Grenoble INP - Ensimag

Documentation de l'extension

PROJET-GL GROUPE 49

BENTAIBI Oussama
AÏT HAMMOU Driss
AJJA Hamza
DAOUD YOUSSEF
LEACHOURI Khalil

2^{nd} Year - 29 janvier 2021

Table des matières

I Introduction

I.1 L'extension Math

Le projet GL a comme but d'implémenter un compilateur de langage deca, à ce but s'ajoute l'implémentation d'une extension au choix de l'équipe, notre choix était d'implémenter l'extension Math qui regroupe des fonctions mathématiques de base (trigonométriques en particulier).
Cette extension doit être implémentée en utilisant le langage deca et en prenant comme type principal les nombres flottants afin de représenter les nombres réels .
Plusieurs algorithmes existent pour implémenter ces fonctions mathématiques et notre mission est de trouver ceux qui fonctionnent avec la meilleure précision possible .
Ainsi nos choix d'algorithmes se basent principalement sur le but de minimiser l'imprécision qu'apporte d'une part le calcul des nombres flottants et d' autre part les approximations adoptées dans ces algorithmes .
La classe Math contient différentes méthodes principales (à côté des méthodes secondaires basiques) :

- Float ulp (float x)
- Float atan (float x)
- Float asin (float x)
- Float cos (float x)
- Float sin (float x)

On note que les angles utilisés dans l'implémentation des fonctions trigonométriques sont exprimés en radian.

I.2 Les nombres flottants

La représentation d'un nombre réel x à virgule en binaire n'est pas si facile que la représentation d'un nombre entier, cette représentation sur 32 bits (des flottants) se compose d'une partie s (1 bit de signe) qui nous indique le signe de ce nombre, une partie e (8 bits) qui représente l'exposant avec un décalage de 127 et une partie m (23 bits) appelée la mantisse, ces parties se regroupent dans la formule suivante :

$$x = (-1)^s \times m \times 2^e$$

Note : Dans la formule ci-dessus $m \geq 1$.

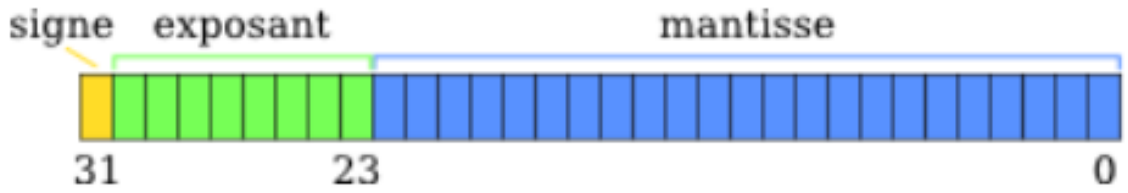

FIGURE 1 – Représentation des flottants en binaire

I.3 la mesure d'erreur et la validation des algorithmes

L'unité d'erreur considérée dans notre implémentation étant ULP (Unit in the Last Place) , on a utilisé la formule suivante qui ajoute cette unité à la formule classique de calcul d'erreur :

$$\textbf{Erreur (en ulp)} = \frac{|x-x'|}{ulp(x)}$$

Où x est la valeur qu'on cherche à approximer , x' et le résultat de notre implémentation et ULP(x) représente la distance entre x et le nombre flottant le plus proche de x .

Cette formule mesure l'écart entre la valeur cherchée et la valeur trouvée en

unité d'ulp .Pour étudier la précision des valeurs x' calculées par nos fonctions , on doit avoir une référence qui est dans notre cas les fonctions de la librairie math de JAVA ,ce qui semble comme une deuxième approximation de la valeur déjà approchée des fonctions de Math.java.
Cependant, en pratique , avoir un écart de quelques ulp est très suffisant pour implémenter des fonctions correctes.

II Fonction ULP

II.1 Implémentation

L'ulp d'un nombre flottant x mesure la distance entre ce nombre et le nombre flottant le plus proche .

La première étape était d'implémenter une fonction simple power(float x,int n) qui calcule la valeur x^n ,ensuite on a implémenté une fonction qui a pour but d'extraire l'exposant d'un nombre flottant .

On a utilisé le fait que la fonction ulp est paire et on a aussi réglé les résultats à retourner pour les valeurs particuliers de x à l'aide de la fonction Math.ulp de Java , par exemple pour un paramètre x=MAX_VALUE=2^{107} qui représente la valeur maximale qu'un nombre flottant peut prendre , notre fonction retourne le même résultat que la fonction de java , à savoir 104.

Pour les autres valeurs on a utilisé la formule suivante , qui représente le fait que le flottant le plus proche à x n'a que le dernier bit de la mantisse qui diffère :

$$1 \text{ ulp} = (0.00...1) \times 2^e = 1.0 * 2^{e-23}$$

Où e est l'exposant de le nombre x

II.2 Test de précision

L'implémentation suivie pour la fonction ulp était excellente puisque on n'a rencontré aucune erreur durant tous les tests.Ainsi, les résultats étaient identiques à ceux de la fonction Math.ulp de Java .La figure ci-dessous le confirme.

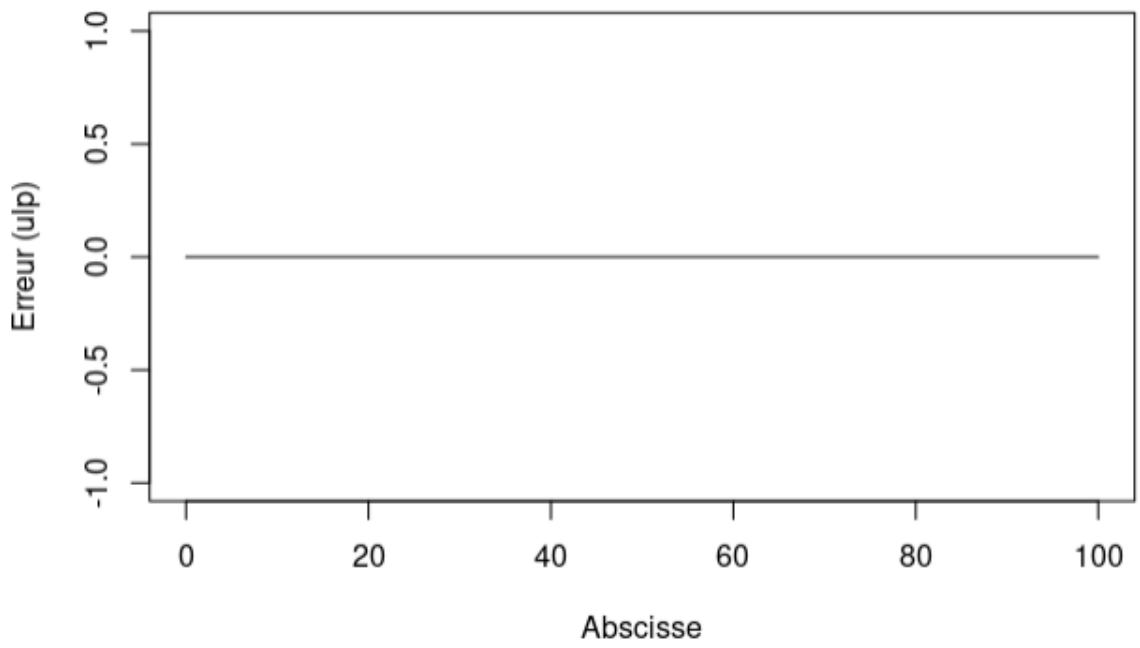

FIGURE 2 – Les erreurs de la fonction ULP

III Fonction Arc Tangente

III.1 Implémentation

La première idée qui nous est arrivée pour implémenter la fonction arc tangente est d'utiliser la formule d'approximation polynomiale classique du développement en série entière de arctan :

$$\forall x \in]-1,1[: arctan(x) = \sum_{n=0}^{+\infty} (-1)^n \frac{x^{2n+1}}{2n+1} \tag{1}$$

L'un des avantages que présente cette formule c'est que les coefficients ne demandent pas assez de calculs comme ceux des autres fonctions trigonométriques(absence des calculs des factoriels).
Par contre, cette formule n'est valable que sur l'intervalle]-1, 1[. C'est pour ça qu'on a pensé à utiliser la formules trigonométrique suivante afin de se rendre à des termes proches de 0 où l'approximation est plus bonne

$$arctan(\frac{1}{x}) + arctan(x) = \frac{\pi}{2} \tag{2}$$

Cependant, on remarque que si on se rapproche de 1 ou -1 l'erreur commence à augmenter. On a donc eu recours à une autre formule trigonométrique qui nous rapproche de 0.

$$arctan(x) = 2 \times arctan(\frac{x}{1+\sqrt{1+x^2}}) \tag{3}$$

Finalement, notre implémentation repose sur les trois formules (1), (2), (3) :

$$\begin{cases} arctan(x) &= \sum_{n=0}^{+\infty}(-1)^n \frac{x^{2n+1}}{2n+1} & \text{si } |x| \leq 0.7 \\ arcatn(x) &= 2 \times arctan(\frac{x}{1+\sqrt{1+x^2}}) & \text{si } 0.7 < |x| \leq 1.3 \\ arctan(x) &= \frac{\pi}{2} - arctan(\frac{1}{x}) & \text{si } |x| > 1.3 \end{cases}$$

III.2 Test de précision

Comme on a dit dans le paragraphe précédent, l'utilisation de la formule classique des séries entières entre -1 et 1 n'est pas précise sur tous l'intervalle vu qu'elle diverge aux alentours de 1 et -1 (voir la figure ci-dessous).

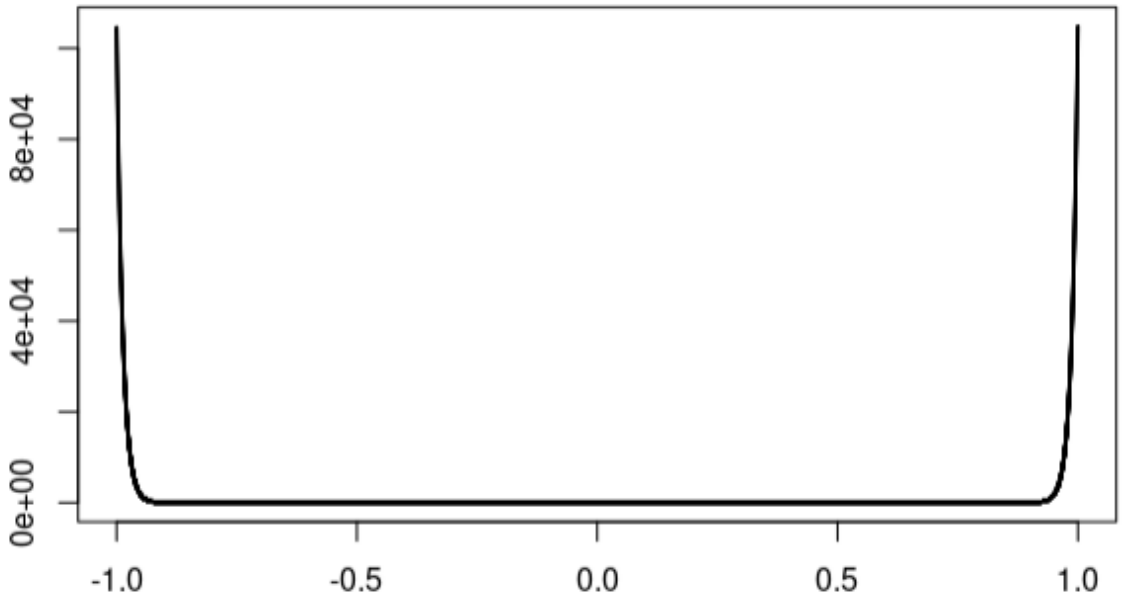

FIGURE 3 – L'erreur en ULP sur]-1, 1[de la fonction arc tangente en utilisant la formule (1)

Cependent la dernière version implémentée de l'arc tangente donne un erreur de 5 ULP au maximum(voir la figure ci dessous) .

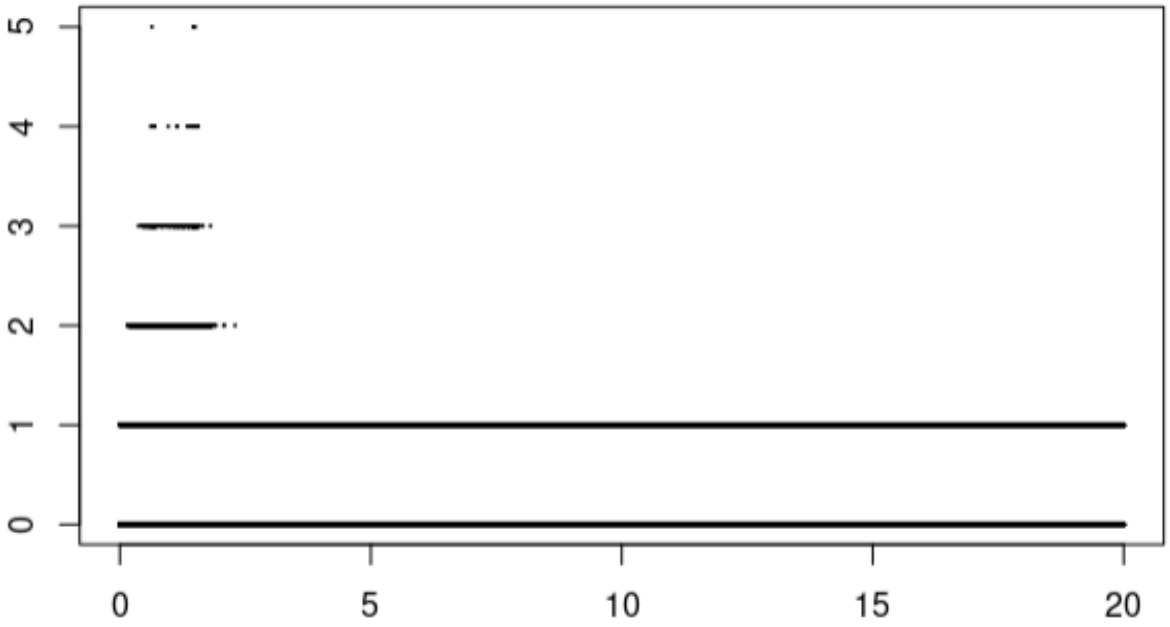

FIGURE 4 – L'erreur en ULP sur [0, 20] de la dernière version implémentée de l'arc tangente

Vous trouverez ci-dessous l'erreur sur des différentes intervalles

Intervalle	Nombre de tests	Pourcentage d'erreur	Erreur moyen	Erreur maximal
[0, 0.12]	125830	0%	0	0
[0.12, 0.45]	346031	3.23%	2	3
[0.45, 0.7]	262145	15.6%	2.13	5
[0.7, 1.3]	629146	11.43%	2	4
[1.3, 20]	629146	0.38%	2.18	5
[10000, 100000]	2880001	0%	0	0

IV Fonction Arc sinus

IV.1 Implémentation

Pour calculer l'arc sinus (une fonction impaire) , on a préféré d'utiliser la fonction arctangent implémentée précédemment avec une très bonne précision au lieu de faire les calculs lourds dans le développement en série entière de l'arcsinus :

$$\sum_{n=0}^{+\infty} \frac{(2n)!}{2^{2n}(n!)^2} \frac{x^{2n+1}}{2n+1}$$

Ainsi , on a utilisé la formule suivante qui relie les deux fonctions arc tangent et arc sinus :

$$arcsin(x) = arctan(\frac{x}{\sqrt{1-x^2}})$$

IV.2 Test de précision

L'approche utilisée pour calculer l'arc sinus a donnée de bonne résultats comme vous pouvez remarquer dans la figure ci-dessous. En effet, on trouve un erreur de 7 ULP au maximum surtout si on se rapproche de 1 ou de -1.

Intervalle	Nombre de tests	Pourcentage d'erreur	Erreur moyen	Erreur maximal
[0, 0.2]	52428	1.83%	2	3
[0.2, 0.5]	78643	10.18%	2	4
[0.5, 0.9]	104857	14.77%	2.14	6
[0.9, 0.98]	20971	0.27%	2.0	2.0
[0.98, 1[	20971	8.20%	2.43	7

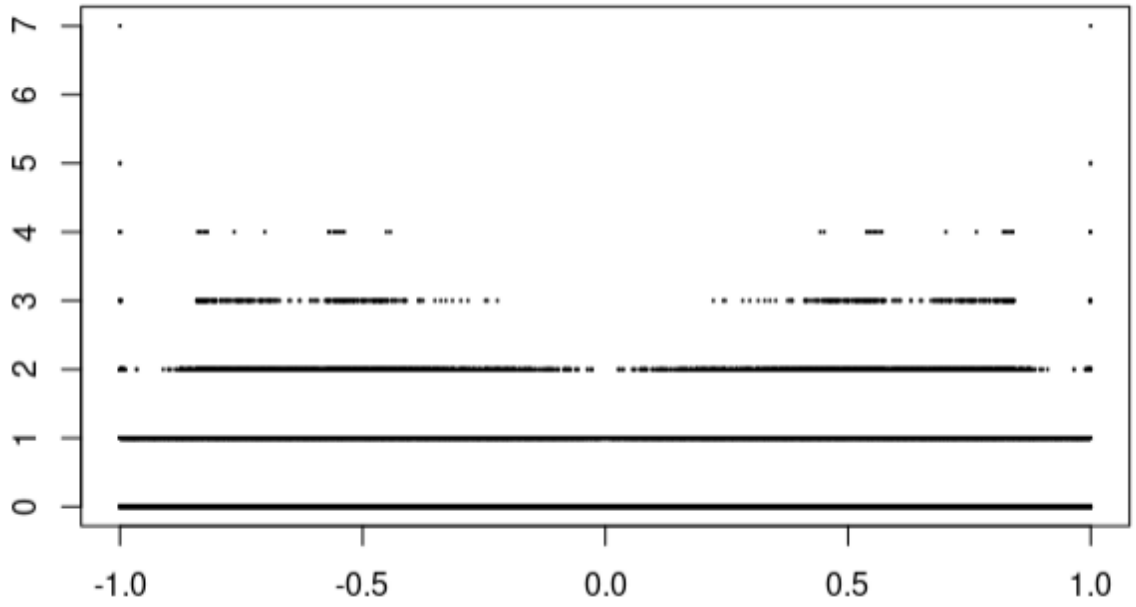

FIGURE 5 – L'erreur en ULP sur]-1, 1[de la fonction arc sinus implméntée

V Fonctions Cosinus et Sinus

V.1 Algorithme de Cordic

L'algorithme de Cordic consiste à déterminer le cosinus et le sinus d'un angle α donné en Radian. Pour ce faire, on cherche les coordonnées x et y du point correspondant à cet angle sur le cercle unité. Autrement dit, On essaye de construire une suite de points $\begin{pmatrix} x_n \\ y_n \end{pmatrix}$ du cercle trigonométrique qui converge vers $\begin{pmatrix} cos(\alpha) \\ sin(\alpha) \end{pmatrix}$ en partant du point $\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ et en lui appliquant des rotations.

En définissant les rotations suivantes $\theta_0 \geq \theta_1 \geq ... \geq \theta_n$ qui diminuent de plus en plus, on calcule à chaque itération i le vecteur $\begin{pmatrix} x_i \\ y_i \end{pmatrix}$ de la manière suivante :

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} cos(\theta_i) & -sin(\theta_i) \\ sin(\theta_i) & cos(\theta_i) \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \tag{4}$$

On définit les facteurs $\sigma_i = 0$ ou $\sigma_i = 1$ qui indiquent le sens de rotation. Et après avoir foctiriser par $cos(\theta_i)$ dans l'équation (4) on obtient :

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = cos(\theta_i) \begin{pmatrix} 1 & -\sigma_i tan(\theta_i) \\ \sigma_i tan(\theta_i) & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

Le but maintenant c'est de choisir les angles θ_i dont on connait la tangente. Ainsi, l'algorithme de Cordic consiste à choisir les angles θ_i tels que $\tan(\theta_i) = 2^{-i}$.

En notant $K_i = \cos(\arctan(\theta_i))$, on obtient :

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = K_i \begin{pmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

On peut ignorer ces facteurs multiplicatifs K_i en utilisant un seul coefficient :

$$K = \prod_{i=0}^{+\infty} K_i \approx 0.60725293510314$$

Finalement, la dernière étape de l'algorithme consiste à savoir le sens de rotation à chaque iteration. Pour ce faire, on vérifie si l'angle

$$\alpha_{i+1} = \alpha_i - \sigma\theta_i = \alpha_i - \sigma \arctan(2^{-i})$$

est positive ou négative de façon à se rapprocher de l'angle α.

V.2 Résultats obtenus

Après avoir implémenter l'algorithme de Cordic pour récupérer les valeurs approchées de cosinus et sinus, on remarque que l'erreur en ULP augmente avec l'augmentation de l'angle(voir les figures ci-dessous), par exemple le pourcentage d'erreur rencontrée lors de l'exécution de sinus sur $[0, \pi]$ est 73.429245%. Ainsi, on a pensé à faire une réduction pour chaque angle à l'intervalle $[0, \frac{\pi}{4}]$ où l'approximation est plus bonne. Pour ce faire, on va utiliser l'algorithme de Cody & Waite.

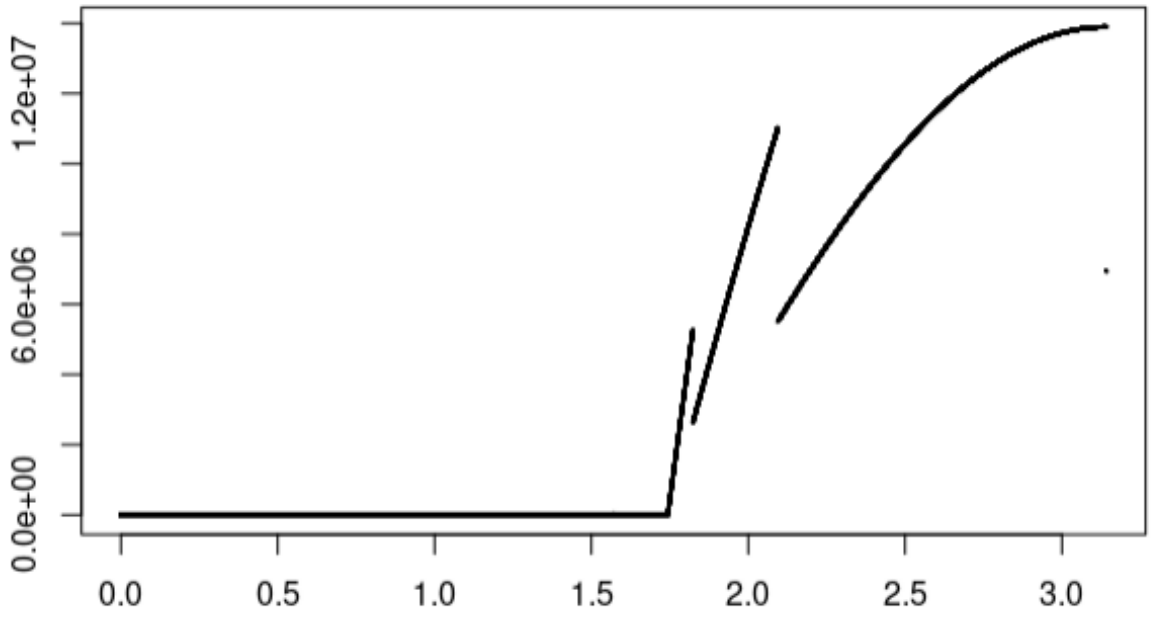

FIGURE 6 – L'erreur en ULP sur $[0, \pi]$ de la fonction **cosinus** implémentée en utilisant l'algorithme de Cordic

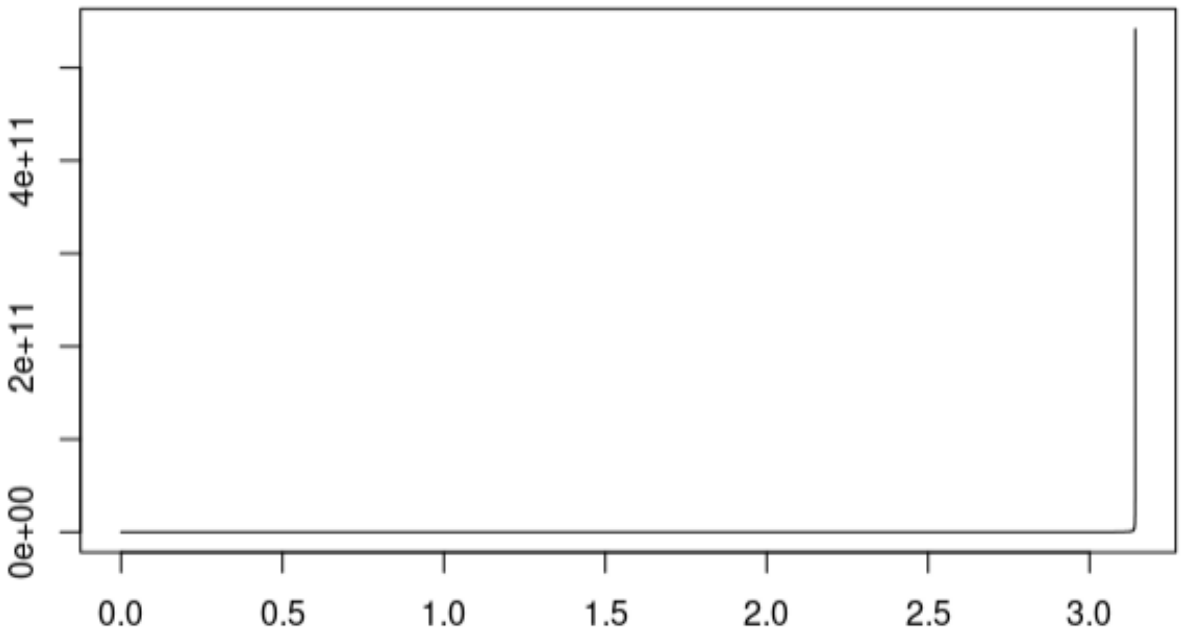

FIGURE 7 – L'erreur en ULP sur $[0,\pi]$ de la fonction **sinus** implémentée en utilisant l'algorithme de Cordic

V.3 Algorithme de Cody and Waite

La méthode de Cody and Waite a pour but de réduire un paramètre flottant x d'une fonction (trigonométrique) à un intervalle où cette fonction a une bonne précision , dans notre cas on cherche à ramener les valeurs supérieures à $\frac{\pi}{4}$ à des valeurs proches de 0, plus précisemment $[0, \frac{\pi}{4}]$.
Pour un nombre flottant x on cherche à trouver la valeur réduite r tels que $r = x - k \times C$ où C est égale à $\frac{\pi}{2}$. Ainsi on cherche à calculer k qui représente la partie entière de $\frac{x}{C}$ et pour avoir plus de précision dans le calcul à faire , on décompose la constante C en des constantes plus petites C1,C2... qui représentent des parties de la mantisse de $\frac{\pi}{2}$.
Ensuite , selon la valeur de k, on utilise une identité trigonométrique pour calculer l'image de x par notre fonction .
En essayant d'implémenter l'algorithme Cody and Waite en Deca , on a remarqué le besoin d'implémenter une fonction round ayant le même fonctionnement que Math.round de Java (retourner la partie entière d'un flottant) puisqu' elle joue un rôle important dans l'implémentation de Cody and Waite .
Pour implémenter une telle méthode on a décidé d'utiliser ASM avec le code suivant :

```
round(float x) asm ("
LOAD -3(LB), R1
INT R1, R1
LOAD R1, R0 ")
```

Cette méthode prend d'abord le paramètre x à arrondir situé à -3(LB) et le pose dans le registre R1 ,ensuite elle le transforme en un entier à l'aide de la commande INT et enfin elle le met dans le registre de retour R0 .

V.4 Résultats de l'algorithme de Cordic et Cody & Waite

On remarque que l'algorithme de Cody & Waite avait un impact sur la précision de notre implémentation des fonctions trigonométriques. Par exemple, l'erreur pour la fonction cosinus a diminué sur $]0, \frac{\pi}{2}]$ par rapport à l'utilisation de Cordic tout seul.

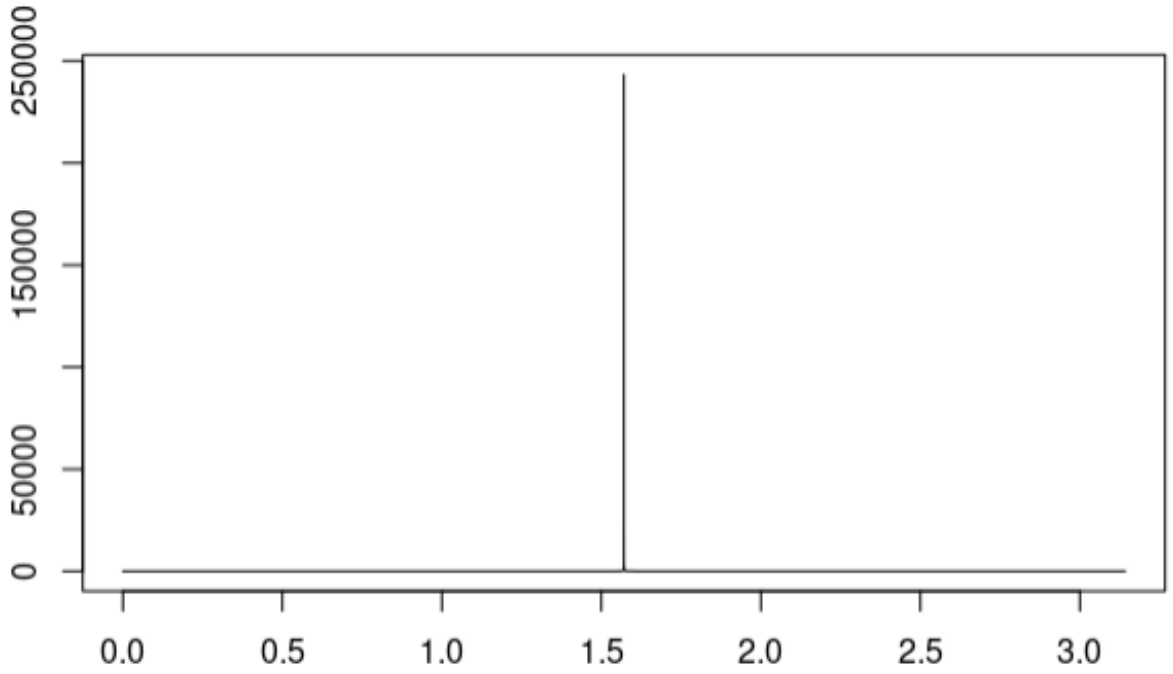

FIGURE 8 – L'erreur en ULP sur $[0, \pi]$ de la fonction **cosinus** implémentée en utilisant l'algorithme de Cordic et Cody & Waite

Dans la figure ci-dessous, on peut bien remarquer la différence après et avant l'utilisation de l'algorithme de Cody & Waite.

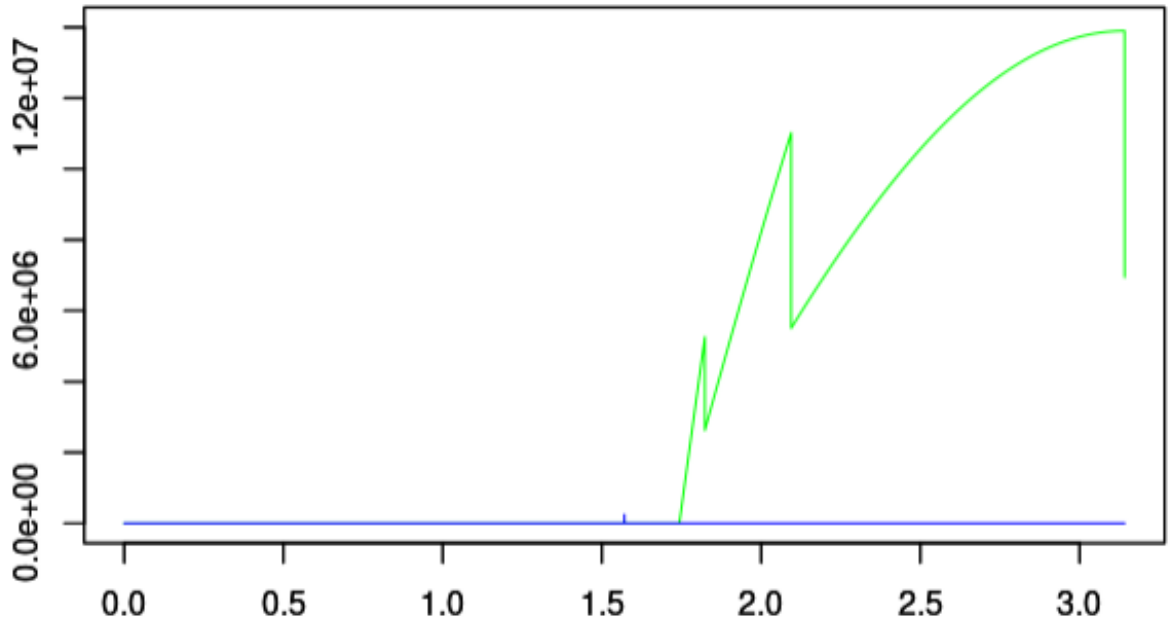

FIGURE 9 – La différence d'erreur pour la fonction **Cosinus** avant et après l'utilisation de l'algorithme de Cody & Waite

On remarque aussi une diminution de l'erreur pour la fonction sinus après l'utilisation l'algorithme de Cody & Waite. En effet, on voit que l'erreur est devenu 32.984924% à la place de 73.429245%. Cependent, 32.98% reste un grand pourcentage et l'erreur est plus grande aus alentours de $\frac{\pi}{2}$ pour la fonction cosinus et π pour la fonction sinus. Ainsi, on a pensé à utilisé le développement en série entière.

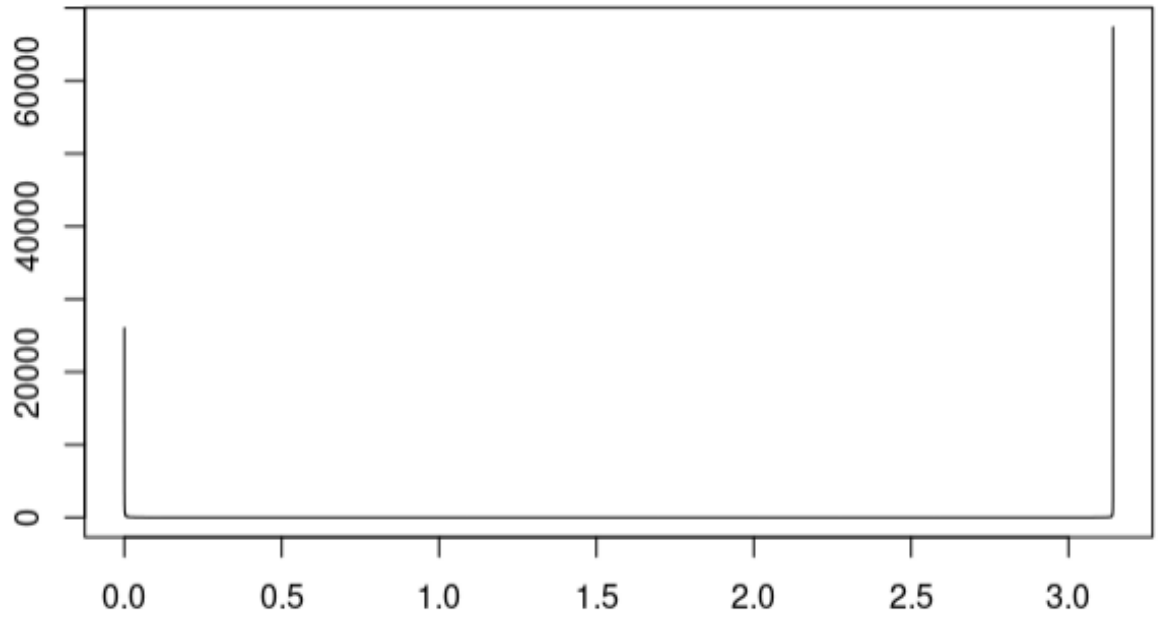

FIGURE 10 – L'erreur en ULP sur $[0,\pi]$ de la fonction **sinus** implémentée en utilisant l'algorithme de Cordic et Cody & Waite

V.5 développement en série entière

La deuxième méthode proposée était l'approximation polynomiale à l'aide de développement en séries entières de cos et sin ,ces approximations implémentées à l'aide de la méthode de Horner pour le calcul polynomiale et de la parité/imparité de ces fonctions , ont une bonne précision autour de 0 , et en particulier sur les intervalles inclus dans $[0,\frac{\pi}{4}]$.$\forall x \in \mathbb{R}$:

$$cos(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$$

$$sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

Cependant , pour des plus grandes valeurs , l'erreur devient inacceptable ainsi on a utilisé la méthode de Cody and Waite expliquée précédemment pour réduire le paramètre (range reduction) et se rendre à un paramètre r proche de 0 et dont le cos/sin est lié à celui de x à l'aide d'une identité trigonométrique.

V.6 Tests de précision

V.6.1 Cosinus

Après avoir testé la fonction cosinus, on trouve de bonnes résultats sauf si on se rapproche des multiples de $\frac{\pi}{2}$. vous pouvez le constatez dans le tableau ci-dessous. En plus de ça plus les nombres commencent à augmenter plus l'erreur augmente. malheureusement, on n'a pas pu reglé ces deux problèmes.

Intervalle	Nombre de tests	Pourcentage d'erreur	Erreur moyen	Erreur maximal
$[0, \frac{\pi}{4}]$	823550	0%	0	0
$[\frac{\pi}{4}, \frac{\pi}{2} - \frac{\pi}{100}]$	820256	0.76%	2.42	5
$[\frac{\pi}{2} - \frac{\pi}{100}, \frac{\pi}{2}]$	32942	27.6%	14.6	17367
$[\frac{\pi}{2} + \frac{\pi}{100}, 3\frac{\pi}{2} - \frac{\pi}{100}]$	3228315	0.036%	2.0	2.0
$[3\frac{\pi}{2} - \frac{\pi}{100}, 3\frac{\pi}{2}[$	32942	79.5%	16.6	52101.0
$[10\pi, 10\pi + \frac{\pi}{2} - \frac{\pi}{100}]$	50443	17.68%	3.67	10
$[100\pi, 100\pi + \frac{\pi}{2} - \frac{\pi}{100}]$	50443	59.89%	6.6	42

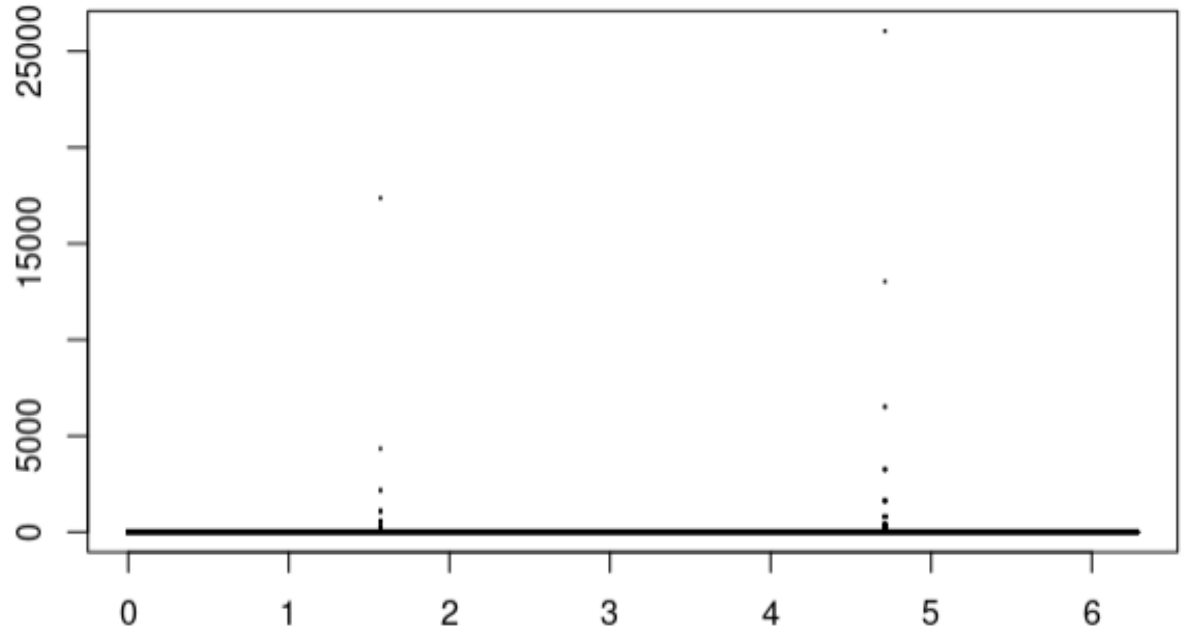

FIGURE 11 – L'erreur en ULP sur $[0, 2\pi]$ de la fonction **cosinus** implémentée en utilisant les séries entières et Cody & Waite

V.6.2 Sinus

Après avoir testé la fonction sinus, on trouve de bonnes résultats sauf si on se rapproche des multiples de π. vous pouvez le constatez dans le tableau ci-dessous. En plus de ça plus les nombres commencent à augmenter plus l'erreur augmente. malheureusement, on n'a pas pu reglé ces deux problèmes.

Intervalle	Nombre de tests	Pourcentage d'erreur	Erreur moyen	Erreur maximal
$[0, \pi - \frac{\pi}{100}]$	3261257	0.082%	2	2
$[\pi - \frac{\pi}{100}, \pi]$	32943	55.27%	30.79	277872.0
$[\pi + \frac{\pi}{100}, 2\pi - \frac{\pi}{100}]$	3228315	0.19%	2	2
$[10\pi + \frac{\pi}{100}, 11\pi - \frac{\pi}{100}]$	100886	18.03%	5.0	22.0
$[100\pi + \frac{\pi}{100}, 101\pi - \frac{\pi}{100}[$	100887	59.51%	13.0	127.0

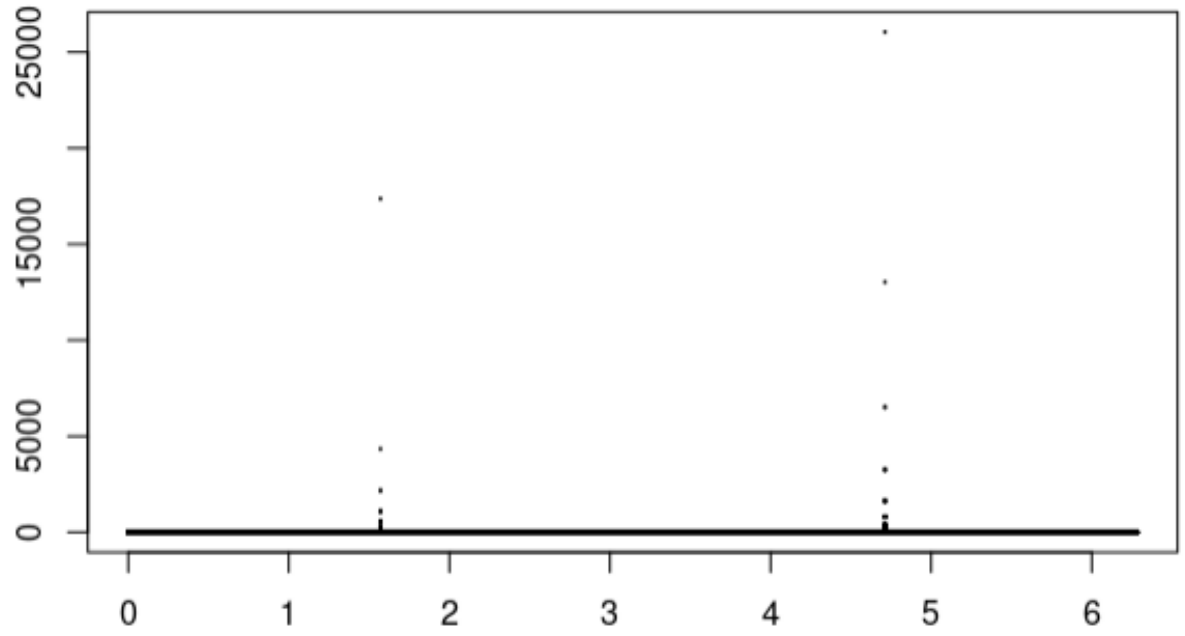

FIGURE 12 – L'erreur en ULP sur $[0,2\pi]$ de la fonction **sinus** implémentée en utilisant les séries entières et Cody & Waite

V.7 Algorithme final

Finalement , nous avons décidé d'adopter l'algorithme de développement en séries entières avec la méthode de Cody and Waite à cause de la bonne précision qu'ils fonctionnent avec et le pourcentage d'erreur qui est faible relativement aux autres algorithmes.

VI Conclusion

L'implémentation de cette extension math nous a permet d'avoir plus de détails sur les calculs des nombres flottants, les mesures d'erreur liées à ces calculs et les façons dont on peut gérer ces erreurs .
Maintenant, dans ce stage avancé de l'implémentation de cette extension , on comprend mieux pourquoi $1.0+1.0 \neq 2.0$:)

Références

[1] **Algorithme de Cordic** http ://cdeval.free.fr/IMG/pdf/cordic.pdf

[2] **Algorithme de Cody & Waite** Jean-Michel Muller, Elementary Functions, Algorithms and Implementation, second edition, pages 177–184, 2005

[3] **Séries de Taylor** https ://fr.wikipedia.org/wiki/Série_de_Taylor

[4] **Méthode de Ruffini-Horner** https ://fr.wikipedia.org/wiki/Méthode_de_Ruffini-Horner