The idea is to reserve as much memory bits for each floating-point number that is needed. Numbers are treated much like strings of text: there is always enough memory to handle short as well as long text. The related bookkeeping is done by hardware and software and is practically invisible to the user. Why not use similar flexible structure also for floating-point numbers. Both the significant and the exponent part of the proposed floating-point number format vary in length. That means that there should be some way to tell the length of both the significant (mantissa) and the exponent. And what is even more import Gustafson introduces a one bit flag, `ubit` which tells whether or not the number is actually exact or not. That single bit is extremely important. It makes a clear difference between `unums` and traditional floating-point numbers. In case the result is exact, that information can be traced thru the computations, while in traditional floating-point numbers there is not any information on exactness. The number may be inexact already in the input or somewhere within the calculations. The poor user is given no information on that, never.

Let us have a simple example. Our program is using a decimal constant 0.1 and we thing it is exact. However as a binary floating-point number, we cannot have exactly 0.1, there simply is not such a number in any length of floating point number we have chosen. Another very common constant is 1/3=0.333333... that does not have neither decimal nor binary finite exact representation. So, even the constants of our software can be inexact, but the user have no idea of this and neither of the exactness or precision of any results of computation.

Observe, that is situation is totally different to that of using integers in calculations. The results of integer calculations are precise if the results fit within the given wordlength i.e. there is no overflow. Overflow is easy to detect and typically cause program to stop.

If the numbers used in arithmetic operations are exact, as they are when using integer, also the result can be calculated exactly without any rounding errors. The situation with floating-point numbers is similar in many cases: exact values in and exact values out. This makes integer and floating-point calculations seamless when there is information of exactness (the `ubit`). When there is rounding error, that information can be encoded in the `ubit` and the length of the significant. The least significant bit (or a few bits) is imprecise and also gives the small interval within which the results must be, if the calculations has been done correctly. That same precision information can be outputted to the user, so that he/she can check that the calculations has been successfully done with respect to rounding errors.

Traditionally interval arithmetic has been used to monitor rounding errors. `unums` can

be used to replace interval arithmetic [26, 1, 2]. The good news is that they are doing it much better and they can be used in any calculations, while interval arithmetic applies only to very simple formulas.

Like interval arithmetic `unums` can be used in global optimization. In addition the varying length format of `unums` also saves memory, time, and energy[25].

1.1 Error example

Professor W. Kahan has presented a simple example of rounding problem:

"Define functions with:

$E(0) = 1, E(z) = (e^z - 1)/z$,

$Q(x) = |x - \sqrt{x^2 + 1}| - 1/(x + \sqrt{x^2 + 1})$,

$H(x) = E(Q(x)^2)$.

Compute $H(x)$ for $x = 15.0, 16.0, 17.0, 9999.0$."

Repeat using more precision and other computers and softwares.

It seems that the solution is always $(0, 0, 0, 0)$—whatever precision or computer or software you use.

However, the right answer is $(1, 1, 1, 1)$. The problem cannot be solved by just adding more and more bits to floating-point calculations (brute force approach). The reason is that there are a lot of holes in the floating-point number space. Human mathematical reasoning or at least computational topology is needed to solve it. `unums` do that automatically. There is not any holes in their encoding of floating-point numbers. Using unums of the average size of just 6bits Gustafson was able to solve the problem[13].

What is the reason that computers or programs are not able to provide essential information on the precision of numerical calculations when using floating-point numbers? It seems that nobody really understood the importance of topology. The IEEE floating-point numbers don't cover the real numbers space but just sample it, while `unums` cover the real numbers from $-\infty$ to ∞ and everything in between without any holes of any size. The whole real axis is covered by a finite set of either open (imprecise, tt ubit=1) or closed (precise, `ubit=0`) numbers letting no holes in between (fig. 1). This use of basic topology, helps in to gain much more reliable results.

Already very short unums can be useful in practical applications to screen possible solutions of mathematical problems like locating zeros of functions. Because unum calculations are mathematically rigorous, we can rely on the results got. If there doesn't seem to be a solution when using short representations, there is no hope to find them using longer representation. That would be only a waste of energy.

1.2 Related work

Unums are just a couple of years old so there is not yet many publications on them. Here you can find most of them: [13, 14, 19, 15, 8, 18, 22].

Interval Arithmetic (IA) was invented in 60's and they have been tried to apply to monitoring of rounding errors ever since, so that there is plenty of research on them. Here some of them: [26, 5, 16, 28, 27, 10, 20, 23, 29, 17, 9, 6, 24]. The idea of IA is simple and easy to implement[4, 3] but unfortunately they have only a limited application area. The main problem is that IA tends to give so pessimistic bounds to the expressions analyzed that it is practically useless. Especially poor suited IA is for iterative evaluations[3].

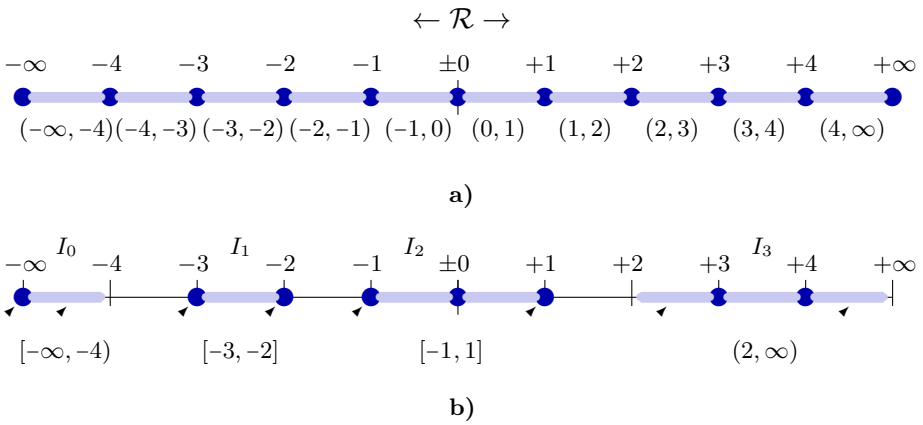

Figure 1: An example of very short unums. **a)** Covering the whole real axis with 11 exact unums, (•) and 10 open intervals ( ). **b)** Four unum intervals I_i, $i \in \{0, 1, 2, 3\}$, representable by unum pairs (shown by arrow heads) of unums in a).

2 `unum++` in C++

Our implementation of `unums` is not exactly identical to that of Gustafson. Actually also Gustafson have presented a new unum-like format he calls unum 2.0 [14], that is quite different to both unums and IEEE floating-point standard. Our implementation uses varying length significant, but the exponent is kept of fixed size. This is because we use our unums as a model of floating-point numbers mainly for the analysis of numerical algorithms. Our long term goal is to finally implement unums in FPGAs for real time numerical processing.

Using todays processors to implement unums is not efficient. The structure of numerical operations has been designed for IEEE floating-point numbers, which are of fixed length while unums can have any length. So the obvious way to do experiments is to use software emulation or hardware implementation using (Field Programmable Gate Arrays) FPGAs, which can be tailored to use any word length.

The `unum++` package presented here is actually based on the author's interval arithmetic [26] packages IP[4] in C and IP++[3] in C++ and thus contains also the basic interval methods.

Table 1 gives some examples of very small integer unums and basic arithmetic operations between them.

2.1 ubit

Our definition of `ubit` differs from that given by Gustafson. Our ubit is an integer that gives the number of imprecise digits of the significant. In case `ubit=0` the number is precise, otherwise the `ubit` gives the number of least significant digits which are imprecise. The implementation of `unums` are here done using decimal digits for debugging purposes. The significant is represented by a varying size array of digits.

If for both operands `ubit=0` i.e. they are precise then the result of the operation is precise and `ubit=0` for it, too. For addition and subtraction `ubit` processing is easy. For multiplication the situation is different because the number of digits tends to double for each multiplication and also the number of imprecise digits in greatly increased. Hence it is reasonable to limit the precision in multiplication or even try to totally avoid multiplication, which is typically done in high performance digital signal processing implementation on

Table 1: Examples of addition, subtraction, and multiplication of values `C0 = 123` (last digit uncertain) and `C1 = 3456` (two last digits uncertain). Uncertain digits shown by overline (like: $\overline{3}$).

var/expr [value]	S	Significant	Exp	ubit	Elen	Slen
`C0 [123]`	+	$12\overline{3}$	10^0	$1(\approx)$	1	3
`C1 [3456]`	+	$34\overline{56}$	10^0	$2(\approx)$	1	4
`C0+C1 [3570]`	+	$35\overline{7}$	10^1	$1(\approx)$	1	3
`C0-C1 [-3330]`	−	$33\overline{3}$	10^1	$1(\approx)$	1	3
`C1-C0 [3330]`	+	$33\overline{3}$	10^1	$1(\approx)$	1	3
`C1-C1 [0]`	+	$\overline{0}$	10^0	$1(\approx)$	1	1
`C1*(-C1) [-11943000]`	−	$11\overline{943}$	10^3	$3(\approx)$	1	5

FPGAs or ASICs.

The proceedings of imprecise digit is done by monitoring carry propagation. For addition and subtraction that is simply done by looking if the most significant imprecise digit will launch a carry. In case there is no carry launched then the ubit value is got by looking the maximum of the ubits i.e. the indexes of the most significant imprecise bits (or digits in our case) of the aligned significants. In the case there is carry propagation the maximum value is increased by the number of digit positions that the carry is actually propagated.

The evaluation of `ubit` for multiplication is similarly based on carry propagation monitoring. Observe, that multiplication is actually a series of additions which means that the process of carry propagation is somewhat more complex (c.f. Table 2) and avoided here like the evaluation of division based on Newton's iteration.

Even the most simple and basic the numerical problems are actually surprisingly complex to solve rigorously. There are 18 special cases that should be handled when solving the simple linear equation ($bx + c = 0$) and tens of cases more for the quadratic equation $ax^2 + bx + c = 0$. How many programs using second order equations are really checking all the possible special cases? Obviously, there are a lot of cases that just wait to be revealed in some critical situation. Thus, profound software verification and testing is really needed and that should be as automatic as possible to save human work and to keep high quality and reliability.

3 Hardware implementation

Because current processors are designed to obey IEEE floating-point standard the implementation of unums at hardware level is not possible. The situation with FPGAs is totally different. Namely they do not have any hardwired floating-point processing at all. All must be implemented so that using IEEE or unums is in principle possible and also in practise. However, the `unums` are so complex that it takes some time until such implementations will emerge. Meanwhile we have to do `unum` processing with software like Gustafson has done for his book. Also a hybrid is possible: to implement the very basic operations needed with `unums` on FPGA and doing the rest with software. This is most practical with FPGAs that already contain an embedded processor the so called SoC-FPGA or System-on-Chip-FPGA. Linux software, like C++, can be run on the SoC ARM processor while the FPGA would

do most hardware beneficial parts of the processing. In addition we can greatly facilitate verification and testing with this kind of dual hardware.

4 Discussion

The great thing of Gustafson's `unum` proposal is that he not only claims that his format is a good alternative to consider for floating-point processing but he also shows by programmed numerical examples how unums work bit by bit in simple but realistic numerical problems.

5 Conclusions and future

This paper discusses the importance of the quality of numerical computational methods for practical engineering. The current situation in computer realisations of computations with floating-point numbers realy need some essential revision. The idea of flexible sized floating-point number with rigorous monitoring of rounding errors might not sound a revolution. But for practical calculations used in critical applications that might give totally new level of confidence and means to increase automation in science and engineering design.

References

[1] Jarmo T. Alander. Intervallimenetelmien soveltuvuudesta käyrien ja kaarevienpintojen tietokonekäsittelyyn [Applicability of interval methods in modeling of curves and sculptured surfaces]. Master's thesis, Helsinki University of Technology, Department of Computer Science, 1984. (in Finnish).

[2] Jarmo T. Alander. On interval arithmetic range approximation methods of polynomials and rational functions. *Computers&Graphics*, 9(4):365–372, 1985.

[3] Jarmo T. Alander. Programmers manual of interval package IP++, Version 0.0 and comparison to IP. Technical Report HTKK-TKO-B62, Helsinki University of Technology, Laboratory of Information Processing Science, 1987.

[4] Jarmo T. Alander, Kari Hyytiä, Juha Hämäläinen, Asko Jaatinen, Olli Karonen, Panu Rekola, and Matti Tikkanen. Programmers manual of Interval Package IP, Version 0.0. Technical Report HTKK-TKO-B59, Helsinki University of Technology, Laboratory of Information Processing Science, 1984.

[5] G. Alefeld and J. Herzberger. *Einführung in die Intervallrechnung.* B. I. Verlag, Mannheim, 1974.

[6] Anon. *IEEE Standard 1788-2015 for Interval Arthmetic.* IEEE, 2015.

[7] David H. Bailey. Big data and its sins, 2015. (conference slides).

[8] Andrea Bocco, Yves Durand, and Florent de Dinechin. Hardware support for unum floating point arithmetic. In *Proceedings of the 2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, Giardini Naxos (Italy), 12.-15. June 2017. IEEE, Piscataway, NJ.

[9] Michael J. Cloud, Ramon E. Moore, and R. Baker Kearfott. *Introduction to Interval Analysis*. SIAM, Philadelphia, 2009.

[10] J. Garloff. Convergent bounds for the range of multivariate polynomials. In K. Nickel, editor, *Interval Mathematics 1985*, volume 212, pages 37–56, Freiburg (Germany), 23.-26. September 1985. Springer-Verlag, Berlin.

[11] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[12] John L. Gustafson. *The End of Error*. Chapman & Hall/CRC Press, Boca Raton, FL, 2015.

[13] John L. Gustafson. *The End of Error*. Chapman & Hall/CRC Press, Boca Raton, FL, 2015.

[14] John L. Gustafson. A radical approach to computation with real numbers. *Supercomputing*, 3(2):–, August 2016.

[15] John L. Gustafson and Isaac T. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing*, 4(2):70–86, 2017.

[16] E. Hansen. A globally convergent interval method for computing and bounding real roots. *BIT*, 18:415–424, 1978.

[17] Peter Hertling. A lower bound for range enclosure in interval arithmetic. *Theoretical Computer Science*, 279(1-2):83–95, May 2002.

[18] Junjie Hou, Yongxin Zhu, Yulan Shen, Mengjun Li, Qian Wu, and Han Wu. Enchancing precision and bandwidth in cloud computing: Implementation of a novel floating-point format in FPGA. In *Proceedings of the 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, New York, NY, 26.-28. June 2017. IEEE.

[19] Laslo Hunhold. The unum number format: Mathematical foundations, implementation and comparison to IEEE 754 floating-point numbers. Bachelor's thesis, Universität zu Köln, 2016.

[20] Eero Hyvönen. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence*, 58(1-3):71–112, 1992.

[21] IEEE 754-2008 – IEEE standard for floating-point arithmetic. Standard 745-2008, IEEE, 2008.

[22] Deepak Ingole, Michal Kvasnica, Himeshi Praveeni De Silva, and John L. Gustafson. Reducing memory footprint in explicat model predictive control using universal numbers. In *Proceedings of the 2017 IFAC World Congress*, pages –, Toulouse (France), 9.-14. 2017. IFAC.

[23] Vladik Kreinovich. With what accuracy can we measure masses if we have an (approximately known) mass standard. *SIGNUM Newsletter*, 31(4):26–34, October 1996.

[24] Ulrich Kulisch. Up-to-date interval arithmetic from closed intervals to connected sets of real numbers. In *Parallel Processing and Applied Mathematics*, volume 9574 of *Lecture Notes in Computer Science*, pages –. Springer-Verlag, Heidelberg, 2016.

[25] Mark P. Mills. The cloud begins with coal big data, big networks, gig infrastructure, and big power, an overview of the electricity used by the global digital ecosystem. Report, Digital Power Group, 2013.

[26] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, Inc., Englewood Cliffs, 1966.

[27] S. P. Mudur and P. A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics & Applications*, 4:7–17, 1984.

[28] H. Ratschek. Centered forms. *SIAM Journal of Numerical Analysis*, 17:656–662, 1980.

[29] Toru Sasaki, Jun Tanida, and Yoshiki Ichioka. Optical implementation of high-accuracy computing based on interval arithmetic and fixed point theorem. *Optical Engineering*, 38(3):485–489, March 1999.

[30] V. Stodden, D. H. Bailey, J. Borwein, R. J. LeVeque, W. Rider, and W. Stein. Setting the default to reproducible, reproducibility in computational and experimental mathematics. report, ICERM Workshop on Reproducibility in Computational and Experimental Mathematics, 10.-14. December 2012.

A Unum examples

Table 2: Multiplication of significants of $e = 2.71828182845904$ and $\pi = 3.1415926535897932$ with one imprecise digit.

		1	1	3	2	4	3	5	4	6	5	8	6	7	7	7	6	6	5	6	6	3	3	1	2	2	2	1			
2																	5	4	3	6	5	6	3	6	5	6	9	1	8	0	8
3																8	1	5	4	8	4	5	4	8	5	3	7	7	1	2	
9														2	4	4	6	4	5	3	6	4	5	6	1	3	1	3	6		
7													1	9	0	2	7	9	7	2	7	9	9	2	1	3	2	8			
9												2	4	4	6	4	5	3	6	4	5	6	1	3	1	3	6				
8											2	1	7	4	6	2	5	4	6	2	7	6	7	2	3	2					
5										1	3	5	9	1	4	0	9	1	4	2	2	9	5	2	0						
3										8	1	5	4	8	4	5	4	8	5	3	7	7	1	2							
5								1	3	5	9	1	4	0	9	1	4	2	2	9	5	2	0								
6							1	6	3	0	9	6	9	0	9	7	0	7	5	4	2	4									
2							5	4	3	6	5	6	3	6	5	6	9	1	8	0	8										
9					2	4	4	6	4	5	3	6	4	5	6	1	3	1	3	6											
5				1	3	5	9	1	4	0	9	1	4	2	2	9	5	2	0												
1				2	7	1	8	2	8	1	8	2	8	4	5	9	0	4													
4		1	0	8	7	3	1	2	7	3	1	3	8	3	6	1	6														
1		2	7	1	8	2	8	1	8	2	8	4	5	9	0	4															
3	8	1	5	4	8	4	5	4	8	5	3	7	7	1	2																
	8	5	3	9	7	3	4	2	2	2	6	7	3	5	5	0	5	1	3	5	4	1	8	2	8	2	7	0	5	2	8
	8	5	3	9	7	3	4	2	2	2	6	7	3	5	6	7	0	6	5	4	6	3	5	5	0	4	1	9	0	0	8

B unum class

```
class unum {
public:
   int   LaTeXdisplay;                // for reporting with LaTeX
   string   Name;                     // ditto
// old interval stuff:
   double  minimum;
        double  maximum;
        double  scalar;
// for unum
   int   S;   // sign
   int   NE;   // length of exponent
   int   NS;   // length of significant
   int   ubit;   // imprecise digits
   int*   Mantis;                     // Significant array
   int*   Expo;   // Exponent array
// constructors:
    unum()  { ... }                   // zero
    unum(int i) { ... }               // precise number from int
    unum(double mini) { ... }         // imprecise floating point number
    unum(double mini, double maxi)
            { ... }                   // floating point interval
    unum(double mini, double maxi, double sca)
            { ... }                   // floating point interval with middle value
    unum(unum& X)
            { ... }                   // new unum from unum
    unum(const unum& X)
            { ... }                   // new unum from unum constant
    unum(string S, int e, int u)
       { ... }   // possibly very long unum given as string
// basic attributes:
```

```
    double inf()    { return minimum; }
    double sup()    { return maximum; }
    double sca()    { return scalar;  }
    double mid()    { return (minimum+maximum)/2; }
    double wid()    { return maximum-minimum; }

// unum specific
    unum normalize();
    unum normalize(int);

    unum dual() ;
    unum undual() ;
    int over(double x) ;
    int approx(int n) ;
    int empty() { return maximum-minimum<eps; }
    int AbsGreaterEqual(int n, const int A[], const int B[]) ;
    int unite() ;
// LaTeX reporting
    string   LaTeX() ;                // display in LaTeX format
    int setLaTeX() ;
    int resetLaTeX() ;
    string   LaTeXtableHeader() ;
    string   LaTeXtableClose() ;
    string   setName(string name) { return Name = name; }
// arithmetic operators:
    unum operator-() ;
    unum operator^(int n) ;
    void     operator=(const unum& X);
    unum operator||(const unum& X) ;
    unum operator&&(const unum& X) ;
    friend   unum operator+(const unum& A, const unum& B) ;
    friend   unum operator-(const unum& A, const unum& B) ;
    friend   unum operator*(double A, const unum& B) ;
    friend   unum operator*(const unum& A, double B) ;
    friend   unum operator*(const unum& A, const unum& B) ;
    friend   unum operator/(double A, const unum& B) ;
    friend   unum operator/(const unum& A, double B) ;
    friend   unum operator/(const unum& A, const unum& B) ;
// input and output:
    friend std::istream& operator>>(std::istream& s, unum& p);
    friend std::ostream& operator<<(std::ostream& s, const unum& p);
// testing:
    void   main();                    // for testing
} ;
```