´l

Documentation de l'extension TRIGO

Groupe GL49:

CHENGUEL Sofièn
FEGUIRI Wael
HASSOUNE Hamza
OLLIER Valentin
SAUNIER Victor

École Nationale Supérieure d'Informatique et de
Mathématiques Appliquées de Grenoble

Janvier 2020

Table des matières

1 Introduction

L'extension TRIGO abordée représente à la fois une partie conséquente et intéressante du projet GL. Le but de celle-ci est de concevoir les fonctions ulp (Unit in the Last Place), sin, cos, arctan et arcsin adaptées aux nombres flottants en langage Deca, et ayant comme contrainte d'être à un ulp près de la bonne valeur qu'on considère dans la suite du projet la valeur renvoyée par les fonctions implémentées en Java (utilisées pour les comparaison). La diffculté de cette partie se manifeste dans le calcul avec une très grande précision (1 ulp près) des valeurs des fonctions, sachant d'une le langage Deca ne supporte que des floats codés sur 32 bits, mais en plus le nombre π est irrationnel (et même transcendant) ce qui signifie qu'il n'est pas représentable sur machine, ceci va compliquer certains calcul par la suite.

2 Fonctions utiles

Afin d'implémenter les cinq fonctions demandées, il s'est avéré nécessaire de passer par certaines fonctions intermédiaires dont on détaillera le rôle et/ou le principe dans cette partie.

2.1 La fonction signe

Cette fonction retourne 0 si son argument est nul, -1 s'il est négatif et $+1$ s'il est positif, elle sera utilisé à toute fin utile dans certaines méthodes implémentées dans le projet.

2.2 La fonction absolue

Renvoie directement la valeur absolue du float qu'elle prend en argument.

2.3 La fonction pow1

Cette fonction calcul uniquement les puissances de (-1) en tirant partie de la parité de l'exposant de celui-ci, elle est utilisée notamment dans le calcul des développement en série de Taylor pour être plus rapide et permettre de calculer plus de valeurs.

2.4 La fonction sqrt

Nous avons implémenté la fonction racine carrée en utilisant la **Méthode de Héron**, celle-ci dérive directement de la méthode de Newton et permet de récupérer la racine carrée avec une bonne précision.
Nous utilisons donc la formule de récurrence suivante :

$$\boxed{a_{n+1} = \frac{1}{2}(a_n + \frac{A}{a_n})}$$

avec :
A : Notre argument (float dont on cherche la racine carrée).
a_0 = A
Un raisonnement purement mathématique permet de montrer que la suite a_n converge vers sqrt(A).
Nous implémentons donc un algorithme itératif avec comme condition d'arrêt $a_{n+1} = a_n$.

2.5 La fonction pow

Afin d'implémenter la fonction puissance nous avons utilisé un schéma récursif avec disjonction de cas : exposant nul, strictement positif ou strictement négatif.
L'idée est simple, elle consiste à ramener tous les cas au cas initial : exposant est nul.
Et la formule de récurrence est comme suit :

$$\begin{cases} pow(x,0) = 1 \\ pow(x,n) = x\, pow(x,n-1) \,\text{si}\, n > 0 \\ pow(x,n) = \dfrac{1}{x}\, pow(x,n+1) \,\text{si}\, n < 0 \end{cases}$$

3 Fonctions recherchées

3.1 La fonction ulp

— Définition :
La fonction ulp est utilisée afin de calculer les erreurs par rapport aux valeurs de référence, elle nous donne une idée sur la précision local en fonction de son argument, celle-ci est en effet variable selon que x se trouve proche de 0 ou pas.

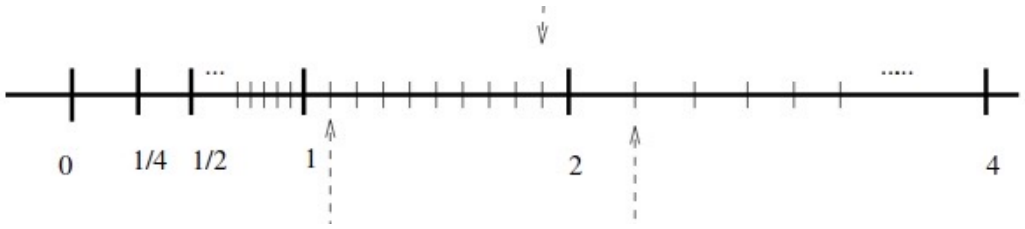

FIGURE 1 – Illustration des flottants représentable et leur répartition sur la droite réel.

La fonction ulp d'un flottant est donnée par la formule suivante :

$$\boxed{ulp(x) = 2^{-p+1+k}}$$

avec :
$p - 1 = 23$: le nombre de bits nécessaires afin de coder la mantisse d'un nombre flottant.

k : l'exposant de la plus grande puissance de 2 inférieure ou égale à x.

— Principe d'implémentation :
DAns un premier temps un algorithme naïf a été implémenté afin de récupérer la valeur de l'ulp, mais l'idée d'utiliser une recherche dichotomique s'est avérée plus efficace et rapide pour le même calcul (moins d'opérations effectuées).
Pour implémenter la fonction ulp il suffit donc de trouver l'entier k tel que :

$$\boxed{2^k \leq x < 2^{k+1}}$$

Pour ce faire, nous avons utilisé dans notre code un algorithme dichotomique,en adoptant la définition de Kahan et en l'adaptant au cas des nombre flottants, ce dernier consiste à trouver l'entier k et calculer par la suite l'image par l'ulp.

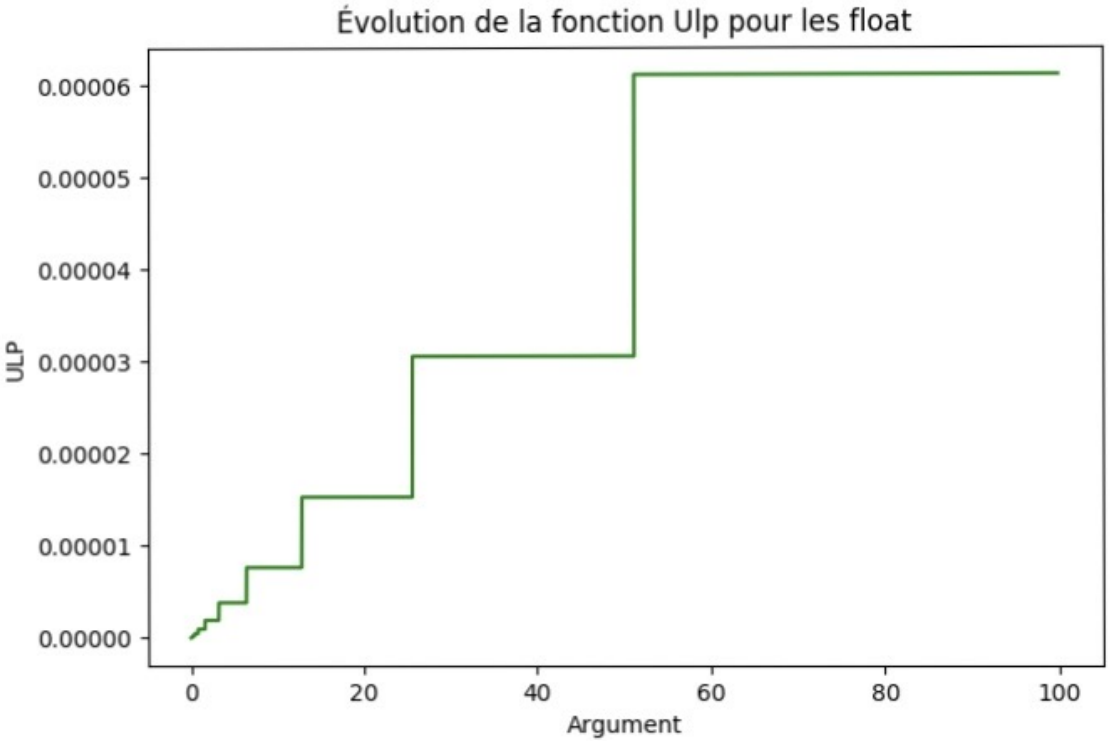

FIGURE 2 – Représentation du graphique de la fonction ulp en escalier.

— Cas particuliers :
La fonction ulp retourne des valeurs particulières, ces dernières ont été traitées à part dans le code (en java), mais celui-ci a été adapté par la suite au contenu du Lexer de notre compilateur en langage deca.
Si x est trop petit pour être représentable en float on retourne la plus petite valeur représentable en float : 2^{-126-p}
Si l'argument est égal à 0 on retourne également le plus petit flottant représentable.

Si l'argument est infinity on retoune infinity.
Si x est très grand pour être représentable on retourne infinity également.

— Résultat :
Après avoir effectué un bon nombre de tests, la fonction codée retourne exactement les mêmes valeurs que la fonction ulp du langage java pour les flottants.

3.2 La fonction sin sur $[0, \frac{\pi}{2}]$

3.2.1 L'algorithme du Cordic

L'algorithme du Cordic utilisé ici (COordinate Rotation DIgital Computer, « calcul numérique par rotation de coordonnées ») consiste principalement en la décomposition d'un angle $\theta \in [0, \frac{\pi}{2}]$ en une série convergente de la forme $\theta = \sum_{i=0}^{\infty} \delta_i \, \epsilon_i$.
Avec δ_i une suite à valeurs dans $\{-1, 1\}$, et ϵ_i une suite réel bien choisie et qui vérifie certaines conditions (à titre d'exemple : $\frac{1}{2}\epsilon_i \leq \epsilon_{i+1} \leq \epsilon_i \ (\forall i \in \mathbb{N})$), afin d'imposer la convergence de la série précédente.
On adopte comme valeurs pour la suite : $\epsilon_i = \arctan(2^{-i})$, en effet elle vérifie les conditions nécessaires pour la convergence de la série générée $\sum_{i=0}^{\infty} \delta_i \, \epsilon_i$, mais surtout elle permet de simplifier les calculs pour l'ordinateur,ce choix n'est donc pas anodin car les calculs à effectuer sont des puissances de 2, ce qui consiste essentiellement à effectuer des opération de décalage de bits.

En effet :
Soit $\theta \in [0, \frac{\pi}{2}]$ et (x, y) un vecteur dans le plan, alors en effectuant une rotation d'angle θ de ce vecteur sur le plan, on obtient le vecteur suivant en utilisant la matrice de rotation :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Ce qui équivaut à écrire, dans le cas où $\theta = \sum_{i=0}^{n-1} \epsilon_i$:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\epsilon_0 & -\sin\epsilon_0 \\ \sin\epsilon_0 & \cos\epsilon_0 \end{pmatrix} \begin{pmatrix} \cos\epsilon_1 & -\sin\epsilon_1 \\ \sin\epsilon_1 & \cos\epsilon_1 \end{pmatrix} ... \begin{pmatrix} \cos(\epsilon_{n-1}) & -\sin(\epsilon_{n-1}) \\ \sin(\epsilon_{n-1}) & \cos(\epsilon_{n-1}) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Le but étant de faire apparaître les $\tan(\epsilon_i)$, et donc en factorisant par $\cos(\epsilon_i)$ dans toutes les matrice du produit, on aboutit à :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \prod_{i=0}^{n-1} \cos(\epsilon_i) \begin{pmatrix} 1 & -\tan(\epsilon_i) \\ \tan(\epsilon_i) & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Dorénavant, en posant : $\theta = \sum_{i=0}^{n-1} \delta_i \, \epsilon_i$, et en remplaçant dans l'égalité précédente : ϵ_i par : $\delta_i \, \epsilon_i$, tout en sachant que la fonction *tan* est impaire et que la suite δ_i prend des valeurs contenues dans $\{-1, 1\}$, on en déduit que :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \prod_{i=0}^{n-1} \cos(\epsilon_i) \begin{pmatrix} 1 & -\delta i \ \tan(\epsilon_i) \\ \delta i \ \tan(\epsilon_i) & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Nous allons alors remplacer les ϵ_i par $\arctan(2^{-i})$ dans l'égalité précédente, ce qui nous donne :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \prod_{i=0}^{n-1} \cos(\arctan(2^{-i})) \begin{pmatrix} 1 & -\delta i\, 2^{-i} \\ \delta i\, 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Ce choix effectué pour les ϵ_i est très optimale. D'abord parce qu'on obtient des puissances de 2, qui se calculent facilement et rapidement par uns machine comme expliqué ci-dessus, mais en plus parce que la suite définie par $(\arctan(2^{-i}))_{i\in\mathbb{N}}$ vérifie les conditions de convergence précédemment définis (la démonstration de sa décroissance rapide est aisément démontrable).
La suite $(\delta_i)_{i\in\mathbb{N}}$ prenant des valeurs dans $\{-1, 1\}$ quant à elle, est définie pour la série convergente suivante : $\theta = \sum_{i=0}^{\infty} \delta_i \arctan(2^{-i})$, comme suit : $\delta_i = signe(\theta - S_i)$, avec $S_i = \sum_{k=0}^{i-1} \delta_k \arctan(2^{-k})$: ce qui représente la somme partielle de θ d'ordre i, on s'arrange clairement afin d'assurer la convergence.

on pose en particulier $(x, y) = (1, 0)$, et $\theta = \sum_{i=0}^{\infty} \delta_i \arctan(2^{-i})$ (en fait tout angle s'écrit sous cette forme), et on obtient :

$$\begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} = \prod_{i=0}^{\infty} \cos(\arctan(2^{-i})) \begin{pmatrix} 1 & -\delta i\, 2^{-i} \\ \delta i\, 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

On pose dans notre cas : $K = \prod_{i=0}^{\infty} \cos(\arctan(2^{-i})) = \prod_{i=0}^{\infty} \frac{1}{\sqrt{1+2^{-2i}}}$, d'où :

$$\begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} = \prod_{i=0}^{\infty} \begin{pmatrix} 1 & -\delta i\, 2^{-i} \\ \delta i\, 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} K \\ 0 \end{pmatrix}$$

Cette dernière égalité se traduit par la construction des suites : $(x_i)_{i\in\mathbb{N}}$ et $(y_i)_{i\in\mathbb{N}}$ qui vérifient $\lim\limits_{n\to+\infty} x_n = \cos\theta$ et $\lim\limits_{n\to+\infty} y_n = \sin\theta$, comme suit :

$$\begin{cases} x_0 = K;\ y_0 = 0;\ z_0 = \theta \\ \delta_i = signe(z_i) \\ x_{i+1} = x_i - \delta_i\, y_i\, 2^{-i} \\ y_{i+1} = y_i + \delta_i\, x_i\, 2^{-i} \\ z_{i+1} = z_i - \delta_i \arctan(2^{-i}) \end{cases}$$

On utilise dans la suite des calculs la valeur de : $K = 0.60725294$ qui représente en effet le produit partiel des $\cos(\arctan(2^{-i}))$ allant jusqu'à 50, le calcul a été effectué indépendemment en utilisant Python.
On itère alors l'algorithme implémenté 50 fois, ce qui signifie qu'on considère que : $\sin(\theta) = y_{50}$.
L'algorithme marche parfaitement loin de 0 (pour $x \geq 0.5$, on a une erreur maximale de 5 ulps) comme le montre la figure 4, néanmoins il s'éloigne beaucoup du résultat attendu quand on s'approche de 0 comme le montre la figure 5.

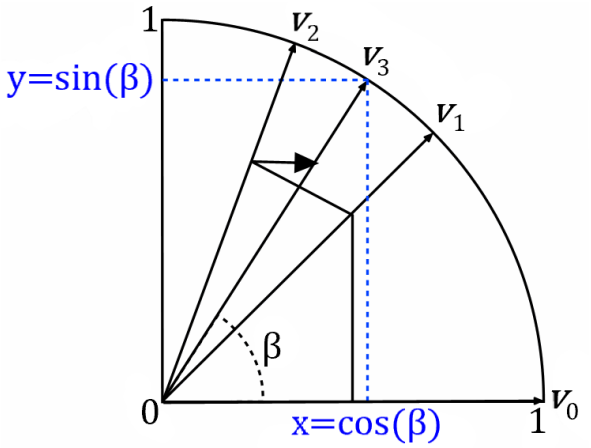

FIGURE 3 – Illustration de la convergence vers l'angle souhaité après plusieurs rotations.

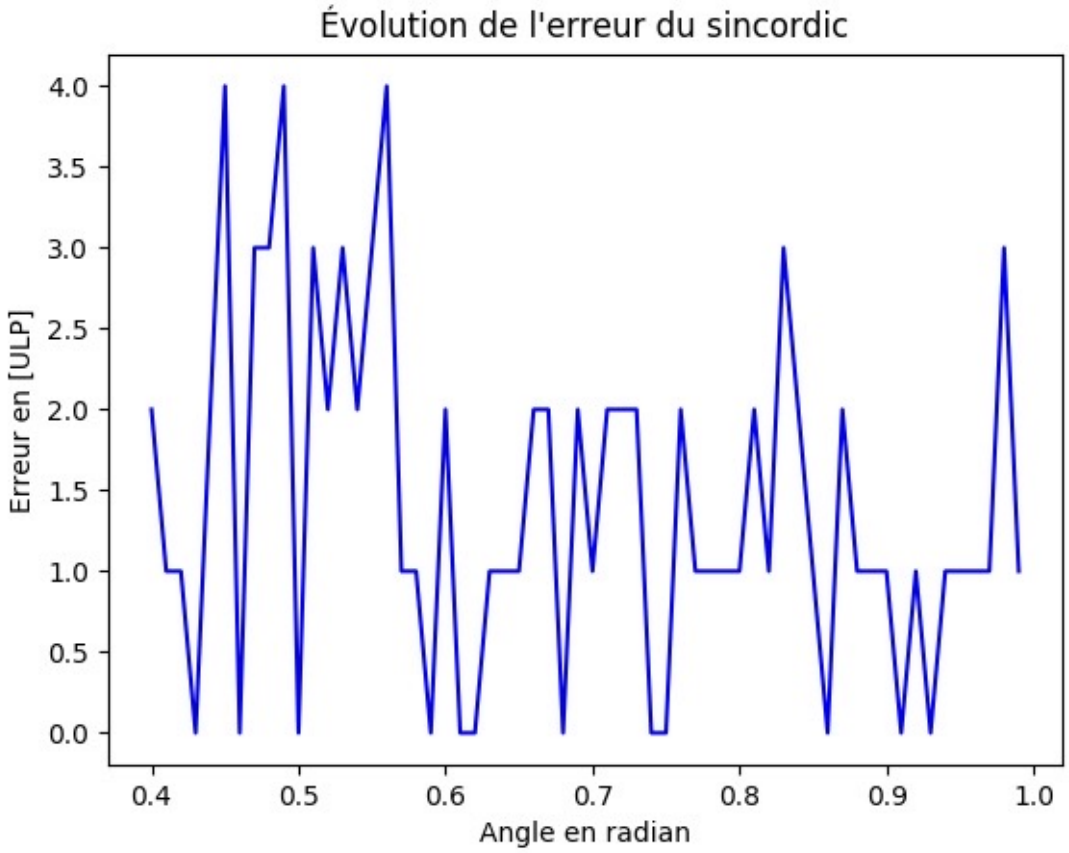

FIGURE 4 – Représentation de l'erreur sur $\sin(x)$ avec l'algorithme du CORDIC, pour $x \geq 0.5$

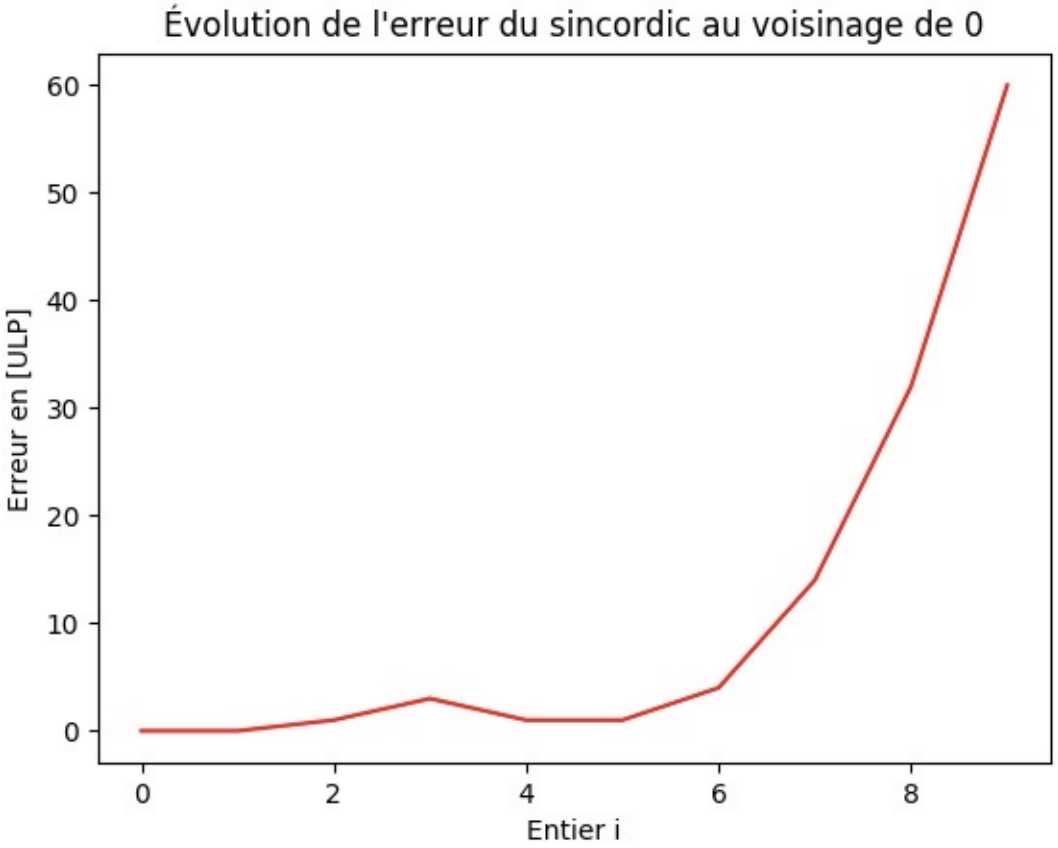

FIGURE 5 – Représentation de l'erreur en ulp sur $\sin(2^{-i})$ avec l'algorithme de Cordic en fonction de i entier naturel (voisinage de zéro)

L'erreur devient très importante pour i assez grand.
Le tableau suivant résume les erreurs en ulps entre le sinus implémenté par l'algorithme du Cordic et celui du langage Java, calculées pour certaines valeurs de i :

i	$\sin_{cordic}(2^{-i})$	$\sin_{java}(2^{-i})$	Err_{ULP}
50	7.4360305×10^{-9}	8.881784×10^{-16}	7.023133×10^{13}
60	7.4360305×10^{-9}	$8.6736174 \times 10^{-19}$	7.1916874×10^{16}
70	7.436030×10^{-9}	$8.4703295 \times 10^{-22}$	7.364288×10^{19}
80	7.436030×10^{-9}	8.271806×10^{-25}	7.541031×10^{22}
90	7.436030×10^{-9}	$8.0779357 \times 10^{-28}$	7.7220155×10^{25}
100	7.436030×10^{-9}	7.888609×10^{-31}	7.707344×10^{28}

Il est donc clair que ce premier algorithme adopté doit être remplacé par une autre méthode pour le calcul des valeurs d'un sin au voisinage de zéro.

3.2.2 Formule de Taylor

Pour remédier à l'erreur au voisinage de 0 de la fonction *sin* générée par l'algorithme précédent, on choisit dans ce cas un développement limité en utilisant la formule de Taylor jusqu'à l'ordre 37, parce qu'on connait l'erreur mathématique exacte qui est en

$o(x^{37})$ mais surtout car les factorielles au delà de cette valeur divergent.
Le développement limité de la fonction *sin* à l'ordre 37 s'écrit :

$$\boxed{\sin(x) = \sum_{k=0}^{k=18} \frac{(-1)^k}{(2k+1)!} x^{2k+1}}$$

Pour effectuer ce calcul sommatoire, il est possible de calculer simplement chaque terme de la somme, et ensuite d'additionner ce qui coûtera en terme de complexité $O(100^2)$.
Mais il existe une méthode dit de Horner qui permet d'effectuer le calcul de la valeur d'un polynôme en un point donné en $O(n)$ avec n le degré du polynôme. Cette méthode a été abandonnée par la suite vu la rapidité de convergence au voisinage de zéro de notre formule.

Le développement limité donne des résultats très précis au voisinage de 0 comme le montre la figure 6 et même pour $x \leq 0.4$ (pas 0 ulp dans ce cas, comme le montre la figure **??**, mais assez précis). Cette dernière affirmation peut être expliquée par le développement de Taylor qui affirme que le sinus s'écrit sous la forme d'une somme infinie avec les termes d'ordre supérieurs qui peuvent être négligés tant que x est petit, et non pas par le développement limité. Ainsi quand x devient grand les termes d'ordre supérieurs ne peuvent plus être négligés, et deviennent au contraire très grands (x^{2k+1}) et par la suite la somme finie diverge au sens informatique (dépasse la valeur maximale autorisée pour les flottants).

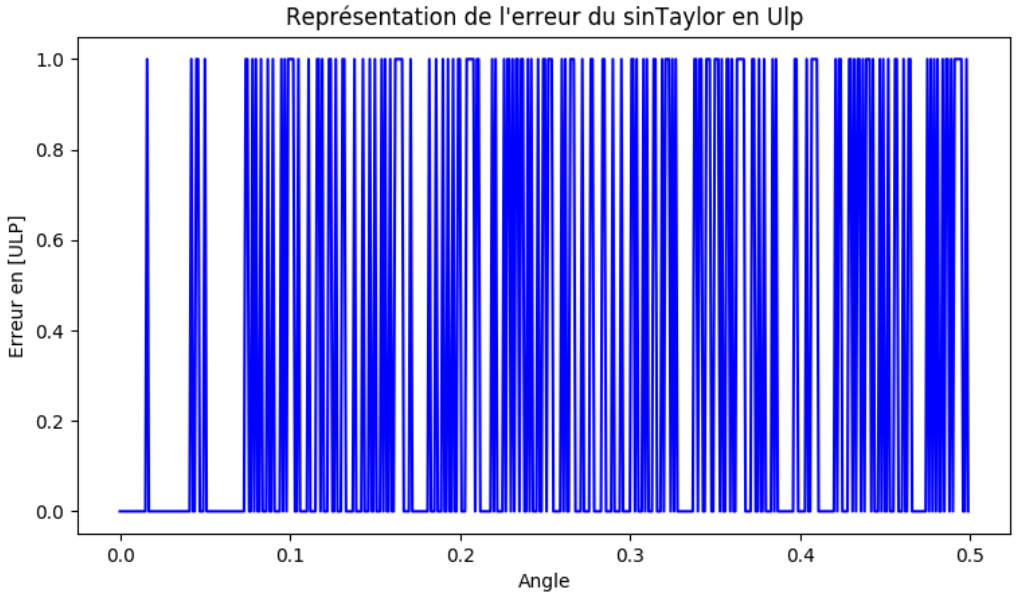

FIGURE 6 – Représentation de l'erreur sur $\sin(x)$ issu de la formule de Taylor pour $x \leq 0.5$

3.2.3 Résumé de la méthodologie :

Au final on a codé la fonction sin selon deux cas : $x \leq 0.4$ et $x \geq 0.5$.
Si $x \leq 0.5$, on utilise le développement en série de Taylor, sinon on tire profit de l'algo-

rithme du Cordic. La valeur 0.5 n'a pas été choisie au hasard, mais suite à une multitude de tests comparant l'erreur (différence entre notre sin et celui du langage Java en valeur absolue divisée par l'ulp de ce dernier) maximale dûe au développement limité en série de Taylor et celle dûe à l'algorithme du Cordic.
Finalement en traçant notre sin sur $[0, \frac{\pi}{2}]$ on obtient la courbe ci dessous, qui se superpose avec celle du langage Java :

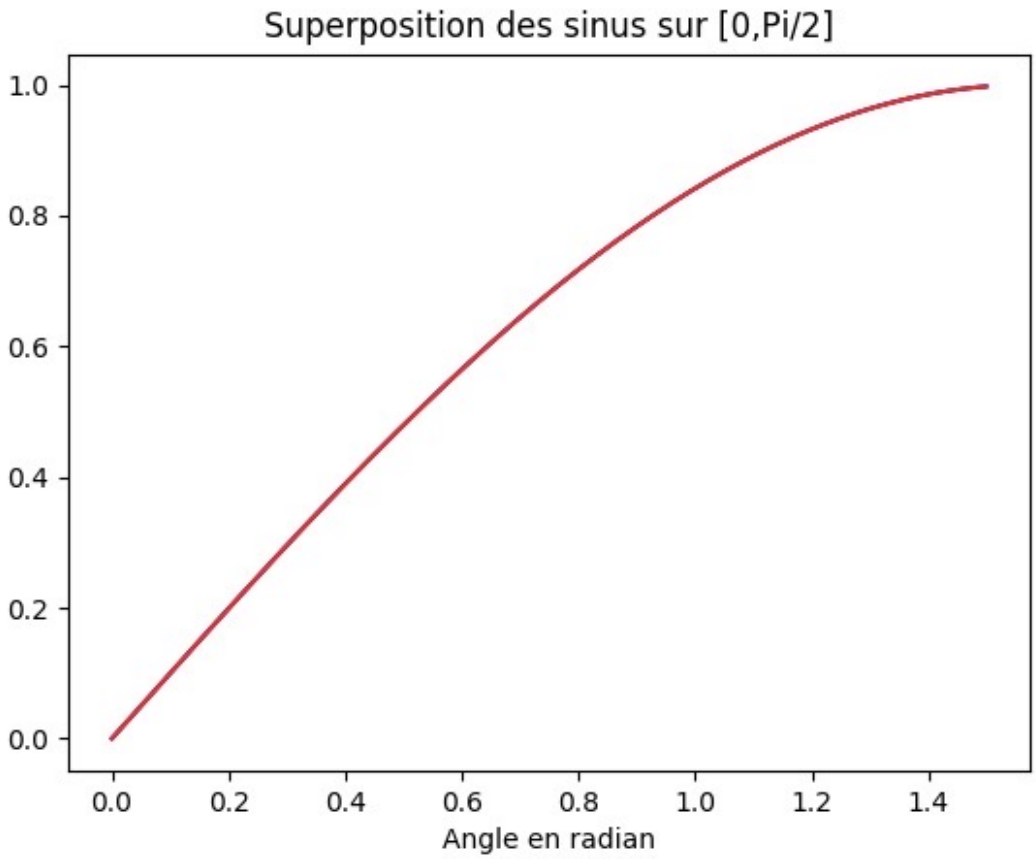

FIGURE 7 – Représentation de la superposition de notre sin (en bleu) et celui de Java (en rouge) sur $[0, \frac{\pi}{2}]$

3.3 La fonction cos sur $[0, \frac{\pi}{2}]$

L'algorithme du Cordic fonctionne aussi pour le cosinus comme vu précédemment. En effet, on sait que : $\lim_{n \to +\infty} x_n = \cos\theta$.
Mais le problème majeur rencontré ici était au voisinage du point $\frac{\pi}{2}$ comme le montre la figure 7. On n'arrivait pas qu'à l'aide de l'algorithme du Cordic à avoir une bonne estimation de $cos(\frac{\pi}{2})$.
On avait en effet une erreur (définie comme étant la différence entre notre cosinus et le cosinus du langage java en valeur absolue divisée par l'ulp du cosinus java) de l'ordre de 10^6 ulp qui est monstrueux.

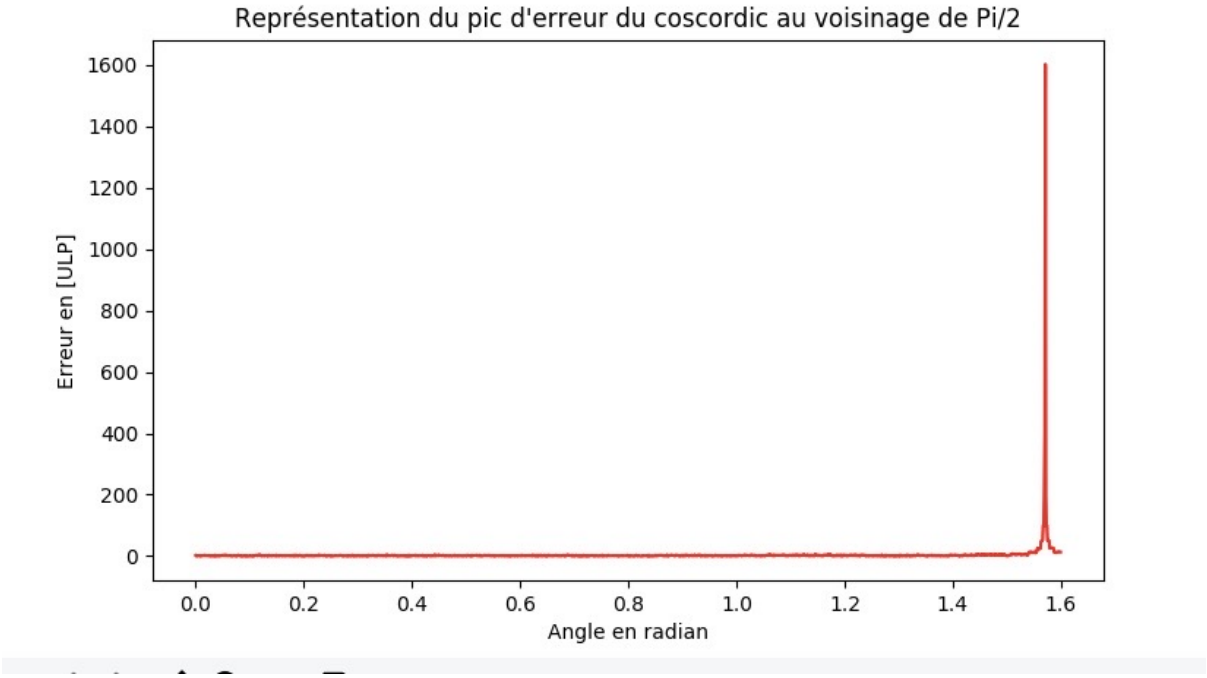

FIGURE 8 – Représentation du pic d'erreur sur le cosinus en utilisant l'algorithme du CORDIC, au voisinage de $\frac{\pi}{2}$

Pour contourner ce problème, et sachant qu'on a implémenté un bon sinus sur tout l'intervalle $[0, \frac{\pi}{2}]$, on a préféré utiliser la relation mathématique liant les deux fonctions :

$$\boxed{\cos(x) = \sin(\frac{\pi}{2} - x)}$$

Mais un problème de précision subsiste quand même, car cette formule est vraie pour un π mathématique, c'est à dire un π d'une précision infinie et non pas un π informatique qui est représenté par un nombre fini de bits.
Néanmoins,à l'aide de cette transformation simple, on obtient des résultats satisfaisant près de $\frac{\pi}{2}$, ainsi que pour les autres valeurs se trouvant dans l'intervalle $[0, \frac{\pi}{2}]$.

Ainsi, on obtient de bons résultats pour la fonction cos sur tout l'intervalle $[0, \frac{\pi}{2}]$. En la traçant avec celle du langage Java, on obtient la figure ci-dessous :

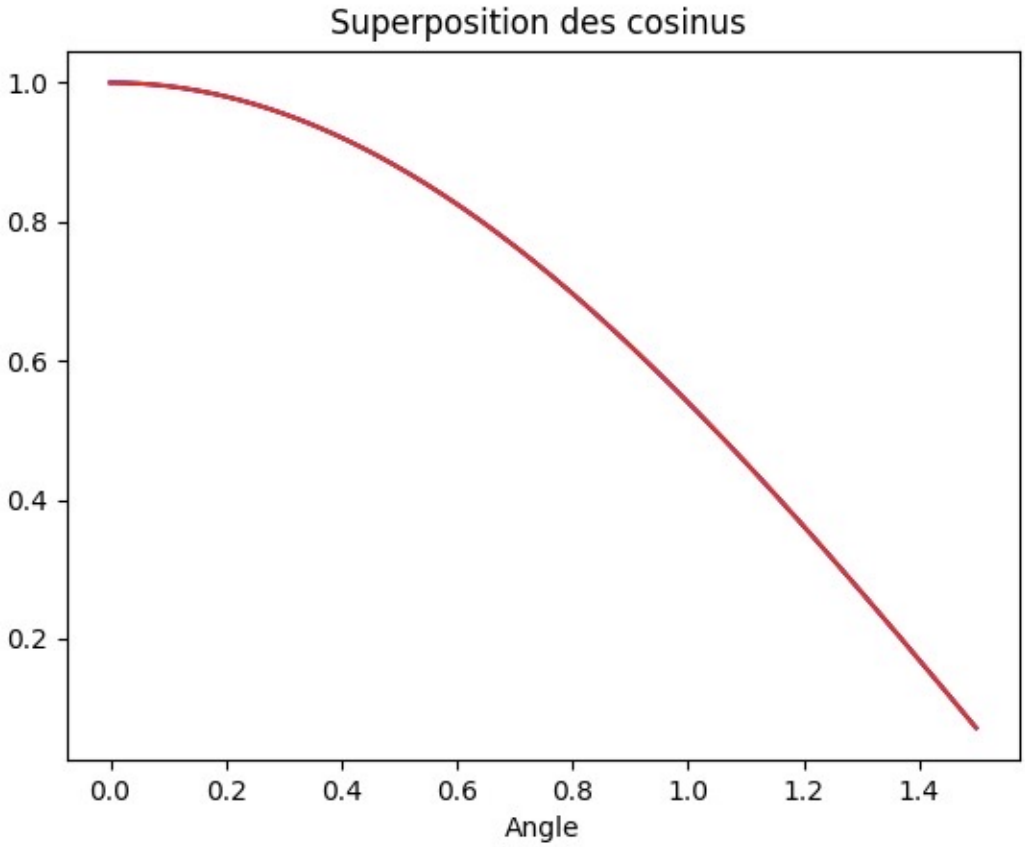

FIGURE 9 – Superposition de notre $cos(x)$ (en bleu) et celui du langage Java (en rouge) sur $[0, \frac{\pi}{2}]$

3.4 La fonction arctan

La fonction arctan est une fonction fondamentale dans notre projet, puisqu'elle est nécessaire pour l'implémentation des autres fonctions sin, cos et arcsin.
En effet, il existe une relation mathématique liant arcsin avec arctan, et ensuite pour coder cosinus et sinus à l'aide de l'algorithme du CORDIC précédemment présenté, il est nécessaire d'avoir les valeurs des $\arctan(2^{-i})$ pour $i \in \{1, ..., 50\}$.

Remarquons tout d'abord que les puissances 2^{-i} se rapprochent de plus en plus de 0. Ce qui nous pousse à penser à programmer en premier lieu le développement limité en série de Taylor de la fonction arctan. Mais le problème qui se pose ici c'est que ce dernier converge très lentement et pour un ordre très grand vu que les termes du développement limité de la fonction arctan sont en $O(\frac{1}{2k+1})$. Ajoutons à cela le fait que dans l'algorithme du CORDIC, il existe une boucle de taille 50, ceci qui fait exploser la complexité, et par la suite ralentit considérablement le temps d'exécution des fonctions implémentées par l'algorithme du CORDIC.
Notons qu'en plus la série de Taylor de l'arctan présente un pic d'erreur illustré sur la figure ci-dessous et que l'on va essayer d'éviter par la suite.

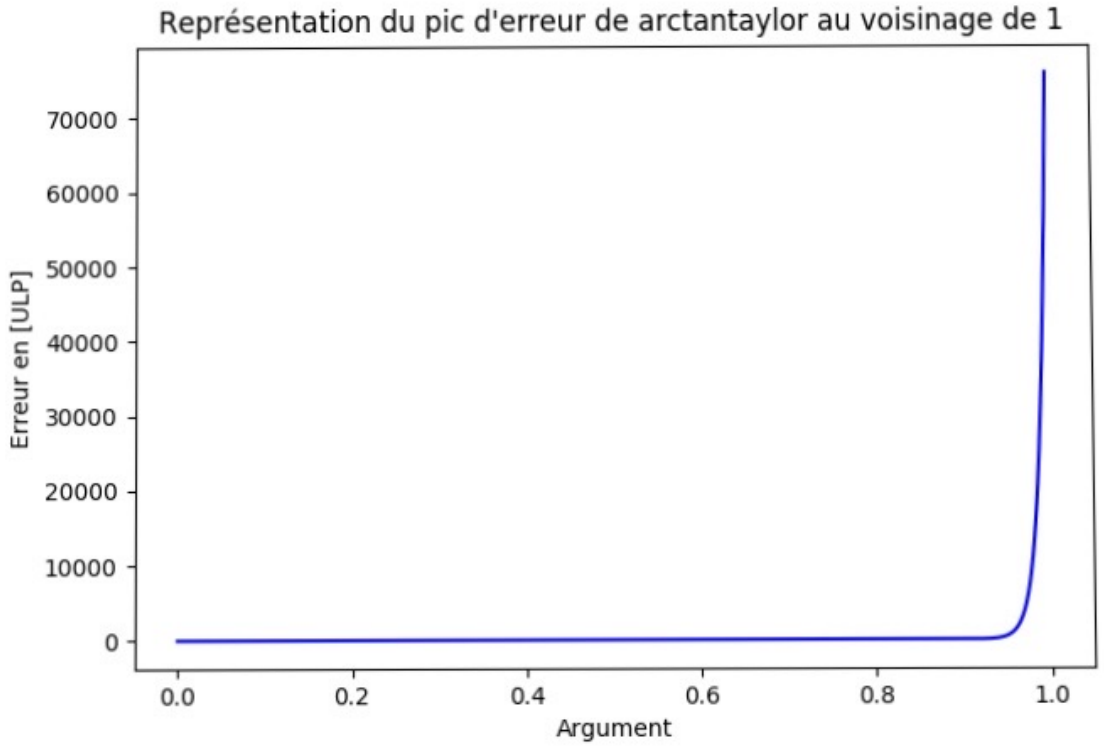

FIGURE 10 – Représentation du pic d'erreur pour arctanTaylor au voisinage de 1.

3.4.1 Algorithme adopté au voisinage de zéro

Afin de calculer l'arctangente pour les petits angles, nous avons utilisé l'algorithme de la moyenne arithmetico-géométrique, au lieu du développement limité en série de Taylor comme justifié précédemment. L'algorithme utilisé s'avère en effet plus précis et plus rapide.

— Définition :
Si a et b sont deux réels tels que $0 < b < a$, alors les deux suites suivantes définies par :

$$\begin{cases} a_{n+1} = \frac{a_n + b_n}{2} \\ b_{n+1} = \sqrt{a_{n+1} b_n} \\ a_0 = a \\ b_0 = b \end{cases}$$

convergent et tendent vers la même limite.

— Application à l'arctan :
Pour trouver l'arctangente on utilise l'algorithme ci-dessous en s'inspirant de la représentation en fractions continues d'Euler qui se déduit facilement du développement en série de Taylor de l'arctangente.
En prenant donc $a_0 = \frac{1}{\sqrt{1+x^2}}$ et $b_0 = 1$

On trouve arctan mathématiquement par la relation :

$$\boxed{\arctan(x) = \lim_{n \to +\infty} \frac{x}{a_n \sqrt{1 + x^2}}}$$

On s'arrête à la 15ème itération dans notre algorithme de la moyenne arithmético-géométrique adapté,ce qui était pour nous assez suffisant en terme de précision.

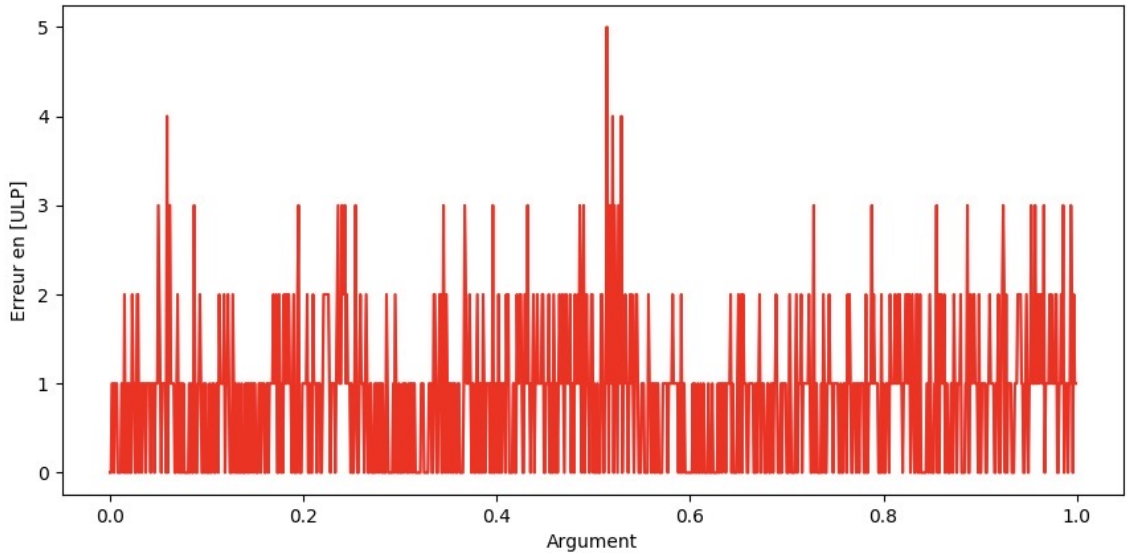

FIGURE 11 – Représentation de l'erreur sur $\arctan(x)$ en fonction de x pour $x \in [0, 1]$

On a maintenant une bonne fonction arctan valable pour $x \in [0, 1]$. L'intervalle choisi est judicieux, puisqu'on a en effet besoin des $\arctan(2^{-i})$ pour $i \leq 50$ pour l'algorithme du CORDIC, à l'aide duquel on a implémenté notre fonction sin.

3.4.2 Calcul pour les grandes valeurs

Afin de tirer profit au maximum de notre algorithme développé ci-dessus et qui marche très bien pour les petites valeurs, on utilise donc la relation mathématique suivante dès que l'argument dépasse 1 :

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

La figure qui suit montre bien la bonne précision de la méthode utilisée.

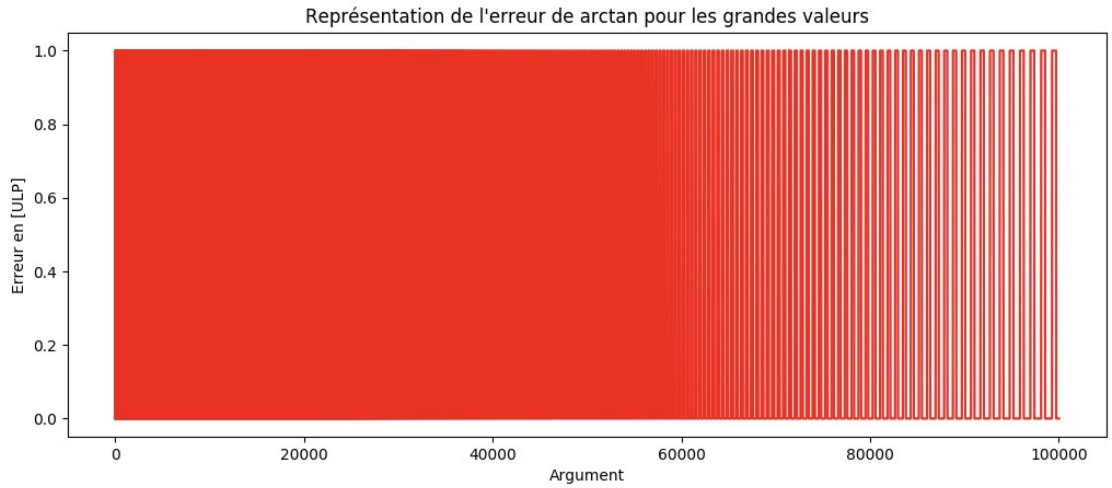

FIGURE 12 – Représentation de l'erreur sur $\arctan(x)$ en fonction de x pour $x \in [1, 100000]$

Dans le cas où x est négatif, il suffit simplement d'utiliser la parité de la fonction arctan qui est impaire.

3.4.3 Résumé de la méthodologie :

Lorsque $0 \leq x \leq 1$ on utilise l'algorithme de la moyenne arithmético-géométrique. Dans le cas contraire, c'est à dire lorsque $x \geq 1$, on utilise la formule mathématique mentionnée ci-dessus.
Et sinon si : $x \leq 0$, on applique simplement la transformation $\boxed{\arctan(x) = -\arctan(-x)}$.
Ainsi on obtient la courbe ci-dessous de la fonction arctan :

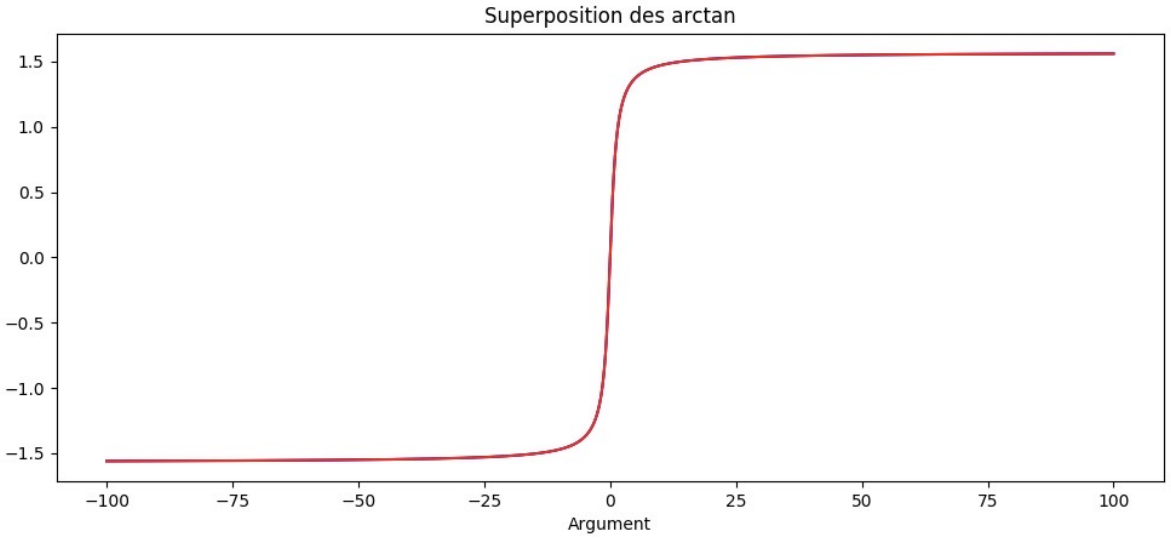

FIGURE 13 – Représentation de la superposition de notre $\arctan(x)$ (en bleu) et celle du langage Java (en rouge) sur l'intervalle $[-100, 100]$

3.5 La fonction arcsin

La fonction arcsin était la plus simple à implémenter, vu qu'il existe une relation mathématique liant celle-ci à la fonction arctan.
On a, en effet :

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

Ainsi si on a une fonction arctan et *sqrt* qui sont précises, on aura forcément et systématiquement une bonne fonction arcsin en terme de précision, et d'ailleurs c'est ce que montrent les deux figure suivantes :

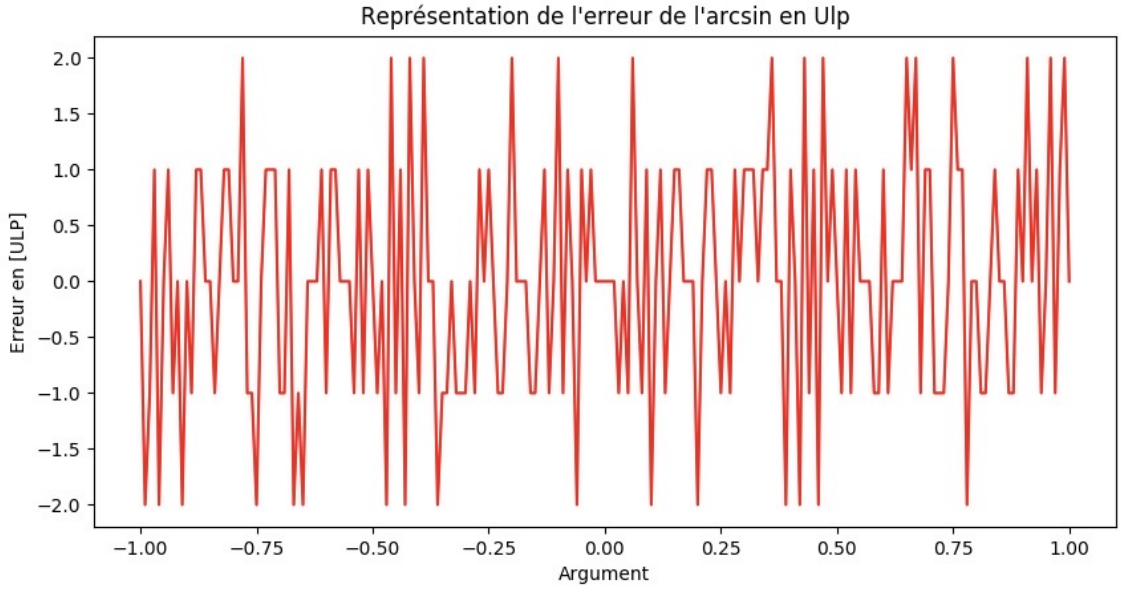

FIGURE 14 – Représentation de l'erreur sur $\arcsin(x)$ en fonction de x pour $x \in [-1, 1]$

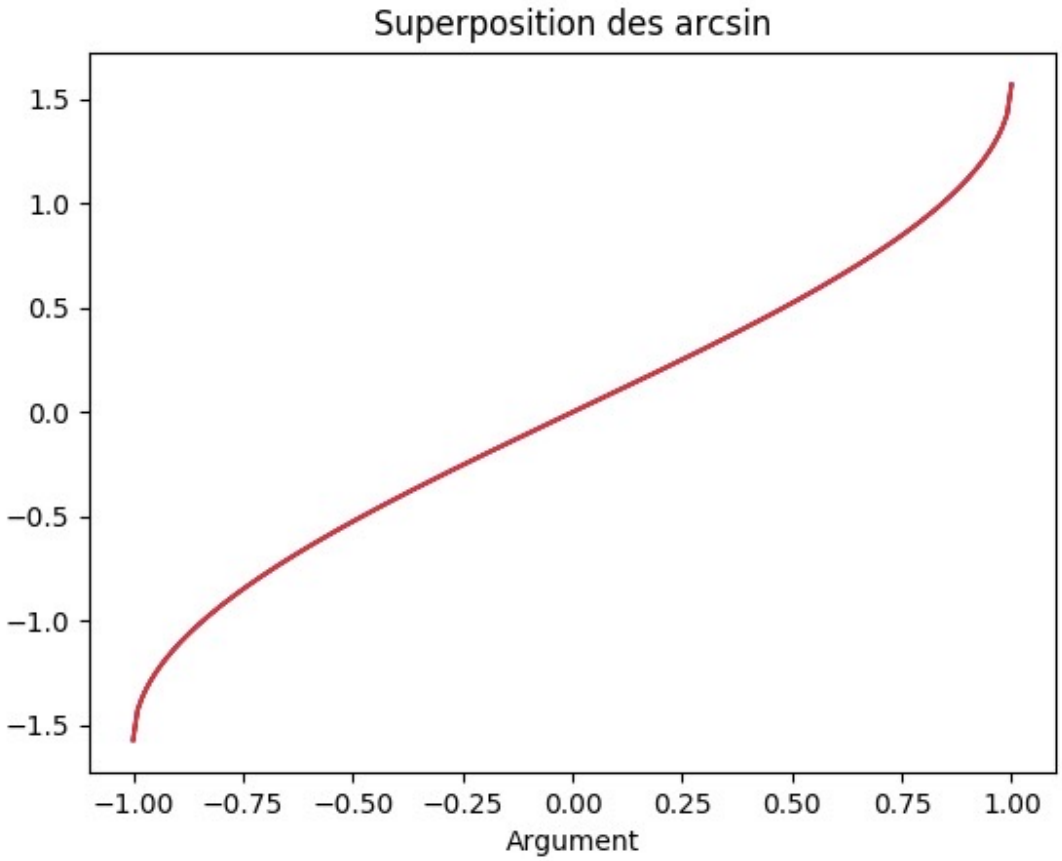

FIGURE 15 – Représentation de la superposition de notre arcsin(x) (en bleu) et celui de Java (en rouge) sur $[-1, 1]$

3.6 La réduction d'angle (pour les grandes valeurs)

Cette partie représente peut être la partie la plus délicate de l'extension TRIGO.
En effet, Les fonctions *sin* et *cos* implémentées pour le moment ne sont valables que pour les x tels que : $0 \leqslant x \leqslant \frac{\pi}{2}$. Or il est difficile de ramener un angle quelconque compris entre 0 et $\frac{\pi}{2}$ avec une grande précision, tout simplement car nous ne disposons pas de la vraie valeur théorique du nombre π sur notre machine.
D'où la nécessité de chercher une méthode efficace qui donne avec une grande précision le modulo $\frac{\pi}{2}$ d'un angle réel quelconque.
En ce qui concerne la validation de notre fonction de réduction d'angle, on comparera en effet la différence en ulp et en valeur absolue entre $sin(x_{reduit})$ et $sin(x)$, en utilisant la fonction sinus du langage Java.

3.6.1 Premier algorithme naïf

La première idée était d'effectuer des soustractions ou bien des additions consécutives de $\frac{\pi}{2}$ sur la valeur de l'argument donné jusqu'à ce que la valeur obtenue soit contenue dans l'intervalle $[0, \frac{\pi}{2}]$.

Une fonction simple à implémenter, néanmoins elle reste non fonctionnelle surtout lorsqu'on travaille avec de très grandes valeurs. Les soustractions ou bien les additions consécutives font que le temps d'exécution s'allonge, sans oublier les imprécisions qui s'accumulent et qui résultent de l'imprécision de notre π.

3.6.2 Deuxième algorithme naïf

Une deuxième idée a été de calculer l'entier $k \in \mathbb{Z}$ tel que : $0 \leqslant x - k\frac{\pi}{2} < \frac{\pi}{2}$.
Autrement dit, on calcule d'abord k vérifiant $k = \left\lfloor \frac{x}{\frac{\pi}{2}} \right\rfloor$, puis on calcule $x - k\frac{\pi}{2}$.
L'algorithme parait correcte et sans défaut à première vue, mais en réalité il en présente deux : Premièrement l'erreur comme définie dans l'introduction explose lorsque l'angle auquel on applique la réduction d'angle est proche d'un multiple de π.
Ensuite, l'erreur explose aussi quand l'angle devient très grand.
Et ces deux défauts sont clairement illustrées dans la figure suivante :

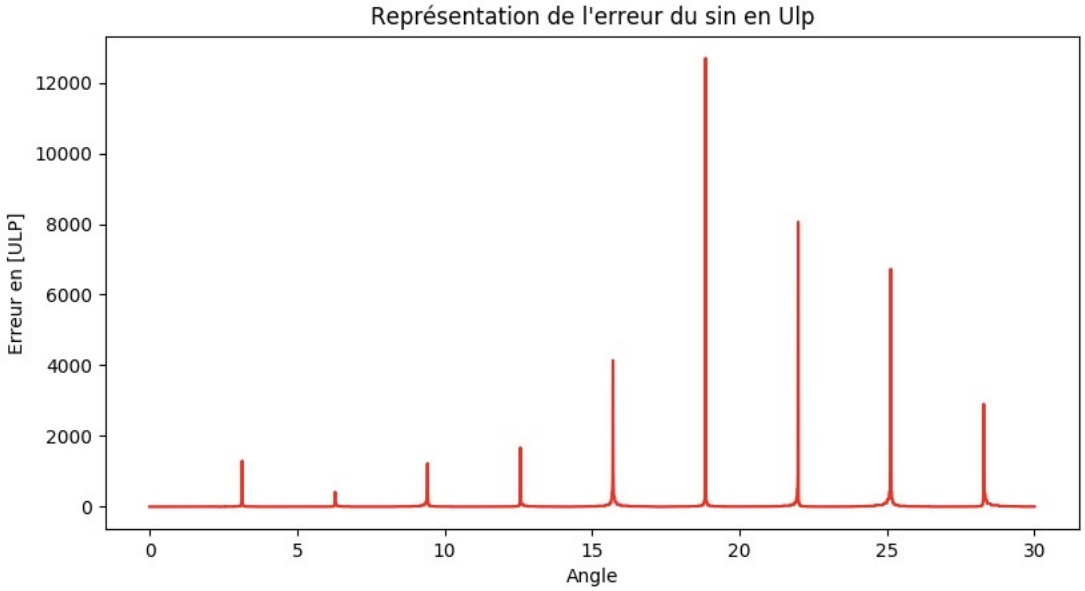

FIGURE 16 – Explosion de l'erreur en ulp au voisinage des multiples de π

Ainsi il paraît clairement que cette méthode ne peut pas être appliquée pour tout angle. Elle peut néanmoins être appliquée pour des angles "petits", et reste valable pour un grand nombre de valeur d'angle loin des valeurs critiques.

3.6.3 Méthode de cody and Wait

Cette méthode consiste tout simplement en la décomposition du nombre π en une somme de deux nombres. L'un qui est proche de π et l'autre qui représente par conséquent la différence entre π et ce dernier. Et ensuite appliquer la méthode précédente.

Cette technique élémentaire permet le gain en précision et la conservation des bits de poids faibles qui paraissent non significatives de π.
Cette méthode présente les mêmes défauts que la méthode précédente, et sont présentés dans les figures suivantes :

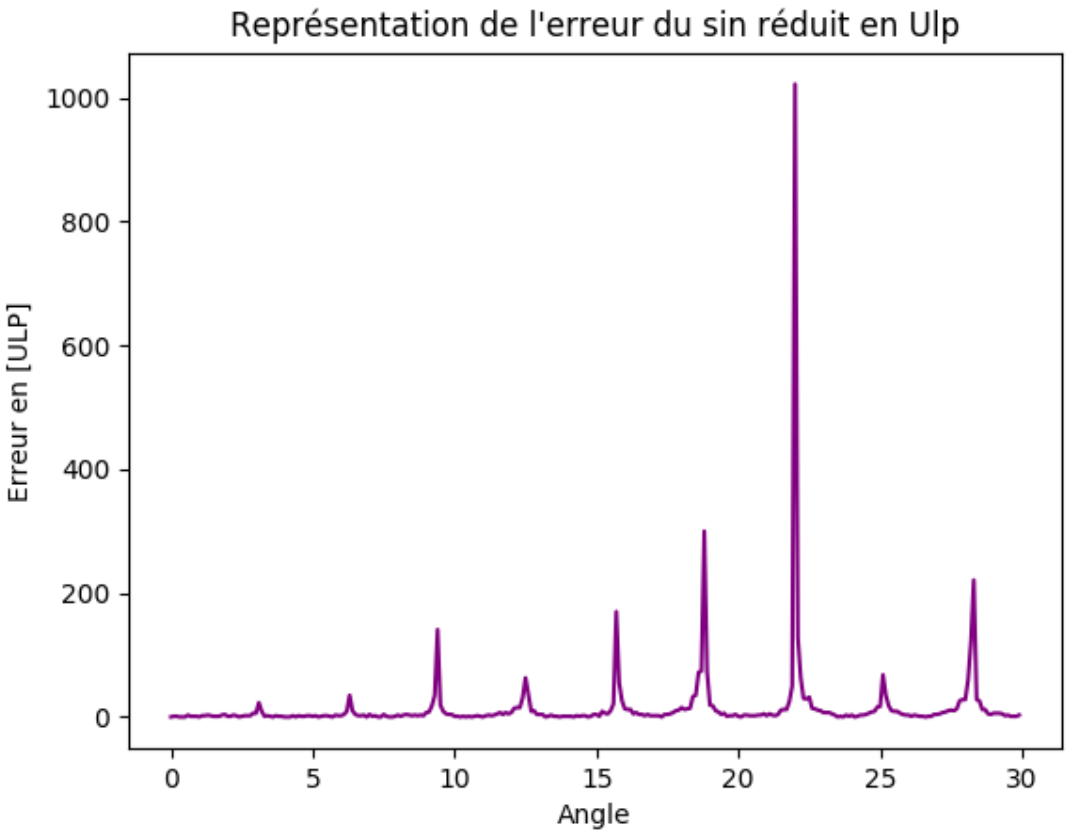

FIGURE 17 – Explosion de l'erreur en ulp au voisinage des multiples de π

Il apparaît clairement que la méthode de Cody and Wait est beaucoup plus efficace que la méthode naïve, pour les petits angles.
En effet l'erreur maximale pour la première méthode avoisine à peu près 1000 ulp, alors que pour la seconde méthode, elle dépasse les 12000 ulp!!

3.6.4 Algorithme de Payne and Hanek

Cet agorithme consiste à stocker un certain nombre de bits du nombre π, qui dépend de la précision voulue de la réduction d'angle.
Il est en effet valable pour les grandes valeurs de x vérifiant : $x \geq 2^{24}$.
L'étude de celui-ci montre que l'erreur (définie comme la différence en valeur absolue entre le cos de l'angle réduit et le cos de l'angle) maximale ne dépasse pas 4 ulp, ce qui représente une très bonne précision. Ainsi l'algorithme est très efficace, néanmoins on n'a pu l'implémenter en Deca puisqu'on avait besoin des tableaux de caractères afin de

manipuler aisément les chaînes de caractères (pour représenter π par une chaîne de bits).

4 Conclusion

Pour conclure, on peut dire qu'on a réussi à implémenter en Deca les fonctions *arctan*, *arcsin*, *sin* et *cos* sur $[0, \frac{\pi}{2}]$, avec une bonne précision égale à 5 ulp au maximum.
On a également réussi à réduire des angles de petite valeur et relativement lointaines des multiples de π avec une précision acceptable. Pour les grandes valeurs, on a pu comprendre un algorithme qui permet la réduction d'angle avec une très grande précision, mais qui ne peut malheureusement être codé en Deca car utilise la notion de tableau.
Par ailleurs l'extension TRIGO nous a permis de comprendre les subtilités rencontrées en calcul informatique et calcul flottant, et que les imprécisions dûes aux erreurs informatiques sont beaucoup plus importantes que celles d'origine mathématiques.

5 Bibliographie

Références

[1] www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2005/RR2005-09.pdf

[2] https ://www.maths-france.fr/MathSup/Cours/FormulaireTrigo.pdf

[3] maths-au-quotidien.fr/lycee/TP/heron.pdf

[4] hal.inria.fr/inria-00071477/document

[5] www.apmep.fr/IMG/pdf/cordic.pdf

[6] https ://fr.wikipedia.org/wiki/CORDIC