Documentation de l'extension

Projet GL Équipe 16
Amine Angar
Mouad Tarbaoui
Dy el Alem
Oussama Kaddami
Youness Lehdili
27 janvier 2021

Contents

1 La Classe Math

Les fonctions implémentées dans le fichier Math.Decah sont :

1. _min(float a, float b): renvoie le minimum entre les flottants passés en argument;
2. _max(float a, float b): renvoie le maximum entre les flottants passés en argument.
3. fact(int n): renvoie le factoriel de l'entier passé en argument;
4. pow(float a, int b) : lève a à la puissance de b et renvoie le résultat sous la forme flottant;

5. _Ulp(float x) : renvoie la taille d'un Ulp de l'argument;
6. _fmod(float x, float y) : calcule le reste de la division entre deux flottants;
7. _abs(float a):renvoie la valeur absolue d'un flottant;
8. _signe(float z) : renvoie le signe d'un flottant;
9. _cos(float x): calcule et renvoie le cosinus du flottant passé en argument;
10. _sin(float x) :calcule et renvoie le sinus du flottant passé en argument;
11. _asin(float x):calcule et renvoie le arcsinus du flottant passé en argument;
12. _sqrt(float a):calcule et renvoie le racine carrée du flottant passé en argument;
13. _atan(float x) :calcule et renvoie le arcstangente du flottant passé en argument.

2 Fonction ulp.

Pour le calcul de l'ulp d'un flottant x (distance entre x et le flottant supérieur le plus proches), nous avons implémenté 2 algorithmes:
Nous avons utilisé la parité de la fonction ulp pour se ramener au cas d'un argument positif. De plus, en langage deca, on ne peut pas prendre en compte les valeurs NaN et les différents infinis comme en Java.
nous traitant à part les cas particuliers :

- $-\infty, +\infty$ sont représenté respectivement par les flottants 1.0f/0.0,
- maxVlue correspond à la valeur maximale d'un flottant(3.4028234663852886E38f)
- minValue correspond à la valeur minimale d'un flottant, soit 1.4E-45f
- NaN est représenté par le flottant 0.0f/0.0f

le premier algorithme consiste à diviser une puissance p de 2 par 2 jusqu'à ce que la somme p + f (avec f le nombre passé en argument),soit égale à f,ulp(f) est la dernier valeur de p qui vérifier $p + f \neq f$ cette méthode donne des résultats conformes à celle donné par java pour les flottants inférieurs à 10^{12}

le deuxième algorithme consiste à calculer la différence entre le flottant supérieur f^+ le plus proche et celui donné en paramètre f, on calcule f^+ en changeant le dernier bit de la mantisse de f, cette méthode donne des résultats parfaitement identiques à celle donné par le module Math.(sur 74.305.743 test)

2.1 Mesure de la précision.

La mesure de précision qu'on a adopter est basé sur l'Unit in the last place, soit _f ,(f) une des fonction qu'on a implémenté(f la même fonction dans le module java), afin de mesurer l'approximation , on calcule le rapport $\frac{|f(x)-\_f(x)|}{ulp(x)}$ pour exprimer l'écart en ulp entre la valeur approchée et la valeur exacte.

2.2 Validation

Afin de s'assurer de la précision de l'approximation qu'on a choisi, Nous avons confronter les résultats avec celles donnés par Java,l'objectif que nous avons fixé est d'atteindre un niveau de précision de quelques ulps (<=3 ulps), le tableau et la figure 2 montrent les résultats que nous avons obtenu pour la fonction ulp(sans erreur).

	intervalle	nombre d'erreur	erreur moyenne en ulp	pas	nombres de tests
ulp	$[0; 1]$	0	0 ulp	2^{-15}	32.768.000
ulp	$[0; 10.00]$	0	0 ulp	2^{-18}	62.144.000
ulp	$[1.000; 1.000.000]$	0	0 ulp	2^{-15}	32.768.000

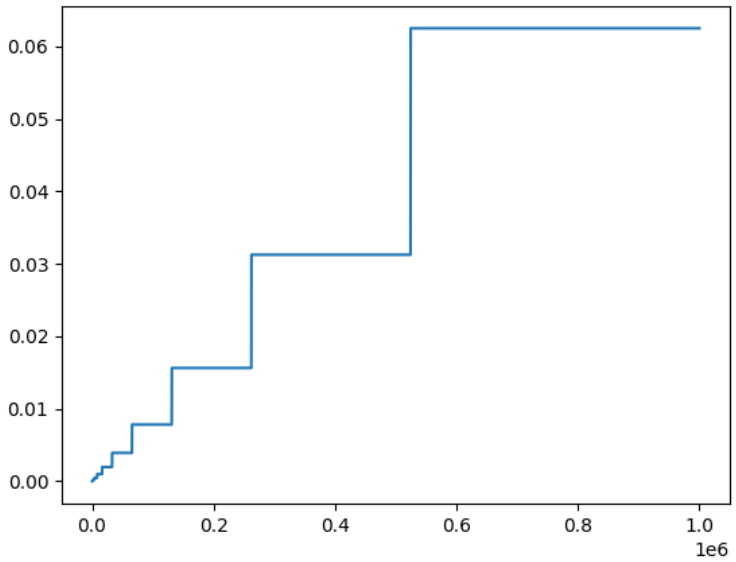

Figure 2: Nombres flottants simple précision.

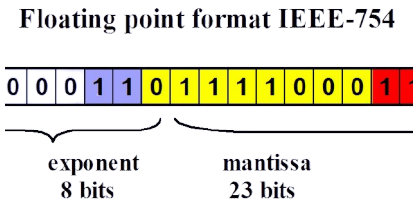

Figure 1: Nombres flottants simple précision.

3 Fonction atan.

Le premier algorithme qu'on tester est celui du développement en série entière, Dans tous les algorithmes utilisés, on utilisera la propriété $arctan(x)+arctan(1/x)=\frac{\pi}{2}$. De plus on se servira du caractère impair de l'arc-tangente. On restreint ainsi l'intervalle à [0,1].
L'implementaton de la fonction arc-tangente qu'on a développer est basé sur le polynôme d'Hermite d'ordre 7, la fonction arctan est impaire , on peut donc se ramener au l'intervalle $[0,+\infty[$, l'implémentation du polynôme d'hermite donne des résultats identiques a celle donné par java à
le principe de cette méthode est le suivant :
le polynôme d'hermite approchant la fonction arc-tangente est définie par la formule :

$$h_m(x)=\int_0^x \frac{(-1)^{m+1}}{4^m}p_m(t)dt \tag{1}$$

avec

$$p_m=\begin{cases} 4-4x^2+5x^4-4x^5+x^6 & \text{si m} = 1 \\ 1-x)^4x^4p_{m-1}(x)+(-4)^{m-1}p_1(x) & \text{sinon.} \end{cases}$$

par exemple:

le polynôme d'ordre 7 qu'on a utilisé permet une précision de $(\frac{1}{4})^{35}=8.470329472543003e-22$ ce polynôme vérifie :

$$|h_m(x)-arctan(x)|\leq(\frac{1}{4})^{5m} \tag{2}$$

la fonction arctan est impaire , on peut donc se ramener au l'intervalle $[0,+\infty[$, l'implementation du polynôme d'Hermite donne des résultats identiques a celle donné par java à 2 ulps prés en moyenne.

Afin de se ramener a des valeurs proches de 0 nous avons adopter la stratégie suivante:
Si $\mid x \mid > 0.6875$ et $\mid x \mid < 1.1875$ nous utilisons la formule trigonométrique suivante pour se ramener à des calculs proche de 0:

$$\arctan x+\frac{\pi}{4}=\arctan\left(\frac{x+1}{1-x}\right)$$

Si $\mid x > 1.1875 \mid$ nous utilisons la formule trigonométrique suivante pour se ramener à des termes proches de 0:

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

le tableau suivant montre le calcul de précision qu'on a effectué sur la fonction arctangeante,(2 ulps en moyenne, 6 ulps au maximum).

3.1 Validation

		intervalle	nombre d'erreur	erreur moyenne en ulp	pas	nombres de tests
Y	atan	[0; 1.000]	0	2 ulp	2^{-15}	1.024.000
	atan	[1.000; 100.00]	0	1 ulp	2^{-18}	3.168.000

le polynôme d'Hermite d'ordre 7 approchant l'arctangente est le suivant :

$$H_7(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \frac{x^{11}}{11} + \frac{x^{13}}{13} - \frac{x^{15}}{15} + \frac{x^{17}}{17} - \frac{x^{19}}{19} + \frac{x^{21}}{21} - \frac{x^{23}}{23} + \frac{x^{25}}{25} - \frac{x^{27}}{27}$$
$$+ \frac{565x^{29}}{16384} - \frac{7x^{30}}{122880} - \frac{16007x^{31}}{507904} - \frac{203x^{32}}{32768} + \frac{18241x^{33}}{270336} - \frac{11879x^{34}}{69632} + \frac{170129x^{35}}{286720} - \frac{68063x^{36}}{36864} +$$
$$\frac{2767847x^{37}}{606208} - \frac{1454473x^{38}}{155648} + \frac{10355263x^{39}}{638976} - \frac{489259x^{40}}{20480} + \frac{5016623x^{41}}{167936} - \frac{1361617x^{42}}{43008} + \frac{5012527x^{43}}{176128} -$$
$$\frac{489259x^{44}}{22528} + \frac{10371647x^{45}}{737280} - \frac{1454473x^{46}}{188416} + \frac{2751463x^{47}}{770048} - \frac{68063x^{48}}{49152} + \frac{178321x^{49}}{401408} - \frac{11879x^{50}}{102400} + \frac{10049x^{51}}{417792} - \frac{203x^{52}}{53248}$$
$$+ \frac{377x^{53}}{868352} - \frac{7x^{54}}{221184} + \frac{x^{55}}{901120}$$

la courbe suivante montre la différence en ulp entre nos valeurs et celle données par Java.

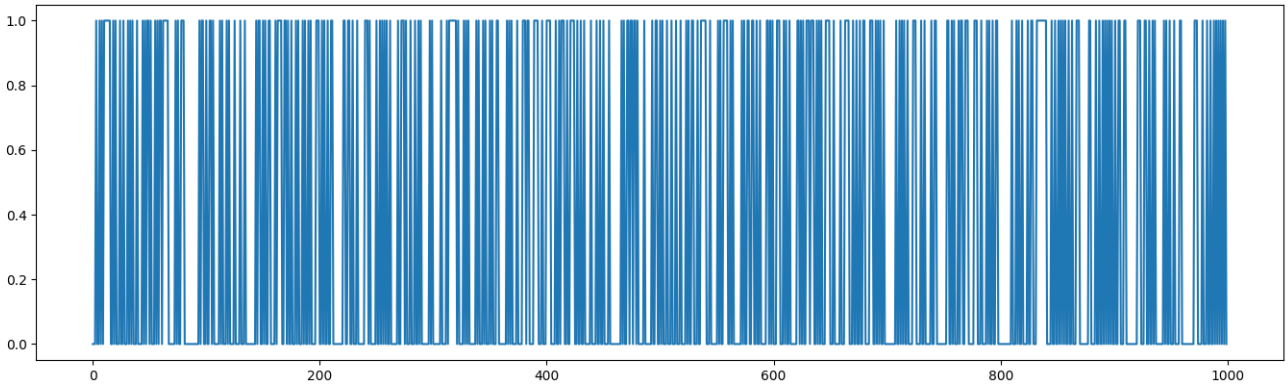

Figure 3: Courbe représentant la différence en ulp comme indiqué dand 1.1.

4 Fonction asin.

Nous avons utilisé le développement en série entière pour calculer une approximation de la fonction arcsinus

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

On aura une imprécision supplémentaire dû au grand nombre d'opérations, plus on s'éloigne de 0 ,plus la précision de l'approximation diminuer, on a utilisé des formules mathématique afin de se ramener à des valeurs proches de 0:
Si $\mid x \mid < 0.72$ nous utilisons la méthode de horner sur le développement en série entière .
Si $\mid x \mid > 0.72$ nous utilisons une formule trigonométrique pour se ramener proche de 0:

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

La méthode de Horner permet de diminuer le nombre d'opérations en comparaison au calcul de

manière basique ce qui améliore la précision de la fonction:

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

4.1 Validation

	intervalle	nombre d'erreur	erreur moyenne en ulp	pas	nombres de tests
asin	$[-1; 1]$	8042	2 ulp	2^{-22}	8.388.608
asin	$[0.72; 1]$	4939	2 ulp	2^{-22}	73401
asin	$[-1; -0.72]$	4944	2 ulp	2^{-22}	73401
asin	$[-0.72; 0.72]$	996	2 ulp	2^{-22}	377488

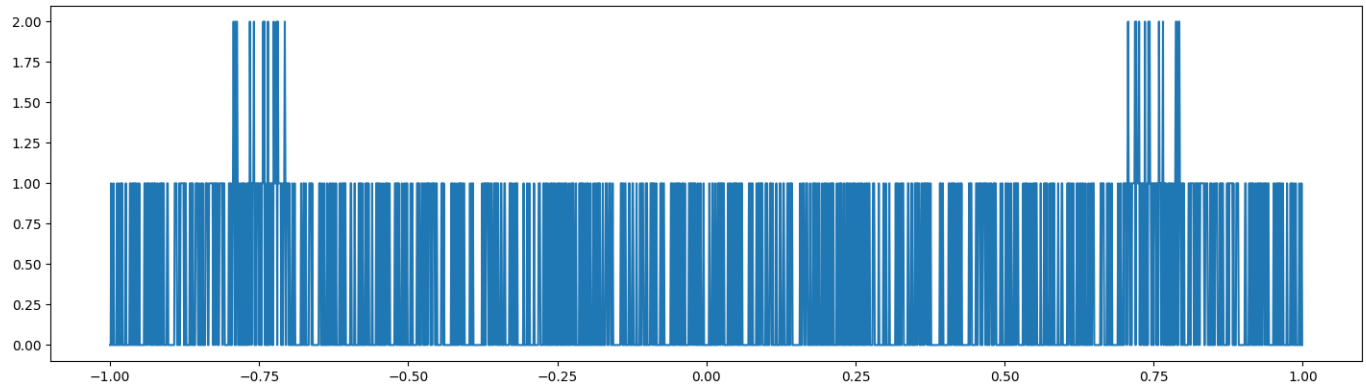

Figure 4: Courbe représentant la différence en ulp comme indiqué dans 1.1.

On remarque que la plupart des valeurs données par cet algorithme sont identiques à 1 ulp prés de celle de Java.

5 Fonction cos.

Afin de calculer une approximation de la fonction cosinus, nous avons commencé la l'algorithme Cordic, sauf que ce dernier donne des approximation a des centaines d'ulp prés.nous avons donc décider d'adopter une autre démarche basé sur le développement en série entière de la fonction cosinus.

$$\cos x = \sum_{n=0}^{+\infty} (-1)^n \frac{x^{2n}}{(2\,n)!} \tag{3}$$

cette fonction donne des valeurs à 2 ulps prés au voisinage de 0, et 200 ulps au voisinage de $\frac{\pi}{2}$, afin d'optimiser cette approximation ,nous avons utilisé des expression mathmatiqe afin de se ramener a des valeurs proches de 0:
sur l'intervalle [1.652,4.5] : $cos(x) = sin(\frac{\pi}{2} - x)$.
sur l'intervalle [4.5,2π] : $cos(x) = -cos(\pi - x)$.

5.1 Optimisation de cos.

Les factorielles et les puissance sont particulièrement lourdes en multiplications, Nous avons cherché à éliminé leur calcul, on remarque dans la série entière du cosinus que la factoriel dans le dénominateur augmente de 2 en 2,on peut donc utiliser la technique de mémorisation afin de conserver la valeur du dénominateur et la multiplier a chaque itération p par (2p)(2p+1), pour le numérateur , la puissance augmente de 2 en 2, et le signe change à chaque itération , il suffit de conserver le numérateur et le multiplier par x^2

5.2 Validation

	intervalle	nombre d'erreur	erreur moyenne en ulp	pas	nombres de tests
cos	$[0; 0.785]$	7820	2 ulp	2^{-22}	3.294.199
cos	$[0.785; 1.047]$	8730	2 ulp	2^{-22}	1.098.066
cos	$[1.047; 2.3561]$	210135	15 ulp	2^{-22}	5.490.331
cos	$[1.047; 3.141]$	210135	2 ulp	2^{-22}	4.290.191
cos	$[3.141; 3.926]$	1031825	2.31 ulp	2^{-22}	6.588.398
cos	$[3.926; 4.5]$	457529	5.35 ulp	2^{-18}	600844
cos	$[4.5; 6.28]$	276015	28.43 ulp	2^{-18}	466617

la fonction cos donne des résultats identique à 2 ulps que celle de Java dans 338 degré du cercle trigonométrique, elle prends des valeur imprécis a une centaine d'ulp prés pour des valeur proches de $\frac{\pi}{2}$ et $\frac{3\pi}{2}$.
la fonction cos utilise sin dans l'intervalle [1.652,4.5] pour passer a des valeurs proches de 0.
la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme cos qu'on a implémenté et celle de Java dans l'intervalle [0,1.652].

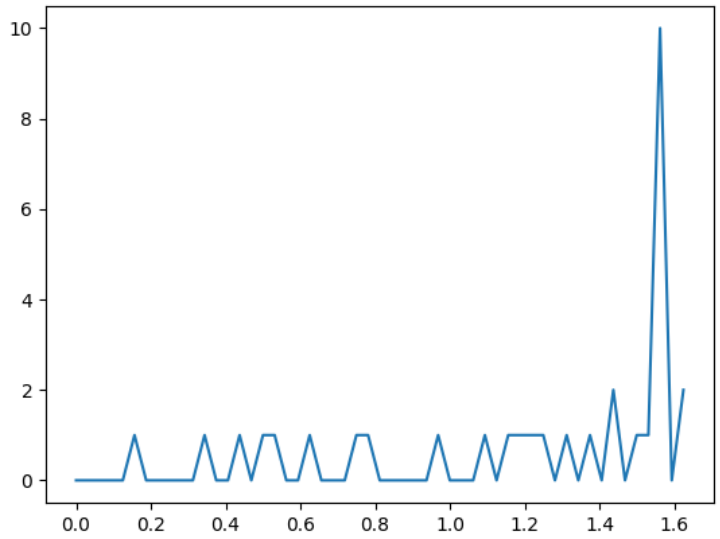

Figure 5: Courbe représentant la différence en ulp comme indiqué dans 1.1 de la fonction cos.

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme cos qu'on a implémenté et celle de Java dans l'intervalle [1.652,4.5].

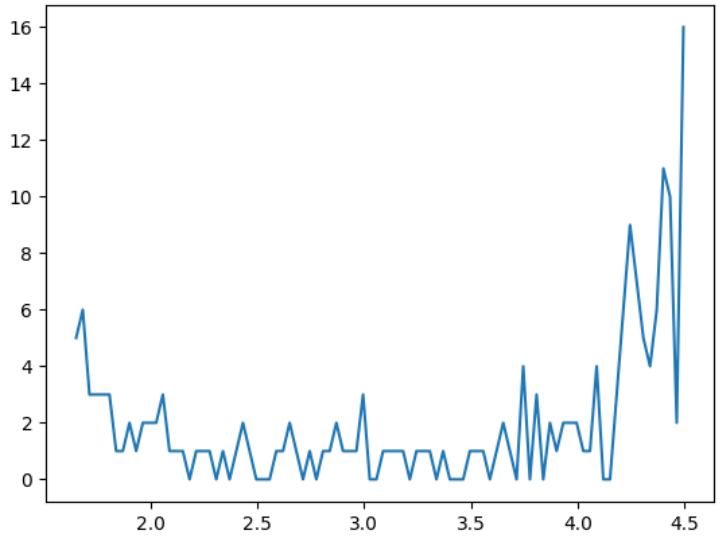

Figure 6: Courbe représentant la différence en ulp comme indiqué dans 1.1 de la fonction cos.

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme cos qu'on a implémenté et celle de Java dans l'intervalle [4.5,6.28].

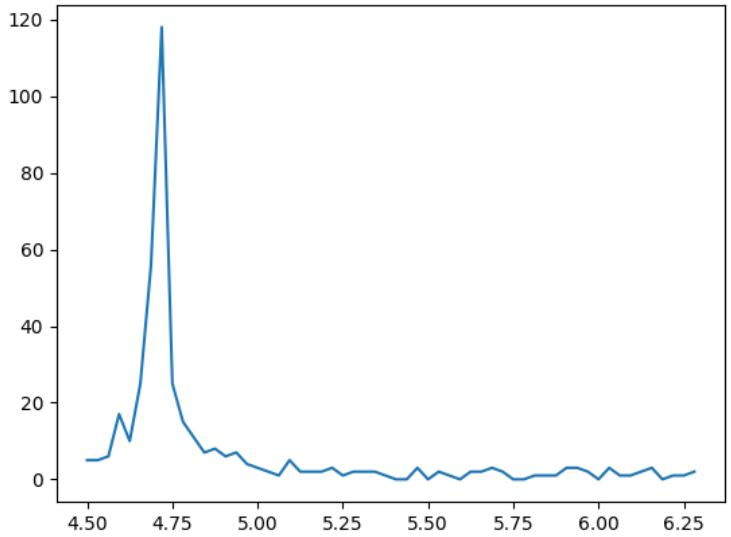

Figure 7: Courbe représentant la différence en ulp comme indiqué dans 1.1 de la fonction cos.

6 Fonction sin.

Afin de calculer une approximation de la fonction sinus, nous avons commencé la l'algorithme Cordic, sauf que ce dernier donne des approximation a des centaines d'ulp prés.nous avons donc décider d'adopter une autre démarche basé sur le développement en série entière de la fonction cosinus.

$$\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!} \tag{4}$$

cette fonction donne des valeurs à 2 ulps prés au voisinage de 0, et 50 ulps au voisinage de $\frac{\pi}{2}$, afin d'optimiser cette approximation ,nous avons utilisé des expression mathématiques afin de se ramener a des valeurs proches de 0:
sur l'intervalle [0.785,1.57] : $sin(x) = cos(\frac{\pi}{2} - x)$.
sur l'intervalle [1.57,2.35] : $sin(x) = -cos(\frac{\pi}{2} + x)$.
sur l'intervalle [2.35,π] : $sin(x) = sin(\pi - x)$.
sur l'intervalle [π,3.926] : $sin(x) = cos(-\frac{\pi}{2} + x)$.
sur l'intervalle [3.926,5.297] : $sin(x) = -sin(-\pi + x)$.
sur l'intervalle [5.297,6.28] : $sin(x) = -cos(\frac{\pi}{2} + x)$.

6.1 Validation

	intervalle	nombre d'erreur	erreur moyenne en ulp	pas	nombres de tests
sin	$[0; 0.785]$	424	2 ulp	2^{-18}	205.888
sin	$[0.785; 1.57]$	2904	2 ulp	2^{-18}	205.888
sin	$[1.57; 2.35]$	34729	2.16 ulp	2^{-18}	205.888
sin	$[2.35; 3.141]$	154329	26 ulp	2^{-22}	205.888
sin	$[3.141; 3.926]$	115609	46.8 ulp	2^{-22}	204.055
sin	$[3.926; 5.49]$	70438	2.17 ulp	2^{-18}	409994
ssin	$[5.49; 6.28]$	186321	40.4 ulp	2^{-18}	207094

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme sin qu'on a implémenté et celle de Java dans l'intervalle [0,0.7853].

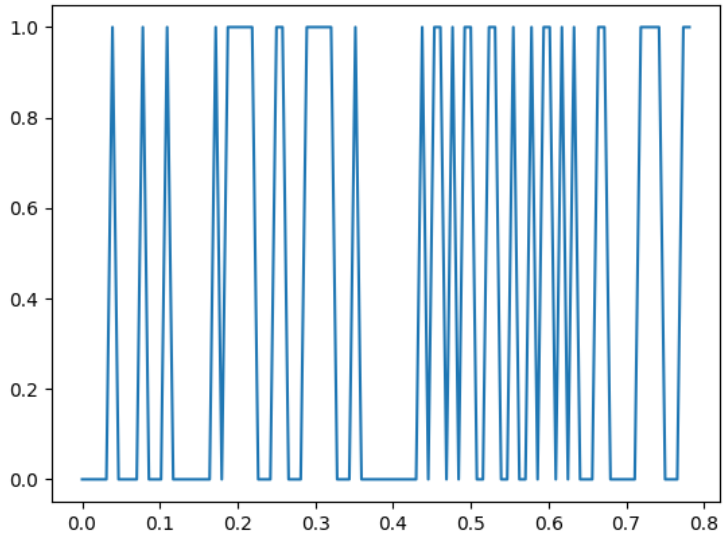

Figure 8: Courbe représentant la différence en ulp comme indiqué dans 1.1.

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme sin qu'on a implémenté et celle de Java dans l'intervalle [0.7853,1.57].

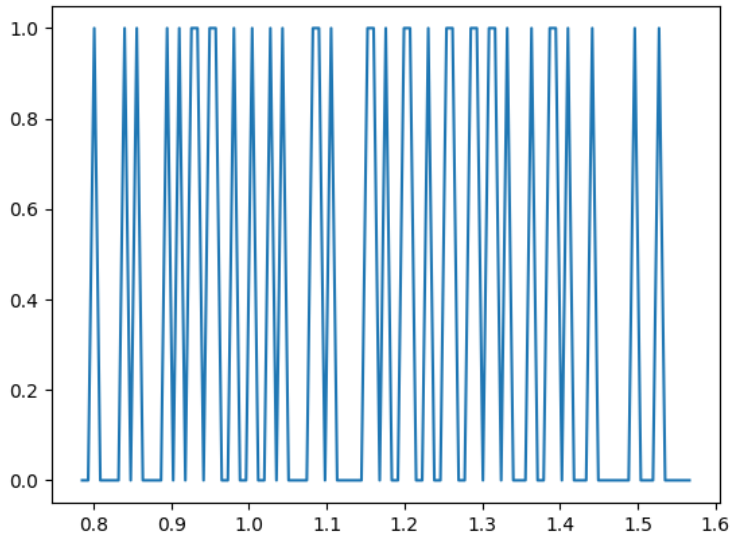

Figure 9: Courbe représentant la différence en ulp comme indiqué dans 1.1.

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme sin qu'on a implémenté et celle de Java dans l'intervalle [1.57,2.35].

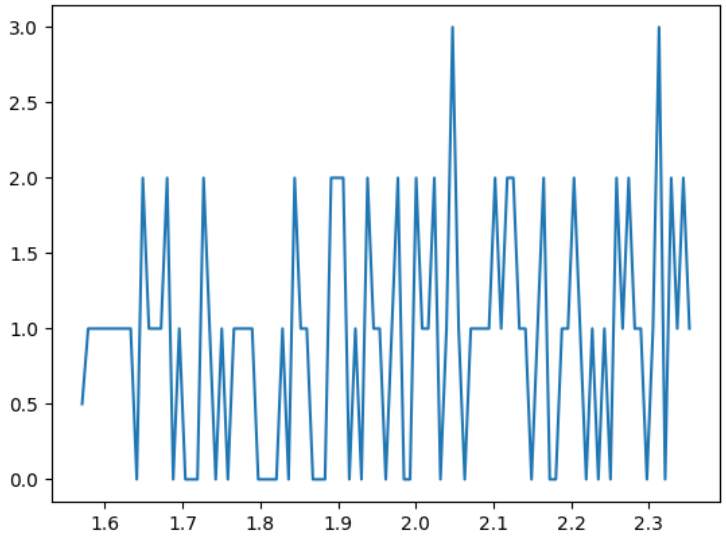

Figure 10: Courbe représentant la différence en ulp comme indiqué dans 1.1.

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme sin qu'on a implémenté et celle de Java dans l'intervalle [2.35,3.14].

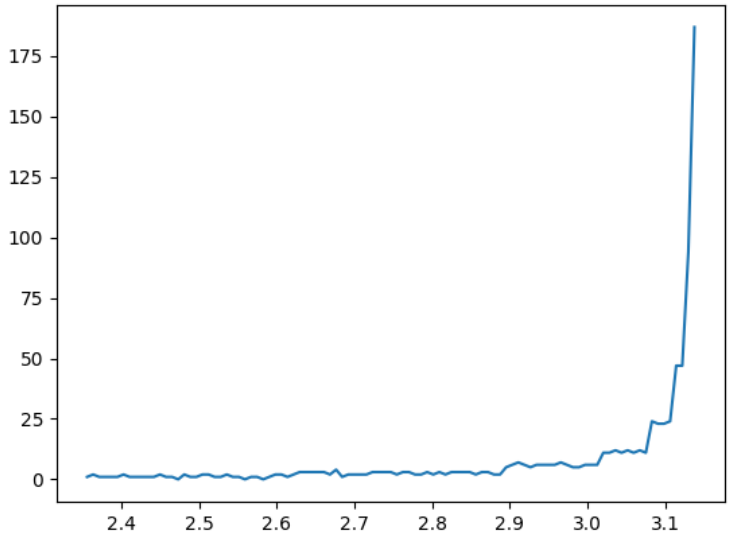

Figure 11: Courbe représentant la différence en ulp comme indiqué dans 1.1.

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme sin qu'on a implémenté et celle de Java dans l'intervalle [3.14,3.92].

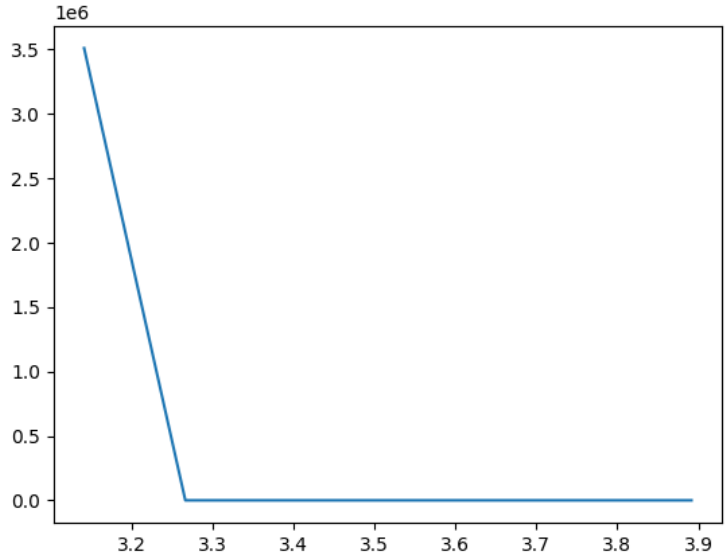

Figure 12: Courbe représentant la différence en ulp comme indiqué dans 1.1.

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme sin qu'on a implémenté et celle de Java dans l'intervalle [3.92,5.49].

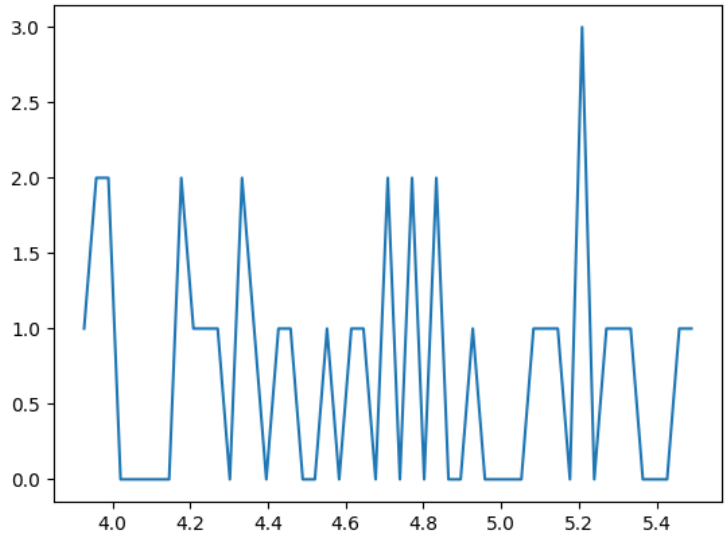

Figure 13: Courbe représentant la différence en ulp comme indiqué dans 1.1.

la courbe suivante représente le différence en ulp entre les valeur donné par l'algorithme sin qu'on a implémenté et celle de Java dans l'intervalle [5.49,6.28].

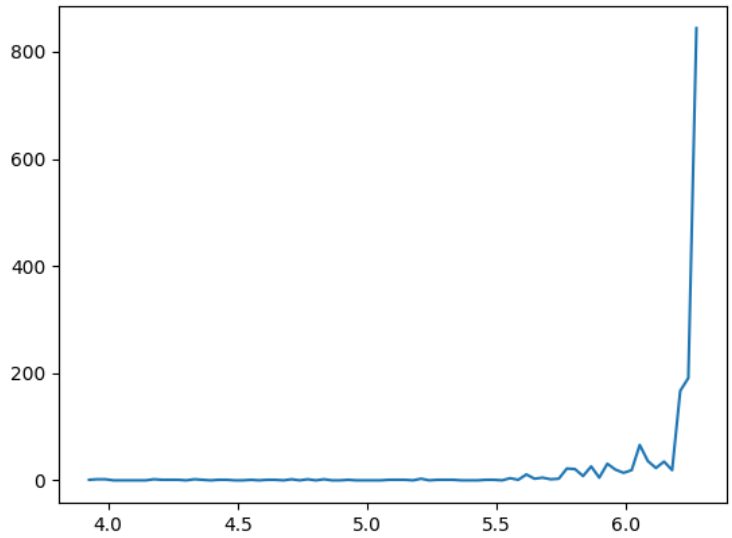

Figure 14: Courbe représentant la différence en ulp comme indiqué dans 1.1.

7 Fonctions intermédiaires

L'implementation des fonctions calculant la valeur absolue, la puissance, le racine carrée... est nécessaire . certaines fonctions sont facile à programmer(min(float f), max(float f), abs(float f).., On pourrait considérer qu'elles viendraient enrichir la bibliothèque maths. Nous avons essayer de garder les noms usuels (abs, pow, sqrt...) en ajoutant un "_" comme le précisait l'énoncé.
La fonction racine est implémentée avec l'algorithme d'Héron avec une douzaine d'itération. Cet algorithme converge vite et précisément, son impact de coût sera donc faible et pèsera peu dans la précision finale.
Le principe de cet algorithme est le suivant:
Pour déterminer la racine carrée du nombre (positif) a, on essaye de faire une approximation de la suite:

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2} \tag{5}$$

de premier terme $x_0 > 0$ choisi si possible « assez proche » de la racine de a, dans notre algorithme on a choisi $x_0 = a$, ce qui implique que plus a est grand plus x_0 est loin de la racine carée de a, et donc plus la précision de l'approximation diminue.

7.1 Validation

Voici la courbe donnée grâce a cet algorithme:

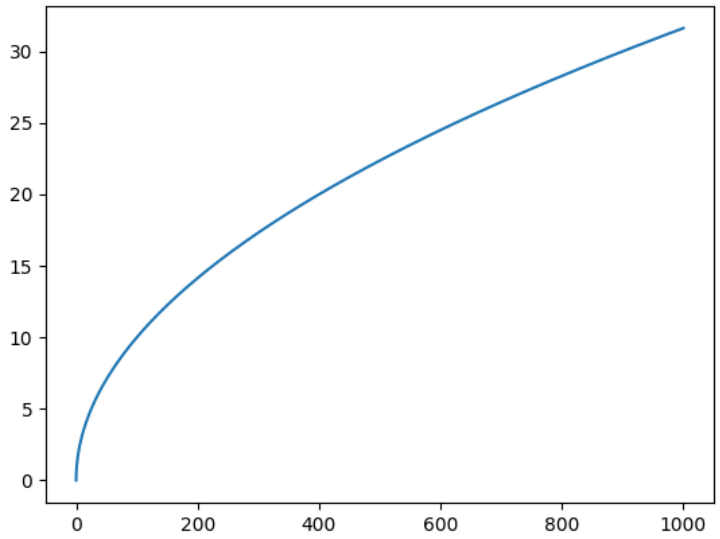

Figure 15: Courbe représentant la différence en ulp comme indiqué dans 1.1.

la fonction $\_fmod$ est utilisé afin de ramener les argument des fonction cosinus et sinus a l'intervalle $[0, 2\pi]$.

8 Validation

Une fois que l'algorithme semble satisfaisant on peut faire tester à notre algorithme des milliers de valeurs pour trouver les valeurs d'entrée qui donnent les sorties les moins précises.c'est le cas des deux fonction cos et sin,ou on a essayer de tester toutes les formule mathématiques afin de trouver les meilleur formules qui donne une différence en ulp les plus faibles.
la validation est relativement facile pour la fonction arcsin puisque son intervalle de définition est borné [-1,1]

8.1 Formules à valider

Nous avons bien vérifie que nos résultats vérifient bien les formules suivantes:

- $cos^2(x) + sin^2(x) = 1$
- $arctan(x) + arctan(\frac{1}{x}) - \frac{\pi}{2} = 0$
- Pour $x \in [-\frac{\pi}{2}, \frac{\pi}{2}], asin(sin(x)) = x$

9 Fonctionnement de l'extension

l'ajout de #include "Math.decah" au début du fichier permet d'utiliser les fonctions indiqués si dessous.
La fonction arcsin doit prendre en argument un flottants compris entre -1 et 1,il faut aussi faire attention aux types des paramètres(float dans la majorité des cas)
les fonction _cos et _sin donnent des résultats identiques à une dizaine d'ulp prés pour des valeur proches de pi/2
pour des flottants très grands, la fonction _fmod donne des résultats imprécis, ce qui peut influencer le résultat des fonctions _cos et _sin,
Ces méthodes sont des approximations de valeurs mathématiques, dont on mesure la précision grâce aux ULP (Unit in the Last Place).
Pour une analyse complète et détaillée des algorithmes et de leurs précisions respectives, il faut se reporter à la documentation de la classe Math.

Calcul du temps des tests:

	intervalle	nombre d'erreur	erreur moyenne en ulp	pas	nombres de tests	temps de calcul
cos	$[0; 0.785]$	7820	2 ulp	2^{-22}	3.294.199	0m4,062s
cos	$[0.785; 1.047]$	8730	2 ulp	2^{-22}	1.098.066	0m1,572s
cos	$[1.047; 2.3561]$	210135	15 ulp	2^{-22}	5.490.331	0m8,434s
cos	$[2.3561; 3.141]$	210135	2 ulp	2^{-22}	4.290.191	0m5,870s
cos	$[3.141; 3.926]$	1031825	2.31 ulp	2^{-22}	6.588.398	0m6,099s
cos	$[3.926; 4.5]$	457529	5.35 ulp	2^{-18}	600844	0m0,441s
cos	$[4.5; 6.28]$	276015	28.43 ulp	2^{-18}	466617	0m0,762s
ulp	$[0; 1]$	0	0 ulp	2^{-15}	32.768.000	0m0,173s
ulp	$[0; 1.000]$	0	0 ulp	2^{-18}	128000	0m0,196s
ulp	$[1.000; 1.000.000]$	0	0 ulp	2^{-15}	7992000	0m1,294s
atan	$[0; 1.000]$	0	2 ulp	2^{-15}	1.024.000	0m2,365s
atan	$[1.000; 100.00]$	0	1 ulp	2^{-18}	3.168.000	0m0,974s
asin	$[-1; 1]$	8042	2 ulp	2^{-22}	8.388.608	0m22,782s
asin	$[0.72; 1]$	4939	2 ulp	2^{-22}	73401	0m0,283s
asin	$[-1; -0.72]$	4944	2 ulp	2^{-22}	73401	0m0,344s
asin	$[-0.72; 0.72]$	996	2 ulp	2^{-22}	377488	0m0,163s
sin	$[0; 0.785]$	424	2 ulp	2^{-18}	205.888	0m0,345s
sin	$[0.785; 1.57]$	2904	2 ulp	2^{-18}	205.888	0m0,314s
sin	$[1.57; 2.35]$	34729	2.16 ulp	2^{-18}	205.888	0m0,374s
sin	$[2.35; 3.141]$	154329	26 ulp	2^{-22}	205.888	0m0,343s
sin	$[3.141; 3.926]$	115609	46.8 ulp	2^{-22}	204.055	0m0,353s
sin	$[3.926; 5.49]$	70438	2.17 ulp	2^{-18}	409994	0m0,543s
ssin	$[5.49; 6.28]$	186321	40.4 ulp	2^{-18}	207094	0m0,349s

10 Analyse bibliographique

- A SEQUENCE OF HERMITE INTERPOLATING-LIKEPOLYNOMIALS FOR APPROXIMATING ARCTANGENTHERBERT A. MEDINA

- https://fr.wikipedia.org/wiki/M
- novel implementation of CORDIC algorithm using Backward Angle Recoding-Yu Hen Hu,Homer H. M. Chern