Introduction à l'arithmétique compensée

Valérie Ménissier-Morain

Université Pierre et Marie Curie - Paris 6
LIP6 - Département CALSCI

RAIM 2012

Un constat simple

Le produit

Le produit de deux nombres réels de N chiffres de mantisse tient sur $2N$ chiffres de mantisse

$$\begin{array}{rll} & 1.x_2x_3\cdots x_N & 2^{e_x} \\ \times & 1.y_2y_3\cdots y_N & 2^{e_y} \\ \hline 1.z_2z_3\cdots z_N \underbrace{0\cdots 0}_{k-1} 1 z_{N+k+1}\cdots z_{2N} & & 2^{e_x+e_y} \end{array}$$

(valeurs absolues)

Un constat simple

Le produit

Le produit de deux nombres réels de N chiffres de mantisse tient sur $2N$ chiffres de mantisse

$$\begin{array}{rr} & 1.x_2x_3\cdots x_N \quad 2^{e_x} \\ \times & 1.y_2y_3\cdots y_N \quad 2^{e_y} \\ \hline 1.z_2z_3\cdots z_N \underbrace{0\cdots 0}_{k-1} 1 z_{N+k+1}\cdots z_{2N} & 2^{e_x+e_y} \end{array}$$

$$1.z_2z_3\cdots z_N\, 2^{e_x+e_y} \quad + \quad 0 \text{ ou } 1.z_{N+k+1}\cdots z_{2N} \underbrace{0\cdots 0}_{k}\, 2^{e_x+e_y-N-k}$$

(valeurs absolues) (arrondi vers le bas pour simplifier) (erreur nulle ou non nulle normalisée)

Un constat simple

Le produit

Le produit de deux nombres réels de N chiffres de mantisse tient sur $2N$ chiffres de mantisse

$$\begin{array}{rr} & 1.x_2x_3\cdots x_N \quad 2^{e_x} \\ \times & 1.y_2y_3\cdots y_N \quad 2^{e_y} \\ \hline 1.z_2z_3\cdots z_N \underbrace{0\cdots 0}_{k-1} 1 z_{N+k+1}\cdots z_{2N} & 2^{e_x+e_y} \end{array}$$

$$1.z_2z_3\cdots z_N\, 2^{e_x+e_y} \quad + \quad 0 \text{ ou } 1.z_{N+k+1}\cdots z_{2N} \underbrace{0\cdots 0}_{k}\, 2^{e_x+e_y-N-k}$$

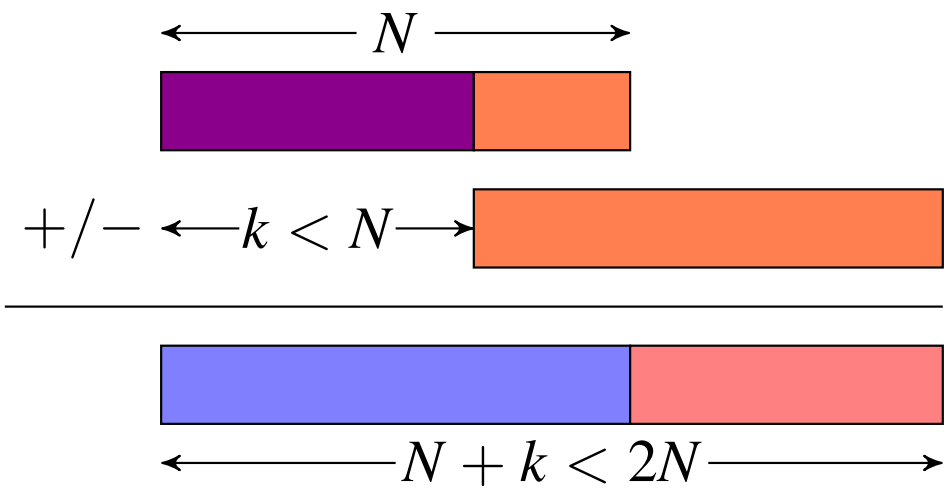
résultat flottant $+$ *erreur flottante*

(valeurs absolues) (arrondi vers le bas pour simplifier) (erreur nulle ou non nulle normalisée)

Un constat simple

La somme

La somme de deux nombres réels de N chiffres de mantisse tient sur $2N$ chiffres de mantisse



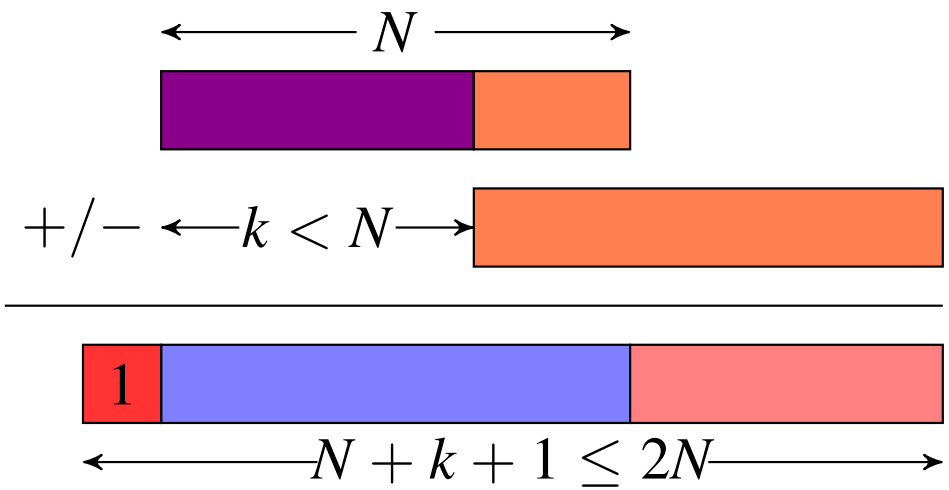
Après dénormalisation

avec chevauchement, sans propagation de retenue jusqu'au bout

Un constat simple

La somme

La somme de deux nombres réels de N chiffres de mantisse tient sur $2N$ chiffres de mantisse



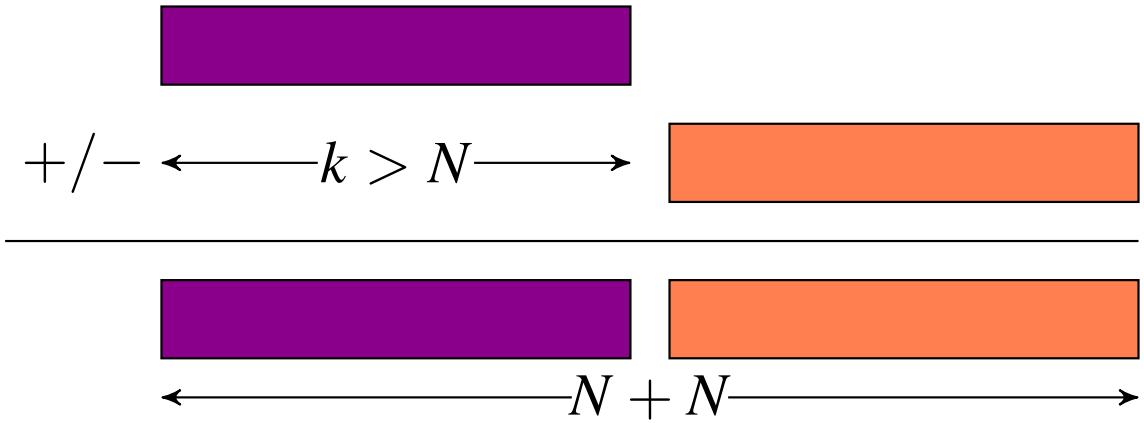
Après dénormalisation

avec chevauchement, avec propagation de retenue jusqu'au bout

Un constat simple

La somme

La somme de deux nombres réels de N chiffres de mantisse tient sur $2N$ chiffres de mantisse



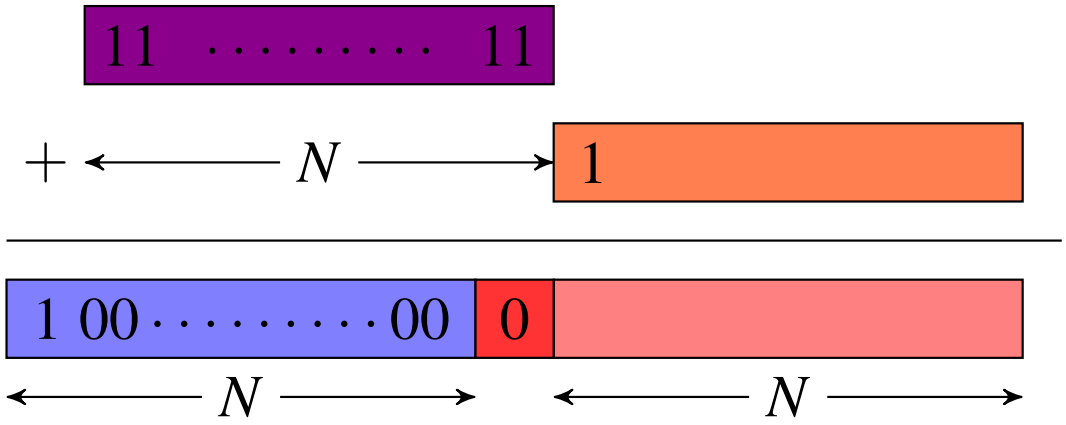
Après dénormalisation

sans chevauchement ni continuité

Un constat simple

La somme

La somme de deux nombres réels de N chiffres de mantisse tient sur $2N$ chiffres de mantisse



Après dénormalisation

sans chevauchement, mais avec continuité et propagation de retenue jusqu'au bout

(arrondi au plus près ou vers $+\infty$)

Un constat simple

Conclusion

Le produit et la somme de deux nombres flottants est la somme du résultat flottant de l'opération et d'un terme d'erreur flottant, sans chevauchement

Un constat simple

Conclusion

Le produit et la somme de deux nombres flottants est la somme du résultat flottant de l'opération et d'un terme d'erreur flottant, sans chevauchement

C'est bien beau ce résultat sur le papier, mais a-t-on un moyen simple de calculer ce terme d'erreur ?

Un constat simple

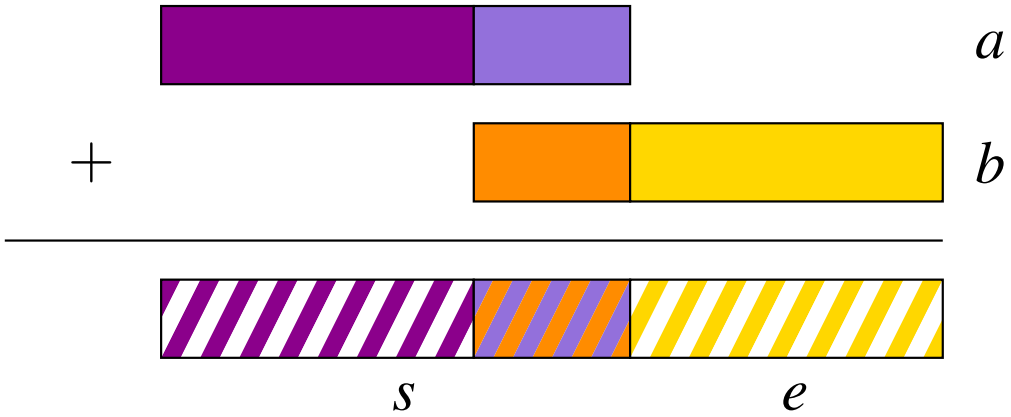
Conclusion

Le produit et la somme de deux nombres flottants est la somme du résultat flottant de l'opération et d'un terme d'erreur flottant, sans chevauchement

C'est bien beau ce résultat sur le papier, mais a-t-on un moyen simple de calculer ce terme d'erreur ? OUI

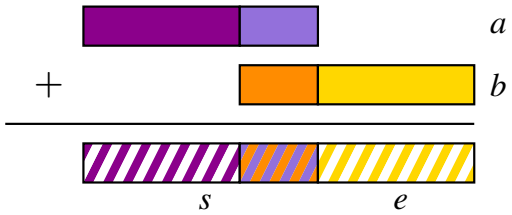
Les EFT (*Error Free Transformations*)

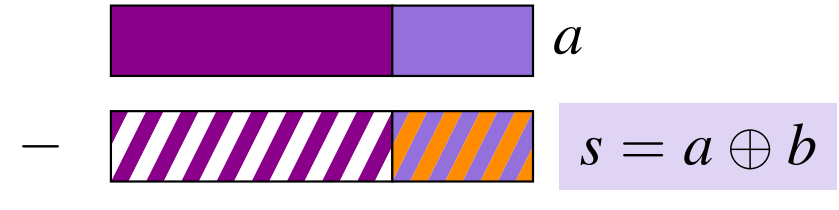
Intuition pour la somme : l'algorithme `FastTwoSum` de Dekker (1971)

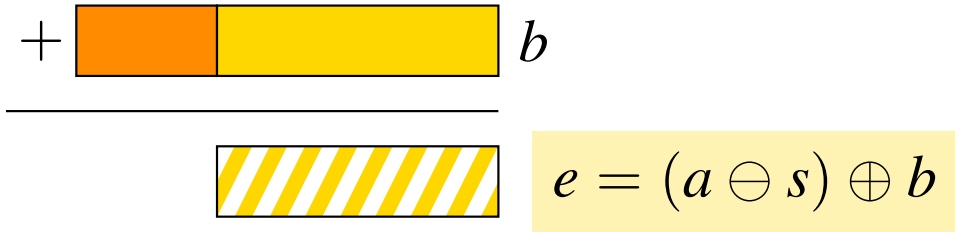


Les EFT (*Error Free Transformations*)

Intuition pour la somme : l'algorithme `FastTwoSum` de Dekker (1971)

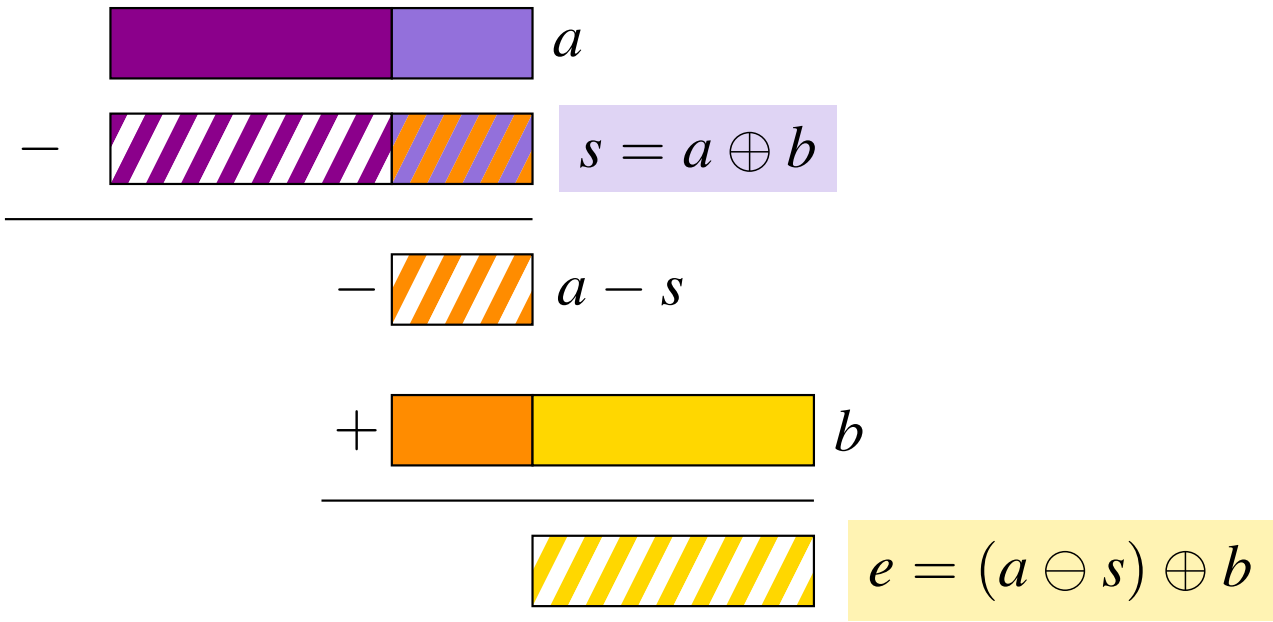






Les EFT (*Error Free Transformations*)

Intuition pour la somme : l'algorithme `FastTwoSum` de Dekker (1971)

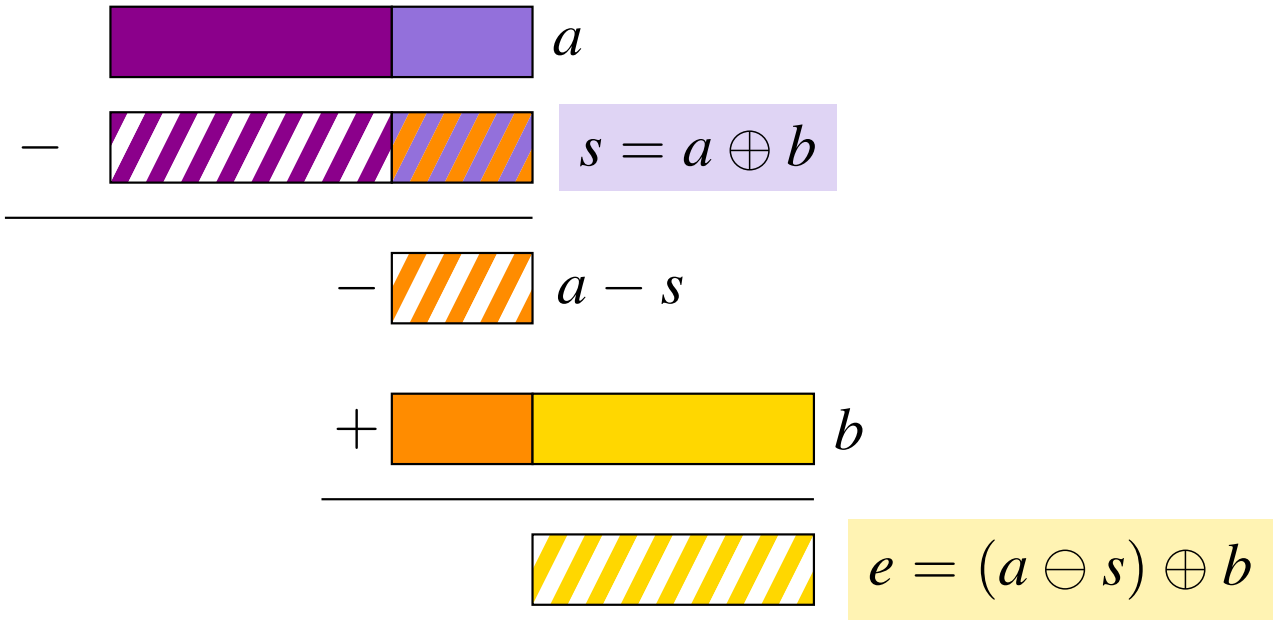


Vrai ? Dans quel mode d'arrondi ?

- Toujours vrai en arrondi au plus près
- Dépend du signe de a et b pour les autres modes d'arrondi.

Les EFT (*Error Free Transformations*)

Intuition pour la somme : l'algorithme `FastTwoSum` de Dekker (1971)

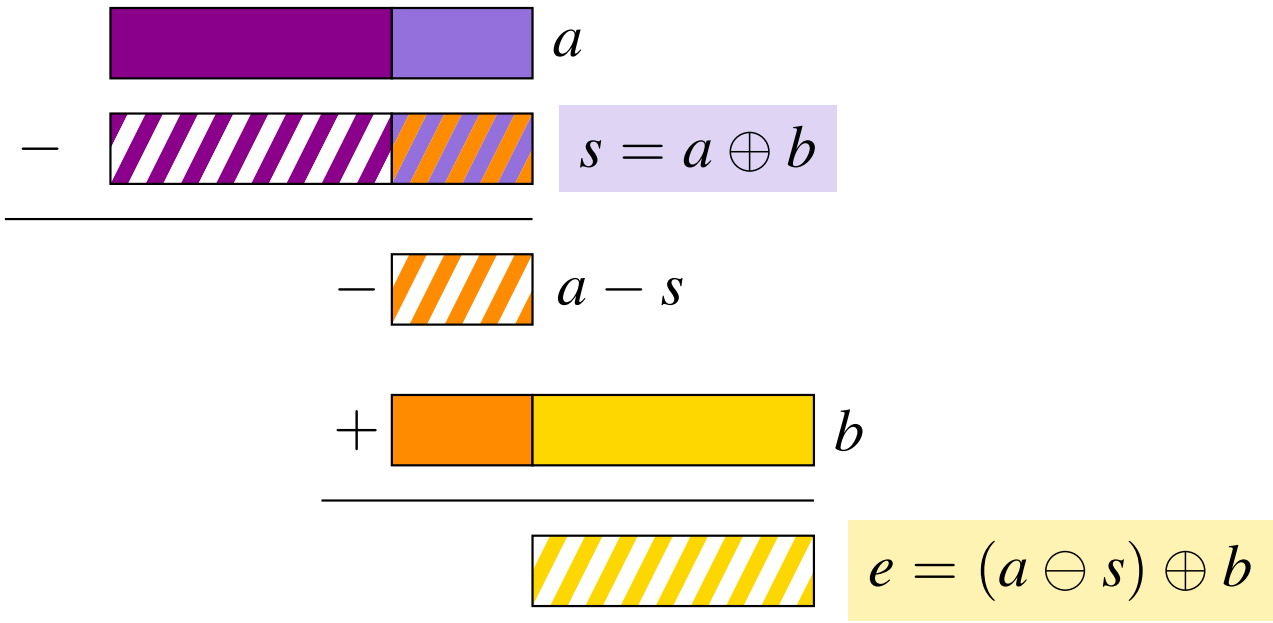


Vrai ? Dans quel mode d'arrondi ?

- Toujours vrai en arrondi au plus près
  Algorithme `FastTwoSum` de Dekker (1971)
- Dépend du signe de a et b pour les autres modes d'arrondi.

Les EFT (*Error Free Transformations*)

Intuition pour la somme : l'algorithme `FastTwoSum` de Dekker (1971)



Vrai ? Dans quel mode d'arrondi ?

- Toujours vrai en arrondi au plus près
  Algorithme `FastTwoSum` de Dekker (1971)
  Suppose que $|a| > |b|$
- Dépend du signe de a et b pour les autres modes d'arrondi.

Les EFT (*Error Free Transformations*)

L'algorithme `TwoSum` de Knuth (1969)

$$s = a \oplus b$$
$$\overline{a} = s \ominus b$$
$$\overline{b} = s \ominus \overline{a}$$
$$e = (a \ominus \overline{a}) \oplus (b \ominus \overline{b})$$

Référence : *Theorem B*, page 236, 3ème édition, tome 2

$$s + e = a + b$$

Les EFT (*Error Free Transformations*)

Les algorithmes `FastTwoSum` et `TwoSum`

Avec la syntaxe MatLab

function [s,e] = FastTwoSum(a,b)
  $s = a \oplus b$
  $e = (a \ominus s) \oplus b$

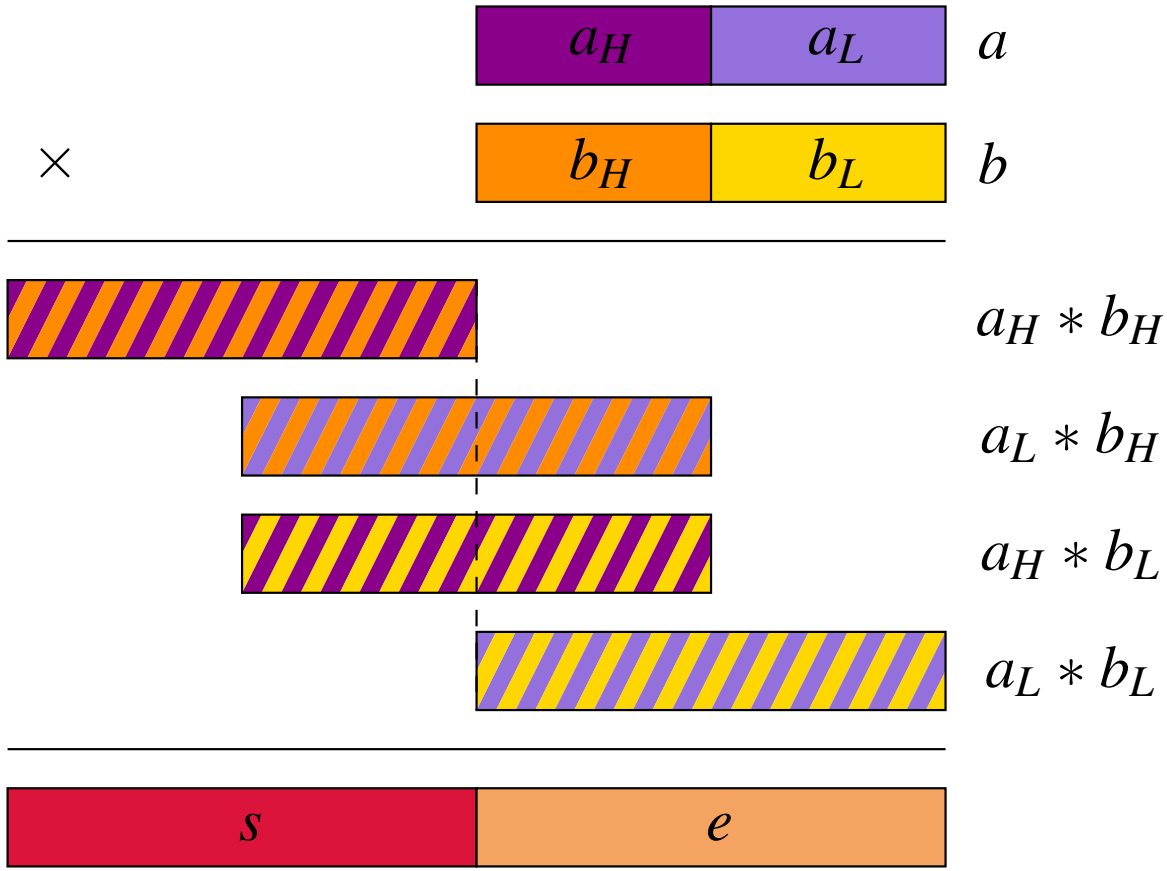
$|a| \geq |b|$

function [s,e] = TwoSum(a,b)
  $s = a \oplus b$
  $\overline{a} = s \ominus b$
  $\overline{b} = s \ominus \overline{a}$
  $e = (a \ominus \overline{a}) \oplus (b \ominus \overline{b})$

Coût : 3 additions pour `FastTwoSum`, 6 pour `TwoSum`

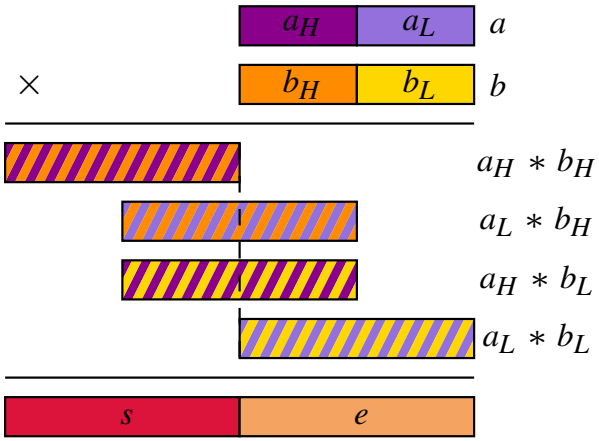
Les EFT (*Error Free Transformations*)

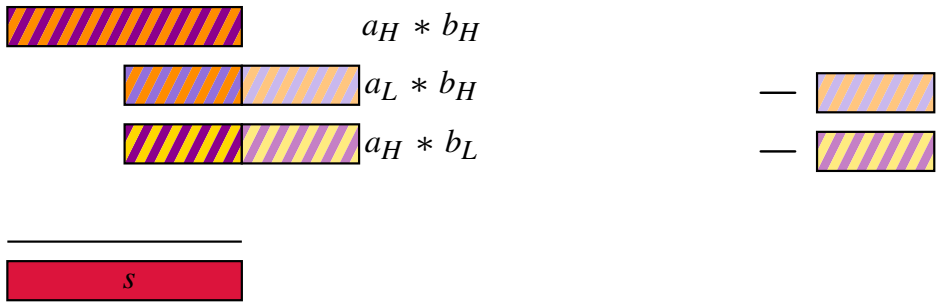
Intuition pour le produit



Les EFT (*Error Free Transformations*)

Intuition pour le produit

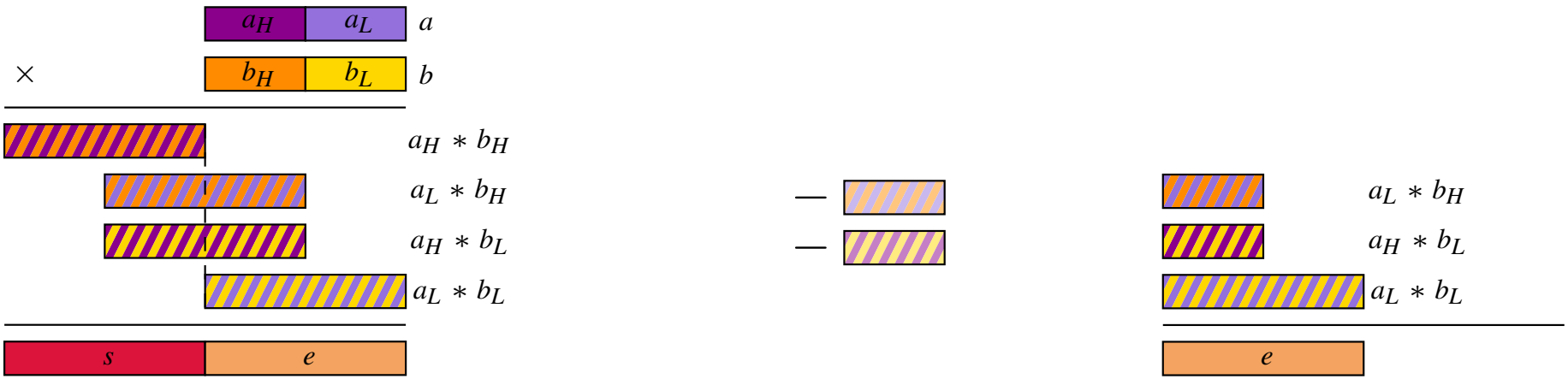




$$((s - a_H * b_H) - a_L * b_H) - a_H * b_L$$

Les EFT (*Error Free Transformations*)

Intuition pour le produit



$$((s - a_H * b_H) - a_L * b_H) - a_H * b_L$$

$$e = a_L * b_L - (((s - a_H * b_H) - a_L * b_H) - a_H * b_L)$$

Les EFT (*Error Free Transformations*)

L'algorithme `TwoProduct` de Dekker (1971)

Avec la syntaxe MatLab

```
function [s,e] = TwoProduct(a,b)
   s = a ⊗ b
   [a_H, a_L] = Split(a)
   [b_H, b_L] = Split(b)
   e = a_L ⊗ b_L ⊖ (((s ⊖ a_H ⊗ b_H) ⊖ a_L ⊗ b_H) ⊖ a_H ⊗ b_L)
```

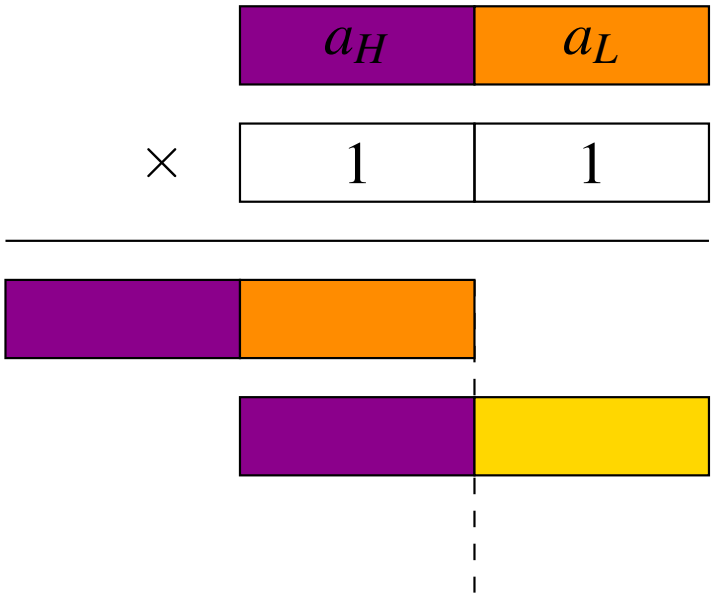
Coût : 5 produits, 4 soustractions + le coût des deux `Split`

Justement comment effectue-t-on le `Split` ?

Les EFT (*Error Free Transformations*)

Couper les nombres en deux : `Split`

Non, ce n'est pas une opération de base fournie en assembleur !



a

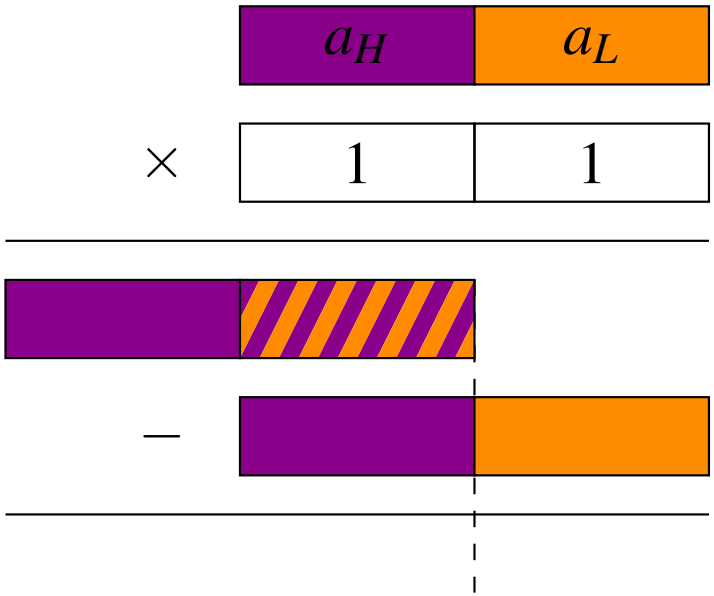
$2^s + 1$

$s = \lceil \frac{n}{2} \rceil$

Les EFT (*Error Free Transformations*)

Couper les nombres en deux : `Split`

Non, ce n'est pas une opération de base fournie en assembleur !



a

$2^s + 1$ $\qquad s = \lceil \frac{n}{2} \rceil$

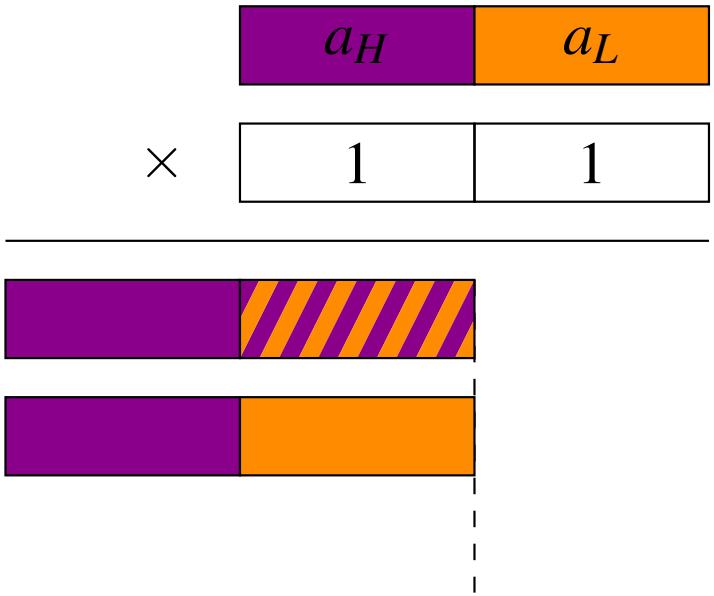
$c = a * (2^s + 1)$

a

Les EFT (*Error Free Transformations*)

Couper les nombres en deux : `Split`

Non, ce n'est pas une opération de base fournie en assembleur !



a

$2^s + 1$

$s = \lceil \frac{n}{2} \rceil$

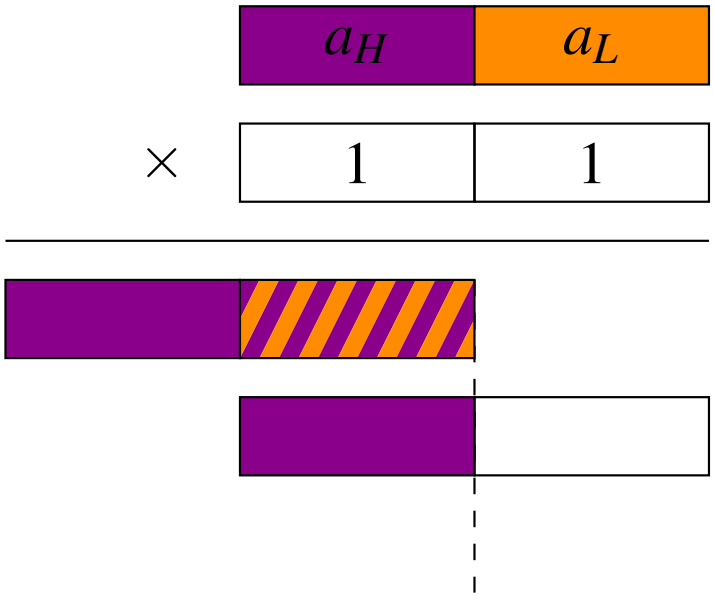
$c = a * (2^s + 1)$

$c - a$

Les EFT (*Error Free Transformations*)

Couper les nombres en deux : `Split`

Non, ce n'est pas une opération de base fournie en assembleur !



a

$2^s + 1$

$s = \lceil \frac{n}{2} \rceil$

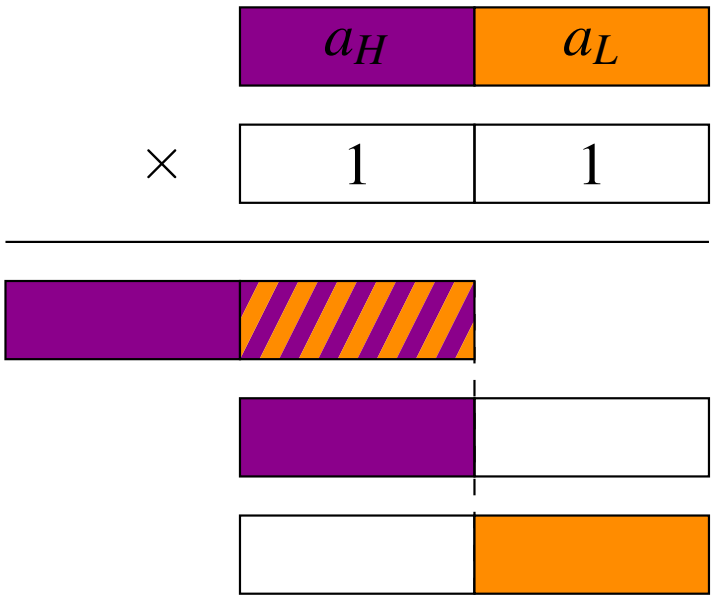
$c = a * (2^s + 1)$

$a_h = c - (c - a)$

Les EFT (*Error Free Transformations*)

Couper les nombres en deux : `Split`

Non, ce n'est pas une opération de base fournie en assembleur !



Les EFT (*Error Free Transformations*)

L'algorithme `Split` de Veltkamp (cité par Dekker 1971)

Avec la syntaxe MatLab

$s = \lceil n/2 \rceil$ % *Précalculé*
$f = \mathrm{fl}(2^s + 1)$ % *Précalculé*

function [a_h, a_l] = Split(a)
  $c = f \otimes a$
  $a_h = c \ominus (c \ominus a)$
  $a_l = a \ominus a_h$

Coût : 1 produit, 3 soustractions donc
l'algorithme `TwoProduct` a un coût de 7 produits et 10 soustractions.

Les EFT (*Error Free Transformations*)

Et si j'ai un FMA ?

FMA (*Fused-Multiply-and-Add*) $\texttt{FMA}(a,b,c) = ab + c$ avec un seul arrondi final
Disponible sur PowerPC, Itanium et Cell
Ajouté à la norme IEEE en 2008, mais pas disponible systématiquement

function $(s,e) = \texttt{TwoProductFMA}(a,b)$
  $s = a \otimes b$
  $e = \texttt{FMA}(a,b,-s)$

$\texttt{FMA}(a,b,-s)$ est l'arrondi de $ab - s$, mais $ab - s$ est un nombre flottant donc $\texttt{FMA}(a,b,-s) = ab - s$
Marche quel que soit le mode d'arrondi

Coût : 1 produit, 1 FMA.

Existe-t-il d'autres EFT ?

Algorithme de division de Pichat et Vignes (1993)

Évidemment le quotient de deux nombres de n chiffres n'a pas une longueur limitée, mais...

Avec

$$q = a \oslash b \text{ et } r \text{ tel que } a = bq + r$$

Pichat et Vignes ont montré que r est un flottant et donné un algorithme pour le calculer

function $[q, r] = \texttt{DivRem}(a, b)$
$q = a \oslash b$
$[s, e] = \texttt{TwoProduct}\ (q, b)$
$r = (a \ominus s) \ominus e$

Évidemment calculer r est trivial si on a un FMA !

On peut s'en servir pour calculer plus de chiffres du quotient

Existe-t-il d'autres EFT ?

Algorithme de racine carrée de Markstein (1990)

Calcule $r = \mathrm{fl}(\sqrt{a})$ et $e = a - r^2$ de la même façon.

Utilisation

On peut obtenir davantage de chiffres de la racine :
en remplaçant r par $r + r'$, on voudrait obtenir $0 = a - (r + r')^2$,
on développe $0 = e - 2rr' - r'^2$ et on prend donc $r' = e/2r$
ce qui augmente la précision de l'approximation de $\sqrt{a}$.

Existe-t-il d'autres EFT ?

L'algorithme pour le FMA (Boldo & Muller 2005)

Le résultat du FMA de 3 nombres de n chiffres est un nombre de $3n$ chiffres

Il faut deux flottants sans chevauchement pour représenter l'erreur, ce qui rend le résultat plus difficilement calculable et moins visiblement utile

Existe-t-il d'autres EFT ?

EFT de calcul d'un déterminant pour la détermination du signe d'un déterminant pour prédire la position d'un point par rapport à une surface. (Ozaki, Ogita & Oishi, 2012)

Et alors ?

- Oui c'est possible de calculer le terme d'erreur d'une addition/soustraction ou d'une multiplication flottante
- Mais c'est coûteux, on multiplie le nombre d'opérations par un facteur au moins 2
- Le terme d'erreur est effectivement négligeable devant l'autre qui donne l'ordre de grandeur du résultat
- Alors qu'est-ce qu'on gagne à transformer la somme ou le produit de deux nombres quelconques en une somme de deux nombres sans chevauchement ? Cela en vaut-il la peine ?

Deux utilisations

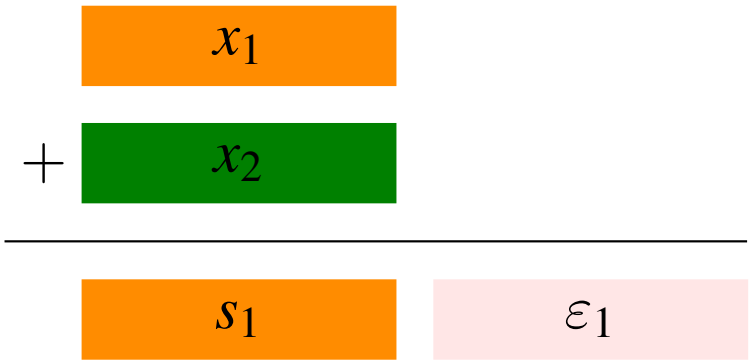
- Tout d'abord au niveau de la seule opération concernée, cela permet le calcul en soft sur deux fois plus de chiffres (utilisation d'origine pour avoir des `double` avec des `float` : titre Dekker *A floating-point technique for extending the available precision*) (cf. Thèse Louvet 3.4)

- Lorsque plusieurs opérations sont combinées pour obtenir un résultat, les erreurs d'arrondi s'accumulent et on va exploiter cette information supplémentaire pour compenser ces erreurs et obtenir un résultat global de meilleure qualité numérique

La somme de n éléments

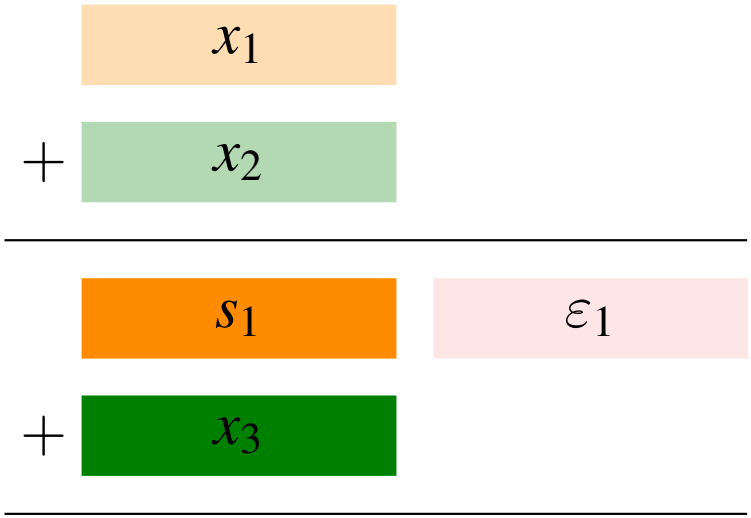

```
function [s]=Sum(x)
    s = x_1
    for i = 1 : n − 1
        s = s ⊕ x_{i+1}
    end
```

La somme de n éléments



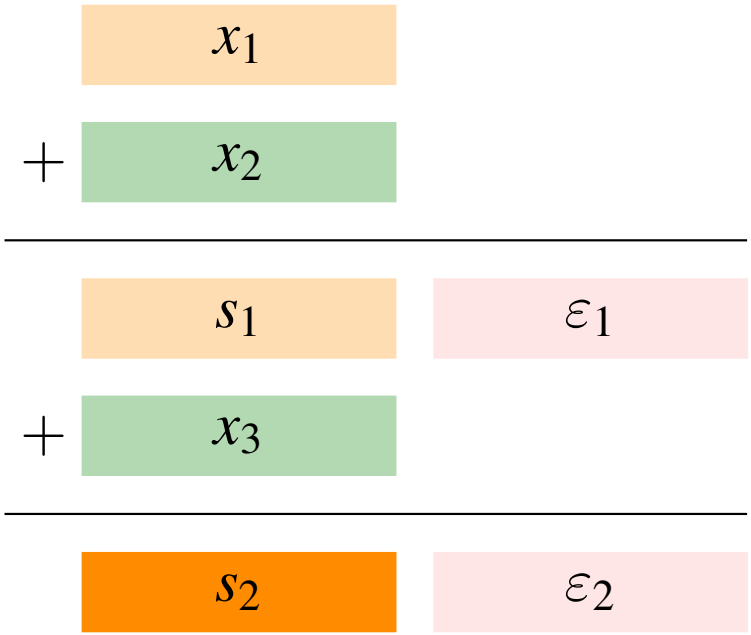

```
function [s]=Sum(x)
  s = x_1
  for i = 1 : n - 1
    s = s ⊕ x_{i+1}
  end
```

La somme de n éléments



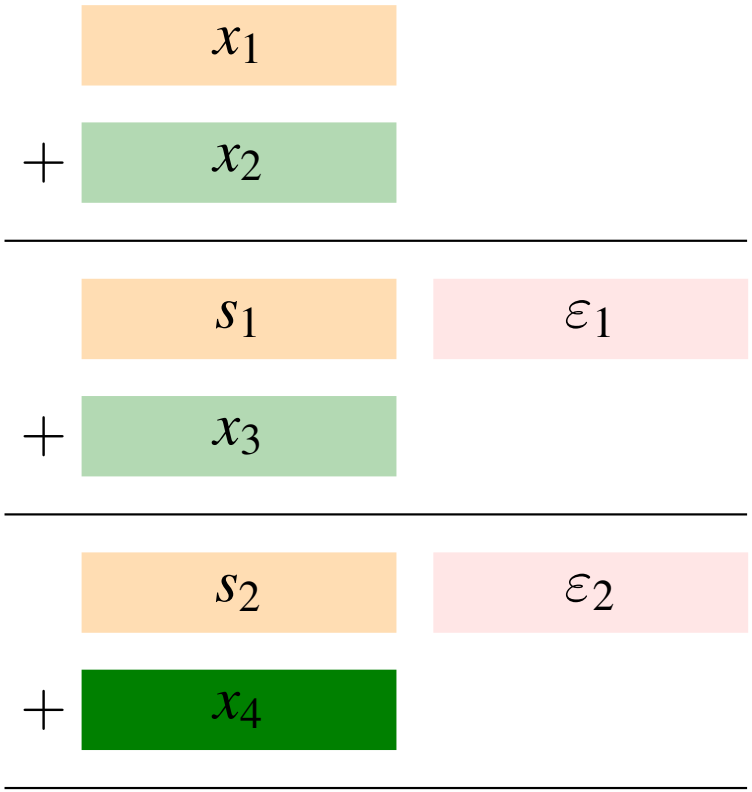

```
function [s]=Sum(x)
    s = x_1
    for i = 1 : n − 1
        s = s ⊕ x_{i+1}
    end
```

La somme de n éléments



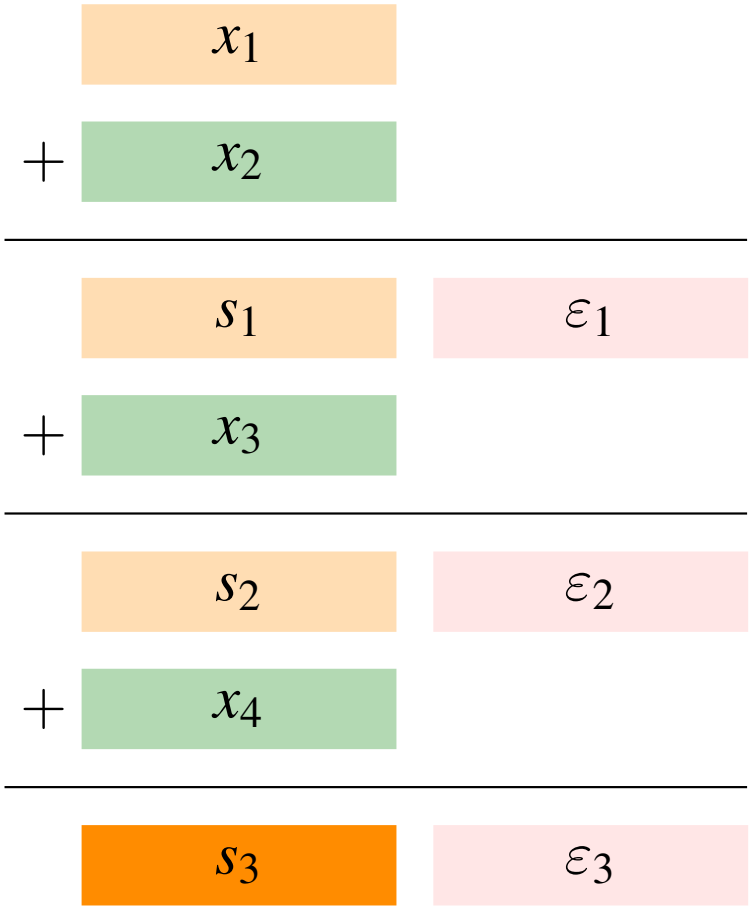

```
function [s]=Sum(x)
    s = x_1
    for i = 1 : n − 1
        s = s ⊕ x_{i+1}
    end
```

La somme de n éléments



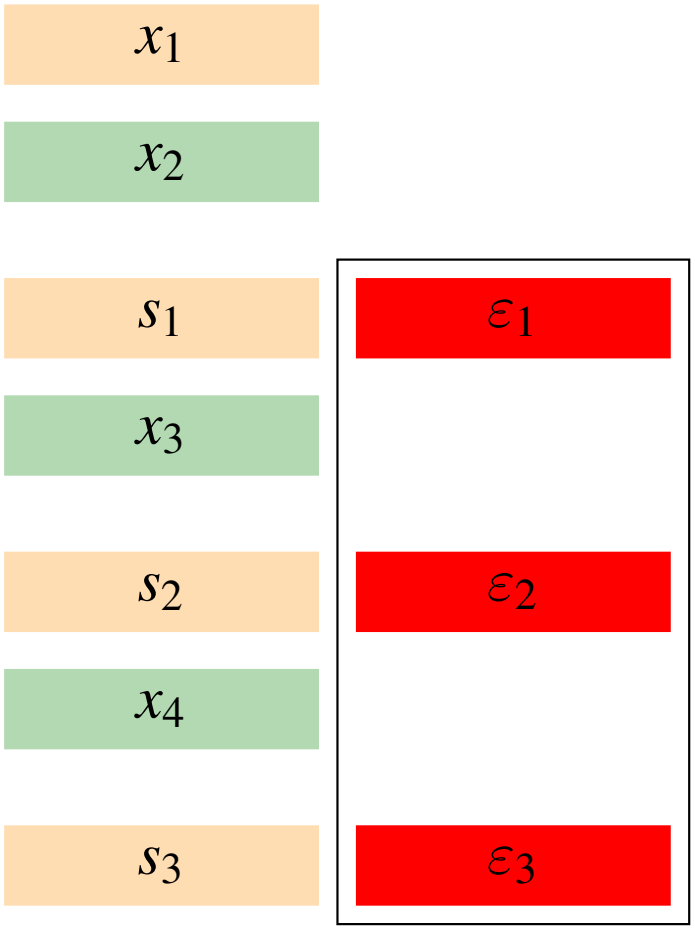

```
function [s]=Sum(x)
    s = x_1
    for i = 1 : n − 1
        s = s ⊕ x_{i+1}
    end
```

La somme de n éléments




```
function [s]=Sum(x)
    s = x_1
    for i = 1 : n − 1
        s = s ⊕ x_{i+1}
    end
```

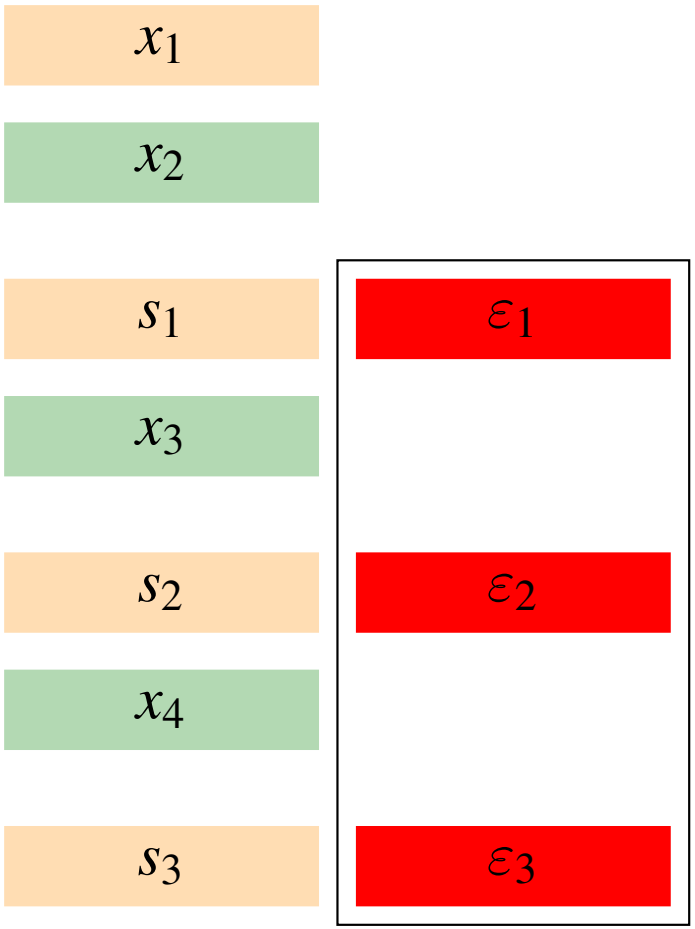
La somme de n éléments




```
function [s]=Sum2(x)
    s = x_1
    for i = 1 : n − 1
        [s, ε_i] =TwoSum(s, x_{i+1})
    end
```

La somme de n éléments

Algorithme Sum2 d'Ogita, Rump & Oishi (2005)

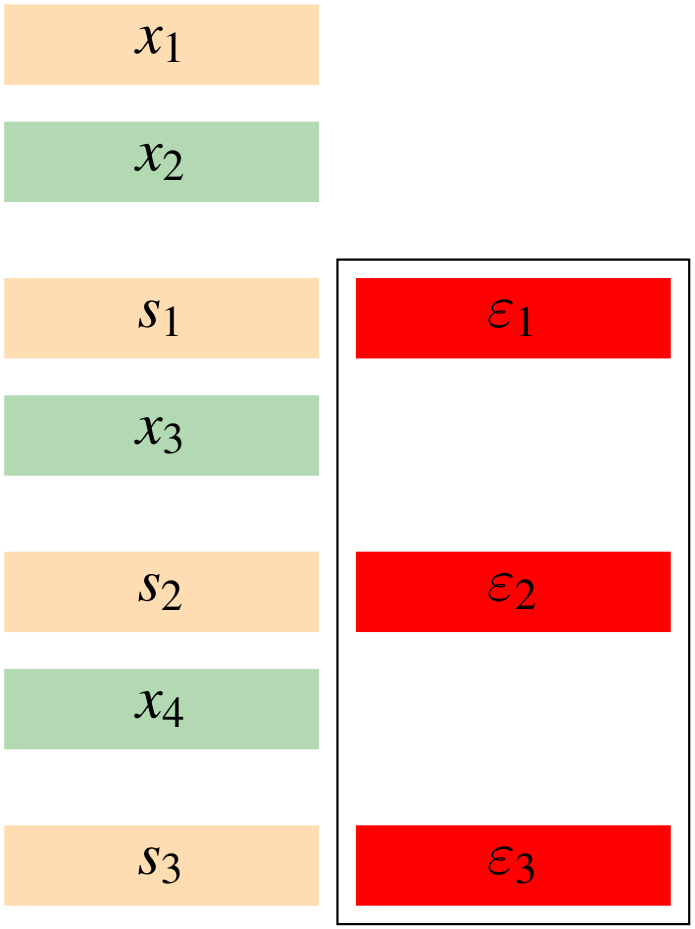



```
function [s]=Sum2 (x)
  s = x_1
  e = 0
  for i = 1 : n − 1
    [s, ε_i] =TwoSum(s, x_{i+1})
    e = e ⊕ ε_i
  end
  s = s ⊕ e
```

Coût : $7(n-1)$ additions

Qualité du résultat : comme si on avait fait le calcul en précision double de la précision courante puis arrondi à la précision courante.

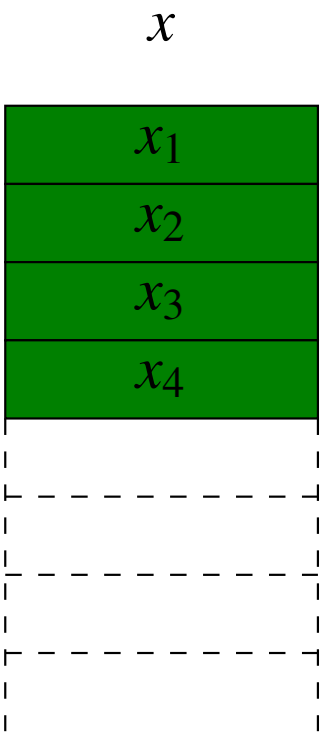
La somme de n éléments



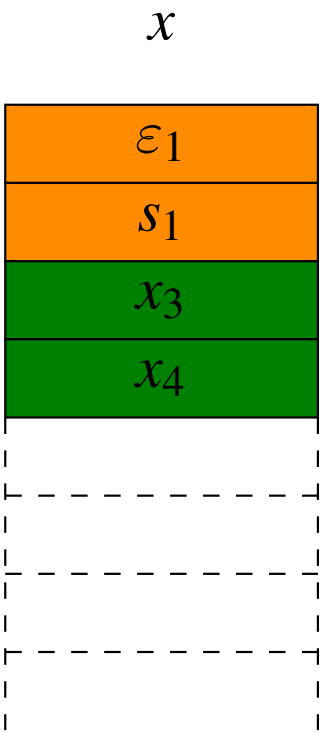
Récursivement

```
function [s] = SumK(x, K, n)
  s = x_1
  if K>1 then
     for i = 1 : n − 1
        [s, ε_i] =TwoSum(s, x_{i+1})
     end
     s = s ⊕ SumK(ε, K − 1, n − 1)
  else s = Sum(x)
  end
```

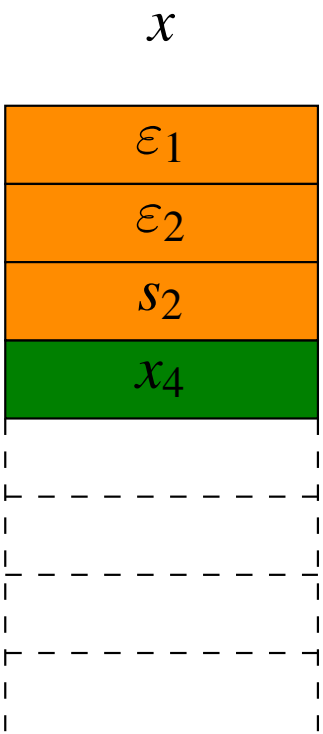
La somme de n éléments



La somme de n éléments

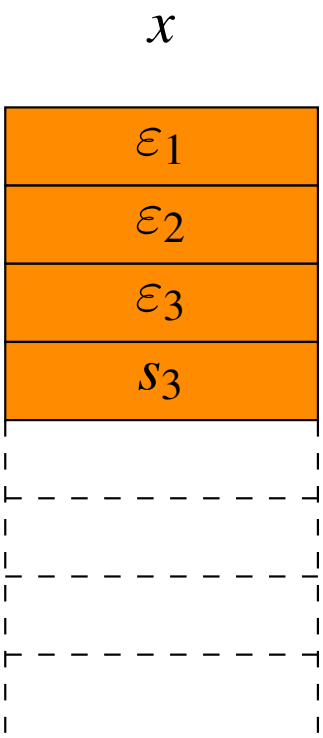


La somme de n éléments



La somme de n éléments

Algorithme SumK d'Ogita, Rump & Oishi (2005)



Par raffinement itératif

```
function [x]=VecSum(x)
  for i = 2 : n
     [x_i, x_{i-1}]=TwoSum(x_i, x_{i-1})
  end

function [s]=SumK(K,x)
  for k = 1 : K - 1
     [x]=VecSum(x)
  end
  s =Sum(x)
```

Coût : $(K-1) \times (n-1)$ `TwoSum` et 1 `Sum` soit $(6K-5)(n-1)$ additions

Qualité du résultat : Comme si on calculait en précision K fois la précision courante puis arrondissait à la précision courante.

Le produit scalaire

function res=Dot(x,y)

$s = x_1 \otimes y_1$

for $i = 1 : n-1$

$s = s \oplus (x_{i+1} \otimes y_{i+1})$

end

res$= s$

Le produit scalaire

function res=Dot2(x,y)
  $[s,e]$=TwoProduct(x_1,y_1)
  for $i = 1 : n-1$
    $[p_i,\varepsilon_\times]$=TwoProduct(x_{i+1},y_{i+1})
    $[s,\varepsilon_+]$=TwoSum(s,p_i)
  end
  res$= s$

Le produit scalaire

L'algorithme `Dot2` d'Ogita, Rump & Oishi (2005)

function res=Dot2(x, y)
  $[s, e]$=TwoProduct(x_1, y_1)
  for $i = 1 : n - 1$
    $[p_i, \varepsilon_\times]$=TwoProduct$(x_{i+1}, y_{i+1})$
    $[s, \varepsilon_+]$=TwoSum(s, p_i)
    $e = e \oplus (\varepsilon_+ \oplus \varepsilon_\times)$
  end
  res$= s \oplus e$

Coût : n TwoProduct, $n-1$ TwoSum, $2n-1$ additions de base, soit $25n-7$ opérations

Qualité du résultat : comme si on avait fait le calcul en précision double de la précision courante puis arrondi à la précision courante.

Le produit scalaire

L'algorithme DotK d'Ogita, Rump & Oishi (2005)

function res=DotK(x, y)
  $[s, e_1]$=TwoProduct(x_1, y_1)
  for $i = 1 : n-1$
    $[p_i, e_i]$=TwoProduct(x_{i+1}, y_{i+1})
    $[s, e_{n-1+i}]$=TwoSum(s, p_i)
  end
  $e_{2n} = s$
  res=SumK$(K-1, e)$

Coût : n TwoProduct, $n-1$ TwoSum, 1 SumK d'un vecteur de longueur $2n$ pour $K-1$, soit $17n + 6(n-1) + (6(K-1)-5)(2n-1) < (12K+1)n$ opérations.

Qualité du résultat : comme si on calculait en précision K fois la précision courante puis arrondissait à la précision courante.

L'évaluation polynomiale avec l'algorithme de Horner

Algorithme de Louvet (2007), Graillat, Langlois & Louvet (2008)

```
function [res]=Horner(p, x)
    s = p_n
    for i = n − 1 : −1 : 0
        s = (s ⊗ x) ⊕ p_i
    end
    res = s
```

```
function [res]=CompHorner(p, x)
    s = p_n
    for i = n − 1 : −1 : 0
        [t, ε_×]=TwoProduct(s, x)
        [s, ε_+]=TwoSum(t, p_i)
        e_i = ε_× ⊕ ε_+
    end
    res = s ⊕ Horner(e, x)
```

et on partage le `Split` de x.

Coût : 1 `Split` de x, n fois (1 `TwoProduct` avec un seul `Split`, 1 `TwoSum`, 1 addition), et pour `Horner` n tours de 2 opérations et enfin 1 addition finale, d'où au total : $4 + (13 + 6 + 1)n + 2n + 1 = 22n + 5$ opérations.

Qualité du résultat : comme si on avait fait le calcul en précision double de la précision courante puis arrondi à la précision courante.

Et bien sûr il y a un algorithme `CompHornerK` (Louvet 2007)

La résolution de système triangulaire (inférieure)

```
function [x]=TRSV(A, b)
  for k = 1 : n
    x_k = (b_k ⊖ Dot(A_{k[1..k-1]}, x_{[1..k-1]})) ⊘ A_kk
  end
```

(Louvet 2007) Deux façons de produire un vecteur plus précis :

- transformer les opérations en EFT au fur et à mesure du calcul et combiner les termes d'erreur successifs pour produire directement une solution x plus précise.
- faire tout le calcul normalement et ensuite, par raffinement itératif, calculer le résidu $r = b - Ax$ en arithmétique compensée, puis résoudre le système triangulaire(inférieur) $Ae = r$ et rendre $x + e$.

Les deux façons sont équivalentes mais la seconde s'itère de façon naturelle.

Autres applications réelles

Travail de Ozaki, Ogita, Rump & Oishi (2011-2012) sur le produit de matrices (EFT,intervalles,compensée ?)

Il est envisageable de transformer automatiquement un code d'arithmétique flottante utilisant $+$ et $\times$ en un code d'arithmétique compensée collectant les termes d'erreur et les combinant pour améliorer le résultat flottant.

Cf. exposé Laurent Thévenoux

Et sur les complexes ?

Graillat & Ménissier-Morain (2007-2012)
En coordonnées cartésiennes

- EFT (`TwoSumCplx`, `TwoProductCplx`)
- `Sum2CPlx`,`Dot2Cplx`,`HornerCplx`