Unum – an expedient extension of IEEE 754

Thomas Risse

Institute of Informatics & Automation, IIA
Faculty EEE & CS
Hochschule Bremen
University of Applied Sciences

March 10^{th}, 2016 @ LSBU

Thomas Risse
IIA, Fak E & I, HSB

Agenda

1. Introduction – Setting the Stage
2. Unum – Universal Numbers
3. Arithmetic with Unums
4. How Unums are Used – Examples
5. Conclusion

Introduction – Setting the Stage

This is about floating point numbers, numbers like -1, 0, 1, $e \approx 2.71828$, $\pi \approx 3.14159$, $1.0 \cdot 10^2$, $h \approx 6.626 \cdot 10^{-34}\,\mathrm{J\,s}$, $\gamma = (6.67408 \pm 0.00031) \cdot 10^{-11}\,\frac{\mathrm{m}^3}{\mathrm{kg \cdot s}^2}$ or Avogadro[1]s constant $N_A \approx 6.022140857(74) \cdot 10^{23}\,\mathrm{mol}^{-1}$, cp [1].

Floating point numbers usually are represented, stored and handled according to the IEEE 754 standard from 1985 [6], amended 2008 [7], s.a. [2].

Some rational numbers $q \in \mathbb{Q}$ can be represented as

$$q = \textbf{sign}\,(\textbf{normalized})\textbf{mantissa} \cdot 2^{\textbf{exponent}}$$

In general the mantissa is normalized in order to avoid ambiguities like $0.5 \cdot 2 = 1 = 32 \cdot 2^{-5}$.

Fixing # of mantissa bits and # of exponent bits determines accuracy and dynamic range.

[1] Amedeo Avogadro (1776-1856) https://en.wikipedia.org/wiki/Amedeo_Avogadro

Introduction – Setting the Stage

That's simple and hardware oriented: e.g. the exponent is stored biased. Since the Intel 8087 of 1985, any processor has some FPU implementing IEEE 754 – to some degree.

$\sqrt{\sqrt{\cdots\sqrt{2}}} = \sqrt[2^n]{2} = 1$, $\sin\pi = 1.22464799147353\text{e-}16$

$\frac{99}{7} - \frac{97}{7} - \frac{2}{7} = -9.99200722162640\text{9e-}16$ etc etc allegedly!

$a \cdot b \neq b \cdot a$ on a 1976 CRAY-1 [5] before: IEEE 754 cannot represent rationals like $0.1 = 0.0\overline{0011}_{(2)}$ **exactly**.

$\Rightarrow$ computational hazards due to representational errors

Even worse, the laws of arithmetic in $\mathbb{Q}$ are no longer guaranteed: e.g. with a two bit mantissa and lots of exponent bits we have $1 = 1.0_{(2)} \cdot 2^0$, $\frac{1}{4} = 1.0_{(2)} \cdot 2^{-2}$ so that $\mathrm{fl}(1 + \frac{1}{4}) = \mathrm{fl}(1.0_{(2)} \cdot 2^0 + 0.01_{(2)} \cdot 2^0) = 1$ and thus

$$\mathrm{fl}\left(\mathrm{fl}(1 + \tfrac{1}{4}) + \tfrac{1}{4}\right) = 1 \neq 1.5 = \mathrm{fl}(1 + \tfrac{1}{2}) = \mathrm{fl}\left(1 + \mathrm{fl}(\tfrac{1}{4} + \tfrac{1}{4})\right)$$

$\Rightarrow$ computational hazards due to rounding errors

Unum – Universal Numbers

Unum [5] is an extension of the IEEE 754 2008 standard [8] of floating point numbers $\mathbb{Q} \ni q = s$ mantissa $2^{exponent}$ (where $s = \pm$ is the sign bit), i.e. first of all

float

s	8bit exponent	24bit mantissa

due to one hidden bit, a total of 4 bytes

double

s	11bit exponent	53bit mantissa

due to one hidden bit, a total of 8 bytes

floats = single precision $\subset$ single extended $\subset$
double = double precision $\subset$ double extended

IEEE floats and doubles are perceived as to represent a proper balance between accuracy, dynamic range and memory requirements (bandwidth), efficient arithmetic and power requirements.

Now, Unum is designed to overcome inherent **deficiencies** of IEEE floats and doubles!

Deficiencies of IEEE 754

- rounding errors jeopardise associativity and distributivity
  - accept $\pm\infty$ to substitute large magnitude finite numbers
  - accept 0 to substitute small magnitude non-zero numbers
  - accept incorrect exact floats for nearby distinct numbers
- handling $\pm\infty$ (all fraction bits 0 and all exponent bits 1)
- handling lots of NaN (fraction $\neq$ **0** and all exponent bits 1)
- underflow/overflow only in processor status word
- potential for parallelization not exploited
- IEEE 754 is not strict (e.g., subnormals optional): different results on different machines! JAVA [11], FMA, EDP, SUN [5]
- inefficient usage of memory results in
  - high memory and transfer requirements
  - high power requirements for transfers

Unum – Universal Numbers

Let's start with a fixed format with es bits for the biased exponent, fs bits for the mantissa with a hidden bit (0 if exponent is all zero, and 1 otherwise) and the ubit

s	es bit exponent	fs bit mantissa	ubit

The ubit indicates whether the stored number is exact (0) or inexact (1: there are more bits to come but there is no room). If ubit==1 then the unum represents the **intervall** between the unum stored and the next bigger exact unum. **no rounding!**

E.g. let es=2 and fs=1. Then
00010 = $\frac{1}{2}$ 00011 = $\left(\frac{1}{2}, 1\right)$ 00100 = 1 and $\frac{2}{3} \in \left(\frac{1}{2}, 1\right)$ is much better than 'lies' like $\frac{2}{3} = \frac{1}{2}$ or $\frac{2}{3} = 1$ in this number system.

E.g. 01110 = $+\infty$ 11110 = $-\infty$
01100 = 4 = maxreal 11100 = -4 = –maxreal
01101 = $(4, \infty)$ = almost ∞ 11101 = $(-\infty, -4)$ = almost $-\infty$
01111 = quiet NaN 11101 = $(4, \infty)$ = signaling NaN

Benefits so far

- no rounding due to ubit: instead an interval
- no underflow: instead intervals $(0, \mathit{smallestsubnormal})$ or $(-\mathit{smallestsubnormal}, 0)$ resp.
- no overflow: instead intervals $(\mathit{maxreal}, \infty)$ or $(-\infty, -\mathit{maxreal})$ resp.
- only exactly two NaN, quiet and signaling

Further potential

variable length formats like for strings, messages, images, memory segments, files, etc. Also, the ratio fs/es in IEEE 754 floats and doubles chosen high-handedly.

To be on the safe side there usually is overkill:
default data type e.g. in MATLAB is double!

Two fields specify bit lengths of exponent and fraction.

variable length unums

Def: esizesize $\in \mathbb{N}_o$ is the number of bits to store the maximum number of bits of the exponent.

Def: fsizesize $\in \mathbb{N}_o$ is the number of bits to store the maximum number of bits of the fraction.

Def: utag = {ubit, exponent size bits, fraction size bits} with |utag| = 1+esizesize+fsizesize

Def: maxubits = $2+\text{esizesize}+\text{fsizesize}+2^{\text{esizesize}}+2^{\text{fsizesize}}$

s	es bit exponent	fs bit fraction	ubit	es-1	fs-1

There are two good messages:

1. memory savings significantly outweigh the additional memory requirements for utags!
2. the computer (and **not** the programmer) manages field sizes!

some (extreme) examples

Def: $\{$esizesize, fsizesize$\}$ defines an *environment*.

E.g. $\{0, 0\}$ takes a total of $1 + 2^0 + 2^0 + 1 = 4$ bits covering the real line by 1111 = signal NaN and 0111 = quiet NaN and
1110 = $-\infty$ 1101 = $(-\infty, -2)$ 1100 = -2 1011 = $(-2, -1)$
1010 = -1 1001 = $(-1, 0)$ 1000 = 0 0000 = 0 ... ...

E.g. $\{3, 4\}$ has a maximum total of
$1 + 2^3 + 2^4 + 1 + 3 + 4 = 33$ bits representing floats with exponents with up to $2^3 = 8$ bits (as in the case of IEEE 754 single prec.) and with fractions with up to $2^4 = 16$ bits (better accuracy as in the case of IEEE 754 half single prec.).

E.g. $\{3, 5\}$ with max 41 bits is better than IEEE 754 single.

E.g. $\{9, 9\}$ allows for exponents and fractions of up to $2^9 = 512$ bits, maxubits = 1+512+512+1+9+9 = 1044 and potential to store integers of the magnitude
$2^{2^{512}} = 2^{4 \cdot 2^{51 \cdot 10}} = 16^{(2^{10})^{51}} \approx 16^{(10^3)^{51}} = 16^{10^{153}}$.

Arithmetic with Unums

Unums represent exact rationals and real open intervals in turns. But Unums avoid the pitfalls of interval arithmetic!

Def: *ubounds* specify real intervals by a (inexact) unum or two unums. Interval arithmetic only provides $[a,b]+[c,d]=[a+c,b+d]$ where $a+b$ is rounded towards $-\infty$ and $b+d$ towards $+\infty$. With Unum there are two addition tables

$+$ *left*	$[-\infty$	$(-\infty$	$[y$	$(y$	$[\infty$
$[-\infty$	$[-\infty$	$[-\infty$	$[-\infty$	$[-\infty$	(*NaN*
$(-\infty$	$[-\infty$	$(-\infty$	$(-\infty$	$(-\infty$	$[\infty$
$[x$	$[-\infty$	$(-\infty$	$[x+y$	$(x+y$	$[\infty$
$(x$	$[-\infty$	$(-\infty$	$(x+y$	$(x+y$	$[\infty$
$[\infty$	(*NaN*	$[\infty$	$[\infty$	$[\infty$	$[\infty$

$+$ *right*	$-\infty]$	$y)$	$y]$	$\infty)$	$\infty]$
$-\infty)$	$-\infty]$	$-\infty]$	$-\infty]$	$-\infty]$	*NaN*)
$x)$	$-\infty]$	$x+y)$	$x+y)$	$\infty)$	$\infty]$
$x]$	$-\infty]$	$x+y)$	$x+y]$	$\infty)$	$\infty]$
$\infty)$	$-\infty]$	$\infty)$	$\infty)$	$\infty)$	$\infty]$
$\infty]$	*NaN*)	$\infty]$	$\infty]$	$\infty]$	$\infty]$

Cases '$(\infty$' and '$-\infty)$' cannot occur and $x-y=x+(-y)$.

If $[x+[y \gtrless \pm\texttt{maxreal}$ then $[x+[y = {}^{(\texttt{maxreal}}_{(-\infty}$.

If $x]+y] \gtrless \pm\texttt{maxreal}$ then $x]+y] = {}^{\infty)}_{-\texttt{maxreal})}$.

Unums vs IEEE 754: varying precision

E.g. in IEEE half $\mathrm{fl}(30 + \frac{1}{256}) = \mathrm{fl}(1.111_{(2)} \cdot 2^4 + 1 \cdot 2^{-8}) = 30$
E.g. in $\{3, 4\}$ $\mathrm{fl}(30+\frac{1}{256}) = 1.111000000001_{(2)} \cdot 2^4 = 30.00390625$
E.g. in $\{3, 4\}$ $\mathrm{fl}(1000+\frac{1}{256}) = (1000, 1000.0078125)$

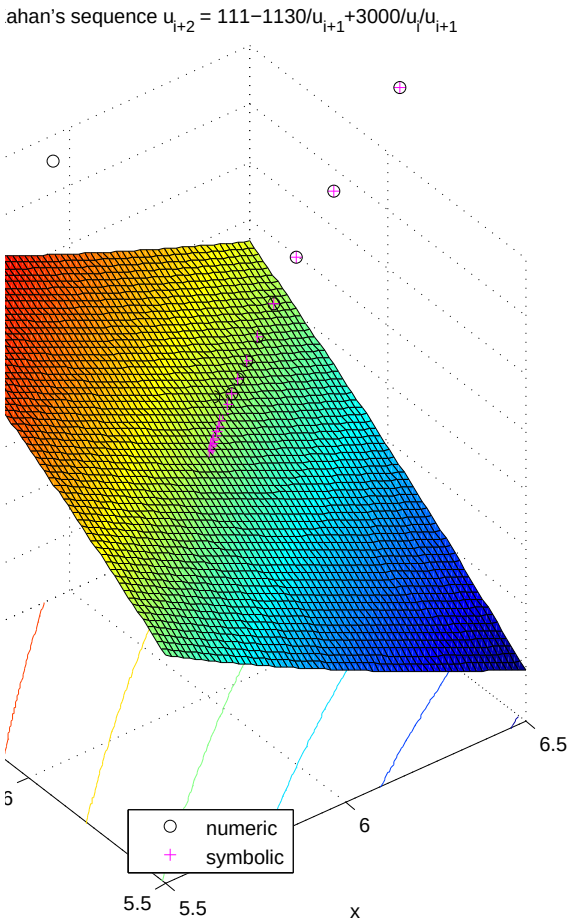
C, JAVA:
```
float s=0.0; for (int i=0; i<1.0e+09; i++) s+=1.0;
```

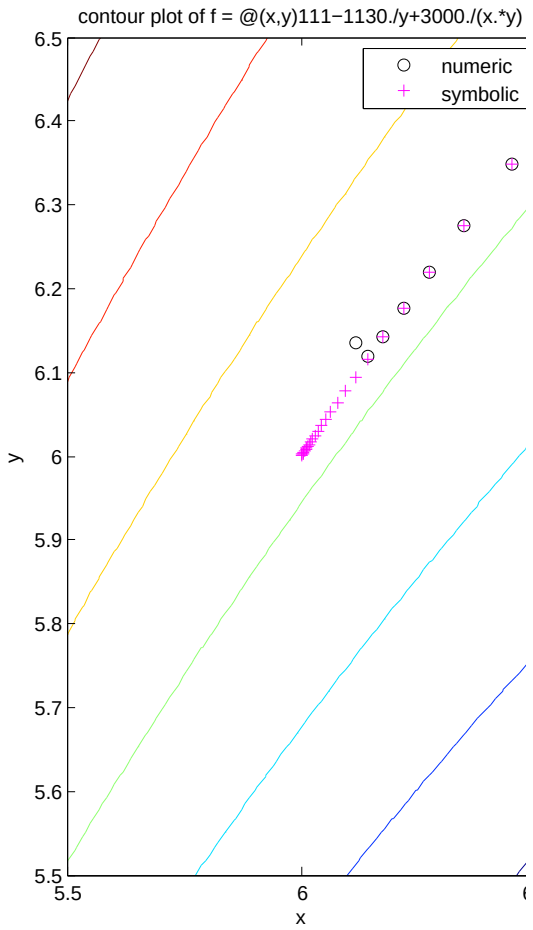
MATLAB:
```
s = single(0); for i=1:10^9, s = single(s+1); end
```

both return $s = 16\,777\,216$ instead of $s = 1\,000\,000\,000$ like Unum.
(There are ways to reduce the impact of rounding, e.g. [9])

Using Unums – Kahan-Examples

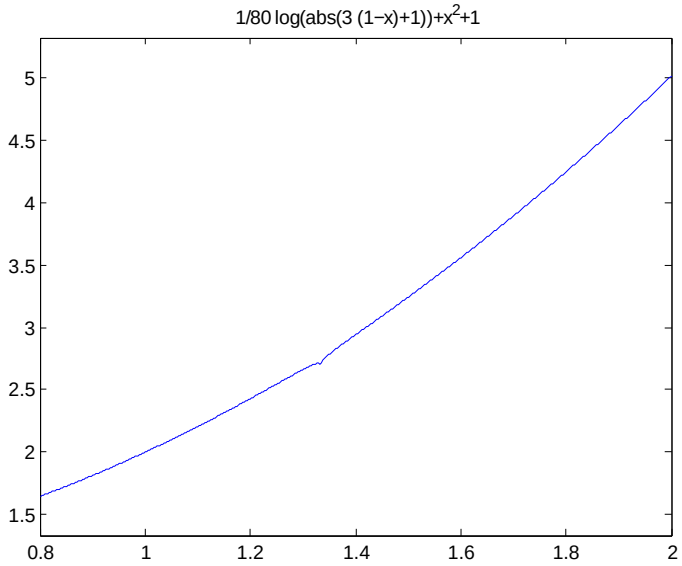
Specify $\left(u_i\right)_{i\in\mathbb{N}_o}$ by $u_{i+2} = 111 - \frac{1130}{u_{i+1}} + \frac{3000}{u_i u_{i+1}}$ recursively with $u_o = 2$ and $u_1 = -4$ (two fixpoints (6,6,6) and 100(1,1,1))

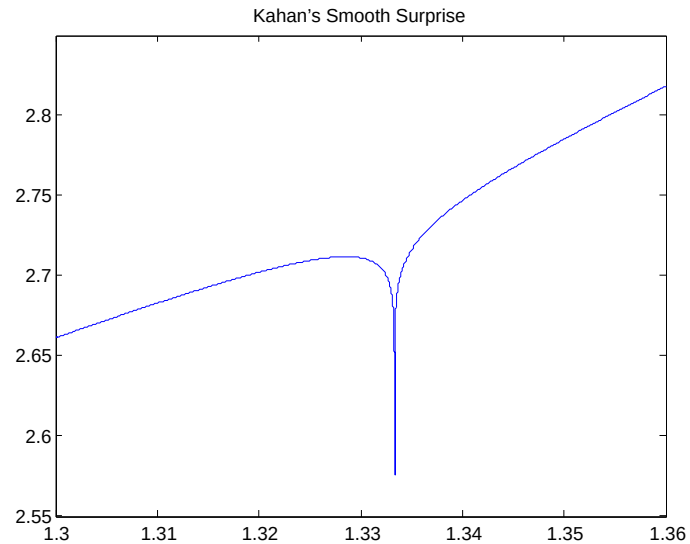




Using Unums – Kahan-Examples

- Evaluate $H(x) := E(Q^2(x))$ with $E(0) = 1$ and $E(x) = \frac{e^x - 1}{x}$ and $Q(x) = \left|x - \sqrt{x^2 + 1}\right| - \frac{1}{x + \sqrt{x^2+1}}$ for $x = (15, 16, 17, 9999)$. MATLAB gives $(0, 0, 0, 0)$ instead of $(1, 1, 1, 1)$ because actually $Q(x) \equiv 0$.
- Find min of $\frac{1}{80} \log \left|3(1 - x) + 1\right| + x^2 + 1$ for $x \in [0.8, 2]$



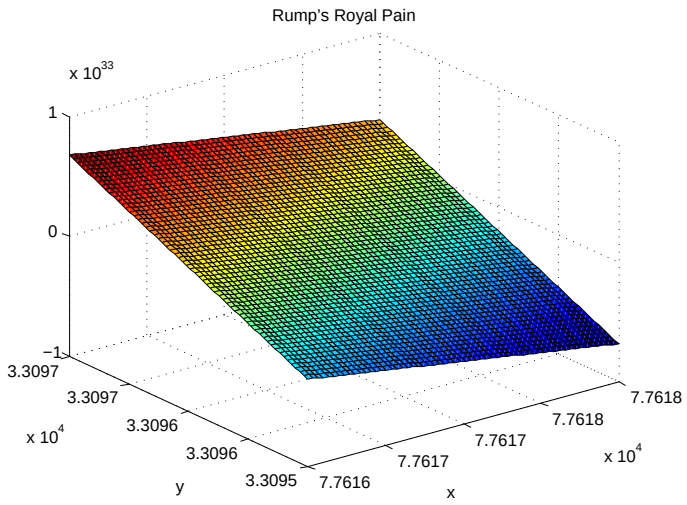


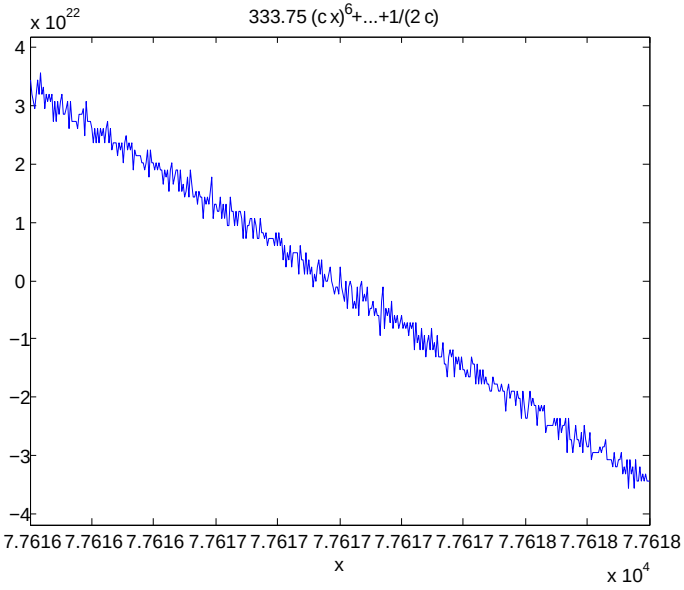
- $5.9604644775390625^{7/8} = 4.76837158203125$ exactly?

  s. `https://sage.informatik.hs-bremen.de/home/pub/189/`

Using Unums – Rump-Example

- For $(x_o, y_o) = (77617, 33096)$ evaluate $f(x, y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}$. MATLAB gives $f(x_o, y_o) = -1.180591620717411e+21$ instead of about $-0.8273960599468213 6$ as e.g. the 2D-section $g(x) = f(x, \frac{y_o}{x_o}x)$ shows.





- Standard example is evaluating $f(x) = (x - 1)^n$ or $g(x) = \sum_{k=0}^{n} \binom{n}{k}(-1)^{n-k}x^k$ with $f = g$ around 1.

Conclusion

- IMHO, Unum is a very convincing concept!
- There is Gustafsons Mathematica notebooks (open source, loyalty free: s.a. `https://www.crcpress.com/downloads/K23295/K23295_Downloads.zip`), a complete Unum Math library (in Mathematica).
- There are e.g. Python implementations, s.a. `https://github.com/jrmuizel/pyunum` and others `https://github.com/REX-Computing/unumjl`, `https://github.com/SFrijters/unum-d`, `https://github.com/tbreloff/Unums.jl`
- An implementation on some FPGA is near at hand.

References

[1] Bureau International des Poids et Mesures, BIPM: On the possible future revision of the International System of Units, the SI; Resolution 1 of the 24th CGPM (2011) www.bipm.org/en/CGPM/db/24/1/

[2] David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic; ACM Computing Surveys, March 1991 www.weblearn.hs-bremen.de/risse/RST/docs/goldberg.pdf

[3] John L. Gustafson: The End of Numerical Error; http://arith22.gforge.inria.fr/slides/06-gustafson.pdf

[4] John L. Gustafson: An Energy-Efficient and Massively Parallel Approach to Valid Numerics; www.slideshare.net/insideHPC/unum-computing-an-energy-efficient-and-massively-parallel-approach-to-valid-numerics

[5] John L. Gustafson: The End of Error – Unum Computing; CRC Press 2015

References'

[6] IEEE 754-1985 – IEEE Standard for Binary Floating-Point Arithmetic
http://ieeexplore.ieee.org/servlet/opac?punumber=2355

[7] IEEE 754-2008 – IEEE Standard for Binary Floating-Point Arithmetic
ieeexplore.ieee.org/servlet/opac?punumber=4610933

[8] about IEEE 754 http://grouper.ieee.org/groups/754/

[9] William Kahan: Further remarks on reducing truncation errors; Communications of the ACM, 8 (1):40, January 1965

[10] William Kahan: Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic; May 31, 1996,
www.cs.berkeley.edu/∼wkahan/ieee754status/ieee754.ps

[11] William Kahan: How JAVA's Floating-Point Hurts Everyone Everywhere; www.cs.berkeley.edu/∼wkahan/JAVAhurt.pdf

[12] William Kahan: Desperately Needed Remedies for the Undebuggability of Large Floating-Point Computations in Science and Engineering; www.eecs.berkeley.edu/∼wkahan/Boulder.pdf