Master 1 : SFPN - Année 2019/2020

PSFPN

Evaluation précise de fonctions rationnelles

Auteur :
Yasmine Ikhelif

Encadrante :
Valérie Ménissier-Morain

Evaluation précise de fonctions rationnelles

Yasmine Ikhelif

May 2020

Table des matières

1 Introduction **2**

I Arithmétique **3**

2 Norme IEEE-754 **3**
2.1 Types et sources d'erreurs 3

3 Arithmétique flottante **4**
3.1 Fused Multiply-Add Operation (FMA) 4

4 Arithmétique complexe en virgule flottante **5**

5 Horner **5**

II Arithmétique compensée **6**

6 Error-Free Transformations **6**

7 Pour les rééls **6**
7.1 EFT de la somme de deux flottants 6
7.2 Résultats expérimentaux 6
7.3 EFT du produit de deux flottants 7
7.4 Résultats expérimentaux 7
7.5 Propriétés 8

8 Pour les complexes **9**
8.1 EFT de la somme de deux flottants complexes 9
8.2 EFT du produit de deux flottants complexes 9
8.3 Résultats expérimentaux 10

9 Schéma de Horner compensé **11**
9.1 Pour les réels 11
9.2 Pour les complexes 12

10 Évaluation de fonctions rationnelles réelles **14**
10.1 Évaluation classique de fonctions rationnelles 14
10.2 Évaluation de fonction rationnelles avec un algorithme compensé 15
10.3 Résultats expérimentaux 15

11 Évaluation de fonctions rationnelles complexes **16**

12 Conclusion **16**

1 Introduction

Les nombres réels sont modélisés sur ordinateur par l'ensemble discret des nombres flottants. Chaque opération d'arithmétique flottante rend un résultat entaché d'erreur. Pour certaines opérations telles que l'addition ou la multiplication de deux nombres flottants cette erreur est elle-même un nombre flottant. Nous allons essayer tout au long de ce travail d'isoler cette erreur afin d'obtenir des résultats plus précis aux opérations arithmétiques fondamentales.

D'autre part, nous allons essayer d'évaluer des fonctions rationnelles et complexes qui sont définies comme étant le quotient de deux polynômes aux coefficients réels ou complexes.

Pour résumer les principaux points que nous avons abordés durant ce projet sont :

— Introduction et assimilation de notions d'arithmétique en virgule flottante
— Introduction à l'arithmétique compensé
— Implémention des algorithmes compensé
— Évaluation des polynômes via Horner et Horner Compensé
— Conception d'algorithme d'algorithmes d'EFT de la division réelle/complexe
— Utilisation des méthodes précédentes pour l'évaluation de fractions rationnelles.

Première partie
Arithmétique

2 Norme IEEE-754

La norme IEEE-754 est une norme sur l'arithmétique à virgule flottante publiée en 1985 [1]. Cette norme permet de spécifier le format de représentation des nombres à virgule flottante, leurs résultats ainsi que les modes d'arrondi.

Pour que notre travail soit le plus clair possible, nous allons tout d'abord commencer par parler de la virgule flottante qui est la méthode utilisée pour représenter les nombres réels sur les ordinateurs. Un nombre flottante x est réel représenté de la forme :

$$x = (-1)^s \times m \times b^e$$

Où $(-1)^s$ est le bit de poids fort. Ce but désigne le signe du nombre, avec $s \in \{0, 1\}$. Si $s = 0$, x est positif. Si $s = 1$ x est négatif. m est la mantisse, b la base ici nous sommes en base 2 et e est l'exposant de x. Il faut noter aussi que le changement de base, de la base 10 pour les nombres réels que nous essayons de modéliser à la base 2 employé par les ordinateurs, peut déjà être une raison de perte de précision.

Il existe deux formats principaux pour la représentation des nombres à virgule flottante en base 2 que nous utilisons dans ce projet et qui répondent à la norme IEE-754 :

Type	Taille	Mantisse	Exposant
Simple	32 bits	23+1 bits	8 bits (-126 à 127)
Double	64 bits	52+1 bits	11 bits (-1022 à 1023

La norme IEE-754 définit aussi pluseiurs modes d'arrodi. Il y a principalement quatre types d'arrondi :
— L'arrondi au plus près : le nombre est arrondi à la valeur la plus proche, avec le bit de signe à 0.
— L'arrondi vers zéro.
— L'arrondi vers plus l'infini.
— L'arrondi vers moins l'infini.

Dans ce projet nous allons choisir le mode d'arrondi au plus proche.

2.1 Types et sources d'erreurs

Les erreurs sont toujours présentes dans le calcul numériques. Les erreurs d'arrondi sont inévitables lorsqu'on travaille en précision finie. On distingue alors trois sources différentes d'erreurs de calcul :
— Erreurs de mesure ou d'estimation (les erreurs dans les données peuvent être grandes si ces dernières sont mal mesurées)
— Erreurs résultant du stockage des données sur ordinateur (erreurs d'arrondi - petites)
— Erreurs résultants d'opération arithmétiques précédentes erronées. (petites ou grandes erreurs)

Les erreurs résultats de mesures ou de stockage peuvent être analysées grâce aux théories de perturbation. Les erreurs d'arrondi résultant d'opérations arithmétiques quant à elles demandes une autre méthode d'analyse. Nous expliquerons ceci dans la suite.

3 Arithmétique flottante

Dans la suite, nous allons suivre suivre une arithmétique à virgule flottante suivant la norme IEEE 754. Nous faisons l'hypothèse qu'il n'y a pas d'*Overflow* ni d'*Underflow*.

On définit l'*Overflow* par le fait que le résultat exact est strictement supérieur au plus grand nombre flottant normalisé. L'*Underflow* quant à lui désigne le fait que le résultat exact soit inférieur au plus petit nombre flottant représentable.

L'ensemble des nombres flottants est défini sur $\mathbb{F}$, l'erreur d'arrondi est définie par u. Dans la norme IEE 754, en double précision $u = 2^{-53}$ et en simple précision $u = 2^{-24}$. On représente par fl(.) le résultat d'une opération en virgule flottante. Dans la norme IEE 754, toutes les opérations satisfont *

Nous avons alors :

$$fl(a \circ b) = (a \circ b)/(1 + \epsilon_1) = (a \circ b)/(1 + \epsilon_2) \text{ pour } \circ = \{+, -, \cdot, /\}, \text{ et } |\epsilon_v| \leq u$$

Ceci implique :

$$|a \circ b - fl(a \circ b)| \leq u|a \circ b|$$

et

$$|a \circ b - fl(a \circ b)| \leq u|fl(a \circ b)| \text{ pour } \circ = \{+, -, \cdot, /\}$$

Nous utilisons la notation standard γ_n pour l'estimation d'arrondi, définie par :

$$\gamma_n := \frac{nu}{1-nu} \text{ pour } n \in \mathbb{N}$$

Tout en admettant explicitement que : $nu \leq 1$

Nous remarquons que $a \circ b \in \mathbb{R}$ et $ab := fl(a \circ b) \in \mathbb{F}$ mais en général, nous n'avons pas $a \circ b \in \mathbb{F}$. Nous admettons que pour les opérations de base : $\{+, -, \times, /\}$ l'erreur d'arrondi d'une opération en virgule flottante est toujours un nombre en virgule flottante.

$$\begin{aligned} x = a \oplus b &\Rightarrow a + b = x + y \text{ avec } y \in \mathbb{F} \\ x = a \ominus b &\Rightarrow a - b = x + y \text{ avec } y \in \mathbb{F} \\ x = a \otimes b &\Rightarrow a \times b = x + y \text{ avec } y \in \mathbb{F} \end{aligned}$$

Nous pouvons ainsi mesurer cette erreur grâce aux *Error-Free Transofrmations(EFT)* que nous décrirons plus loin.

3.1 Fused Multiply-Add Operation (FMA)

Certains ordinateurs possèdent une *Fused-Multiply Add* (FMA) opération qui permet une multiplication flottante suivi d'une addition ou soustraction, $x*y+z$ ou $x*y-z$, comme étant une seule opération flottante [1]. Une opération de type FMA ne commet donc qu'une seule erreur d'arrondi.

$$fl(x \times y \pm z) = (x \times y \pm z)/(1 + \epsilon), \ \ |\epsilon| \leq u$$

Dans la suite, nous expliciterons tous les algorithmes qui emploient cette méthode. Nous comparerons aussi le nombre total d'opérations que ces derniers effectueront aux algorithmes qui n'utilisent pas la FMA. Nous la définissions alors ici afin que la suite soit plus claire

4 Arithmétique complexe en virgule flottante

On note par $\mathbb{F} + i\mathbb{F}$ l'ensemble des nombres complexes en virgule flottante. Comme pour les réels, nous allons noter $fl(.)$ le résultat d'un calcul en virgule flottante. Pour $x, y \in \mathbb{F} + i\mathbb{F}$ nous avons :

$$fl(x \circ y) = (x \circ y)(1 + \epsilon_1) = (x \circ y)/(1 + \epsilon_2) \text{ pour } \circ = \{+, -\} \text{ et } |\epsilon_1|, |\epsilon_2| \leq u$$

et

$$fl(x.y) = (x.y)/(1 + \epsilon_1) \text{ avec } |\epsilon_1)| \leq \sqrt{2}\gamma_2$$

Ce qui implique :

$$|a \circ b - fl(a \circ b)| \leq u|a \circ b|$$

et

$$|a \circ b - fl(a \circ b)| \leq u|a \circ b - fl(a \circ b)| \text{ pour } \circ = \{+, -\}$$

Et pour la multiplication

$$|(x.y - fl(x.y)| \leq \sqrt{2}\gamma_2$$

Pour les quantités $\tilde{\gamma_n}$ d'estimation d'erreur :

$$\tilde{\gamma}_n := \frac{n\sqrt{2}\gamma_2}{1 - n\sqrt{2}\gamma_2}$$

Nous utiliserons aussi les inégalités suivantes dans la suite des algorithmes :

$$(1 + \sqrt{2}\gamma_2)(1 + \tilde{\gamma}_n) \leq (1 + \tilde{\gamma}_{n+1})$$

et

$$(1 + \sqrt{2}\gamma_2)\tilde{\gamma}_{n-1} \leq \tilde{\gamma}_n$$

5 Horner

Dans cette partie, nous allons brièvement expliquer l'évaluation de polynômes via l'algorithme de Horner. Considérons un polynôme P de degré n dont les coefficients sont supposés flottants :

$$P(x) = \sum_{i=0}^{n} p_i x^i$$

Avec la méthode de Horner nous allons calculer l'image du polynôme P en un point x_0. Elle permet d'obtenir la division euclidienne de $P(x)$ par $(x - x_0)$. Ce qui est utile pour la factorisation de polynômes.

L'algorithme de Horner effectue les calculs comme suit :

$$P(x_0) = ((...(a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + ...a_1)x_0 + a_0$$

Nous expliciterons dans la suite un algorithme à implémenter en MATLAB pour l'évaluation de polynômes via le schéma de Horner.

Deuxième partie

Arithmétique compensée

6 Error-Free Transformations

Dans cette partie nous allons présenter différents algorithmes d'arithmétique compensée. L'arithmétique compensée est une arithmétique qui vise à minimiser les erreurs d'arrondi. Comme nous l'avons présenté précédemment, pour deux flottants $a, b \in \mathbb{F}$ le résultat de l'opération $a \circ b \in \mathbb{R}$ alors que nos calculs se font sur $\mathbb{F}$. Nous avons démontré précédemment que cette opération engendre une erreur $y \in \mathbb{F}$, nous allons chercher à isoler cette erreur y et à la réinjecter dans le résulat de notre calcul afin d'avoir un résultat plus précis. Ceci est le but principal de l'arithmétique compensée, cette opération sera explicitée dans tous les algorithmes que nous allons définir dans la suite. Les *Error-Free Transformations* sont donc des méthodes et algorithmes qui implémentent cette propriété. Tous les algorithmes de cette partie sont donc des *Error-Free Transformations* (EFT).

7 Pour les rééls

7.1 EFT de la somme de deux flottants

Le premier algorithme que nous allons présenter est l'algorithme **TwoSum** de Knuth [2]. Cet algorithme effectue 6 flops (opérations en virgules flottante)

Algorithme 1 : EFT de la somme de deux flottants

function $[x, y]$: **TwoSum**(a, b)

$$
\begin{aligned}
x &= (a \oplus b) \\
z &= (x \ominus b) \\
y &= (a \ominus (x \ominus z)) \oplus (b \ominus z)
\end{aligned}
$$

Si nos deux flottants en entrée a et b sont tels que $|a| \leq |b|$, alors nous pouvons utiliser l'algorithme **FastTwoSum** de Dekker [3] pour calculer l'EFT de l'addition. Cet algorithme a les mêmes propriétés que l'algorithme 1, mais ne nécessite que 3 flops. Néanmoins, si nous considérons la valeur absolue et la comparaison comme des opérations, alors il en effectue 6. En pratique, **FastTwoSum** est 50% plus lent que **TwoSum** dû aux ramifications.

Algorithme 2 : EFT de la somme de deux flottants lorsque $|a| \leq |b|$

function $[x, y]$: **FastTwoSum** $((a, b))$

$$
\begin{aligned}
x &= (a \oplus b) \\
y &= (b \ominus (x \ominus a))
\end{aligned}
$$

7.2 Résultats expérimentaux

Nous avons testé ces fonction sur deux flottants a et b générés au aléatoirement.

$$
\begin{aligned}
a &= 2563657.579765784564 \\
b &= 4565768798045375867957.67687978756453453
\end{aligned}
$$

Nous testons les deux fonction précédentes sur a et b et comparons le résultat à l'addition par défaut de MATLAB.

$$
\begin{aligned}
TwoSum(a, b) = 4.5658 \times 10^{21} \; y = 2.3465 \times 10^{-6} \\
FastTwoSum = 4.5658 \; times 10^2 1 \; y = 2.3465 \times 10^- 6 \\
a + b = 4.5658 \times 10^{21}
\end{aligned}
$$

Nos résultats confirment bien que ces fonctions renvoient l'addition des deux flottants a et b. Tous les deux isolent l'erreur absolue y que nous pouvons réinjecter dans le résultat.

7.3 EFT du produit de deux flottants

En ce qui concerne les EFT de produit de deux flottants, nous devons d'abord séparer les flottants en deux parties. Pour ce faire, nous utilisons la fonction **Split** de Dekker [3], décrite dans l'algorithme 4. Si q est le nombre de bits de la mantisse, alors nous considérons $r = \lceil q/2 \rceil$. L'algorithme 4 va donc séparer un nombre flottant a en deux parties x et y, tous les deux auront au plus $r-1$ bits différents de zéro tel que $a = x + y$. Par exemple, pour la norme IEE 754, $q = 53$ et $r = 27$, donc les nombres en sortie auront au plus $r - 1 = 26$ bits différents de zéro. Tel que le bit de signe est utilisé pour la séparation.

Algorithme 3 : Séparation d'un nombre flottant en deux parties

function $[x, y]$: **Split**(a)

$z = a \otimes (2^r + 1)$
$x = z \ominus (z \ominus a)$
$y = a \ominus x$

L'algorithme 4 de Veltkamp auquel nous allons nous intéresser maintenant est utilisé pour calculer l'EFT du produit de deux flottants. Il est plus communément appelé **TwoProduct** [3] et effectue 17 flops.

Algorithme 4 : EFT du produit de deux nombres flottants

function $[x, y]$: **TwoProd**(a, b).

$x = a \otimes b$
$[a_h, a_l] =$ **Split**(a)
$[b_h, b_l] =$ **Split**(b)
$y = a_l \otimes b_l \ominus (((x \ominus a_h \otimes b_h) \ominus a_l \otimes b_h) \ominus a_h \otimes b_l)$

Nous pouvons réécrire l'algorithme **TwoProd** d'une manière plus simple pour les processeurs qui disposent d'une FMA (décrite précédemment). Pour a, b et c and $\mathbb{F}$, $fma(a, b, c)$ est le résultat exact de $a \times b + c$ arrondi au flottant le plus proche. Alors, $y = a \times b - a \otimes b = fma(a, b, -(a \otimes b))$ et **TwoProd** peut être remplacé par l'algorithme ci-dessous qui n'effectue que 2 flops. Dans notre code nous utilisons la fonction prédéfinie fma() de la librairie Fixed-Point Designer. Nous définissons alors l'algorithme **TwoProductFMA** de Ogita, Rump et Oishi [4].

Algorithme 5 : EFT du produit de deux nombres flottants avec une FMA

function$[x, y]$: **TwoProductFMA**(a, b)

$x = a \otimes b$
$y = fma(a, b, -x)$

Nous pouvons remarquer alors que **TwoSum**, **TwoProduct** et **TwoProductFMA** n'effectuent que des opérations en virgule flottante bien optimisables. Ils n'utilisent pas de branches ni n'accèdent aux mantisses cce qui généralement demande beaucoup de temps.

Dans la suite, nous faisons l'hypothèse qu'aucune opération FMA n'est utilisée ailleurs que dans l'algorithme **TwoProductFMA**. Le but étant de concevoir un algorithme dont la preuve est valable dans n'importe quel ordinateur dépendant de la norme IEE-754. Toutes les conclusions dans la suite ainsi que le nombre de flops suivent cette hypothèse.

7.4 Résultats expérimentaux

Nous allons tester ces fonctions sur les deux flottants a et b de la partie précédente.

$$a = 2563657.579765784564$$
$$b = 4565768798045375867957.67687978756453453$$

$$TwoProd(a, b) = 1.1705 \times 10^{28} \; y = -8.0523 \times 10^{11}$$
$$TwoProductFMA(a, b) = 1.1705 \times 10^{28} \; y = -8.0523 \times 10^{11}$$
$$a \times b = 1.1705 \times 10^{28}$$

Nous remarquons ici aussi que les fonctionnent correctement. Elles calculent bien le produit de a et b et isolent bien la valeur de l'erreur y . Cependant, nous n'avons pas remarquer de différence entre l'exécution de TwoProd et TwoProdFMA, $maiscelanechangepaslefaitquetout$ TwoProdFMA$effectuemoinsd'oprations$.

7.5 Propriétés

Nous allons maintenant nous intérésser aux propriétés des deux algorithmes précédents, **TwoSum** et **TwoProd**, y compris en présence d'*underflow*

Théorème 1 :

Soient $a, b \in \mathbb{F}$ et $x, y \in \mathbb{F}$ tels que $[x, y] = \mathbf{TwoSum}(a, b)$. Alors, même en présence d'*underflow*.

$$a + b = x + y \ , \qquad x = a \oplus b, \qquad |y| \leq u|x|, \qquad |y| \leq u|a + b|$$

L'algorithme **TwoSum** effectue 6 flops.

Soient $a, b \in \mathbb{F}$ et $x, y \in \mathbb{F}$ tels que $[x, y] = \mathbf{TwoProd}(a, b)$. Alors, en l'absence d'*underflow*.

$$a + b = x + y \ , \qquad x = a \otimes b, \qquad |y| \leq u|x|, \qquad |y| \leq u|a \times b|$$

et en présence d'*underflow*.

$$a \times b = x + y + 5\eta \qquad x = a \otimes b, \ ; |y| \leq u|x| + 5v, \qquad |y| \leq u|a \times b| + 5v, \qquad |\eta| \leq v.$$

L'algorithme **TwoProd** effectue 17 flops.

8 Pour les complexes

Nous allons présenter présenter les algorithmes de EFT sur les complexes, que nous allons utiliser dans la suite pour nos évaluations. Comme pour les réels, nous allons présenter des algorithmes d'addition et de multiplication.

Dans tous les algorithmes qui suivent, nous allons considérer deux flottants x et y, où $x = a + ib$ et $y = c + id$. Avec a et c les parties réelles de x resp. y et b, d les parties imaginaires.

8.1 EFT de la somme de deux flottants complexes

L'algorithme utilisé pour l'EFT de la somme de deux flottants complexes est **TwoSumCplx** [5] qui est assez similaire à l'algorithme pour les réels **TwoSum** défini précédemment.

Algorithme 6 : EFT de la somme de deux flottants complexes

function $[s, e] = \mathbf{TwoSumCplx}(x, y)$

$[s_1, e_1] = TwoSum(a, c$
$[s_2, e_2] = TwoSum(b, d)$
$s = s_1 + is_2$
$e = e_1 + ie_2$

Théorème 2
Soient $x, y \in \mathbb{F} + i\mathbb{F}$ et soient $s, e \in \mathbb{F} + i\mathbb{F}$ tels que $[s, e] = TwoSumCplx(x, y)$ Alors :

$$x + y = s + e, s = fl(x + y), |e| \leq u|s|, |e| \leq u|x + y|$$

L'algorithme **TwoSumCplx** effectue 12 flops.

Preuve D'après le théorème 1, nous avons avec TwoSum $s_1 + e_1 = a + c$ et $s_2 + e_2 = b + d$, il en résulte que $s + e = x + y$. Comme $s = s_1 + s_2$ et $e = e_1 + e_2$, $x = a + ib$ et $y = c + id$

8.2 EFT du produit de deux flottants complexes

On ne peut pas directement appliquer l'algorithme TwoProd aux flottants complexes. Il faut d'abord séparer les parties les parties réelles des parties imaginaires et les traiter séparément pour ensuite rassembler le résultat à la fin.

Algorithme 7 : EFT du produit de deux nombres complexes

function $[p, e, f, g]$ = TwoProdCplx(x, y)

$[z_1, h_1] = TwoProduct(a, c)$
$[z_2, h_2] = TwoProduct(b, d)$
$[z_3, h_3] = TwoProduct(a, d)$
$[z_4, h_4] = TwoProduct(b, c)$
$[z_5, h_5] = TwoSum(z_1, z_2)$
$[z_6, h_6] = TwoSum(z_3, z_4)$
$p = z_5 + iz_6$
$e = h_1 + ih_3$
$f = h_2 + ih_4$
$g = h_5 + ih_6$

Il est possible d'optimiser l'algorithme précédent en ajoutant un appel direct à la fonction Split(). A chaque appel à l'algorithme TwoProd() on fait deux séparations pour a, b, c, d en appelant directement la fonction Split() on n'en fait qu'un seul.

Algorithme 8 : Produit de deux flottants complexes avec un seul Split.

function $[p, e, f, g]$ = TwoProdCplxSingleSplitting(x, y)

$[a_1, a_2] = Split(a)$
$[b_1, b_2] = Split(b)$
$[c_1, c_2] = Split(c)$
$[d_1, d_2] = Split(d)$
$z_1 = fl(a.c)$ $z_2 = fl(b.d)$ $z_3 = fl(a.d)$ $z_4 = fl(b.c)$
$h_1 = fl(a_2.c_2 - (((z_1 - a_1.c_1) - a_2.c_1) - a_1.c_2))$
$h_2 = fl(b_2.d_2 - (((z_2 - b_1.d_1) - b_2.d_1) - b_1.d_2))$
$h_3 = fl(a_2.d_2 - (((z_3 - a_1.d_1) - a_2.d_1) - a_1.d_2))$
$h_4 = fl(b2.c2 - (((z4 - b1.c1) - b2.c1) - b1.c2))$
$[z5, h5] = TwoSum(z1, z2), [z6, h6] = TwoSum(z3, z4)$
$p = z5 + iz6$
$e = h1 + ih3$
$f = h2 + ih4$
$g = h5 + ih6$

Algorithme 9 : Multiplication de deux flottants complexes avec FMA

function$[p, e, f, g]$ = TwoProdFMACplx(x, y)

$[z_1, h_1] = TwoProdFMA(a, c)$
$[z_2, h_2] = TwoProdFMA(b, d)$
$[z_3, h_3] = TwoProdFMA(a, d)$
$[z_4, h_4] = TwoProdFMA(b, c)$
$[z_5, h_5] = TwoSum(z_1, z_2)$
$[z_6, h_6] = TwoSum(z_3, z_4)$
$p = z_5 + iz_6$
$e = h_1 + ih_3$
$f = h_2 + ih_4$
$g = h_5 + ih_6$

Nous nous sommes servi principalement de l'algorithme de multiplication sans FMA.

8.3 Résultats expérimentaux

Nous avons testé ces fonctions complexes sur différents nombres complexes de différentes tailles, mais tous étaient des flottants. Ces algorithmes font tous les bons calculs et renvoient les bons résultats des additions et multiplications.

Tous les tests faits ainsi que leurs résultats se trouvent dans le fichier **Tests.m**

9 Schéma de Horner compensé

9.1 Pour les réels

Pour définir un schéma de Horner compensé, nous allons nous appuyer sur l'algorithme de Horner classique auquel nous allons ajouter les EFT de somme et de produit précédentes.

Tout d'abord, nous allons expliciter l'algorithme de Horner "classique"

Algorithme 10 : Évaluation polynomiale avec l'algorithme de Horner

function res = **Horner**(p, x)

$s_n = s_n$
$for\ i = n-1 : -1 : 0$
$\quad s_i = s_{i+1} \otimes x \otimes a_i$
end
$res = s_0$

Nous allons définir maintenant une EFT pour l'algorithme précédent. Il s'agit de l'algorithme **EFTHorner**

Algorithme 11 : EFT de l'évaluation polynomiale via l'algorithme de Horner

function$[\mathbf{Horner}(p, x), p_\pi, p_\sigma] = \mathbf{EFTHorner}(p, x)$
$s_n = a_n$
for $i = n-1 : -1 : 0$
$[p_i, \pi_i] =$ **TwoProduct**(s_{i+1}, x)
$[s_i, \sigma_i] =$ **TwoSum**(p_i, a_i)
π_i : le coefficients de degré i dans p_π.
σ_i : le coefficient de degré i dans $p\sigma$ end
Horner$(p, x) = s_0$

Maintenant, nous allons expliciter l'algorithme de Horner compensé qui utilise les algorithmes **TwoSum** et **TwoProd** dans l'algorithme 11 de Graillat, Langlois et Louvet [6].

Algorithme 12 : Schéma de Horner compensé

function res = **CompHorner**

$s_n = s_n$
$r_n = 0$
$for\ i = n-1 : -1 : 0$
$\quad [p_i, \pi_i] = \mathbf{TwoProd}(x, s_{i+1}$
$\quad [s_i, \sigma_i] = \mathbf{TwoSum}(p_i, a_i)$
$\quad r_i = r_{i+1} \otimes x \oplus (\pi_i + \sigma_i)$
end
$res = s_0 \oplus r_0$

Nous pouvons modifier cet algorithme pour le rendre plus rapide en remplaçant TwoProd par TwoProdFMA. Si cette modification est effectuée dans la suite, elle sera explicitée.

9.2 Pour les complexes

Nous allons définir deux nouveaux algorithmes dans cette section pour évaluer des polynômes dont les coefficients sont des flottants complexes. Nous considérerons un nombre flottant complexe $x = a + ib$

Le premier algorithme va être un schéma de Horner pour l'évaluation d'un polynôme dont les coefficients sont des flottants complexes.

Algorithme 13 : Horner complexe

function res = HornerCplx(p, x)
$rn = a_n//a_n = c + id$
$rn_r = c$
$rn_i = d$
$for\ i = 2 : 1 : n$
$rn_r = (rn_r * x) + real(p(i))$
$rn_i = (rn_i * z) + imag(p(i));$
$res = rn_r + i * rn_i$

Nous allons maintenant définir une implémentation de l'algorithme précédent avec une FMA

Algorithme 14 : Horner complexe avec FMA

function res = HornerCplxFMA(p, x)
$rn = a_n//a_n = c + id$
$rn_r = c$
$rn_i = d$
$for\ i = 2 : 1 : n$
$rn_r = fma(rn_r, z, real(p(j)))$
$rn_i = fma(rn_i, z, imag(p(j)));$
$res = rn_r + i * rn_i$

Maintenant, nous allons définir une EFT de l'algorithme de Horner sur les complexes.

Algorithme 15 : EFT du schéma de Horner complexe

function$[res, p_\pi, p_\mu, p_v, p_\sigma] = EFTHorner(p, x)$
$s_n = a_n$
for $i = n - 1 : -1 : 0$
$[p_i, \pi_i, \mu_i, v_i] = TwoProductCplx(s_{i+1}, x)$
$[s_i, \sigma_i] = TwoSumCplx(p_i, a_i)$
end
$res = s_0$

Où nous désignons par p_i, π_i, μ_i, v_i et σ_i les coefficients du degré i des polynômes p, p_π, p_v, p_σ

Nous allons maintenant décrire un algorithme qui calculera la somme de quatre polynômes différents. On appellera dans cette algorithme l'algorithme **Accsum** [4] que nous expliciterons dans la suite. Nous supposons ici que tous les polynômes sont de degré n

Algorithme 16 : Évaluation de la somme de quatre polynômes de degré n

function $c = HornerSumAcc(p, q, r, s, x)$
$v_n = Accsum(p_n + q_n + r_n + s_n)$
for $i = n - 1 : -1 : 0$
$v_i = fl(v_{i+1} \times x + Accsum(p_i + q_i + r_i + s_i))$
end
$c = v_0$

Algorithme 17 : Somme précise avec *faithful rounding**

function $res = AccSum(p)$
$[\tau_1, \tau_2, p'] = Transform(p)$
$res = fl(\tau_1 + (\tau_1 + (\sum_{i=1}^{n} p'_i)$

Cette fonction calcule la somme des éléments d'un vecteur p avec pour mode d'arrondi le "faithful rounding" qui veut dire, que pour l'approximation res que calcule la fonction et la vraie valaur $s = \sum p_i$ il n'y ait pas de nombre flottant entre et que si s est un nombre flottant, alors $res = s$. Cette fonction est très intéressante et permet de calculer une somme encore plus précise que ce que nous avons pu voir jusque là, malheureusement faute de temps, nous n'avons pas pu l'exploiter dans le reste de notre travail, mais le code adapté de cette fonction est disponible dans le fichier **AccSum.m**

Pour revenir à l'algorithme de Horner, nous avons défini une nouvelle fonction **EFTHornerCplx** qui est une EFT de l'évaluation polynomiale sur les complexes.

10 Évaluation de fonctions rationnelles réelles

Nous allons dans cette partie présenter deux algorithmes d'évaluation de fonctions rationnelles réelles. Nous définissons les fonctions rationnelles par la division de deux polynômes. Nous commencerons par un algorithme classique puis nous présenterons un autre algorithme avec compensation.

Dans la suite de cette partie nous allons considérer deux polynômes p et q tels que :

$$p = \sum_{i=0}^{n} a_i x^i \text{ et } q = \sum_{i=0}^{n} b_i x^i$$

Les polynômes p et q sont de degré n, leur coefficients sont des nombres flottants.
On définit la fonction rationnelle f comme suit :

$$f(x) = p(x)/q(x)$$

Alors, pour évaluer $f(x)$ il faudra d'abord évaluer $p(x)$ et $q(x)$ avec l'algorithme de Horner et ensuite faire une division des deux résultats obtenus [7].

Pour ces évaluations, nous allons comparer la division effectuée avec l'algorithme de division de MATLAB à la division effectuée avec nos propres algorithmes de division, pour les réels ainsi que les complexes. Ces fonctions se nomment **TwoDiv** et **TwoDivCplx** leur code se trouve dans les fichiers du même nom dans le dossier de code. Ces deux fonctions sont des EFT.

10.1 Évaluation classique de fonctions rationnelles

Pour une évaluation "classique" de fonctions rationnelles, nous allons décrire un algorithme qui repose sur le schéma de Horner "classique". On appellera alors la fonction Horner définie plus haut sur les polynômes p et q et le résultat renvoyé sera la division de ces deux évaluations. Cette procédure est décrite dans l'algorithme RatEval ci-dessous :

Algorithme 18 : Évaluation classique de fonctions rationnelles

function res = RatEval(p, q, x)

$res = Horner(p, x) \oslash Horner(q, x)$

Nous avons un autre version de cet algorithme où nous utiliserons l'EFT de la division.
Nous avons défini une algorithme pour faire une EFT de la division de deux réels. Cet algorithme **TwoDiv** est fortement inspiré de **TwoProd** dans le sens où il fait les opérations inverse. Nous avons donc testé l'évaluation des fonctions rationnelles réelles avec la division par défaut de MATLAB ainsi que par l'algorithme **TwoDiv**.

Algorithme 19 : Évaluation classique de fonctions rationnelles avec EFT

function res = EFTRatEval(p, q, x)

$rp = Horner(p, x)$
$rq = Horner(q, x)$
$res = TwoDiv(rp, rq)$

La différence entre ces deux algorithmes est qu'avec le deuxième il nous sera possible d'isoler l'erreur absolue y

10.2 Évaluation de fonction rationnelles avec un algorithme compensé

Pour cette partie nous allons suivre la même approche que la partie précédente mais au lieu d'utiliser l'algorithme de Horner classique, nous allons utiliser l'algorithme de Horner compensé [7].

Algorithme 20 : Evaluation de fonctions rationnelles avec un algorithme compensé

function res = CompRatEval(p, q, x)

$res = CompHorner(p, x) \oslash CompHorner(q, x)$

Comme pour l'évaluation classique, nous allons créer une deuxième version qui va utiliser une EFT de la division.

Algorithme 21 : Evaluation de fonctions rationnelles avec un algorithme compensé et EFT de la division

function res = EFTCompRatEval(p, q, x)

$rp = CompHorner(p, x)$
$rq = CompHorner(q, x)$
$res = TwoDiv(rp, rq)$

10.3 Résultats expérimentaux

Nous avons testé les fonctions sur un polynôme P de taille 1000. Les coefficients de ce polynôme sont tous des flottants compris entre 10^{-17} et 10^{17}. Nous avons évalué ce polynôme en un point $x = 2346547 * 10^{-12}$ et nous avons obtenu les résultats suivants :

$$Horner(p, x) = 1.0000 \times 10^{17}$$
$$CompHorner(p, x) = 1.0000 \times 10^{17} y = 4.0735 \times 10^{-11}$$
$$polyval(p, x) = 1.0000 \times 10^{17}$$

Nous avons obtenu les mêmes résultats qu'avec la fonction `polyval` de MATLAB. Malheureusement, faute de temps nous n'avons pas pu explorer ces résultats d'avantage.

Nous avons ensuite défini un autre polynôme q de degré 1000 aussi dont les coefficients sont tous des flottants assez petits mais tous positifs. Alors, pour l'évaluation de ce polynôme nous avons obtenus les résultats suivants :

$$Horner(q, x) = 0.5972$$
$$CompHorner(q, x) = 0.5972 \; y = -1.3402 \times 10^{-23}$$
$$polyval(q, x) = 0.5972$$

Nous constatons que les évaluations sont toutes bonnes.

Maintenant en ce qui concerner la division, nous avons testé deux approche. La première classique avec la division intégrée et la seconde avec notre EFT de division **TwoDiv**

$$\text{p(x)/q(X)} = 1.6745\text{e+}17$$
$$Horner(p, x)/Horner(q, x) = 1.6745 \times 10^{17}$$
$$CompHorner(p, x)/CompHorner(q, x) = 1.6745e \times 10^{17}$$

Avec EFT :
$$TwoDiv(Horner) = 1.6745 \times 10^{17} \; y = -4.1520 \times 10^{-9}$$
$$TwoDiv(CompHorner) = 1.6745 \times 10^{17} \; y = -4.1520 \times 10^{-9}$$

On peu déduire de ces tests que nos algorithmes fonctionnent correctement, seulement ils peuvent être plus lents que les algorithmes de MATLAB. Il faudrait faire d'avantage de tests sur des polynômes de tailles différentes pour pouvoir se rendre compte de l'impact que les EFT peuvent avoir et au bout de quel rang elle peuvent devenir intéressantes.

11 Évaluation de fonctions rationnelles complexes

Dans cette partie, nous allons faire les mêmes opérations que pour les réels mais nous utiliserons les fonctions HornerCplx pour l'évaluation classique et et EFTHornerCplx pour l'évaluation compensée.

Nous avons défini une fonction **TwoDivCplx** qui est une EFT de la division de deux nombres flottants complexes. Cette fonction appelle la fonction **TwoDiv** qui est une EFT de la division de deux flottants réels. La fonction **TwoDivCplx** donne de bons résultats pour des paires de flottants complexes de différentes tailles, elle isole bien l'erreur des parties réelles et imaginaire et envoie un paramètre contenant l'erreur absolue totale. Nous suivons ici la même approche que pour les fonctions rationnelles réelles mais nous utiliserons plutôt les fonctions définies pour les complexes.

La démarche à suivre sera donc pour deux polynômes aux coefficients flottants complexes p et q de les évaluer séparément avec **HornerCplx**, **EFTHornerCplx** et la fonction de MATLAB `polyval()`

Il faudra ensuite faire la division des résultats obtenus, pour ce faire, nous utiliserons la fonction **TwoDivCplx** sur les paires de résultats de chaque algorithme. Nous comparerons ensuite les résultats obtenus à ceux obtenus avec la division classique. Le but étant de savoir si notre fonction **TwoDivCplx** fait bien son travail et avec des tests supplémentaires pouvoir l'améliorer ou déduire un rang à partir duquel elle devient plus intéressante que la division classique.

Nous avons obtenus de bons résultats pour la division seulement il faudrait pouvoir tester les algorithmes d'avantages pour pouvoir se faire une meilleure idée.

12 Conclusion

Grâce à ce projet nous avons pu acquérir de nouvelles connaissances nous nous somme familiariser avec de nouvelles notions en arithmétique compensée. Nous avons découvert de nouveaux algorithmes performant qui permettent de gagner en précision et de palier aux erreurs d'arrondi inévitables lorsque l'on travaille avec des flottants. Nous avons pu élaborer de nouveaux algorithmes de division, même si nous n'avons pas eu assez de temps de tous bien les tester et les optimiser.

Références

[1] Nichols J. Higham. *Accuracy and stability of numerical algorithms.* the Society for Industrial and Applied Mathematics, 1996.

[2] D.E Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms.* Addison-Wesley, Reading, 1998.

[3] T. J. Dekker. A floating-point technique for extending the available precision. *Numerical Mathematics*, 18 :224–242, 1971.

[4] Siegfried M. Rump Takeshi Ogita and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6) :1955–1988, 2005.

[5] Valérie Ménissier-Morain Stef Graillat. Accurate summation, dot product and polynomial evaluation in complex floating point arithmetic. *Inormation and computationf*, 216 :57–71, 2012.

[6] N. Louvet S. Graillat and Ph. Langlois. Compensated horner scheme. *Inormation and computationf*, 216 :57–71, 2005.

[7] Stef Graillat. An accurate algorithm for evaluating rational functions. *Applied Mathematics and Computation*, 337(10) :494–503, 2018.