# Extension : MATH

GL28

Janvier 2020

## Table des matières

# 1 Introduction

## 1.1 Rappel des objectifs

On cherche à implémenter une bibliothèque *MATH* pour calculer quelques fonctions mathématiques.

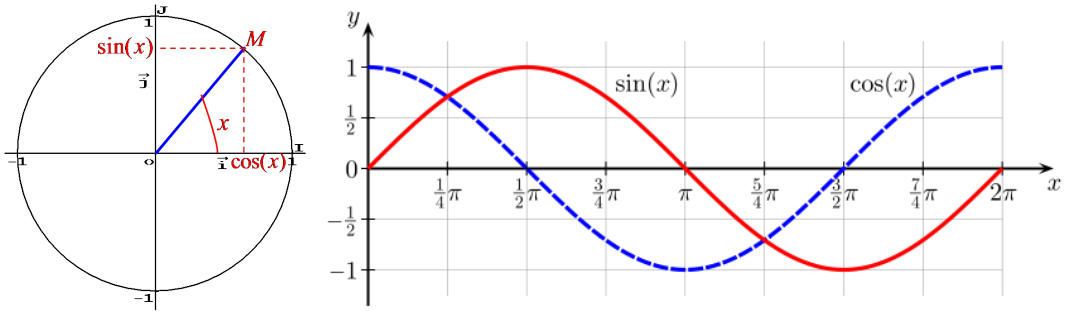
```
float sin(float f);
float cos(float f);
float asin(float f);
float atan(float f);
float ulp(float f);
```

## 1.2 Un point sur les fonctions

### 1.2.1 Sinus et Cosinus

$$\begin{array}{ccccccccc} \sin & : & \mathbf{R} & \to & [-1,1] & \cos & : & \mathbf{R} & \to & [-1,1] \\ & & x & \mapsto & \sin(x) & & & x & \mapsto & \cos(x) \end{array}$$

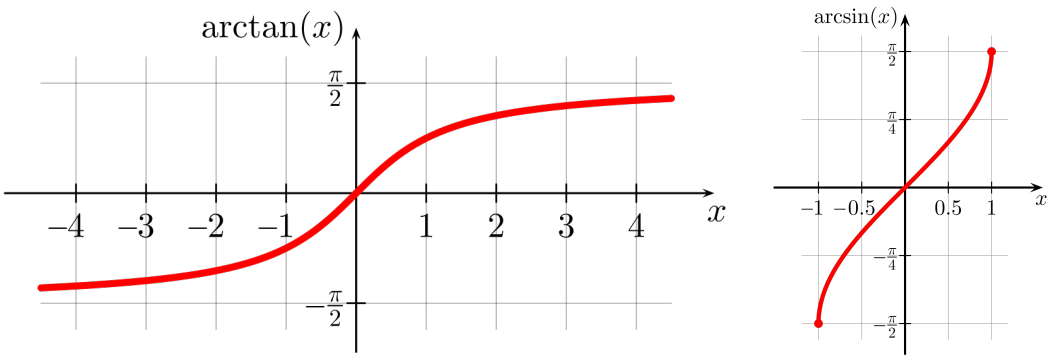
Si sur un cercle de centre $(0,0)$ et de rayon 1 dans un repère orthonormé, alors un point M formant avec l'axe des abscisses un angle $x$ alors l'abscisse de M est $\cos x$ et son ordonnée est $\sin x$.




### 1.2.2 Arctan et Arcsin

$$\begin{array}{ccccccccc} \arctan & : & \mathbf{R} & \to & ]-\frac{\pi}{2},\frac{\pi}{2}[ & \arcsin & : & [-1,1] & \to & [-\frac{\pi}{2},\frac{\pi}{2}] \\ & & x & \mapsto & \arctan(x) & & & x & \mapsto & \arcsin(x) \end{array}$$

arcsin est l'application réciproque de sin sur l'intervalle $[-\frac{\pi}{2},\frac{\pi}{2}]$ et arctan est l'application réciproque de tan sur l'intervalle $]-\frac{\pi}{2},\frac{\pi}{2}[$.




### 1.2.3 Unit in the last place (ULP)

L'ULP n'est pas une fonction mathématique à proprement parler, mais plutôt une fonction "informatique". Soit $x$ un nombre réel, alors il se trouve entre deux flottants tels que $a \leq x \leq b$. Ainsi

$$ulp(x) = b - a$$

Cenpendant, si $x$ est un flottant, alors, en nottant, $x^+$ le flottant directement au dessus de $x$ on a :

$$ulp(x) = x^+ - x$$

Cette fonction nous servira également pour mesurer la précision. Par exemple on pourra donner le nombre de flottant qui sépare notre implémentation de sinus avec une autre au lieu de donner un nombre réel qui ne veut pas dire grand chose.

## 1.3 Précision et rapidité

L'idéal est de trouver un bon compromis entre précision et rapidité. En effet avant d'implémenter une fonction de la sorte, il nous faut une erreur maximale. C'est-à-dire une erreur limite qu'on s'engage à ne jamais dépasser.

Pour illustrer prenons par exemple la fonction suivante :

```
float sin(float f) {
    return 0;
}
```

On peut garrantir que cette fonction est la plus rapide pour calculer le sinus d'un nombre. Malheureusement son erreur est de 1. Cette exemple, quelque peu extrême permet de montrer qu'il existe un lien entre rapidité et précision. De la même manière que pour *le principe d'Heisenberg* très célèbre en mécanique quantique on a une relation du type

$$rapidite \times precision \leq cste$$

C'est à dire que si nous voulons une fonction sinus qui donne un résultat plus rapidement alors nous devons sacrifier de la précision. Inversement si nous voulons plus de précision alors, nous devrons faire plus de calculs et celà prendra ainsi plus de temps. En somme, on cherche à trouver *rapidite* et *precision* de sorte à avoir

$$rapidite \times precision = cste$$

## 1.4 La norme IEEE 754

En informatique, l'IEEE 754 est une norme sur l'arithmétique à virgule flottante mise au point par le Institute of Electrical and Electronics Engineers. Elle est la norme la plus employée actuellement pour le calcul des nombres à virgule flottante avec les CPU et les FPU. [1]


$$v = (-1)^s \times 2^{e-127} \times (1 + m \times 2^{-23})$$

avec $e$ l'exposant et $m$ la mantisse.

## 1.5 Un point sur les calculs de coûts

Dans ce qui suit, nous allons calculer le coût théorique de nos algorithmes. Cependant nous prendrons en compte que les opérations de calculs (ADD, SUB, MUL, DIV). En effet toutes les autres opérations peuvent différer d'un compilateur à l'autre et nous nous en préocupperons donc pas. Nous ne prendrons pas en compte l'instruction FMA puisque non implémentée dans notre compilateur. Pour pouvoir ce rendre compte de la réalité, nous tracerons le nombre de cycles internes sur l'execution de certains programmes lors de l'analyse de nos résultats en *Deca*.

# 2 Prototypage en Java

Pour éviter de se rendre compte à la dernière minute qu'un algorithme est meilleur qu'un autre, nous avons décidé de tester nos algorithmes de calcul à l'aide du langage *Java*. Cependant, il est bien sûr évident que le compilateur *Java* sera bien plus rapide et performant que notre compilateur *Deca*. Par exemple, il semblerait que pour faire des opérations entre flottants, *Java* convertisse les flottants 32 bits en flottants 64 bits pour être certain de ne faire aucune erreur d'arrondi.

Notre pari est donc que si un algorithme est meilleur qu'un autre en *Java* alors il sera également meilleur en *Deca*.

## 2.1 Sinus et Cosinus

### 2.1.1 Développement de Taylor

**Méthode** En utilisant les développements de Taylor on a pour $x$ assez petit, on a :

$$\sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}$$

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40320}$$

Pour évaluer ce polynôme, on utilise la méthode de Horner, c'est à dire qu'on factorise partiellement le polynôme pour chaque terme.

Graphiquement, on remarque qu'on est à moins de 1 ULP de différence avec les fonctions Java pour $n = 7$ pour sin et $n = 8$ pour cos sur l'intervalle $[0, \frac{\pi}{4}]$.

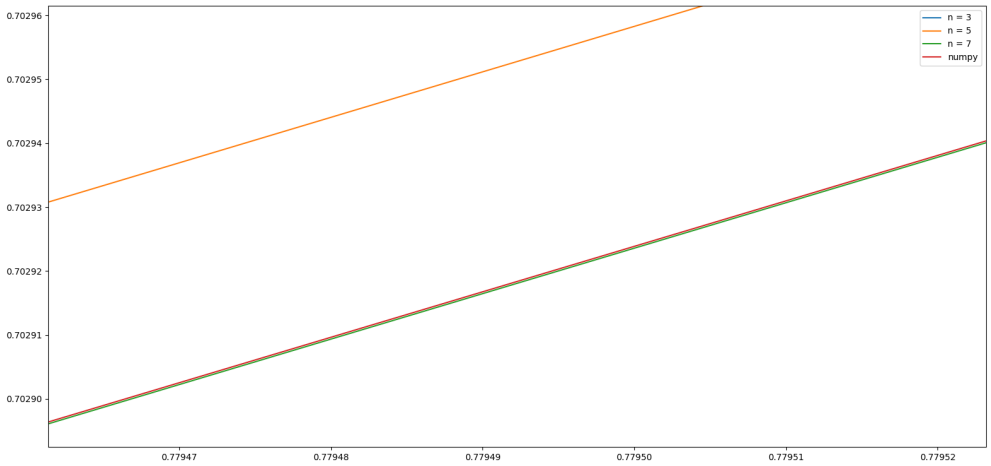



FIGURE 1 – Comparaison des sinus pour différents ordres au voisinage de $\frac{\pi}{4}$

Maintenant qu'on a notre sinus et cosinus opérationnels entre 0 et $\frac{\pi}{4}$, on a toute l'information suffisante pour calculer n'importe quel sinus ou cosinus. En effet on peut utiliser les formules suivants :

$\forall x \in R$

1. $\sin(-x) = -\sin(x)$ et $\cos(-x) = \cos(x)$
2. $\sin(x + 2\pi) = \sin(x)$ et $\cos(x + 2\pi) = \cos(x)$
3. $\sin(x + \pi) = -\sin(x)$ et $\cos(x + \pi) = -\cos(x)$
4. $\sin(x + \frac{\pi}{2}) = \cos(x)$ et $\cos(x + \frac{\pi}{2}) = -\sin(x)$
5. $\sin(\frac{\pi}{2} - x) = \cos(x)$ et $\cos(\frac{\pi}{2} - x) = \sin(x)$

Maintenant, soit $x \in R$,

— avec (1) on peut ramener $x$ dans $R^+$
— avec (2) on peut ramener $x$ dans $[0, 2\pi]$
— avec (3) on peut ramener $x$ dans $[0, \pi]$
— avec (4) on peut ramener $x$ dans $[0, \frac{\pi}{2}]$
— avec (5) on peut ramener $x$ dans $[0, \frac{\pi}{4}]$

En réalité on va seulement écrire une fonction sinus qui fait toutes ces transformations et pour cosinus on se contentera d'utiliser la formule (4), i.e. appeler sinus avec un autre argument.

**Résultats** En traçant l'erreur en ULP i.e. $\left|\frac{notreFonction - fonctionJava}{ulp}\right|$, on obtient le graphe suivant :

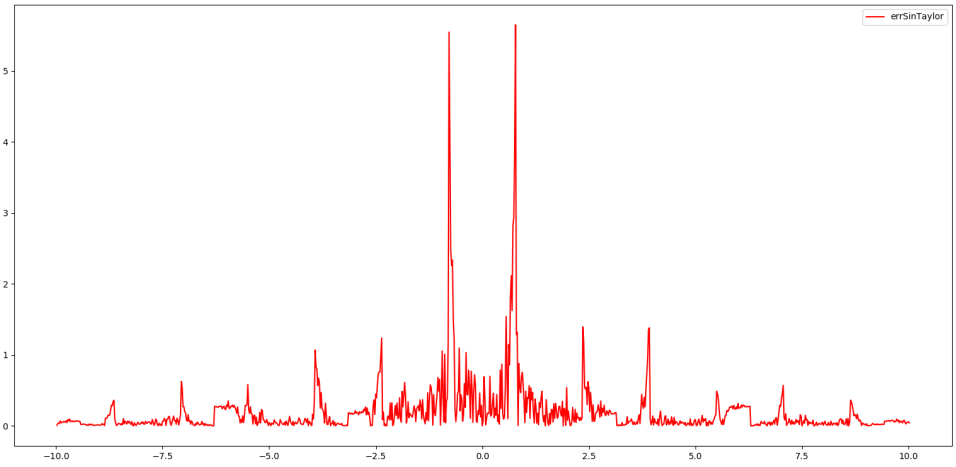



FIGURE 2 – Tracé de l'erreur de Sinus en ULP

On constate 2 pics d'ULP au voisinage de $-\frac{\pi}{4}$ et $\frac{\pi}{4}$ ce qui est cohérent étant donné que c'est à cet endroit que le développement limité est le moins précis. Hormis ce détail, on reste toujours en dessous de 1 ULP ce qui est très satisfaisant.

**Coût** Procédons étape par étape pour calculer le coût de cet algorithme on va suivre la liste précédente :

1. on utilise la parité des fonctions : 1 multiplication pour prendre l'opposé
2. on doit appliquer un "modulo flottant" : si on écrit $f = kp + x$ alors on va faire environ $\left|\frac{f}{p}\right|$ soustractions, on voit que c'est cette étape qui nous coûte beaucoup. En l'occurence dans notre cas $p = 2\pi$
3. 2 soustractions (1 pour le test et l'autre pour la modification si necessaire), 1 multiplication pour changer le signe si nécessaire
4. 2 soustractions (même chose qu'avant)
5. 2 soustractions (même chose qu'avant)
6. enfin il reste à calculer le polynôme, en applicant la méthode de Horner, on est donc à 8 multiplications et 4 additions

On a donc au total $\left|\frac{f}{p}\right| + 10$ multiplications et 10 additions / soustractions. D'après la documentation on a donc :

$$N_{cycle} = 20\left|\frac{f}{p}\right| + 20 \times 10 + 2 \times 10 = 20\left|\frac{f}{p}\right| + 220$$

### 2.1.2 Algorithme de Cordic

**Méthode** L'algorithme de Cordic pour calculer $\tan\theta \in [0, \frac{\pi}{2}]$ est le suivant :
on commence avec :

$$\begin{cases} x_0 = x_0 \\ y_0 = 0 \end{cases}$$

Il suffit ensuite de trouver une suite décroissante $(\theta_i)_{0 \leq i \leq n}$ tel que $\sum_{i=0}^{n} \theta_i = \theta$. Ainsi on a les relations suivantes :

$$\begin{cases} x_k = x_0 \cos(\theta_0 + \theta_1 + \ldots + \theta_{k-1}) \\ y_k = x_0 \sin(\theta_0 + \theta_1 + \ldots + \theta_{k-1}) \end{cases}$$

Ainsi on a en appliquant la relation si dessus en $n$ :

$$\frac{y_k}{x_k} = \tan\theta$$

Pour plus de détail confère la référence [3].

**Résultats**

Nous obtenons l'erreur suivante pour le sinus calculé avec Cordic :

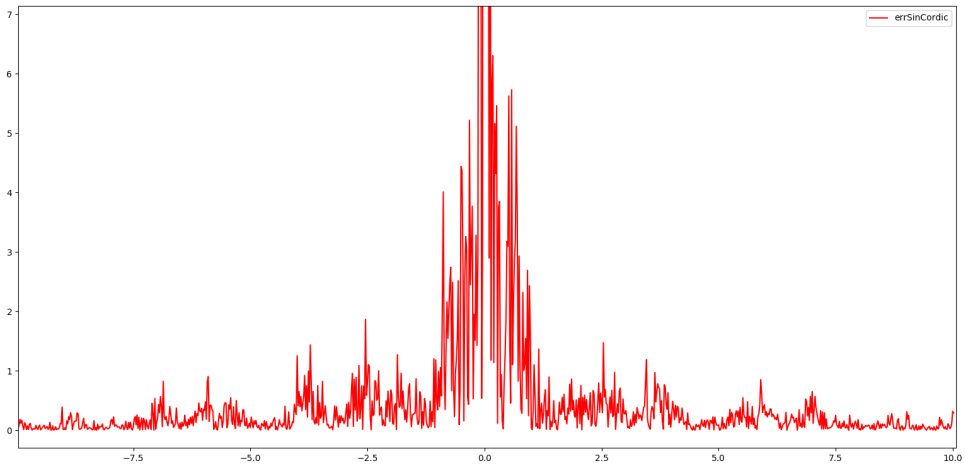



FIGURE 3 – Tracé de l'erreur de Sinus en ULP

On constate une erreur légérement supérieure au calcul en utilisant le développement de Taylor.

### 2.1.3 Polynômes de Tchebychev

**Méthode** Soit $T_n$ le $n$-ième polynôme de Tchebychev, alors on a :

$$\cos(x) = J_0(1) + 2\sum_{n \geq 1} (-1)^n J_{2n}(1) T_{2n}(x)$$

avec

$$J_{2n}(1) = \sum_{l \geq 0} \frac{(-1)^l}{4^{l+n}(2n+l)!}$$

On peut ensuite négliger tous les termes tels que $l \geq 1$ étant donné la vitesse de décroissance du terme général de cette série et on obtient ainsi :

$$J_{2n}(1) \simeq \frac{1}{4^n(2n)!}$$

Ainsi il reste plus qu'à tronquer la première somme de sorte à avoir une précision convenable.

**Résultats** On obtient la courbe suivante :

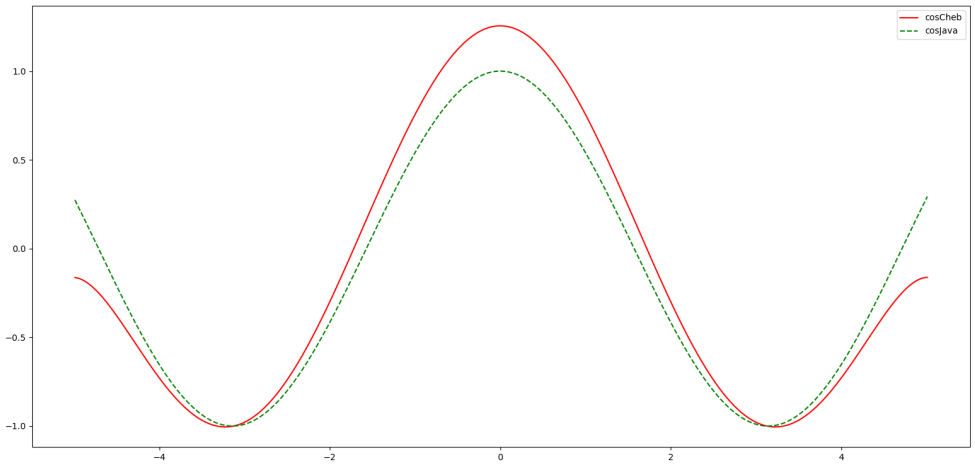



FIGURE 4 – Tracé superposé du sinus de Java et de celui de Tchebychev

On voit sans forcer que cette méthode est très mauvaise. Nous n'avons pas réussi à trouver où était l'erreur. Vient-elle de l'implémentation ou de la formule initiale ? Après quelques temps de recherche inutiles nous avons laissé tombé cette méthode. Nous ne perdrons donc pas de temps à calculer son coût.

## 2.2 Arctan et Arcsin

Les formules suivantes vont nous être utiles pour cette section :

$$\arcsin(-x) = -\arcsin x \tag{1}$$

$$\arctan(-x) = -\arctan x \tag{2}$$

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right) \tag{3}$$

$$\arctan(x) = \arcsin(\frac{x}{\sqrt{1+x^2}}) \tag{4}$$

$$\arctan(x) + \arctan\frac{1}{x} = \pm\frac{\pi}{2} \tag{5}$$

### 2.2.1 Développement de Taylor

**Méthode** De même que pour sinus et cosinus, pour $x$ suffisamment petit on a :

$$\arctan x \simeq x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9}$$

$$\arcsin x \simeq x + \frac{x^3}{3} + 3\frac{x^5}{40} + 5\frac{x^7}{112} + 35\frac{x^9}{1152}$$

En admettant que ces développements soient valables pour $x \in [0, 1]$, alors on peut utiliser alors avec (1), (2) et (5), on peut recouvrir l'ensemble de définition des deux fonctions.

**Résultats** Voici ce qu'on obtient :

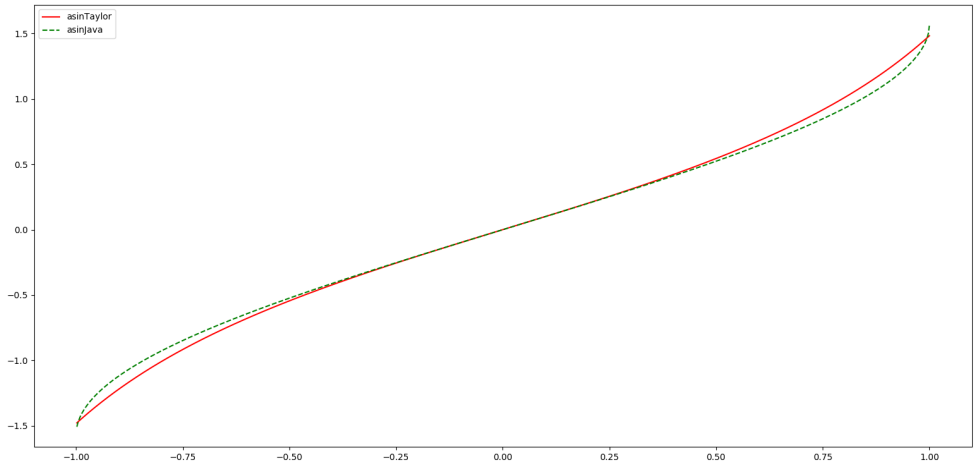



FIGURE 5 – Tracé de Arcsin sur $[-1, 1]$

On remarque donc que l'approximation pour Arcsin est très mauvaise, il n'y a pas besoin de tracer l'erreur pour se rendre compte de celà. On pourrait se dire qu'augmenter l'ordre pourrait rendre le résultat meilleur, mais ce n'est pas réellement le cas et puis on devrait faire encore plus de calculs coûteux.

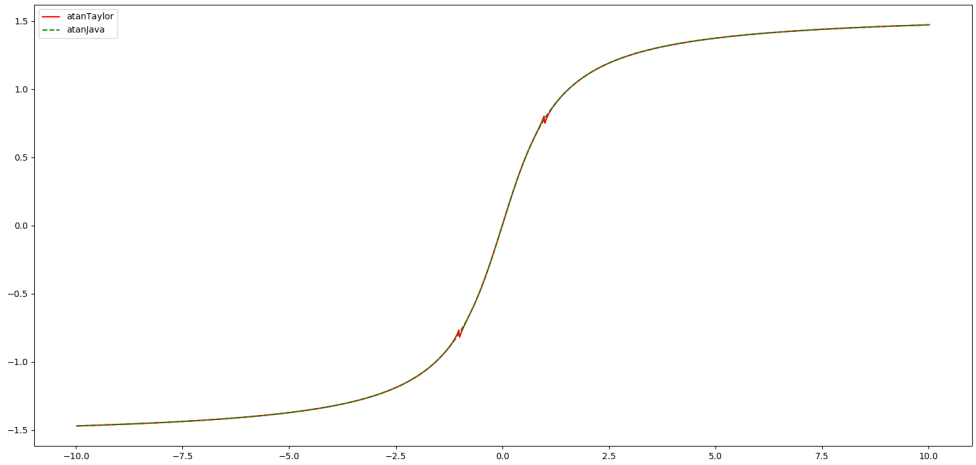



FIGURE 6 – Tracé de Arctan sur $[-10, 10]$

Pour Arctan le résultat est satisfaisant à l'oeil, sauf en 1 et $-1$ qui sont les points où le développement limité est le moins précis. En effet $\arctan(1) \simeq 0.79$ alors que notre méthode donne 0.83. On ne peut donc pas utiliser cet algorithme seul pour calculer Arctan.

**Coût** Pour calculer les développements limités, de la même manière que pour sinus, on trouve 9 multiplications et 4 additions avec la méthode de Horner pour les 2 fonctions. Enfin, on oublie pas la multiplication pour le changement de signe dans le cas où $x < 0$. Et puis, exclusivement pour Arctan, si $x > 1$ on a 1 division et 1 soustraction pour appliquer (5). On remarque donc que malgré ses imprécisions, l'algorithme a le mérite d'être très efficace. en dehors de $\{-1, 1\}$.

### 2.2.2 Calcul par dichotomie

**Méthode** Pour calculer Arctan, on a utilisé la méthode de Dichotomie sur l'intervalle $[0, 1]$ en résolvant l'équation $h(x) = \tan(x) - f = 0$ ce qui permet d'obtenir $x = \arctan(f)$ (On peut utiliser cette méthode puisque $h$ est continue sur l'intervalle $[0, 1]$ et $h(0)h(1) < 0$.

Pour cela on utilise notre fonction tan calculée avec l'algorithme de Cordic dans la partie précédente.

**Résultats**

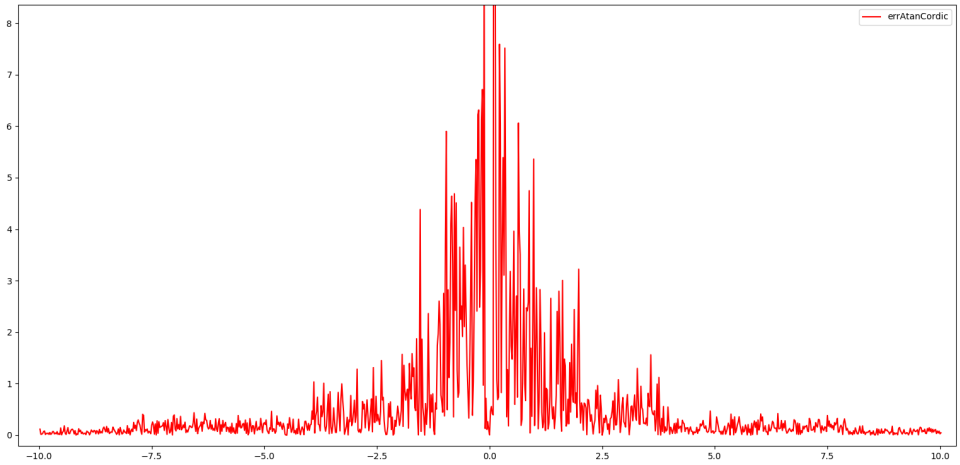



FIGURE 7 – Tracé de l'erreur d'Arctan sur $[-10, 10]$

On remarque que pour Arctan, l'erreur est bonne lorsque $|x| \to +\infty$, en effet on est à moins de 1 ULP. Cependant au voisinage de 0, la précision est très brouillon, on est entre 4 et 9 ULP.

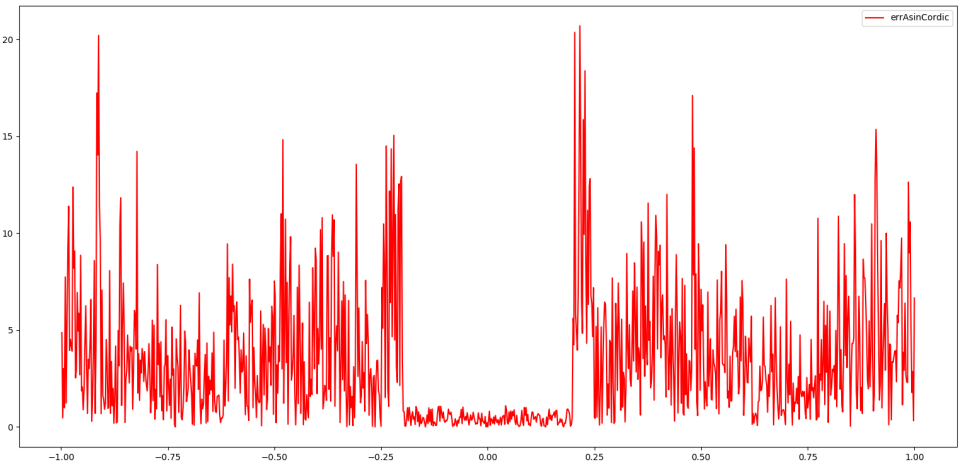



FIGURE 8 – Tracé de l'erreur d'Arcsin sur $[-1, 1]$

Pour Arcsin, l'erreur est acceptable au voisinage de 0 : on est en dessous de 1 ULP, cependant pour les autres valeurs on tourne autour de 10 ULP ce qui est très mauvais. Nous allons donc oublier cette implémentation dans la suite car nous verrons que la méthode de Newton a de bien meilleur résultats.

**Coût** Sans rentrer dans les détails, cette méthode est assez coûteuse puisqu'à chaque itération de la dichotomie on doit calculer une tangente.

### 2.2.3 Méthode de Newton

Soit $y \in R$ connu, nous recherchons alors un nombre $x$ vérifiant $\tan x = y$. Alors $x = \arctan y$. La méthode de Newton permet de résoudre $f(x) = 0$. Posons alors :

$$f(x) = \tan(x) - y$$

ainsi

$$f'(x) = \frac{1}{\cos^2(x)}$$

$f$ étant $C^2$, cette méthode nous assure que la suite :

$$\begin{cases} u_0 = 0 \\ u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)} \end{cases}$$

converge vers $\arctan y$. Et en remplaçant, on obtient :

$$u_{n+1} = u_n + \cos u_n (y \cos u_n - \sin u_n)$$

Puisque $|u_n| < \frac{\pi}{2}$ il n'y a pas de soucis à se faire pour le domaine de définition de $f'$. Après expérimentations, on a trouvé que $N = 10$ itérations étaient plus que suffisantes pour que $u_n$ converge.

Enfin pour trouver Arcsin on utilise la formule (3). Mais pour appliquer cette formule il nous faut une racine carrée, pour celà on utilise la suite suivante qui a la propriété de converger vers $\sqrt{a}$ :

$$\begin{cases} u_0 = 0.1 \\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right) \end{cases}$$

Des tests du même genre que pour déterminer l'ordre suffisant sur le développement de Taylor nous montre que $N = 15$ itérations sont suffisantes pour calculer cette racine carrée.

**Résultats** On obtient les résultats suivants :

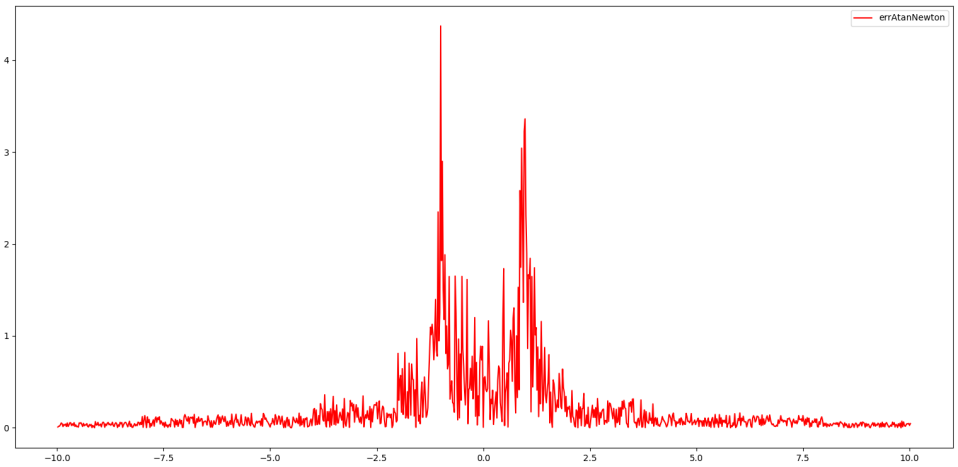



FIGURE 9 – Tracé de l'erreur d'Arctan en ULP

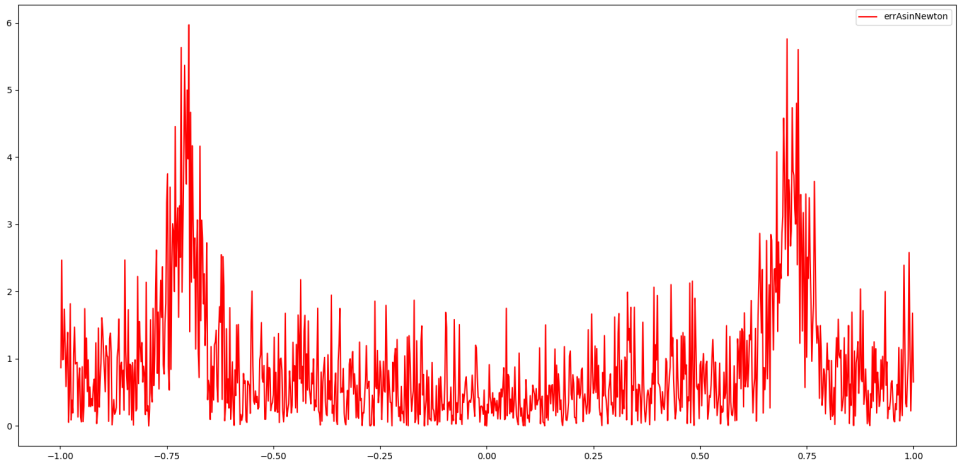



FIGURE 10 – Tracé de l'erreur d'Arcsin en ULP

On remarque qu'autour de 1 et $-1$ on a de nouveaux des pics d'ULP. Il s'agit du même phénomène que pour sinus et cosinus. Pour ces deux valeurs la méthode converge mais beaucoup plus lentement. Globalement on reste en dessous de 2 ULP, ce qui est très satisfaisant.

**Coût** Admettons que nous fassions donc $N = 10$ itérations de notre suite. Pour une itération on doit :

- calculer sinus et cosinus : on se trouve sur $[0, 1]$ donc $\frac{f}{p} = 0$ ainsi on a $220 \times 2 = 440$ cycles pour l'instant
- 2 multiplications et 2 additions / soustractions
- 1 division et 1 soustraction pour appliquer (5) si nécessaire
- 1 multiplication pour appliquer (2) si nécessaire.

On a donc pour une itération $440 + 3 \times 20 + 3 \times 2 + 1 \times 40 = 543$ dans le pire des cas pour une itération. Admettons qu'on en fasse donc 10, on a alors

$$N_{cycle} = 5430$$

C'est donc bien plus coûteux que notre méthode avec les développements de Taylor. Ainsi une bonne idée pourrait être de de se servir de la méthode de Newton pour calculer l'Arctan au voisinnage de 1 et $-1$ et d'utiliser la méthode des développements de Taylor en dehors de ces points.

Cepedant pour calculer la racine carrée qui nous permet d'utiliser la formule (3) on fait 2 additions, 1 multplication et 1 division par itération, et comme on fait 15 itérations alors on ajoute $62 \times 15 = 930$ cycles. Ainsi on oublie pas la division, la multiplication et l'addition de la formule (3) et pour Arcsin on obtient

$$N_{cycle} = 5430 + 930 + 40 + 20 + 2 = 6422$$

## 2.3 ULP

### 2.3.1 Logarithme

**Méthode** Soit $x$ un flottant codé selon la norme IEEE-754, alors $x = (-1)^s \times 2^{e-127} \times (1 + m \times 2^{-23})$. Montrons que $ulp(x) = x^+ - x = z_{mach} \times 2^{e-127}$ où $z_{mach} = 2^{-23}$.

Pour faire ce calcul nous devons distinguer le cas où $m = 1111...1111$ et les autres cas.

**Cas 1, m saturée de 1** Dans ce cas $m = 2^{23} - 1$ et donc :

$$x = (-1)^s \times 2^{e-127} \times (2 - 2^{-23})$$

de plus, $m^+ = 0$ et donc comme l'exposant est incrémenté :

$$x^+ = (-1)^s \times 2^{e-127+1}$$

Ainsi en faisant la différence on obtient :

$$ulp(x) = 2^{e-127+1} - 2^{e-127} \times (2 - 2^{-23}) = z_{mach} \times 2^{e-127}$$

**Cas 2, m quelconque** Dans ce cas, $m^+ = m + 1$ donc :

$$ulp(x) = (-1)^s \times 2^{e-127} \times (1 + (m+1) \times 2^{-23}) - (-1)^s \times 2^{e-127} \times (1 + m \times 2^{-23})$$

d'où :

$$ulp(x) = z_{mach} \times 2^{e-127}$$

Ainsi grâce à cette formule, il suffit de trouver le $e$ pour trouver l'ulp d'un nombre. Pour celà on va procéder par décalages successifs pour "simuler" un logarithme d'où le nom de cette méthode. Ainsi pour trouver l'ulp de $x \neq 0$ on va procéder de la manière suivante :

- on initialise $t = 2^0$, à la fin de notre algorithme celle ci doit valoir $2^{e-127}$
- maintenant on procède à un test sur $x$, si $x \in [1, 2[$ alors on a fini car $e = 0$
- si $x < 1$ alors tant que $x < 1$ on fait $x \leftarrow 2x$ et $t \leftarrow \frac{t}{2}$
- si $x \geq 2$ alors tant que $x \geq 2$ on fait $x \leftarrow \frac{x}{2}$ et $t \leftarrow 2t$

Ainsi en sortie de cet algorithme $t$ vaut $2^{e-127}$, il suffit donc de renvoyer $t \times z_{mach}$

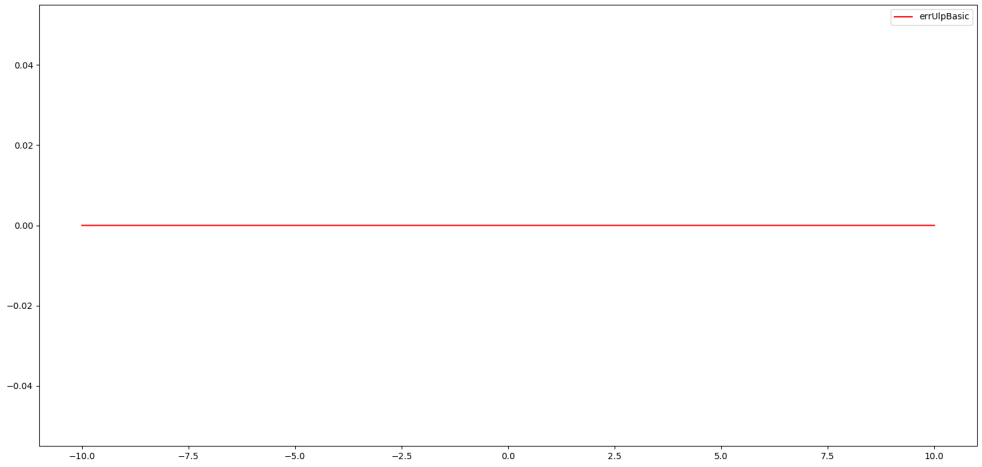



FIGURE 11 – Tracé de l'erreur d'ULP absolue

**Résultats** Celà peut parraître surprenant mais l'erreur en Java est de 0 quelque soit le point.

**Coût** Le coût est assez simple à calculer : en effet pour chaque itération, il y a 1 test d'égalité donc 1 soustraction, 1 multiplication par 2 et 1 division par 2, celà fait donc 62 cycles internes. Le nombre d'itérations quand à lui est le nombre de fois qu'il faut multiplier ou diviser $x$ pour atteindre l'intervalle $[1, 2[$ i.e. $|\lfloor \log_2(x) \rfloor| + 1$. Au total on a donc

$$N_{cycle} = 62(|\lfloor \log_2(x) \rfloor| + 1)$$

# 3 Implémentation en Deca

Nous avons donc choisi les meilleurs algorithmes pour chaque fonction et les avons implémentés en *Deca*. Désormais tous les graphes suivants concerneront les fonctions *Deca*.

## 3.1 Sinus et Cosinus

Nous avons donc implémenté la méthode des développements de Taylor.

**Précision** Pour les précisions nous traceront toujours cette courbe : $\left|\frac{fonctionDeca - fonctionJava}{ulp}\right|$

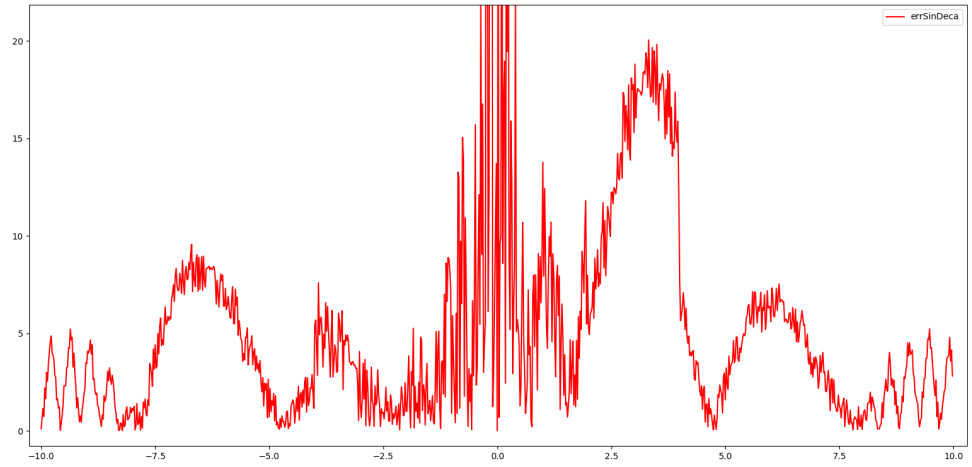



FIGURE 12 – Tracé de l'erreur de Sinus en ULP

On constate donc qu'on a toujours une précision inférieure à 20 ULP. On perd quand même beaucoup en précision par rapport à nos essais en Java, c'était attendu mais dommage. On constate un pic en 0, il ne signifie pas grand chose, en effet il s'agit d'une division de 0 par 0 (c'est évidemment en 0 que notre fonction Deca est la plus proche de la fonction Java, de plus en 0 ulp est très proche de 0 et donc en divisant 0 par 0, on ne peut rien interpréter en ce point).

**Coût** Pour les précisions nous traceront la courbe $N_{cycle} = f(x)$ où nous feront varier $x$ dans un intervalle convenable. Pour celà nous utilisons l'option "-d" de ima et un script qui s'occupe d'écrire les programmes *deca*, de les compiler et exécuter.

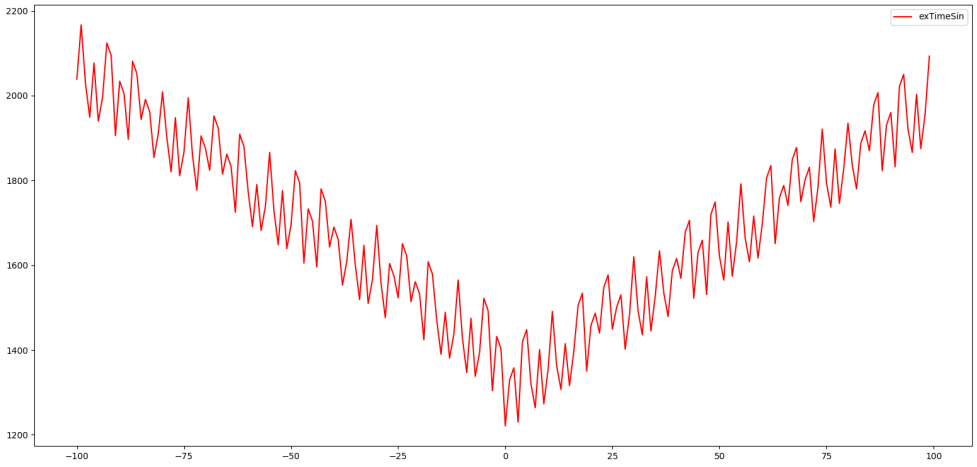



FIGURE 13 – $N_{cycle}$ en fonction de la valeur calculée pour Sinus

Ainsi on remarque donc que notre formule $N_{cycle} = 20\left|\frac{f}{p}\right| + 220$ est vrai dans la forme. En effet les valeurs sont fausses (il faut garder à l'esprit qu'on a volontairement laissé de côté toutes les instructions autre que ADD, SUB, MUL, DIV) mais l'allure de la courbe est exacte. Ceci permet de nous rendre compte que le problème de notre implémentation est le modulo flottant. En effet, plus la valeur qu'on cherche à calculer est grande plus il faut faire de soustraction pour la ramener entre 0 et $2\pi$.

On ne trace pas Cosinus, car les résultats sont exactement les mêmes étant donné notre implémentation.

## 3.2 Arctan et Arcsin

Pour Arctan et Arcsin, nous avons donc décidé d'implémenter la méthode de Newton.

### Précision

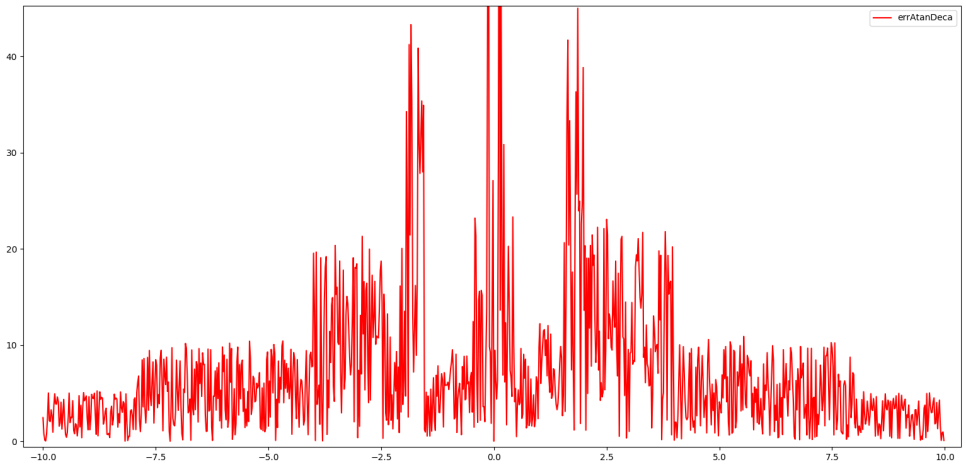



FIGURE 14 – Tracé de l'erreur de Arctan en ULP

On constate donc une précision inférieure à 20 ULP. C'est de nouveau bien plus que notre prototype en Java mais celà reste acceptable. De plus on remarque que la précision tend vers 0 lorsque $|x|$ tend vers $+\infty$.

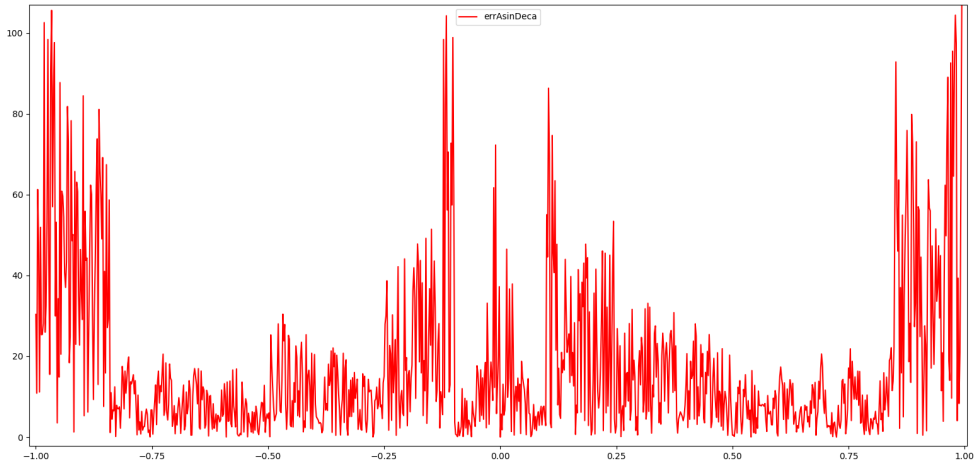



FIGURE 15 – Tracé de l'erreur de Arcsin en ULP

Pour Arcsin le résultat est décevant, en effet nous atteignons des pics à 100 ULP. Cependant comme Arcsin est calculé grâce à Arctan et que nous avons vu qu'Arctan reste acceptable, nous pouvons en déduire que les pics d'imprécisions sont surement dus au calcul de la fonction racine carrée.

### Coût

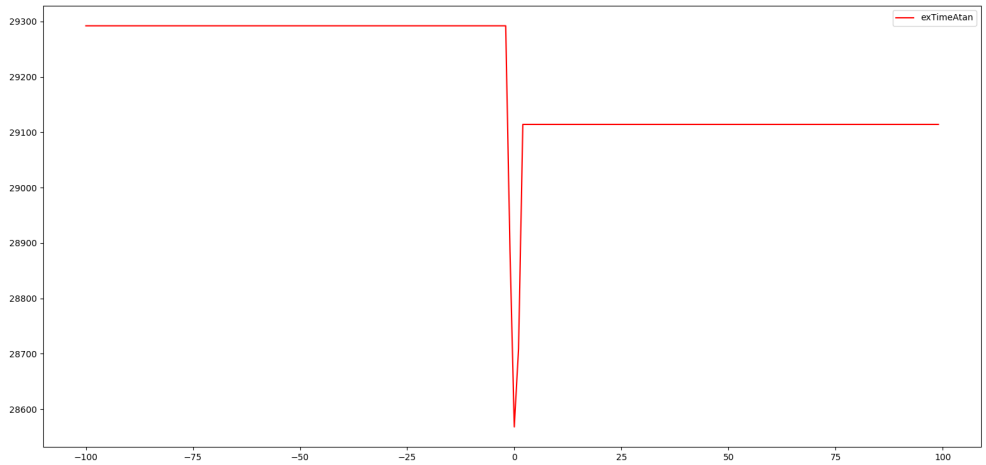



FIGURE 16 – $N_{cycle}$ en fonction de la valeur calculée pour Arctan

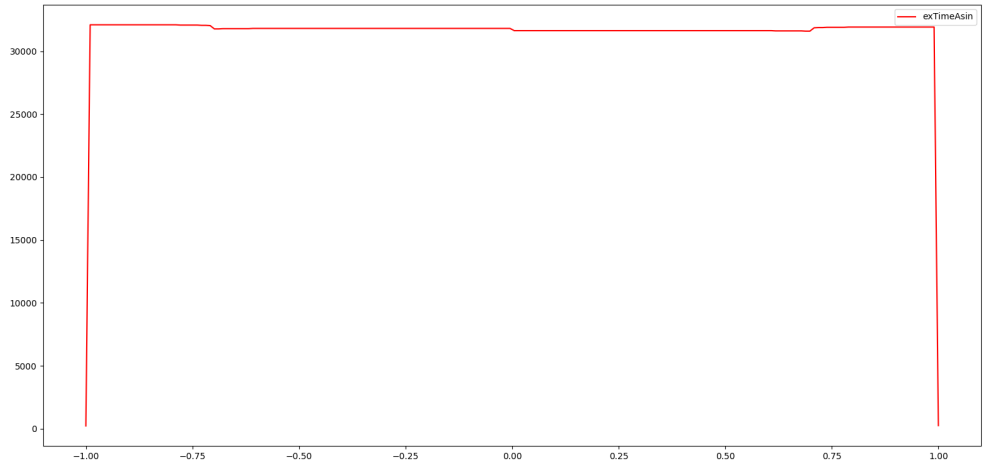



FIGURE 17 – $N_{cycle}$ en fonction de la valeur calculée pour Arcsin

Une fois de plus on retrouve les résultats précédents théoriques : $N_{cycle}^{Arctan} < N_{cycle}^{Arcsin}$ et ces deux valeurs sont des constantes. C'est donc très intéressant car celà signifie que pour des valeurs très grandes de x (pour Arctan en l'occurence), le temps de calcul, bien qu'important, restera le même.

## 3.3 ULP

### Précision

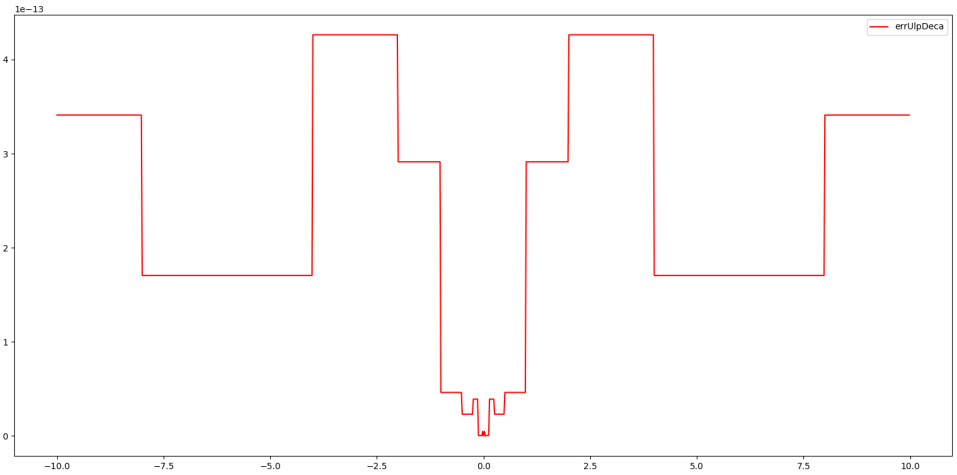



FIGURE 18 – Tracé de l'erreur de ULP absolue

Cette fois ci on a tracé l'erreur absolue et non l'erreur en ULP. On remarque donc qu'on a une différence de $10^{-13}$ pour $|x| > 1$ la où $ulp(x) > 10^{-7}$ ce qui est très satisfaisant.

Cependant en 0 on ne peut pas en dire autant étant donné que l'ulp de 0 est beaucoup plus petite que $10^{-13}$

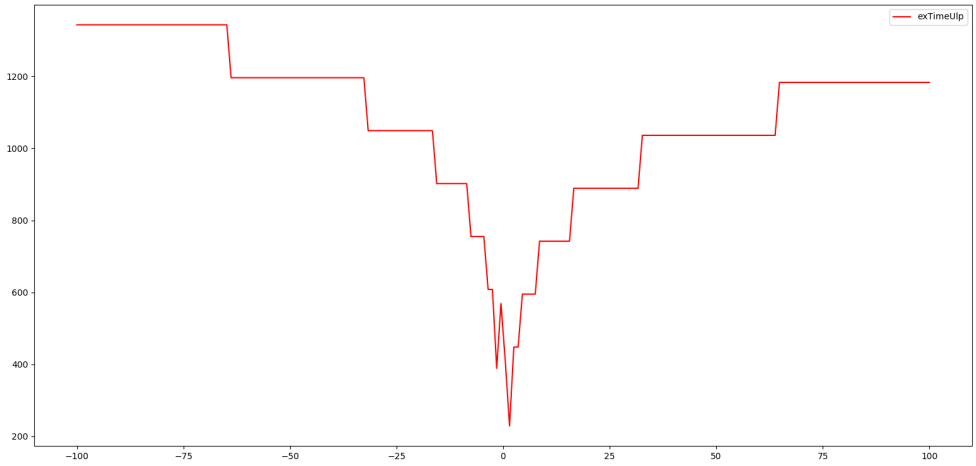



FIGURE 19 – $N_{cycle}$ en fonction de la valeur calculée pour ULP

On retrouve de nouveau notre courbe théorique : l'aspect "en escalier" vient de la partie entière et on reconnaît vaguement la courbe d'un logarithme. C'est une bonne chose car le logarithme croit très lentement ce qui nous permet de calculer l'Ulp pour de grandes valeurs sans avoir à attendre.

**NB** Pour l'Ulp, nous avons oublié le cas où $f = 0$ où $f$ est l'argument de la fonction. En effet notre algorithme va multiplier $f$ par 2 et partir en boucle infinie. C'est dommage mais ce n'est pas très difficile à corriger.

# 4 Conclusion

Pour Sinus et Cosinus nous atteignons dans le pire des cas 20 ULP, ce qui est acceptable mais loin d'être satisfaisant. On pourrait peut-être augmenter l'ordre du développement limité mais surtout on pourrait utiliser l'instruction FMA pour éviter de perdre de l'information lors de nos multiplications / additions (cette remarque est valable aussi pour nos autres fonctions).

Pour Arctan, comme précedemment on pourrait essayer d'augmenter le DL ou d'utiliser FMA. Cependant pour améliorer la rapidité on pourrait essayer une disjonction de cas. En effet on a vu lors de notre prototypage que la fonction Arctan qui utilisait le développement de Taylor était beaucoup plus rapide pour une précision similaire en dehors de $\{-1, 1\}$. Du coup il serait peut-être judicieux de distinguer le cas $x$ au voisinage de 1 et le cas contraire.

Pour Arcsin, nous avons bien vu que la précision est très mauvaise, une première idée serait de vérifier le nombre d'itérations nécessaire à la convergence de notre fonction racine

carrée. Nous avons fait $N = 15$ itérations mais c'était peut-être une faute d'inattention et qu'en réalité il faudrait plus d'itérations.

Enfin pour Ulp, bien que la méthode semble assez simple, elle est en somme très rapide et relativement précise. Il aurait été intéressant d'utiliser les instructions de décalages mais *Deca* n'en propose pas contrairement au langage assembleur.

D'une manière générale, nous aurions aimé passer plus de temps sur l'extension. En effet nous sommes assez déçu de nos résultats dans l'ensemble et nous aurions aimé nous plonger un peu plus dans le calcul flottant, pour minimiser toutes les erreurs d'arrondis. Une première idée serait par exemple, lorsqu'on multiplie deux flottants entre-eux, stocker le dépassement dans une autre case mémoire, empêchant ainsi les erreurs d'arrondis pour les calculs futurs.

# Références


[1] *IEEE 754*. Wikipedia, https ://fr.wikipedia.org/wiki/IEEE_754

[2] *Unit in the last place*. Wikipedia, https ://en.wikipedia.org/wiki/Unit_in_the_last_place

[3] *Calcul informatique des fonctions trigonométriques* http ://www.trigofacile.com/maths/trigo/calcul/cordic/cordic.htm