On various ways to split a floating-point number

Claude-Pierre Jeannerod    Jean-Michel Muller    Paul Zimmermann
Inria, CNRS, ENS Lyon, Université de Lyon, Université de Lorraine
France

ARITH-25
June 2018
Splitting a floating-point number

\[ X = ? \]

All products are computed exactly with one FP multiplication (Dekker product)

\[ \sqrt{a^2 + b^2} \rightarrow 2^k \sqrt{\left( \frac{a}{2^k} \right)^2 + \left( \frac{b}{2^k} \right)^2} \]

Dekker product (1971)
Splitting a floating-point number

- In each "bin", the sum is computed exactly.

- Matlab program in a paper by Zielke and Drygalla (2003),

- analysed and improved by Rump, Ogita, and Oishi (2008),

- reproducible summation, by Demmel & Nguyen.

- absolute splittings (e.g., $\lfloor x \rfloor$), vs relative splittings (e.g., most significant bits, splitting of the significands for multiplication);

- no bit manipulations of the binary representations (would result in less portable programs) → only FP operations.
IEEE-754 compliant FP arithmetic with radix $\beta$, precision $p$, and extremal exponents $e_{\text{min}}$ and $e_{\text{max}}$;

$F$ = set of FP numbers. $x \in F$ can be written

$$x = \left( \frac{M}{\beta^{p-1}} \right) \cdot \beta^e,$$

$M, e \in \mathbb{Z}$, with $|M| < \beta^p$ and $e_{\text{min}} \leq e \leq e_{\text{max}}$, and $|M|$ maximum under these constraints;

significand of $x$: $M \cdot \beta^{-p+1}$;

RN = rounding to nearest with some given tie-breaking rule (assumed to be either “to even” or “to away”, as in IEEE 754-2008);
Notation and preliminary definitions

Definition 1 (classical ulp)

The unit in the last place of $t \in \mathbb{R}$ is

$$\text{ulp}(t) = \begin{cases} 
\beta^{\lfloor \log_\beta |t| \rfloor - p + 1} & \text{if } |t| \geq \beta^{e_{\text{min}}}, \\
\beta^{e_{\text{min}} - p + 1} & \text{otherwise}.
\end{cases}$$

Definition 2 (ufp)

The unit in the first place of $t \in \mathbb{R}$ is

$$\text{ufp}(t) = \begin{cases} 
\beta^{\lfloor \log_\beta |t| \rfloor} & \text{if } t \neq 0, \\
0 & \text{if } t = 0.
\end{cases}$$

(introduced by Rump, Ogita and Oishi in 2007)
Notation and preliminary definitions

- $x = 1.xxxxxxxxx \cdot 2^e$
- $ufp(x) = 1.00000000 \cdot 2^e$
- $ulp(x) = 0.00000001 \cdot 2^e$

Guiding thread of the talk: *catastrophic cancellation is your friend.*
Absolute splittings: 1. nearest integer

Uses a constant $C$. Same operations as Fast2Sum, yet different assumptions.

Algorithm 1

Require: $C, x \in \mathbb{F}$

$s \leftarrow \text{RN}(C + x)$

$x_h \leftarrow \text{RN}(s - C)$

$x_\ell \leftarrow \text{RN}(x - x_h)$ \{optional\}

return $x_h \{\text{or } (x_h, x_\ell)\}$

First occurrence we found: Hecker (1996) in radix 2 with $C = 2^{p-1}$ or $C = 2^{p-1} + 2^{p-2}$. Use of latter constant referred to as the 1.5 trick.

Theorem 3

Assume $C$ integer with $\beta^{p-1} \leq C \leq \beta^p$. If $\beta^{p-1} - C \leq x \leq \beta^p - C$, then $x_h$ is an integer such that $|x - x_h| \leq 1/2$. Furthermore, $x = x_h + x_\ell$. 
An interesting question is to compute \([x]\), or more generally \([x/\beta^k]\).

**Algorithm 2**

**Require:** \(x \in \mathbb{F}\)

- \(y \leftarrow \text{RN}(x - 0.5)\)
- \(C \leftarrow \text{RN}(\beta^p - x)\)
- \(s \leftarrow \text{RN}(C + y)\)
- \(x_h \leftarrow \text{RN}(s - C)\)

**return** \(x_h\)

**Theorem 4**

*Assume \(\beta\) is even, \(x \in \mathbb{F}\), \(0 \leq x \leq \beta^{p-1}\). Then Algorithm 2 returns \(x_h = [x]\).*
Relative splittings

- expressing a precision-$p$ FP number $x$ as the exact sum of a $(p - s)$-digit number $x_h$ and an $s$-digit number $x_\ell$;
- first use with $s = \lfloor p/2 \rfloor$ (Dekker product, 1971)
- another use: $s = p - 1 \rightarrow x_h$ is a power of $\beta$ giving the order of magnitude of $x$. Two uses:
  - evaluate $\text{ulp}(x)$ or $\text{ufl}(x)$. Useful functions in the error analysis of FP algorithms;
Relative splittings

- expressing a precision-$p$ FP number $x$ as the exact sum of a $(p - s)$-digit number $x_h$ and an $s$-digit number $x_\ell$;
- first use with $s = \lfloor p/2 \rfloor$ (Dekker product, 1971)
- another use: $s = p - 1 \rightarrow x_h$ is a power of $\beta$ giving the order of magnitude of $x$. Two uses:
  - evaluate $\text{ulp}(x)$ or $\text{ufp}(x)$. Useful functions in the error analysis of FP algorithms;
  
→ exact information
Relative splittings

- expressing a precision-$p$ FP number $x$ as the exact sum of a $(p - s)$-digit number $x_h$ and an $s$-digit number $x_\ell$;
- first use with $s = \lfloor p/2 \rfloor$ (Dekker product, 1971)
- another use: $s = p - 1 \rightarrow x_h$ is a power of $\beta$ giving the order of magnitude of $x$. Two uses:
  - evaluate $\text{ulp}(x)$ or $\text{ulf}(x)$. Useful functions in the error analysis of FP algorithms;
  - exact information
- power of $\beta$ close to $|x|$: for scaling $x$, such a weaker condition suffices, and can be satisfied using fewer operations.
Relative splittings

- expressing a precision-$p$ FP number $x$ as the **exact sum** of a $(p - s)$-digit number $x_h$ and an $s$-digit number $x_\ell$;
- first use with $s = \lfloor p/2 \rfloor$ (Dekker product, 1971)
- another use: $s = p - 1 \rightarrow x_h$ is a power of $\beta$ giving the **order of magnitude** of $x$. Two uses:
  - evaluate $\operatorname{ulp}(x)$ or $\operatorname{ufp}(x)$. Useful functions in the error analysis of FP algorithms;
  - power of $\beta$ close to $|x|$: for scaling $x$, such a weaker condition suffices, and can be satisfied using fewer operations.

→ **exact information**

→ **approximate information**
Veltkamp splitting

$x \in \mathbb{F}$ and $s < p \rightarrow$ two FP numbers $x_h$ and $x_\ell$ s.t. $x = x_h + x_\ell$, with the significand of $x_h$ fitting in $p - s$ digits, and the one of $x_\ell$ in $s$ digits ($s - 1$ when $\beta = 2$ and $s \geq 2$).

Algorithm 3 Veltkamp’s splitting.

Require: $C = \beta^s + 1$ and $x$ in $\mathbb{F}$

\[
\gamma \leftarrow \text{RN}(Cx) \\
\delta \leftarrow \text{RN}(x - \gamma) \\
x_h \leftarrow \text{RN}(\gamma + \delta) \\
x_\ell \leftarrow \text{RN}(x - x_h) \\
\text{return } (x_h, x_\ell)
\]

- Dekker (1971): radix 2 analysis, implicitly assuming no overflow;
- extended to any radix $\beta$ by Linnainmaa (1981);
- works correctly even in the presence of underflows;
- Boldo (2006): $Cx$ does not overflow $\Rightarrow$ no other operation overflows.

Remember: catastrophic cancellation is your friend!
Veltkamp splitting: FMA variant

If an FMA instruction is available, we suggest the following variant, that requires fewer operations.

Algorithm 4 FMA-based relative splitting.

Require: \( C = \beta^s + 1 \) and \( x \) in \( \mathbb{F} \)

\[
\begin{align*}
\gamma & \leftarrow \text{RN}(Cx) \\
\xh & \leftarrow \text{RN}(\gamma - \beta^s x) \\
\xel & \leftarrow \text{RN}(Cx - \gamma) \\
\text{return } & (\xh, \xel)
\end{align*}
\]

Remarks

- \( \xel \) obtained in parallel with \( \xh \)
- even without an FMA, \( \gamma \) and \( \beta^s x \) can be computed in parallel,
- the bounds on the numbers of digits of \( \xh \) and \( \xel \) given by Theorem 5 can be attained.

Theorem 5

Let \( x \in \mathbb{F} \) and \( s \in \mathbb{Z} \) s.t. \( 1 \leq s < p \). Barring underflow and overflow, Algorithm 4 computes \( \xh, \xel \in \mathbb{F} \) s.t. \( x = \xh + \xel \). If \( \beta = 2 \), the significands of \( \xh \) and \( \xel \) have at most \( p - s \) and \( s \) bits, respectively. If \( \beta > 2 \) then they have at most \( p - s + 1 \) and \( s + 1 \) digits, respectively.
Extracting a single bit (radix 2)

- computing \( \text{ufp}(x) \) or \( \text{ulp}(x) \), or scaling \( x \);
- Veltkamp’s splitting (Algorithm 3) to \( x \) with \( s = p - 1 \): the resulting \( x_h \) has a 1-bit significand and it is nearest \( x \) in precision \( p - s = 1 \).
- For computing \( \text{sign}(x) \cdot \text{ufp}(x) \), we can use the following algorithm, introduced by Rump (2009).

\[
\begin{align*}
\text{Algorithm 5} \\
\text{Require: } & \beta = 2, \varphi = 2^{p-1} + 1, \psi = 1 - 2^{-p}, \text{ and } x \in \mathbb{F} \\
q & \leftarrow \text{RN}(\varphi x) \\
r & \leftarrow \text{RN}(\psi q) \\
\delta & \leftarrow \text{RN}(q - r) \\
\text{return } & \delta
\end{align*}
\]

Very rough explanation:
- \( q \approx 2^{p-1}x + x \)
- \( r \approx 2^{p-1}x \)

\( q - r \approx x \) but in the massive cancellation we loose all bits but the most significant.
These solutions raise the following issues.

- If $|x|$ is large, then an overflow can occur in the first line of both Algorithms 3 and 5.
- To avoid overflow in Algorithm 5: scale it by replacing $\varphi$ by $\frac{1}{2} + 2^{-p}$ and returning $2^p \delta$ at the end. However, this variant will not work for subnormal $x$.

→ to use Algorithm 5, we somehow need to check the order of magnitude of $x$.

- If we are only interested in scaling $x$, then requiring the exact value of $\text{ufp}(x)$ is overkill: one can get a power of 2 “close” to $x$ with a cheaper algorithm.
Extracting a single bit (radix 2)

Algorithm 6 \text{sign}(x) \cdot \text{ulp}(x) \text{ for radix 2 and } |x| > 2^{e_{\text{min}}}.

Require: \( \beta = 2, \psi = 1 - 2^{-p} \), and \( x \in \mathbb{F} \)

\begin{align*}
    a & \leftarrow \text{RN}(\psi x) \\
    \delta & \leftarrow \text{RN}(x - a) \\
    \text{return} & \delta
\end{align*}

Theorem 6

If \( |x| > 2^{e_{\text{min}}} \), then Algorithm 6 returns

\[
\text{sign}(x) \cdot \begin{cases} 
    \frac{1}{2} \text{ulp}(x) & \text{if } |x| \text{ is a power of 2}, \\
    \text{ulp}(x) & \text{otherwise}.
\end{cases}
\]

Similar algorithm for \text{ufp}(x), under the condition \( |x| < 2^{e_{\text{max}}-p+1} \).
Underflow-safe and almost overflow-free scaling

- $\beta = 2$, $p \geq 4$;
- RN breaks ties “to even” or “to away”;

Given a nonzero FP number $x$, compute a scaling factor $\delta$ s.t.:
- $|x|/\delta$ is much above the underflow threshold and much below the overflow threshold (so that, for example, we can safely square it);
- $\delta$ is an integer power of 2 ($\rightarrow$ no rounding errors when multiplying or dividing by it).

Algorithms proposed just before: simple, but underflow or overflow can occur for many inputs $x$. 
Underflow-safe and almost overflow-free scaling

Following algorithm: underflow-safe and *almost* overflow-free in the sense that only the two extreme values $x = \pm(2 - 2^{1-p}) \cdot 2^{\text{emax}}$ must be excluded.

\begin{algorithm}
\textbf{Algorithm 7}

\textbf{Require:} $\beta = 2$, $\Phi = 2^{-p} + 2^{-2p+1}$, $\eta = 2^{\text{emin}-p+1}$, and $x \in \mathbb{F}$

\begin{align*}
y & \leftarrow |x| \\
e & \leftarrow \text{RN}(\Phi y + \eta) \\
y\sup & \leftarrow \text{RN}(y + e) \\
\delta & \leftarrow \text{RN}(y\sup - y) \\
\text{return} \ & \delta
\end{align*}
\end{algorithm}
Underflow-safe and almost overflow-free scaling

First 3 lines of Algorithm 7: algorithm due to Rump, Zimmermann, Boldo and Melquiond, that computes the FP successor of $x \not\in [2^{e_{\min}}, 2^{e_{\min}+2}]$. We have,

**Theorem 7**

For $x \in \mathbb{F}$ with $|x| \neq (2 - 2^{1-p}) \cdot 2^{e_{\max}}$, the value $\delta$ returned by Algorithm 7 satisfies:

- if RN is with “ties to even” then $\delta$ is a power of 2;
- if RN is with “ties to away” then $\delta$ is a power of 2, unless $|x| = 2^{e_{\min}+1} - 2^{e_{\min}-p+1}$, in which case it equals $3 \cdot 2^{e_{\min}-p+1}$;
- if $x \neq 0$, then

$$1 \leq \left| \frac{x}{\delta} \right| \leq 2^p - 1.$$

→ makes $\delta$ a good candidate for scaling $x$;
→ in the paper: application to $\sqrt{a^2 + b^2}$. 
## Experimental results

Although we considered floating-point operations only, we can compare with bit-manipulations.

The C programs we used are publicly available (see proceedings).

Experimental setup: Intel i5-4590 processor, Debian testing, GCC 7.3.0 with -O3 optimization level, FPU control set to rounding to double.

### Computation of round or floor:

<table>
<thead>
<tr>
<th>Method</th>
<th>Round</th>
<th>Floor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithms 1 and 2</td>
<td>0.106s</td>
<td>0.173s</td>
</tr>
<tr>
<td>Bit manipulation</td>
<td>0.302s</td>
<td>0.203s</td>
</tr>
<tr>
<td>GNU libm rint and floor</td>
<td>0.146s</td>
<td>0.209s</td>
</tr>
</tbody>
</table>

Note: Algorithms 1 and 2 require $|x| \leq 2^{51}$ and $0 \leq x \leq 2^{52}$ respectively.
## Relative splitting of a double-precision number

Splitting into $x_h$ and $x_\ell$:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$x_h$</th>
<th>$x_\ell$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 3</td>
<td>26 bits</td>
<td>26 bits</td>
<td>0.108s</td>
</tr>
<tr>
<td>Algorithm 4</td>
<td>26 bits</td>
<td>27 bits</td>
<td>0.106s</td>
</tr>
<tr>
<td>Algorithm 4 with FMA</td>
<td>26 bits</td>
<td>27 bits</td>
<td>0.108s</td>
</tr>
<tr>
<td>Bit manipulation</td>
<td>26 bits</td>
<td>27 bits</td>
<td>0.095s</td>
</tr>
</tbody>
</table>

Algorithms 3 and 4 assume no intermediate overflow or underflow.
Conclusion

- systematic review of splitting algorithms
- found some new algorithms, in particular with FMA
- many applications for absolute and relative splitting
- in their application range, these algorithms are competitive with (less-portable) bit-manipulation algorithms
Motivation

Question of Pierrick Gaudry (Caramba team, Nancy, France):

Multiple-precision integer arithmetic in Javascript.

Javascript has only a 32-bit integer type, but 53-bit doubles!

Storing 16-bit integers in a double precision register, we can accumulate up to $2^{21}$ products of 32 bits, and then have to perform $\text{floor}(x/65536.0)$ to normalize.

The Javascript code `Math.Floor(x/65536.0)` is slow on old internet browsers (Internet Explorer version 7 or 8)!

The Javascript standard says it is IEEE754, with always round to nearest, ties to even.

Pierrick Gaudry then opened the “Handbook of Floating-Point Arithmetic”...
First algorithm (designed by Pierrick Gaudry):

Assume $0 \leq x < 2^{36}$ and $x$ is an integer

We can compute $\text{floor}(x)$ as follows:

Let $C = 2^{36} - 2^{-1} + 2^{-17}$.

$s \leftarrow \text{RN}(C + x)$

Return $\text{RN}(s - C)$

Question: can we get rid of the condition “$x$ integer”?