Arithmétique flottante IEEE 754 et amélioration de la précision des calculs

Nicolas Louvet - UCB Lyon 1
LIP (UMR CNRS - ENS Lyon - UCB Lyon 1 - INRIA 5668)
EPI INRIA Arénaire

Qualité numérique des calculs flottants

Comment garantir la sûreté des logiciels numériques, ou en améliorer la qualité, tout en optimisant les performances ?

1. Certifier la qualité numérique du résultat calculé :
   - précision du résultat calculé, ou l'absence de dépassement de capacité
   - établir une preuve de la qualité du résultat
2. Améliorer la qualité numérique du résultat :
   - utilisation d'arithmétiques spécifiques : multi-précision
   - techniques de compensation à base de « transformations exactes »
3. Optimiser les performances des codes numériques :
   - exploiter au mieux les nouvelles architectures...
   - ...tout en certifiant la qualité numérique

Introduction

- Le standard IEEE 754-1985 :
  - largement répandu, assez bien supporté ;
  - permet l'implantation fiable, portable et efficace de programmes flottants.
- Objectifs de la révision du standard :
  - prendre en compte de nouvelles instructions : le Fused Multiply-Add (FMA) ;
  - formaliser les formats décimaux et de grandes précisions ;
  - prendre en compte des avancées : arrondi correct des fonctions élémentaires ;
  - corriger certaines ambiguïtés du standard de 1985.

  IEEE 754-2008 : apports et évolutions par rapport à la version précédente ?
- Comment tirer parti de l'arithmétique IEEE 754 pour améliorer la précision de programme flottants, tout en maintenant un faible surcoût ?
  - Comment utiliser le Fused Multiply-Add ?
  - Méthode proposée : compensation des erreurs d'arrondi.
  - Exemple de l'évaluation de polynômes.

Révision de la norme IEEE-754

Nombres flottants

Un format flottant est en particulier[1] caractérisé par :

- sa base $\beta \in \mathbf{N}$, $\beta \geq 2$,
- sa précision $p \in \mathbf{N}$, $p \geq 2$,
- deux exposants extrêmes $e_{\min}, e_{\max} \in \mathbf{Z}$, $emin < 0 < e_{\max}$.

La norme IEEE 754-2008 prévoit $\beta \in \{2, 10\}$, et fixe $e_{\min} = 1 - e_{\max}$.

Un nombre à virgule flottante dans ce format est de la forme :

$$x = (-1)^s \times \underbrace{(x_0, x_1 x_2 \ldots x_{p-1})_\beta}_{m} \times \beta^e,$$

- m est la mantisse de x,
- $e \in \mathbf{Z}$, $e_{\min} \leq e \leq e_{\max}$ est son exposant,
- $s \in \{0, 1\}$ détermine son signe.

[1] Il reste à définir les valeurs spéciales et l'encodage.

Représentation normalisée : représentation de x dont l'exposant est minimal.

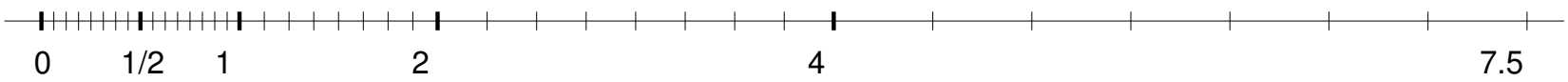
- Nombre flottant normal : $1 \leq m < \beta$, *i.e.*, x s'écrit

$$x = \pm(x_0, x_1 x_2 \dots x_{p-1})_\beta \times \beta^e, \quad \text{avec} \quad x_0 \neq 0.$$

- Nombre flottant sous-normal : $m < 1$ et $e = e_{\min}$, x s'écrit

$$x = \pm(0, x_1 x_2 \dots x_{p-1})_\beta \times \beta^{e_{\min}}.$$

Exemple avec $\beta = 2$, $p = 4$, $e_{\min} = -1$ et $e_{\max} = 2$:



A l'ensemble des nombre flottants, sont adjointes des valeurs spéciales :

- $-\infty$ et $+\infty$,
- « *Not a Number* » (NaN) : pour indiquer une opération invalide.

Formats prévus par la norme

La norme spécifie le codage binaire de formats d'échanges :

- flottants binaires sur 16, 32, 64 ou k bits ($k \geq 128$ multiple de 32) : binary$k$
- flottants décimaux codés k bits (k multiple de 32) : decimal$k$.

Des formats basiques sont définis, qui sont tous des formats d'échanges :

nom	binary32 (single)	binary64 (double)	binary128	decimal64	decimal128
p	24	53	113	16	34
$e_{\min}$	127	1023	16383	384	6144
$e_{\max}$	-126	-1022	-16382	-383	-6143

Un environnement conforme doit implanter au moins un de ces formats basiques.

La norme recommande des formats étendus, dont le codage n'est pas spécifié : « *extended* and *extendable* precision format ».

Attributs

Notion d'attribut d'un bloc de code, attaché à ce bloc pour en spécifier les propriétés numériques ou le comportement vis-à-vis des exceptions.

Attributs obligatoires :

- *rounding-direction attributes*.

Deux exemples d'attributs recommandés :

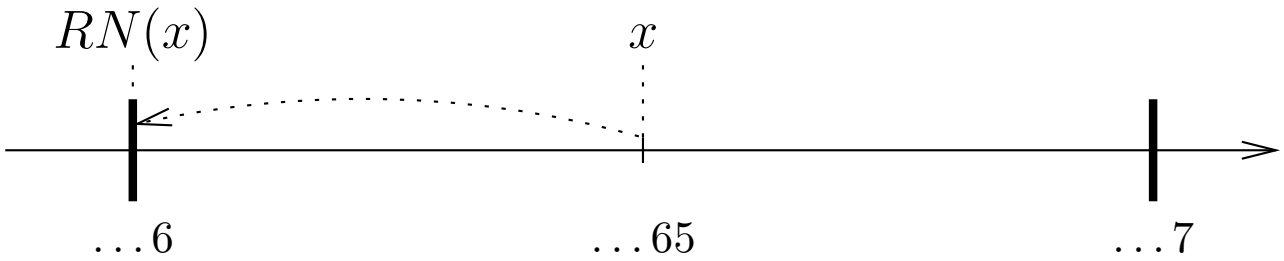
- *preferred width attribute*,
- *value-changing optimization attribute*.

Le standard précise que les attributs d'un bloc de code :

- doivent pouvoir être fixés à la compilation (directives de compilation),
- devraient pouvoir être modifiés à l'exécution (appels de fonction).

Rounding-direction attributes

IEEE 754-1985	IEEE 754-2008	imposé
directed rounding mode	**directed rounding attributes**	
rounding toward $+\infty$	roundTowardPositive	$\times$
rounding toward $-\infty$	roundTowardNegative	$\times$
rounding toward 0	roundTowardZero	$\times$
round-to-nearest mode	**attribute to nearest**	
round-to-nearest-even	roundTiesToEven	$\times$



- Utilisation de roundTiesEven comme attribut par défaut :
  $\hookrightarrow$ imposée en binaire, recommandée en décimal.
- roundTiesToAway est obligatoire en décimal : « pour la finance. »

Preferred width attributes : problématique

```
int main(void) {
  double a = 1848874847.0;
  double b = 19954562207.0;
  double c;
  c = a*b;
  printf("c=%20.19\n", c);
}
```

Avec GCC 4.1.2 20061115 sur IA 32 bits :

sans directives	c=3.689348814741910323 2e+19
-mfpmath=387	c=3.689348814741910323 2e+19
-mfpmath=sse	c=3.689348814741911142 4e+19

Preferred width attributes : problématique

```
int main(void) {
  double a = 1848874847.0;
  double b = 19954562207.0;
  double c;
  c = a*b;
  printf("c=%20.19\n", c);
}
```

Avec GCC 4.1.2 20061115 sur IA 32 bits :

sans directives	c=3.689348814741910323 2e+19
-mfpmath=387	c=3.689348814741910323 2e+19
-mfpmath=sse	c=3.689348814741911142 4e+19

Preferred width attributes : problématique

```
int main(void) {
  double a = 1848874847.0;
  double b = 19954562207.0;
  double c;
  c = a*b;
  printf("c=%20.19\n", c);
}
```

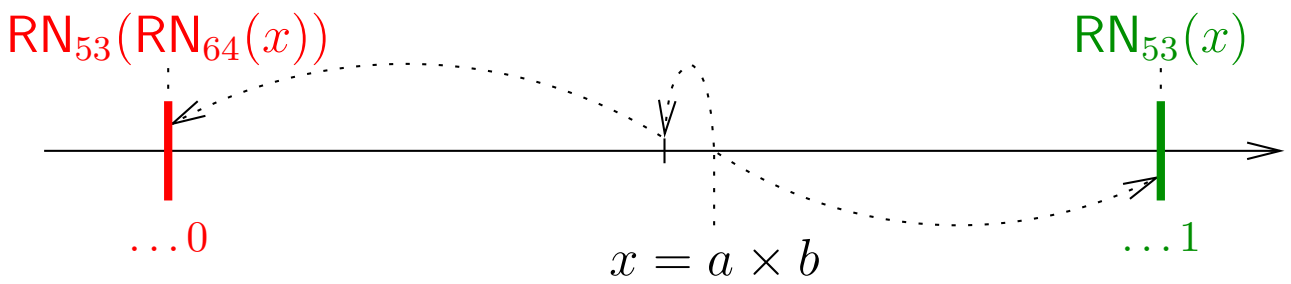
Avec GCC 4.1.2 20061115 sur IA 32 bits :

sans directives	c=3.689348814741910323**2**e+19
-mfpmath=387	c=3.689348814741910323**2**e+19
-mfpmath=sse	c=3.689348814741911142**4**e+19

`-mfpmath=387` : calcul sur x87 avec 64 bits de précision, puis arrondi à 53 bits.

`-mfpmath=sse` : calcul sur SSE, avec 53 bits de précision.

Problème typique de double arrondi :



Preferred width attributes

Si on considère une expression comme

$$((a+b)\times c)+(d+e))\times f\,,$$

le résultat calculé dépend du format utilisé pour les calculs intermédiaires.

Quel format doit être utilisé pour les résultats intermédiaires de l'évaluation d'une expression arithmétique ?

Le standard recommande deux attributs :

- preferredWidthNone – Le format de destination d'une opération est le format de taille maximale parmi ceux des opérandes.
- preferredWidthFormat – L'utilisateur peut spécifier un format « préféré » pour un bloc de code. Le format de destination est celui de taille maximale entre le format préféré et ceux des opérandes.

Arithmétique

Un environnement conforme doit fournir :

- les opérations de conversion entre les différents formats qu'il propose,
- les opérations arithmétiques élémentaires : $+, -, \times, /$ et $\sqrt{\cdot}$,
- l'opération Fused Multiply-Add (FMA) :

$$\mathrm{FMA}(x, y, z) = \circ(x \times y + z), \quad \text{avec un seul arrondi.}$$

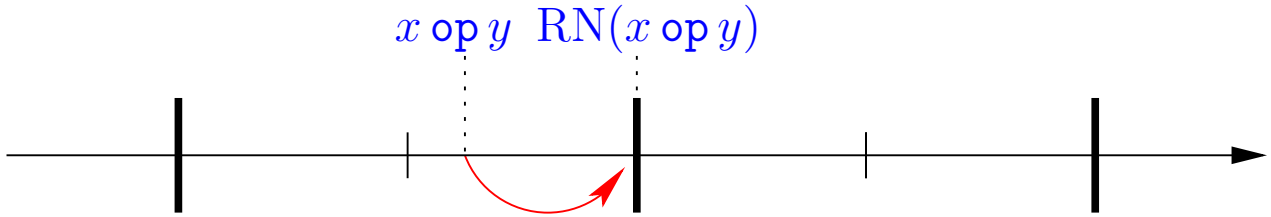
Toutes ces opérations doivent être implantées avec arrondi correct :
le résultat calculé est le résultat exact arrondi selon l'attribut d'arrondi.

L'arrondi correct est recommandé pour les fonctions élémentaires :
e^x, $e^x - 1$, 2^x, $2^x - 1$, 10^x, $10^x - 1$ $\ln(x)$, $\ln(1 + x)$, $\log_2(x)$, $\log_2(1 + x)$,
$\log_{10}(x)$, $\log_{10}(1 + x)$, $\sqrt{x^2 + y^2}$, $1/\sqrt{x}$, $(1 + x)^n$, x^n, $x^{1/n}$, $\sin(x)$, $\cos(x)$ ...

Modélisation de l'arithmétique

Soient $a, b \in \mathbf{F}$ and $\text{op} \in \{+, -, \times, /\}$ une opération arithmétique.

- Le résultat de chaque opération doit être arrondi correctement.



Chaque opération peut être entachée d'une erreur d'arrondi.

- Modèle standard de l'arithmétique flottante :

$$\text{RN}(a \,\text{op}\, b) = (1 + \varepsilon)(a \,\text{op}\, b), \quad \text{avec} \quad |\varepsilon| \leq \mathbf{u},$$

où $\mathbf{u} = \frac{\beta^{1-p}}{2}$ est l'unité d'arrondi (en arrondi au plus proche).

Fused Multiply-Add

Avantages :

- divise le nombre d'opérations arithmétiques et d'erreurs d'arrondi par deux,
- permet de mettre au point des algorithmes pour améliorer la précision.

Inconvénients :

- introduit des ambiguités quant à l'ordre d'évaluation,
- détruit certaines propriétés numériques.

Ex : Le discriminant de $ax^2 - 2bx + c = 0$ peut s'évaluer de trois façons,

$$\Delta_1 = \mathrm{RN}(\mathrm{RN}(b^2) - \mathrm{RN}(ac)), \quad \Delta_2 = \mathrm{RN}(\mathrm{RN}(b^2) - ac), \quad \Delta_3 = \mathrm{RN}(b^2 - \mathrm{RN}(ac)).$$

Supposons $b^2 \geq ac$:

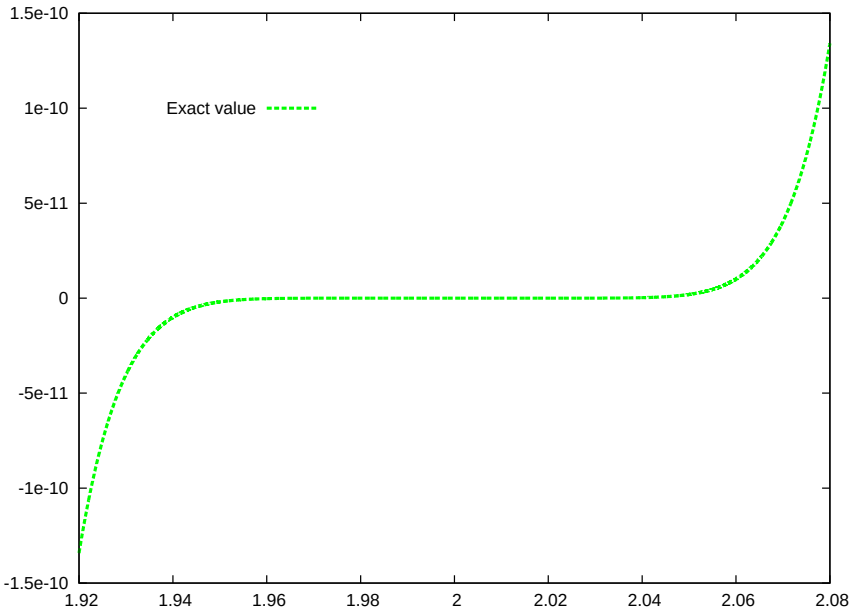
- Par monotonie de l'arrondi, $b^2 \geq ac \Rightarrow \mathrm{RN}(b^2) \geq \mathrm{RN}(ac) \Rightarrow \Delta_1 \geq 0$.
- On peut avoir $\Delta_2 < 0$, par exemple si $b^2 = ac$ et $\mathrm{RN}(b^2) < b^2$!

Amélioration de la précision du résultat

Exemple : évaluation de polynômes

Evaluation de polynômes univariés à coefficients flottants :

- l'évaluation peut être entachée d'erreurs d'arrondis,
- notamment au voisinage d'une racine multiple.

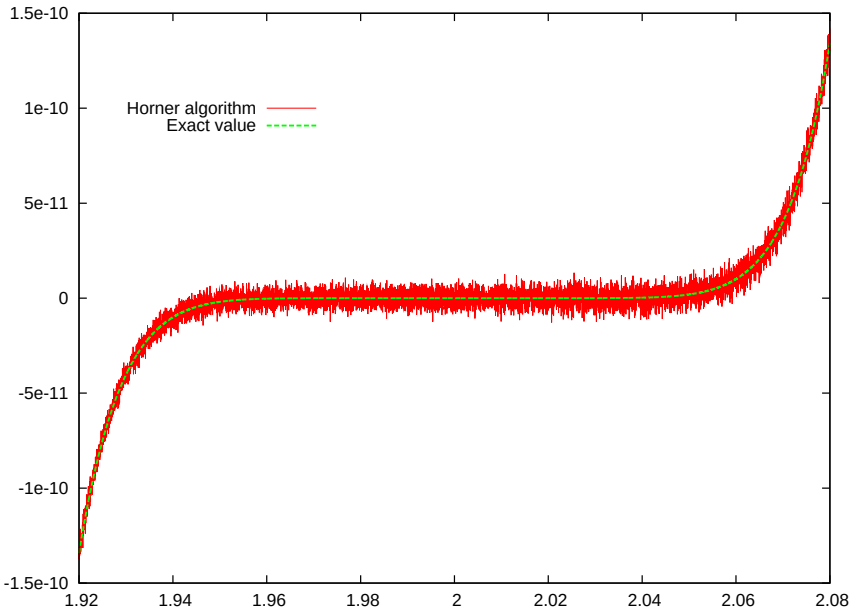


$p(x) = (x-2)^9$ sous forme développée au voisinage de la racine $x = 2$.

Exemple : évaluation de polynômes

Evaluation de polynômes univariés à coefficients flottants :

- l'évaluation peut être entachée d'erreurs d'arrondis,
- notamment au voisinage d'une racine multiple.

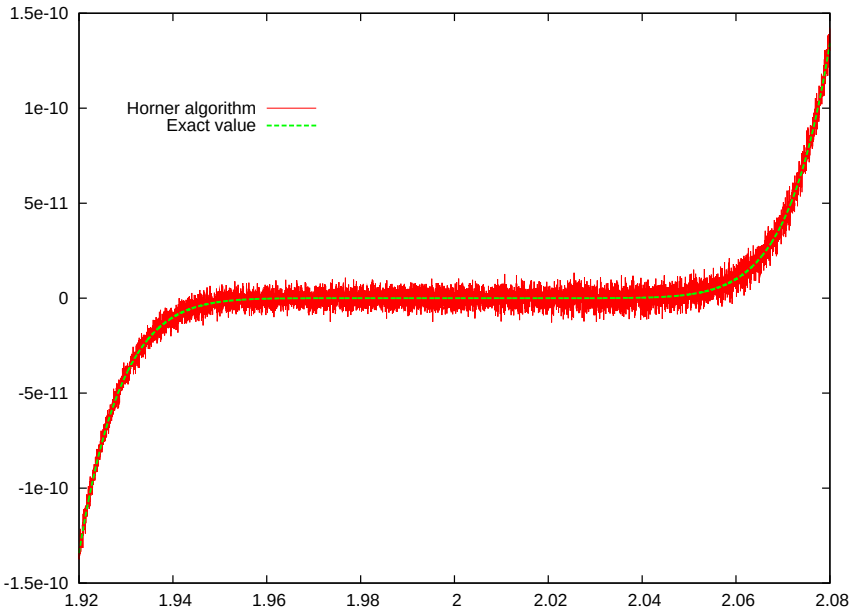


$p(x) = (x-2)^9$ sous forme développée au voisinage de la racine $x = 2$.
Evaluation avec le schéma de Horner en double précision IEEE.

Exemple : évaluation de polynômes

Evaluation de polynômes univariés à coefficients flottants :

- l'évaluation peut être entachée d'erreurs d'arrondis,
- notamment au voisinage d'une racine multiple.



$p(x) = (x-2)^9$ sous forme développée au voisinage de la racine $x = 2$.
Evaluation avec le schéma de Horner en double précision IEEE.

Dans cette partie :

- arithmétique IEEE 754 binaire,
- arrondi au plus proche,
- pas de dépassement de capacité.

Que peut apporter le FMA ?

Soit p le polynôme $p(x) = \sum_{i=0}^{n} a_i x^i$, avec $a_i \in \mathbf{F}$, $x \in \mathbf{F}$.

Horner sans FMA

function $r_0 = \text{Horner}(p, x)$

$h_n = a_n$

for $i = n-1 : -1 : 0$

$r_i = r_{i+1} \otimes x \oplus a_i$

Horner avec FMA

function $r_0 = \text{HornerFMA}(p, x)$

$h_n = a_n$

for $i = n-1 : -1 : 0$

$r_i = \text{FMA}(r_{i+1}, x, a_i)$

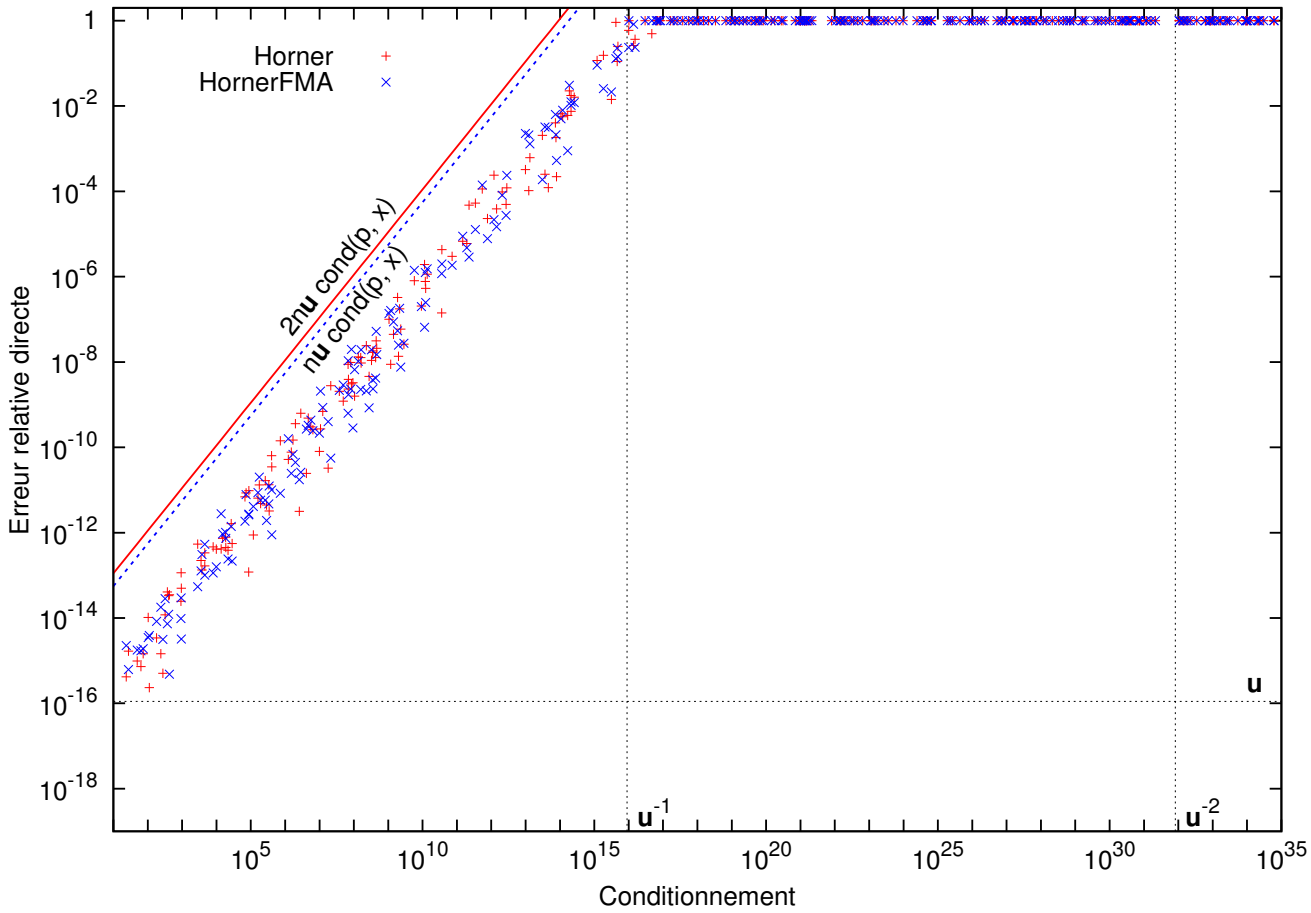
Majoration de l'erreur relative de l'évaluation :

$$\frac{|\text{Horner}(p,x) - p(x)|}{|p(x)|} \lesssim \begin{array}{ll} 2n\mathbf{u} \times \text{cond}(p,x) & \text{sans FMA} \\ n\mathbf{u} \times \text{cond}(p,x) & \text{avec FMA} \end{array}$$

$\text{cond}(p,x)$ est le conditionnement de l'évaluation : $\text{cond}(p,x) = \frac{\sum |a_i x^i|}{|p(x)|} \geq 1$.

Expérience numérique

En double précision IEEE 754, avec des polynômes de degré 50 :



Aucune différence de précision observable entre Horner et HornerFMA. . .

Comment améliorer la précision du résultat ?

- Le conditionnement d'un problème mesure sa sensibilité à des perturbations de ses entrées. Règle pour estimer la précision du résultat calculé :

$$\text{précision du résultat} \lesssim \text{conditionnement} \times \mathbf{u}.$$

Comment améliorer la précision du résultat ?

- Le conditionnement d'un problème mesure sa sensibilité à des perturbations de ses entrées. Règle pour estimer la précision du résultat calculé :

$$\text{précision du résultat} \lesssim \text{conditionnement} \times \mathbf{u}.$$

- Solution pour améliorer la précision : augmenter la précision de travail.

  En matériel :
  - précision étendue, par exemple les registres 80 bits de la fpu x87.

  De façon logicielle :
  - utilisation de bibliothèques en précision arbitraire (précision de travail fixée par le programmeur) : MP, MPFUN/ARPREC, MPFR.
  - expansions de longueur fixe : double-double ($\mathbf{u}^2$), quad-double ($\mathbf{u}^4$) (Briggs, Bailey *et al.*).

Comment améliorer la précision du résultat ?

- Le conditionnement d'un problème mesure sa sensibilité à des perturbations de ses entrées. Règle pour estimer la précision du résultat calculé :

$$\text{précision du résultat} \lesssim \text{conditionnement} \times \mathbf{u}.$$

- Solution pour améliorer la précision : augmenter la précision de travail.
  En matériel :
  - précision étendue, par exemple les registres 80 bits de la fpu x87.

  De façon logicielle :
  - utilisation de bibliothèques en précision arbitraire (précision de travail fixée par le programmeur) : MP, MPFUN/ARPREC, MPFR.
  - expansions de longueur fixe : double-double ($\mathbf{u}^2$), quad-double ($\mathbf{u}^4$) (Briggs, Bailey *et al.*).

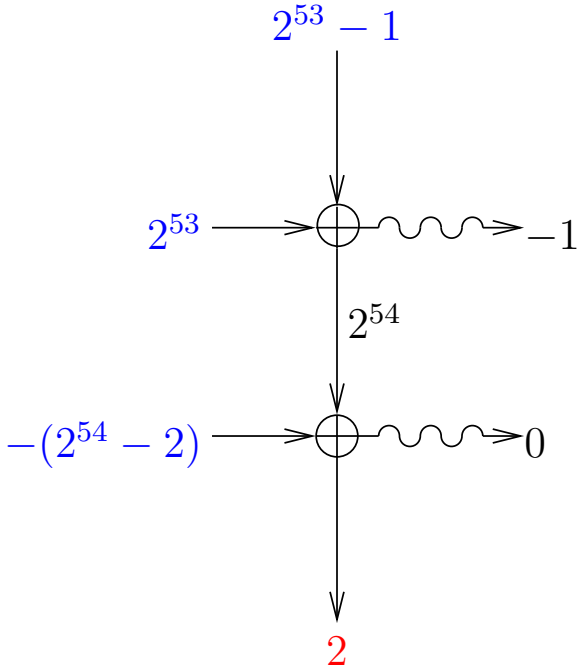
- Algorithmes compensés : correction des erreurs d'arrondi générées.

Exemple : sommation compensée

Flottants binary64 : $x_1 = 2^{53} - 1$, $x_2 = 2^{53}$ and $x_3 = -(2^{54} - 2)$.

Somme exacte : $x_1 + x_2 + x_3 = 1$.

Sommation classique



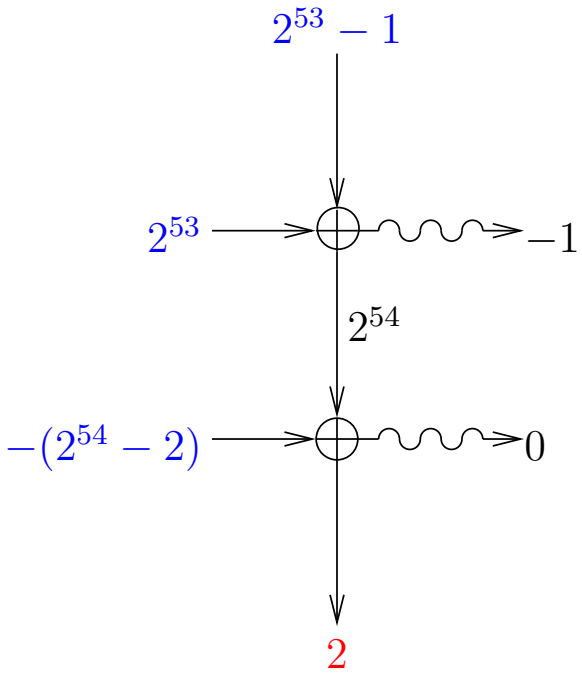
Erreur relative = 1

Exemple : sommation compensée

Flottants binary64 : $x_1 = 2^{53} - 1$, $x_2 = 2^{53}$ and $x_3 = -(2^{54} - 2)$.

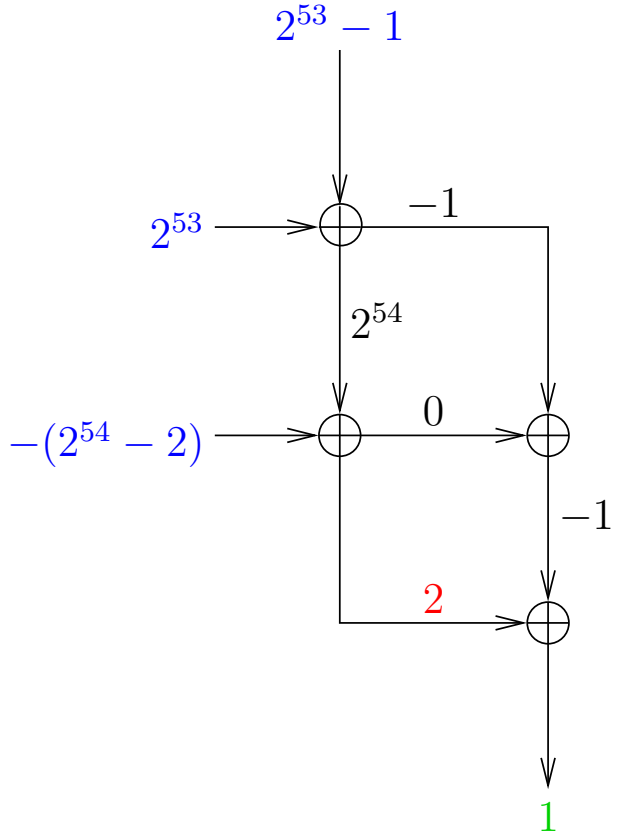
Somme exacte : $x_1 + x_2 + x_3 = 1$.

Sommation classique



Erreur relative = 1

Compensation des erreurs d'arrondi



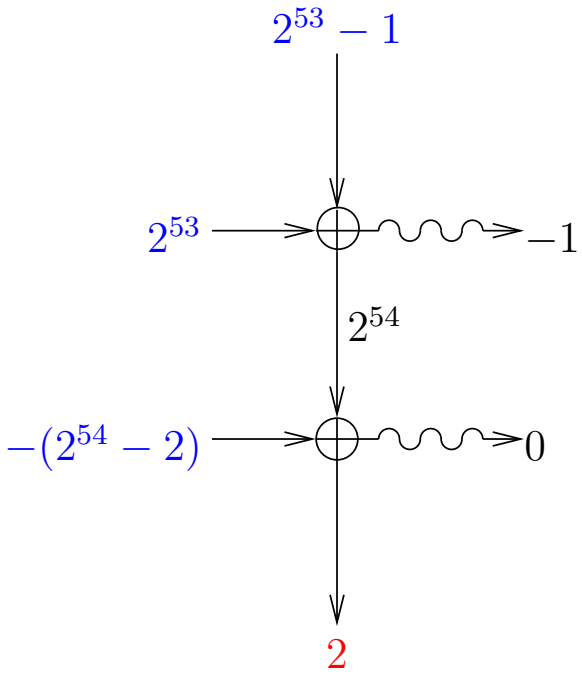
Le résultat exact est calculé

Exemple : sommation compensée

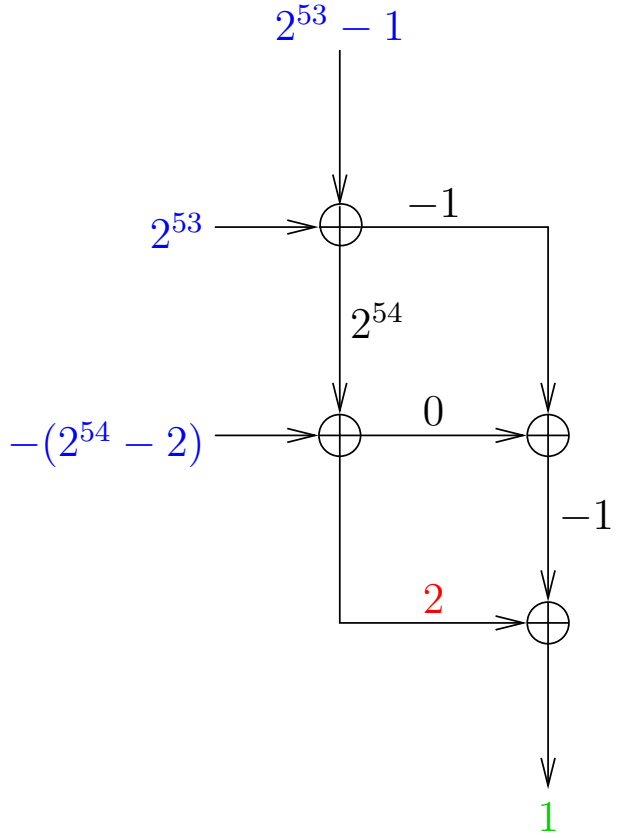
Flottants binary64 : $x_1 = 2^{53} - 1$, $x_2 = 2^{53}$ and $x_3 = -(2^{54} - 2)$.

Somme exacte : $x_1 + x_2 + x_3 = 1$.

Sommation classique



Compensation des erreurs d'arrondi



Erreur relative = 1

Le résultat exact est calculé

Ces erreurs d'arrondi sont calculées grâce aux transformations exactes.

Transformations exactes

Les transformations exactes sont des algorithmes pour calculer les erreurs d'arrondi générées en n'utilisant que la précision de travail.

$+$	$(x, y) = \mathit{Fast2Sum}(a, b)$ avec $\lvert a\rvert \geq \lvert b\rvert$ t. q. $x = a \oplus b$ and $a + b = x + y$	3 flops	Dekker (71)
$+$	$(x, y) = \mathit{2Sum}(a, b)$ t. q. $x = a \oplus b$ and $a + b = x + y$	6 flops	Knuth (74)
$\times$	$(x, y) = \mathit{2Prod}(a, b)$ t. q. $x = a \otimes b$ and $a \times b = x + y$	17 flops	Dekker (71)

pour $a, b, x, y \in \mathbf{F}$.

Addition exacte : Fast2Sum

Soient $a, b \in \mathbf{F}$.

Fast2Sum (Dekker, 71)

Hypothèse : $|a| \geq |b|$.

function $(x, y) = \text{Fast2Sum}(a, b)$

$x = a \oplus b$

$z = x \ominus a$

$y = b \ominus z$

2Sum ne requiert que 3 opérations flottantes, mais le tri des entrées peut être coûteux sur les machines pipelinées utilisant la prédiction de branchement. . .

Addition exacte : 2Sum

On utilise donc généralement l'algorithme 2Sum :

Algorithm 2Sum

function $[x, y] = 2\text{Sum}(a, b)$

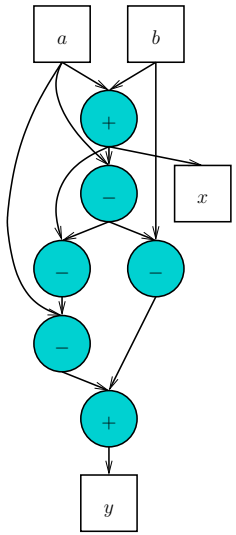
$x = a \oplus b$

$b' = x \ominus a$

$a' = s \ominus b'$

$\delta_a = a \ominus a' \quad \delta_b = b \ominus b'$

$y = \delta_a \oplus \delta_b$



2Sum est un algorithme qui n'effectue que des additions/soustractions et sans branchement, de profondeur 5 avec 6 opérations.

Existe-t-il d'autres algorithmes donnant le même résultat que 2Sum ?

L'algorithme 2Sum est minimal

Théorème (KLLM :2010)

En arith. flottante binaire ($p \geq 2$), parmi tous les algorithmes n'effectuant que des additions/soustractions et calculant le même résultat que 2Sum,

- chacun requiert au moins 6 opérations ;
- chaque algorithmes avec 6 opérations se réduit à 2Sum ;
- chacun a une profondeur d'au moins 5.

Obtenu en énumérant tous les algorithmes d'au plus 6 opérations.

2Sum est donc minimal parmi les algorithmes sans branchement, n'effectuant que des additions/soustractions.

Multiplication exacte

Calcul de $x, y \in \mathbf{F}$ tels que $x = a \otimes b$ et $x + y = a \times b$.

2Prod (Veltkamp et Dekker, 71)

function $[x, y] = \mathrm{2Prod}(a, b)$

% $C = 2^s + 1$ constante pré-calculée

$z_a = C \otimes a\,;$ $\quad z_b = C \otimes b$

$a_h = z_a \ominus (z_a \ominus a)$ $\quad b_h = z_b \ominus (z_b \ominus b)$

$a_\ell = a \ominus a_h$ $\quad b_\ell = b \ominus b_h$

$x = a \otimes b$

$y = a_\ell \otimes b_\ell \ominus (((x \ominus a_h \otimes b_h) \ominus a_\ell \otimes b_h) \ominus a_h \otimes b_\ell))$

opérations : 17

profondeur : 8

Multiplication exacte

Calcul de $x, y \in \mathbf{F}$ tels que $x = a \otimes b$ et $x + y = a \times b$.

2Prod (Veltkamp et Dekker, 71)

function $[x, y] = \mathrm{2Prod}(a, b)$

% $C = 2^s + 1$ constante pré-calculée

$z_a = C \otimes a$; $z_b = C \otimes b$

$a_h = z_a \ominus (z_a \ominus a)$ $b_h = z_b \ominus (z_b \ominus b)$

$a_\ell = a \ominus a_h$ $b_\ell = b \ominus b_h$

$x = a \otimes b$

$y = a_\ell \otimes b_\ell \ominus (((x \ominus a_h \otimes b_h) \ominus a_\ell \otimes b_h) \ominus a_h \otimes b_\ell))$

opérations : 17

profondeur : 8

Mais on peut faire plus efficace avec un FMA :

2ProdFMA

function $[x, y] = \mathrm{2ProdFMA}(a, b)$

$x = a \otimes b$

$y = \mathrm{FMA}(a, b, -x)$ % $y = a \times b - x$

opérations : 2

profondeur : 2

Transformation exacte pour Horner

Soit $p(x) = \sum_{i=0}^{n} a_i x^i$ de degré n, $a_i, x \in \mathbf{F}$.

Algorithme (Horner)

function $r_0 = \text{Horner}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$p_i = r_{i+1} \otimes x$ % erreur $\pi_i \in \mathbf{F}$

$r_i = p_i \oplus a_i$ % erreur $\sigma_i \in \mathbf{F}$

end

On définit p_π et p_σ tels que :

$$p_\pi(x) = \sum_{i=0}^{n-1} \pi_i x^i \quad \text{et} \quad p_\sigma(x) = \sum_{i=0}^{n-1} \sigma_i x^i$$

Transformation exacte pour Horner

Soit $p(x) = \sum_{i=0}^{n} a_i x^i$ de degré n, $a_i, x \in \mathbf{F}$.

Algorithme (Horner)

function $r_0 = \text{Horner}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$p_i = r_{i+1} \otimes x$ % erreur $\pi_i \in \mathbf{F}$

$r_i = p_i \oplus a_i$ % erreur $\sigma_i \in \mathbf{F}$

end

On définit p_π et p_σ tels que :

$$p_\pi(x) = \sum_{i=0}^{n-1} \pi_i x^i \quad \text{and} \quad p_\sigma(x) = \sum_{i=0}^{n-1} \sigma_i x^i$$

Théorème

$$\underbrace{p(x)}_{\text{exact value}} = \underbrace{\text{Horner}(p, x)}_{\in \mathbf{F}} + \underbrace{(p_\pi + p_\sigma)(x)}_{\text{erreur directe}}.$$

Transformation exacte pour Horner

Soit $p(x) = \sum_{i=0}^{n} a_i x^i$ de degré n, $a_i, x \in \mathbf{F}$.

Algorithme (Horner)

function $r_0 = \text{Horner}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$p_i = r_{i+1} \otimes x$ % err. $\pi_i \in \mathbf{F}$

$r_i = p_i \oplus a_i$ % err. $\sigma_i \in \mathbf{F}$

end

Algorithme (EFT for Horner)

function $[r_0, p_\pi, p_\sigma] = \text{EFTHornerFMA}(p, x)$

$r_n = a_n$

for $i = n-1 : -1 : 0$

$[p_i, \pi_i] = \text{2ProdFMA}(r_{i+1}, x)$

$[r_i, \sigma_i] = \text{2Sum}(p_i, a_i)$

$p_\pi[i] = \pi_i \quad p_\sigma[i] = \sigma_i$

end

Théorème

$$\underbrace{p(x)}_{\text{exact value}} = \underbrace{\text{Horner}(p, x)}_{\in \mathbf{F}} + \underbrace{(p_\pi + p_\sigma)(x)}_{\text{erreur directe}}.$$

Schéma de Horner compensé

$(p_\pi + p_\sigma)(x)$ est exactement l'erreur directe affectant Horner(p, x).

$\Rightarrow$ on calcule une valeur approchée de $(p_\pi + p_\sigma)(x)$ comme terme correctif.

Algorithme (Schéma de Horner compensé)

function $\overline{r} = \text{CompHornerFMA}(p, x)$

$[\widehat{r}, p_\pi, p_\sigma] = \text{EFTHornerFMA}(p, x)$ % $\widehat{r} = \text{Horner}(p, x)$

$\widehat{c} = \text{HornerFMA}(p_\pi \oplus p_\sigma, x)$

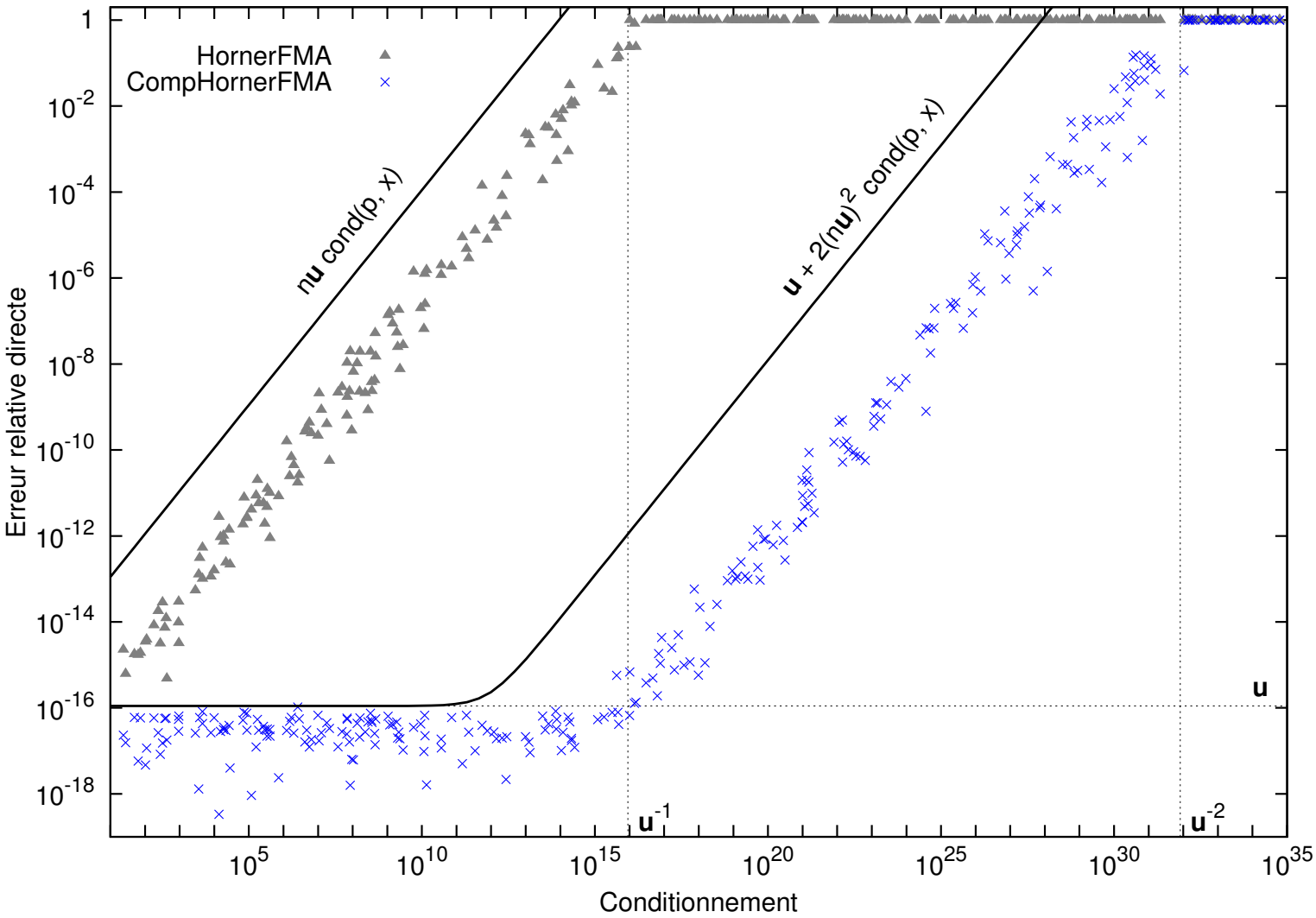
$\overline{r} = \widehat{r} \oplus \widehat{c}$

Théorème

Si p est un polynôme à coefficient flottants, et x un flottant,

$$\frac{|\text{CompHornerFMA}(p, x) - p(x)|}{|p(x)|} \leq \mathbf{u} + 2(n\mathbf{u})^2 \text{cond}(p, x) + \mathcal{O}(\mathbf{u}^3).$$

erreur relative $\lesssim \mathbf{u} + \mathbf{u}^2 \times$ conditionnement



Résultat aussi précis que s'il avait été obtenu par le schéma de Horner classique en précision doublée, avec un arrondi final vers la précision courante.

Implantation

CompHornerFMA

```
double CompHornerFMA(double *P, int n, double x) {
  double p, r, c, t, pi, sig, min, max;
  int i;
  for(i=n-1; i>=0; i--) {
    /* TwoProdFMA(p, pi, s, x); */
    p = r * x;
    pi = FMS(r, x, p);
    /* 2Sum(r, sigma, p, P[i]); */
    r = p + P[i];
    t = r - p;
    sig = (p - (r - t)) + (P[i] - t);
    /* Computation of the error term */
    c = FMA(c, x, pi+sig);
  }
  return(r+c);
}
```

$\hookrightarrow 10n + 1$ opérations.

Implantation

CompHornerFMA

```
double CompHornerFMA(double *P, int n, double x) {
  double p, r, c, t, pi, sig, min, max;
  int i;
  for(i=n-1; i>=0; i--) {
    /* TwoProdFMA(p, pi, s, x); */
    p = r * x;
    pi = FMS(r, x, p);
    /* 2Sum(r, sigma, p, P[i]); */
    r = p + P[i];
    t = r - p;
    sig = (p - (r - t)) + (P[i] - t);
    /* Computation of the error term */
    c = FMA(c, x, pi+sig);
  }
  return(r+c);
}
```

$\hookrightarrow$ $10n + 1$ opérations.

CompHorner

```
double CompHorner(double *P, int n, double x) {
  double p, r, c, pi, sig;
  double x_hi, x_lo, hi, lo, t;
  int i;
  /* Split(x_hi, x_lo, x) */
  t = x * _splitter_;
  x_hi = t - (t - x); x_lo = x - x_hi;
  r = P[n]; c = 0.0;
  for(i=n-1; i>=0; i--) {
    /* TwoProd(p, pi, s, x); */
    p = r * x;
    t = r * _splitter_;
    hi = t - (t - r);
    lo = r - hi;
    pi = (((hi*x_hi - p) + hi*x_lo)
          + lo*x_hi) + lo*x_lo;
    /* TwoSum(s, sigma, p, P[i]); */
    r = p + P[i];
    t = r - p;
    sig = (p - (r - t)) + (P[i] - t);
    /* Computation of the error term */
    c = c * x + (pi+sig);
  }
  return(r+c);
}
```

$\hookrightarrow$ $22n + 5$ opérations.

Performances pratiques

Décomptes d'opérations flottantes :

Algorithme	Nombre d'opérations	« surcoût »
HornerFMA	n	1
CompHornerFMA	$10n + \mathcal{O}(1)$	10
DDHorner avec FMA	$16n + \mathcal{O}(1)$	16

Surcoûts mesurés[2] :

environnement	$\frac{CompHornerFMA}{HornerFMA}$	$\frac{DDHorner}{HornerFMA}$
Itanium 2, ICC 9.1	1.5	5.9
Itanium 2, GCC 4.1.1	2.8	6.7

[2] jeu de 39 polynômes, degré de 10 à 200 par pas de 5, surcoût moyen

Conclusion

Standard IEEE 754-2008 :

- cadre pour l'implantation de programmes flottants binaires et décimaux ;
- standardise une opération importante : le Fused Mutiply-Add ;
- arrondi correct : permet de prouver des propriétés, améliore la portabilité ;
- les recommandations seront-elles suivient d'effets ?

Amélioration de la précision des calculs et FMA :

- Le FMA n'améliore pas toujours la précision...
- ...mais donne une transformation exacte efficace pour le produit !
- La compensation des arrondis donne des algos plus précis efficaces :
  - sommation : Kahan, Møller (1965), Pichat (1972), Neumanier (1974), Priest (1992), Ogita-Rump-Oishi (2005).
  - produits scalaires : Ogita-Rump-Oishi (2005).
  - évaluation polynomiale : Graillat-Langlois-Louvet (2009)