GRENOBLE-INP ENSIMAG

Documentation technique de l'extension TRIGO

GL53

Ayman ABOUELOULA

Kacem JEDOUI

Redouane YAGOUTI

Omar BENCHEKROUN

Nabil BENSRHIER

Janvier 2020

Table des matières

1 Introduction **2**

2 Fonctions utiles **2**
- 2.1 la fonction racine . . . 2
- 2.2 La fonction valeur absolue . . . 2
- 2.3 La fonction factorielle . . . 2
- 2.4 La fonction power . . . 3

3 Un peu de théorie **3**
- 3.1 Cordic . . . 3
- 3.2 Taylor-Lagrange . . . 4
- 3.3 Chebychev . . . 4

4 Réduction du rang : Méthode de Cody & Waite **4**

5 Implémentation des fonctions cibles **5**
- 5.1 la fonction ulp . . . 5
  - 5.1.1 Norme IEEE-754 sur 32 bits . . . 5
  - 5.1.2 Calcul de Précision . . . 5
- 5.2 la fonction sinus . . . 6
  - 5.2.1 Algorithme de Cordic . . . 6
  - 5.2.2 Algorithme de Taylor . . . 6
  - 5.2.3 Algorithme de Chebychev . . . 7
  - 5.2.4 Synthèse . . . 8
- 5.3 la fonction cosinus . . . 9
  - 5.3.1 Algorithme de Cordic . . . 9
  - 5.3.2 Algorithme de Taylor . . . 10
  - 5.3.3 Algorithme de Chebychev . . . 11
  - 5.3.4 Synthèse . . . 12
- 5.4 la fonction arctangeante . . . 13
  - 5.4.1 Algorithme de Cordic . . . 13
  - 5.4.2 Algorithme de Taylor . . . 14
  - 5.4.3 Algorithme de Chebychev . . . 14
  - 5.4.4 Synthèse . . . 15
- 5.5 la fonction arcsinus . . . 16
  - 5.5.1 Taylor . . . 16
  - 5.5.2 Synthèse . . . 17

6 Conclusion **19**

1 Introduction

L'extension TRIGO permet d'enrichir le Projet-GL, elle présente une solution pour implémenter les fonctions trigonométriques essentielles : **sinus, cosinus, arctangeante et arcsin**, un algorithme de quantification d'erreurs **ulp** a été implémenté, la condition étant d'essayer de respecter la contrainte 1 ulp de différence avec les fonctions trigonométrique dans JAVA

Le challenge de cette partie, c'est d'essayer d'être le plus précis possible, tout en respectant les particularités du langage déca : utilisation de int et de float uniquement, pour les floats supérieurs à 2π, un algorithme de réduction d'erreur est implémenté pour recadrer les valeurs dans le bon intervalle.

Cette documentation présentera les détails de notre implémentation, nos choix pour chaque fonction trigonométrique, on détaillera les limites de nos fonctions et on présentera des solutions d'optimisations.

2 Fonctions utiles

Les algorithmes choisis, demandent des fonctions intermédiaires qu'on détaillera ici.

2.1 la fonction racine

On a implémenté la fonction qui calcul la racine carrée d'un nombre réel x quelconque par la limite de la suite définie par :

$$x_{n+1} = \frac{1}{2}(x_n + \frac{x}{x_n})$$

Avec $x_0 = x$. On trouve que la suite x_n tend vers x, pour notre implémentions on choisit de s'arrêter à $n = 100$.

2.2 La fonction valeur absolue

On a implémenté également la fonction valeur absolue d'un réel par l'algorithme suivant

```
float abs(x):
        if x > 0:
            return x
        else :
            return -x
```

2.3 La fonction factorielle

On calcule de manière récursive la fonction factoriel d'un entier naturel positif par l'algorithme suivant :

```
int factoriel(n):
        if n == 0:
            return 1
        else:
            return n * factoriel(n - 1)
```

2.4 La fonction power

On a implémenté aussi la fonction qui calcule la puissance d'un nombre réel par l'algorithme suivant :

```
float power(value1, value2):
        counter = 0
        result = 1
        if value2 < 0
            return power(1/value1, -value2)
        else:
            while (value2 > counter):
                result = result * value1
                counter = counter + 1;
            return result
```

3 Un peu de théorie

On présentera ici les détails sur la théorie derrière les algorithmes utilisés, en exposant les avantages que proposent mais aussi leurs limitation, les détails d'intégration seront présentés dans la section.5

3.1 Cordic

L'algorithme de Cordic (COordinate Rotation DIgital Computer) diffère de l'algorithme de Taylor qui utilise un développment limité pour établir les fonctions trigonométriques, en effet, il permet d'exploiter les spécificités trigonométriques de ces fonctions, et les relations qui les relies.

Cordic, developpé par Jack Volder en 1956, permet d'approcher un point $\mathbf{M} = cos(\theta) + sin(\theta)$ par un point $\mathbf{N}$ appartenant au cerle unité $\mathbf{I} = (0, 1)$, qui va subir une successions de rotations, de plus en plus petites.

le sens de rotation de ce point dépends de sa positions par rapport au point M, si $\mathbf{N} < \mathbf{M}$ la rotation sera dans le sens direct trigonométrique ou le sens anti-horaire sinon dans le sens indirect, la première rotation est toujours dans le sens trigonométrique car $\mathbf{I}$ commence en $(0, 1)$

On note γ_i l'angle correspondant à la $(i+1)^{\text{ème}}$ itération, on commence avec $i = 0$ pour alléger les expressions, on suppose que les angles sont positifs, on note $s_i = \pm 1$, s est positif s'il s'agit du sens trigonométrique sinon il est négatif.

On note $\mathbf{N_n}, n \in \mathbf{N}$ la suite des points abtenus après $(n+1)$ orientations, et θ_n l'angle formé par $\mathbf{N}_n$.

$$\forall n \in \mathbf{N}, \theta_n = \sum_{i=0}^{n} s_i \gamma_i$$

. La $n + 1^{\text{ème}}$ rotation, du point $\mathbf{N_n}$, de coordonnés $(\cos\theta_n, \sin\theta_n)$ fourni une approximation de $\cos et \sin$.

Définition : On appelle *rotation d'angle* γ, l'application linéaire du corp des complexes $\mathbf{C}$, $rot_\gamma : \mathbf{C} \to \mathbf{C}$ définie par $u \to \exp^{i\gamma} u$, en identifiant $\mathbf{C}$ à $\mathbf{R} \times \mathbf{R}$, et en notant $u = (x, y)$, on obtient la matrice suivante.

$$\begin{pmatrix} x \\ y \end{pmatrix} \xrightarrow{rot_\gamma} \begin{pmatrix} \cos\gamma & -\sin\gamma \\ \sin\gamma & \cos\gamma \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{1}$$

A partir de cette matrice et en utilisant les formules trigonométriques, on peut élargir la matrice et les calculs pour pouvoir calculer les fonctions trigonométriques, on détaillera les algorithmes correspondants pour le calcul de chaque fonction dans la section 5

3.2 Taylor-Lagrange

Définition : Soit n un entier. Soit f une fonction $\mathbb{R}$ dans $\mathbb{R}$, définie sur un intervalle ouvert I contenant un point a, dérivable $n1$ fois sur I, et dont la dérivée $n^{ème}$ en a existe. On appelle polynôme de Taylor d'ordre n en a de f, le polynôme

$$P(x) = \sum_{i=0}^{n} f^{(i)}(a) \frac{(x-a)^i}{i!} \tag{2}$$

On appelle le reste de Taylor d'ordre n de f la fonction R_n qui à $x \in I$ associe :

$$R_n(x) = P_n(x) - f(x) \tag{3}$$

Théorème : Soient I un intervalle ouvert contenant 0, et n un entier. Soit f une fonction dérivable $n1$ fois sur I, et dont la dérivée $n^{ème}$ en 0 existe. Soit R_n son reste de Taylor d'ordre n en 0. Au voisinage de 0, R_n est négligeable devant x^n :

$$R_n(x) = o(x^n) \tag{4}$$

Dans notre cas les fonctions sin, cos, *arctan* et *arcsin* sont infiniment dérivables, donc le développement de Taylor est applicable.

3.3 Chebychev

Théorème de Weierstrass :

Soit f un fonction continue sur $[a, b]$. Pour tout $\epsilon > 0$ il existe une fonction polynomiale p tel que :

$$\|p - f\|_\infty < \epsilon$$

Le théorème de Chebychev consiste à interpoler le polynôme p précédent, les valeurs de cette interpolation ont été prises directement des références.

4 Réduction du rang : Méthode de Cody & Waite

La méthode présentée par Cody & Waite consiste à trouver deux valeurs C_1 et C_2 pour approcher un nombre C tel que C_1 soit très proche de C et qui n'a pas trop de nombre après la virgule, et que C_2 est la différence entre C et C_1. Ainsi on évalue :

$$x - kC_1 - RN(kC_2) \tag{5}$$

Pour $C = \pi$ (dans la norme IEEE-754 sur 32 bits) on utilise les coefficients suivants :

$$C_1 = \frac{201}{64} = 3,140625$$

$$C_2 = 9,67653589793.10^{-4}$$

5 Implémentation des fonctions cibles

5.1 la fonction ulp

5.1.1 Norme IEEE-754 sur 32 bits

En Deca les flottants sont codés sous la norme IEEE-754 sur 32 bits. Ces 32 bits sont stockés de cette façon : Un flottant x est ainsi représenté par :

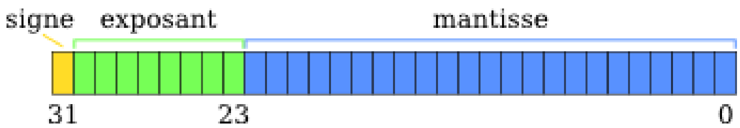

FIGURE 1 – Stockage d'un Flottant sur 32 bits

$$x = s \times 2^e \times m$$

Avec :

- $s \in \{1, -1\}$: représente le signe de x
- e : l'exposant avant son décalage de 127
- m : $1 + mantisse$ représente la partie significative

Exemple : 0b 0 01111100 01000000000000000000000
01111100 représente $124 - 127 = -3$ et la partie significative est 1,01 (en binaire) qui est égale à $1,25$ en décimal, ainsi le nombre représenté est $+1,25 \times 2^{-3}$

5.1.2 Calcul de Précision

Unit of Least Precision est une fonction qui donne pour un réel x la différence entre son flottant est entre le flottant le plus proche (qui soit supérieur à x). Il existe plusieurs façon de calculer cette fonction, celle qu'on a utilisé se base sur la formule explicite de l'ulp :

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

Avec $p - 1 = 23$ et k est l'exposant de la plus grande puissance de 2 inférieure à x, ainsi notre implémentation se base sur le fait de trouver k tel que $2^k \leq x < 2^{k+1}$.

On prendra en considération les cas particulier de la fonction ulp :

- Le plus grand nombre est $2^{128} - 2^{104}$
- Le plus petit flottant positif est 2^{-149}

Pour calculer la précision d'une approximation $f(x)$ par rapport à une valeur réelle $f_{real}(x)$ on utilisera l'erreur suivante :

$$erreur = \frac{|f(x) - f_{real}(x)|}{ulp(f_{real}(x))} \tag{6}$$

5.2 la fonction sinus

5.2.1 Algorithme de Cordic

Pour calculer la fonction sinus, on a besoin d'étendre l'équation 1 pour l'appliquer à cette foncion. en utilisant la formule $tan(\theta) = \frac{\sin\theta}{\cos(\theta}$, on peut factorise par cos, on obtient alors

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \cos(\gamma_i) \begin{pmatrix} 1 & -s_i \tan\gamma_i \\ -s_i \tan\gamma & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{7}$$

on peut maintenant remplacer l'expression avec arctan

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \cos(\arctan(2^{-i}) \begin{pmatrix} 1 & -s_i 2^{-i} \\ -s_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{8}$$

En posant (x,y)=(1,0) et $\theta = \sum s_i \arctan(2^{-i})$, on peut généraliser l'équation pour exprimer tout angle θ.

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \prod_{i=0}^{\infty} \cos(\arctan(2^{-i}) \begin{pmatrix} 1 & -s_i 2^{-i} \\ -s_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{9}$$

On pose K $= \prod_{i=0}^{\infty} \cos(\arctan(2^{-i}))$ et $\lim\limits_{i\to+\infty} x_i = \cos\theta$ et $\lim\limits_{i\to+\infty} y_i = \sin\theta$, On obtient donc

$$\boxed{\left\{ \begin{array}{r} x_0 = K; y_0 = 0; z_0 = \theta \\ d = signe(z_i) \\ x_{i+1} = x_i + s_i y_i 2^{-i} \\ y_{i+1} = y_i + s_i x_i 2^{-i} \\ z_i + 1 = z_i - \arctan(2^{-i}) \end{array} \right.} \tag{10}$$

On a choisi de prendre K = 0.60725293510314F ; et 32 itérations, on stoque les valeurs de z_i dans un tableau qu'on note lookup[i], on aura 32 valeurs de $\arctan(2^{-i})$ stockées.

```
float sinCordic(float theta):
    x = K,y = 0,z = theta ,v = 1.0f
    pour i = 0; i < iterations; i++ :
        d = (z >= 0) ? +1 : -1;
        x = x - d * y * v;
        y = y + d * x * v;
        z = z - d * lookup[i];
        v *= 1/2;
    return y;
```

Pour cette méthode on trouve des erreurs en Ulp de l'ordre de 10^7, c'est donc une approximation à éviter.

5.2.2 Algorithme de Taylor

L'approximation de Sinus par Taylor est donnée par l'algorithme suivant :

```
float coefTaylorSinus(int n, float x):
    return power(-1, n) * power(x, 2*n + 1)/factoriel(2*n+1)

float taylorSinus(float x, int n):
```

```
s = 0;
for (int i = 0; i < n; i++):
    s += coefTaylorSinus(i, x)
return s
```

Ce qui est important pour cette méthode c'est qu'elle donne de très bonnes approximations au voisinage de 0 (pas ailleurs) de la fonction Sinus en terme d'erreur en Ulp (Voir Figure 2)

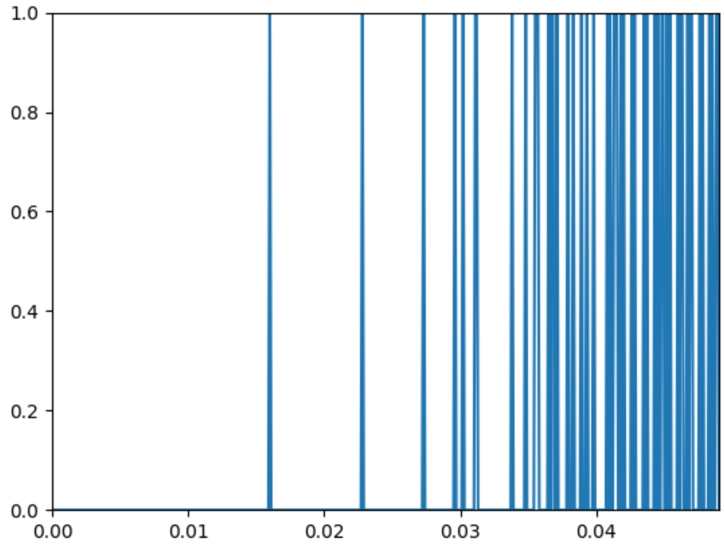

FIGURE 2 – Erreur en Ulp de Sin au voisinage de 0 (Taylor)

5.2.3 Algorithme de Chebychev

Première Étape : On calcule la fonction sin sur l'intervalle $[\frac{-1}{32}, \frac{1}{32}]$ par l'approximation polynomiale suivante :

$$sin(r) = r + a_1 r^3 + a_2 r^5 + a_3 r^7 + a_4 r^9 \tag{11}$$

L'implémentation de cet étape a été faite via l'algorithme suivant :

```
float ChebychevSinus(r):
    a1 = -0.16666666666666666666421f
    a2 = 0.008333333333333312907f
    a3 = -0.0001984126983563939f
    a4 = 0.00000275566861f

    return r+a1*power(r,3)+a2*power(r,5)+a3*power(r,7)+a4*power(r,9)
```

Deuxième Étape : Pour l'instant on sait calculer Sin sur l'intervalle $[\frac{-1}{32}, \frac{1}{32}]$, on veut étendre ce calcul à l'intervalle $[0, \frac{\pi}{4}]$. Pour ceci il existe des coefficients appelés **Breakpoints** qui sont de la forme $c_{jk} = 2^{-j} \times (1 + \frac{k}{8})$ avec $j \in \{1, 2, 3, 4\}$ et $k \in \{0, 1, 2..., 7\}$.

On pose pour la suite $r = x - c_{jk}$ avec $|r| \leq \frac{1}{32}$ et $x \leq \frac{\pi}{4}$. On retrouve $sin(x)$ à travers la formule suivante :

$$sin(x) = sin(c_{jk})cos(r) + cos(c_{jk})sin(r) \tag{12}$$

Les c_{jk} étant des constantes, on les stocke dans des variables en utilisant les fonction sinus et cosinus de la bibliothèque "Math" de Java. En ce qui concerne $cos(r)$, on verra qu'il sera calculé par la même façon sur l'intervalle $[\frac{-1}{32}, \frac{1}{32}]$ indépendamment de Sin.

Troisième Étape : Cette étape consiste à prolonger le calcul sur $[0, \frac{\pi}{2}]$ pour pouvoir ensuite calculer le sinus de n'importe quel flottant par la réduction du rang. La formule utilisée ici est $sin(2 \times x) = 2 \times cos(x) \times sin(x)$

5.2.4 Synthèse

Pour la méthode de Chebychev, on remarque qu'au voisinage de 0, il y a plus d'erreur en Ulp que Taylor (voir figure 3), ainsi on va préférer d'utiliser Taylor au voisinage de 0, et Chebychev ailleurs.

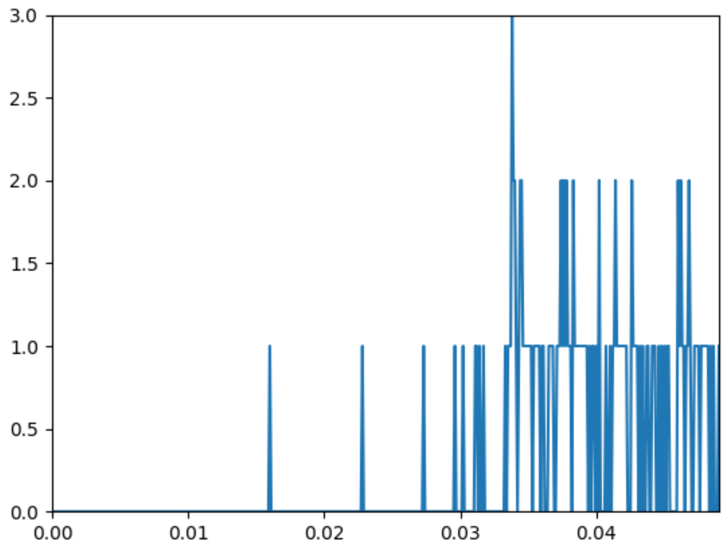

Figure 3 – Précision Ulp au voisinage de 0 pour Chebychev

En combinant ces deux méthodes, on trouve la représentation de Sinus (voir figures 4 et 5) qui est bien proche de celle de Java.

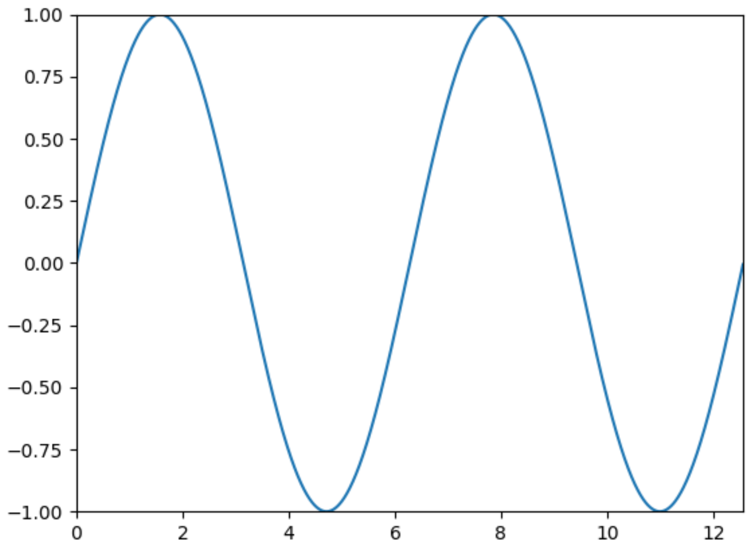

FIGURE 4 – Sinus entre 0 et $\frac{\pi}{4}$

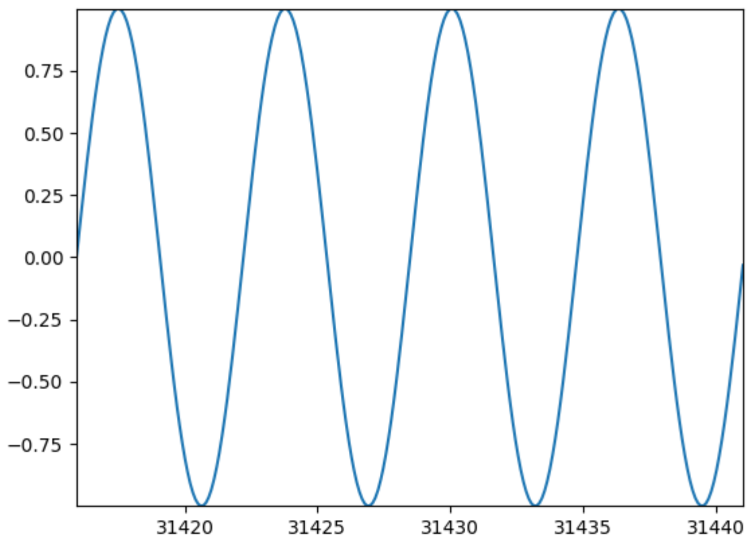

FIGURE 5 – Sinus pour un grand nombre

5.3 la fonction cosinus

5.3.1 Algorithme de Cordic

Pour la fonction cos, on utilise le même principe que la fonction sin, la seule différence est la valeur retournée

```
float cosCordic(float theta):
    x = K,y = 0,z = theta,v = 1.0f
    pour i = 0; i < iterations; i++:
        d = (z >= 0) ? +1 : -1;
        x = x - d * y * v;
        y = y + d * x * v;
        z = z - d * lookup[i];
        v *= 1/2
    return x
```

On remarque des erreurs en Ulp qui ne sont pas très satisfaisantes (voir figure **??**

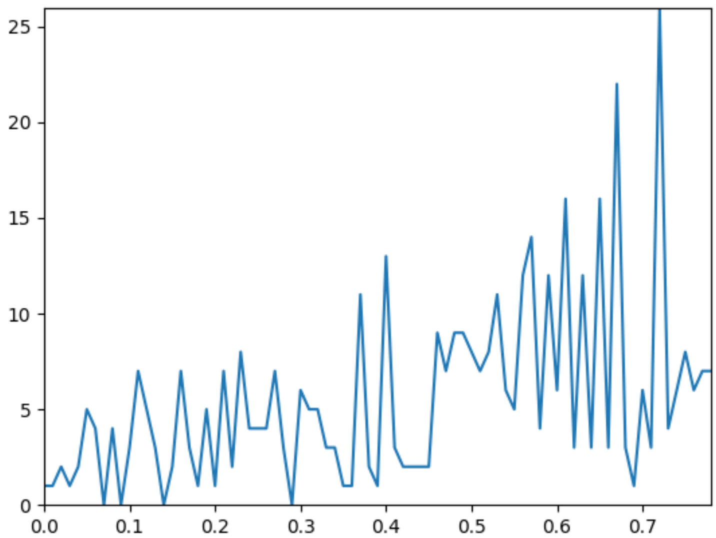

FIGURE 6 – Ulp de Cos entre 0 et $\frac{\pi}{4}$ (Cordic)

5.3.2 Algorithme de Taylor

L'approximation de Taylor de la fonction cosinus est intéressante au voisinage de $\frac{\pi}{2}$, elle est donnée par l'algorithme suivant :

```
float coefTaylorCosinus(int n, float x):
    return power(-1, n) * power(x, 2*n)/factoriel(2*n)

float taylorCosinus(float x, int n):
    s = 0;
    for (int i = 0; i < n; i++):
        s = s + coefTaylorCosinus(i, x)
    return s
```

On remarque qu'au voisinage de $\frac{\pi}{2}$ il y a moins d'erreur en Ulp (voir figure 7), ainsi on a tendance à utiliser cette méthode au voisinage de $\frac{\pi}{2}$.

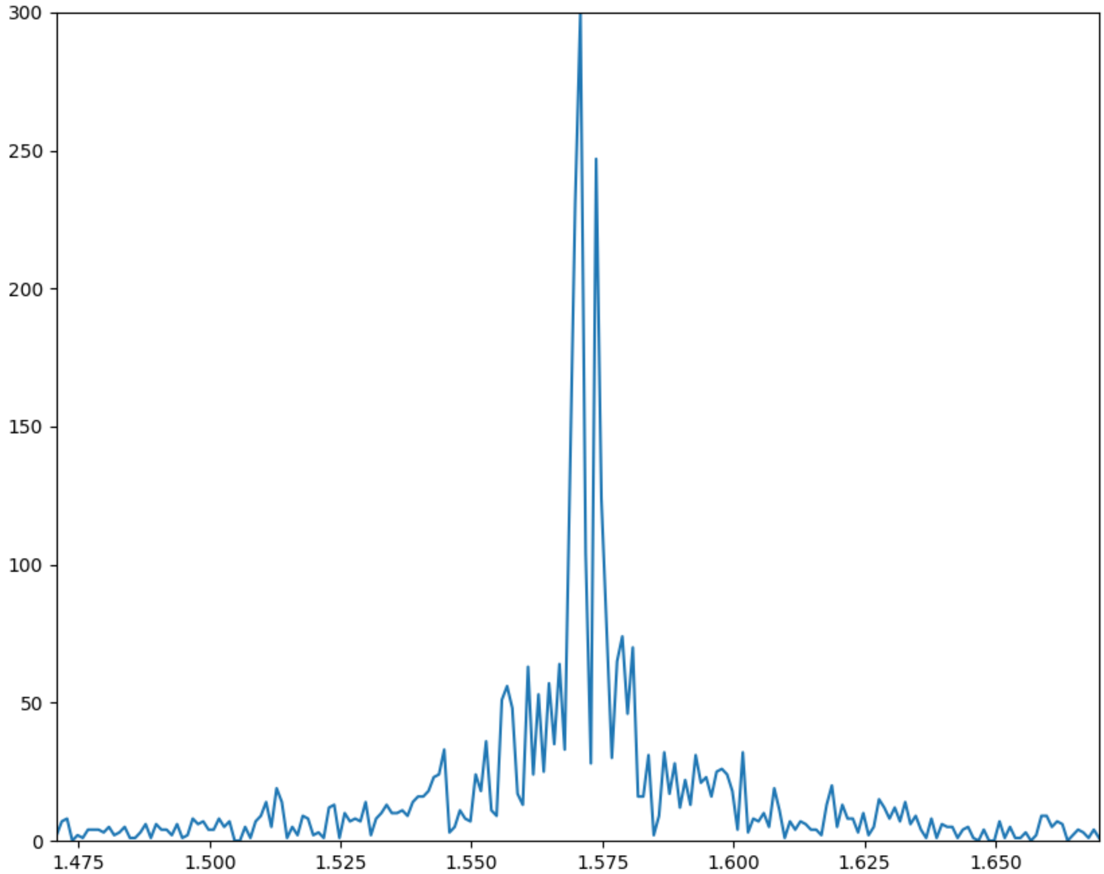

FIGURE 7 – Ulp de Cos au voisinage de $\frac{\pi}{2}$ (Taylor)

5.3.3 Algorithme de Chebychev

Première Étape : On calcule la fonction *cos* sur l'intervalle $[\frac{-1}{32}, \frac{1}{32}]$ par l'approximation polynomiale suivante :

$$cos(r) = 1 + a_1r^2 + a_2r^4 + a_3r^6 + a_4r^8 + a_5r^{10} \tag{13}$$

L'implémentation de cet étape a été faite via l'algorithme suivant :

```
float ChebychevCosinus(r)
  a1 = -0.4999999999999999999999999995425696f
  a2 = 0.04166666666666666666660027496f
  a3 = -0.00138888888888885759632f
  a4 = 0.0000248015872951505636f
  a5 = -0.000000275567182072f
```

```
return 1+a1*power(r,2)+a2*power(r,4)+
a3*power(r,6)+a4*power(r,8)+a5*power(r,10)
```

Deuxième Étape : Pour l'instant on sait calculer Cos sur l'intervalle $[\frac{-1}{32}, \frac{1}{32}]$, on veut étendre ce calcul à l'intervalle $[0, \frac{\pi}{4}]$. Pour ceci on utilise les coefficients appelés **Breakpoints** évoqués précedemment.

On pose pour la suite $r = x - c_{jk}$ avec $|r| \leq \frac{1}{32}$ et $x \leq \frac{\pi}{4}$. On retrouve $cos(x)$ à travers la formule suivante :

$$cos(x) = cos(c_{jk})cos(r) - sin(c_{jk})sin(r) \quad (14)$$

Les c_{jk} étant des constantes, on les stocke dans des variables en utilisant les fonction sinus et cosinus de la bibliothèque "Math" de Java. En ce qui concerne $sin(r)$, il a été déjà calculé par la même façon sur l'intervalle $[\frac{-1}{32}, \frac{1}{32}]$ indépendamment de Cos.

Troisième Étape : Cette étape consiste à prolonger le calcul sur $[0, \frac{\pi}{2}]$ pour pouvoir ensuite calculer le sinus de n'importe quel flottant par la réduction du rang. La formule utilisée ici est $cos(2 \times x) = 1 - 2 \times sin(x)^2$

5.3.4 Synthèse

On remarque que la précision en Ulp de Cos par l'approximation polynomiale est donne des résultats plus satisfaisant (à l'exception du voisinage de $\frac{\pi}{2}$) avec des ulp au maximum égale à 2. Figure 8

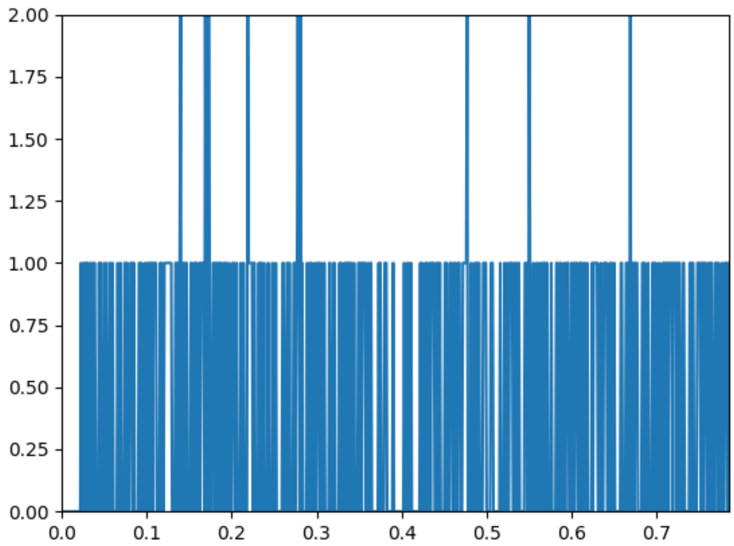

FIGURE 8 – Ulp de Cos au voisinage de $\frac{\pi}{2}$ (Chebychev)

Si on trace la fonction Cos entre 0 et 3π on remarque des valeurs proches de la fonctions réelle (voir figure 9). Par contre on remarque des valeurs aberrantes dans quelques multiples de π qu'on n'a pas pu gérer.

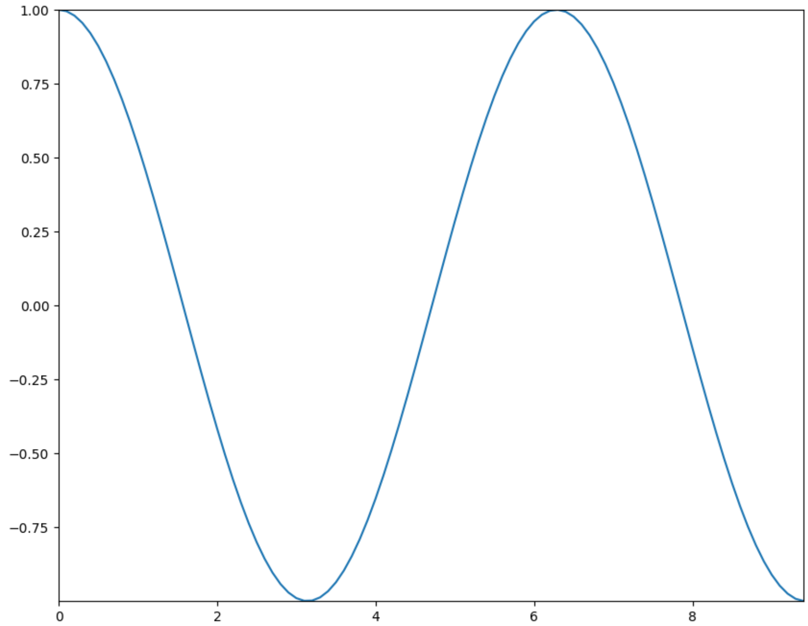

FIGURE 9 – Cos entre 0 et 3π

5.4 la fonction arctangeante

5.4.1 Algorithme de Cordic

L'implémentation de la fonction arctangeante avec Cordic, est en effet très facile, en jouant sur les paramètres on obtient une valeur approchée de arctan $\frac{y_0}{x_0}$

$$\boxed{\begin{cases} x_0 = x_0; y_0 = y_0; z_0 = 0 \\ d = signe(z_i) \\ x_{i+1} = x_i + s_i y_i 2^{-i} \\ y_{i+1} = y_i + s_i x_i 2^{-i} \\ z_i + 1 = z_i - \arctan(2^{-i}) \end{cases}} \tag{15}$$

```
float atanCordic(x0, y0):
    x = x0,y = y0,z = 0,v = 1.0f
    pour (i = 0; i < 32; i++):
        d = (z >= 0) ? +1 : −1
        x = x − d ∗ y ∗ v
        y = y + d ∗ x ∗ v
        z = z − d ∗ lookup[i]
        v = v / 2
    return z
```

Pour cette approximation, on remarque des erreurs en Ulp qui ne sont pas très grandes (figure 10). On peut améliorer cette précision en utilisant une approximation polynomiale qu'on verra par la suite.

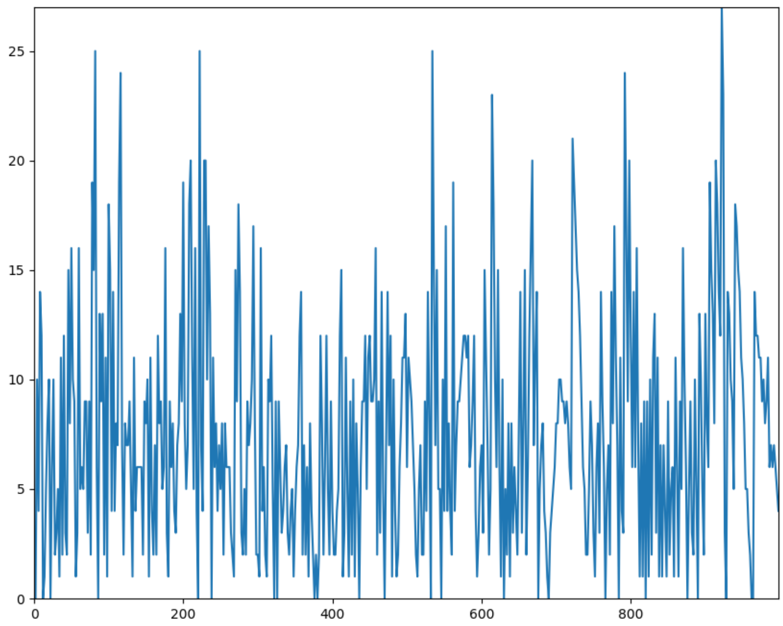

Figure 10 – Précision en Ulp entre 0 et 1000 de Atan

5.4.2 Algorithme de Taylor

L'approximation de l'Arctan par Taylor se fait par l'algorithme suivant :

```
float coefTaylorArctan(int n, float x):
    return power(-1, n) * power(x, 2n + 1)/(2n + 1)

float taylorArctan(float x, int n):
    s = 0;
    for (int i = 0; i < n; i++):
        s = s + coefTaylorArctan(i, x)
    return s
```

En terme de précision au voisinage de 0 on remarque que Taylor donne des résultat satisfaisants comme sur la figure 11, c'est la méthode qu'on utilisera au voisinage de 0.

5.4.3 Algorithme de Chebychev

Première Etape : Dans cette section l'approximation de l'Arctan se fait en utilisant une fraction polynomiale sous la forme $\frac{P(x)}{Q(x)}$ avec les coefficients suivants avec c_i les coefficients de P et les q_i les coefficients de Q :

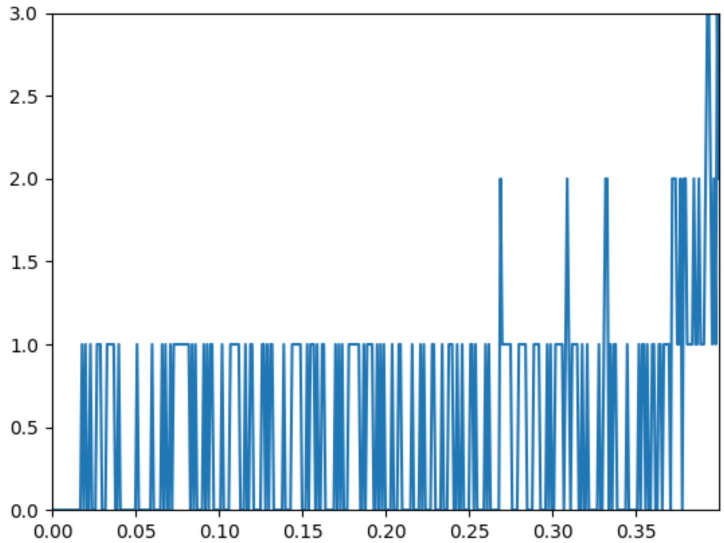

FIGURE 11 – Précision en Ulp de Arctan au voisinage de 0 (Taylor)

```
c1 = 137.77239855699296707280354 5 f;
c2 = 311.60920898000609003552488 12 f;
c3 = 243.26579338982082642067336 48 f;
c4 = 76.11794481216112334509648 66 f;
c5 = 8.13270595820026244909963 98 f;
c6 = 0.13420779282300591906970 38 f;
q1 = 137.77239855699296707280354 93 f;
q2 = 357.53334183233707905977774 92 f;
q3 = 334.88909428920125936798827 67 f;
q4 = 135.92274503357668110432440 01 f;
q5 = 22.23061618444266557469256 796 f;
```

Dans cette phase on remarque que la précision au voisinage de 0 est moins performante par rapport à Taylor (voir figure 7)

Deuxième Etape : On étend le calcul de la fonction Arctan sur R tout entier par la formule :

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

On constate que l'approximation de Chebychev apporte une très grande précision même pour des nombres qui sont très grands comme dans la figure 13.

5.4.4 Synthèse

La conclusion qu'on peut faire c'est d'utiliser la méthode de Taylor au voisinage de 0, et la méthode de Chebychev ailleurs, on obtient alors le graphe inscrit dans la figure 14

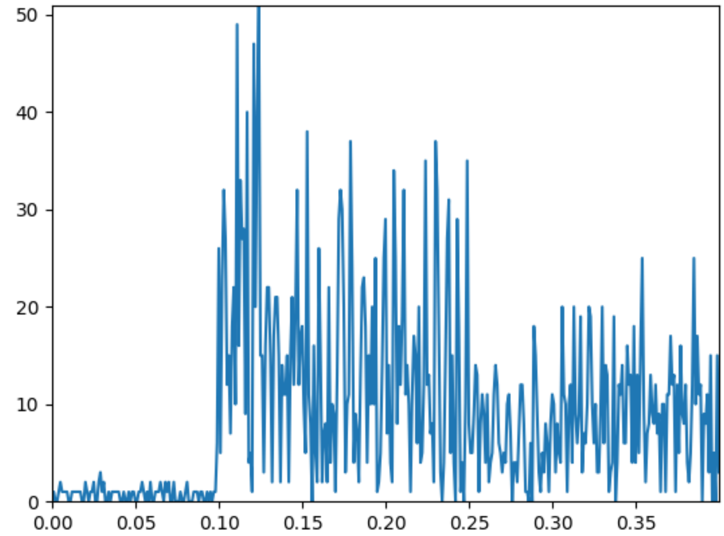

Figure 12 – Erreur en Ulp de Atan au voisinage de 0 (Chebychev)

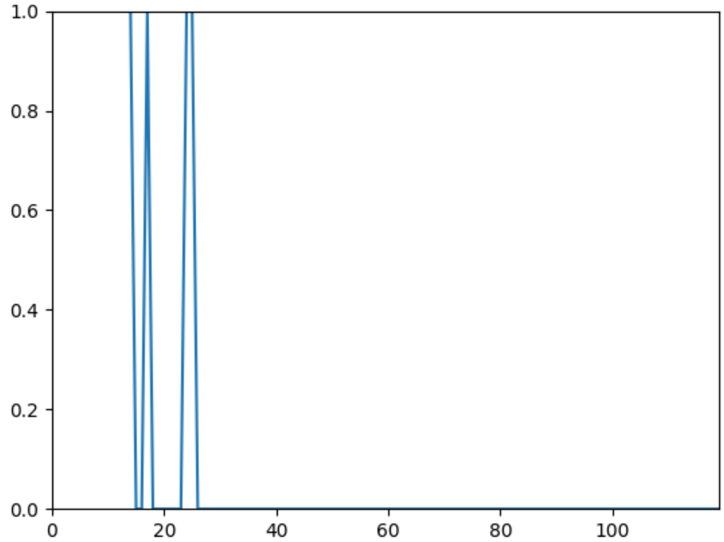

Figure 13 – Précision Ulp de Arctan pour les puissances de 2

5.5 la fonction arcsinus

5.5.1 Taylor

L'algorithme implémenté pour le calcul de l'arcsin en utilisant la méthode de Taylor est ainsi :

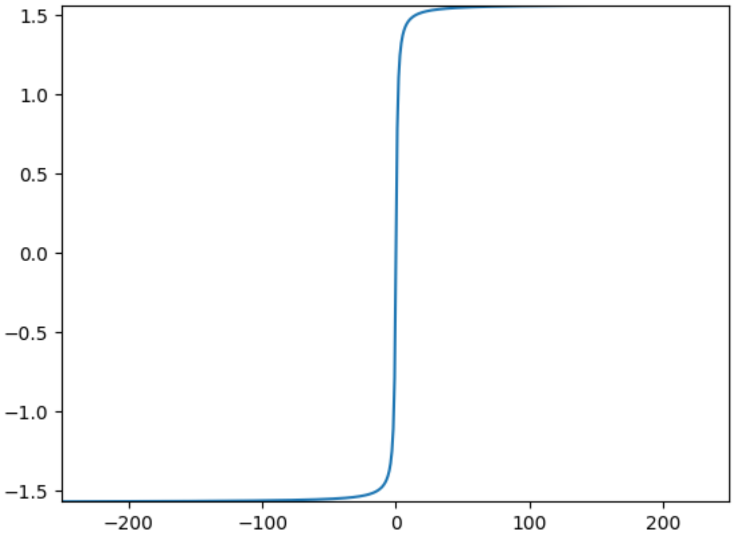

FIGURE 14 – Stockage d'un Flottant sur 32 bits

```
float coefTaylorArcsin(int n, float x):
    return power(-1,n)*power(x,2n+1)*factoriel(2n)
    /((2n+1)*power(2,2n)*power(factoriel(n),2))

float taylorArcsin(float x, int n):
    s = 0
    for (int i = 0; i < n; i++):
        s = s + coefTaylorArcsin(i, x)
    return s
```

Mais on remarque que le développement limité de Taylor donne de grandes erreurs en Ulp comme on peut le constater sur la figure 15

5.5.2 Synthèse

Pour implémenter la fonction arcsin, il suffit de se référer à cette égalitée trigonométrique.

$$arcsin(x) = 2 \times arctan(\frac{x}{1 + \sqrt{1 - x^2}} \tag{16}$$

On se sert alors de la fonction d'Arctan calculée par l'approximation polynomiale de Chebychev et par le développement limité de Taylor (au voisinage de Zéro), c'est la plus bonne approximation que l'on peut faire dans notre cas vu que la fonction **Arctan** donne un nombre très satisfaisant d'erreurs en Ulp, et que la fonction **racine** est très proche de celle de Java.

Il est bien clair que cette seconde méthode porte beaucoup moins d'erreur que celle de Taylor comme vous le voyez sur la figure 16.

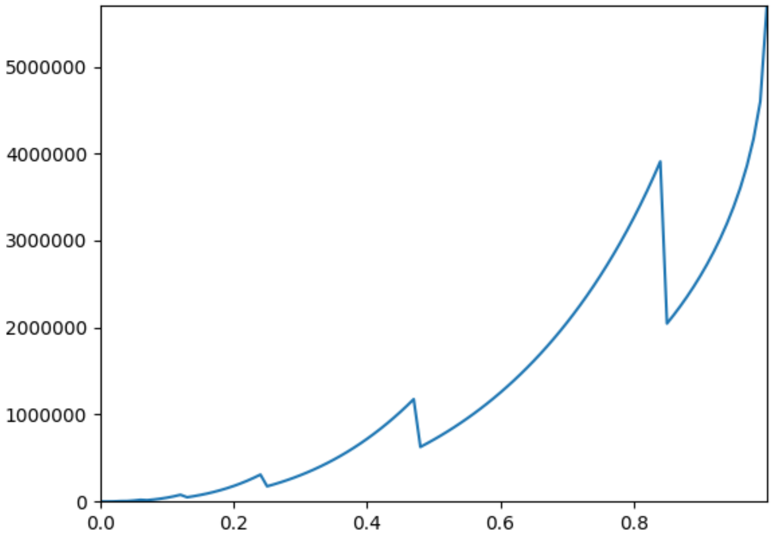

FIGURE 15 – Erreur en Ulp de Arcsin entre -1 et 1

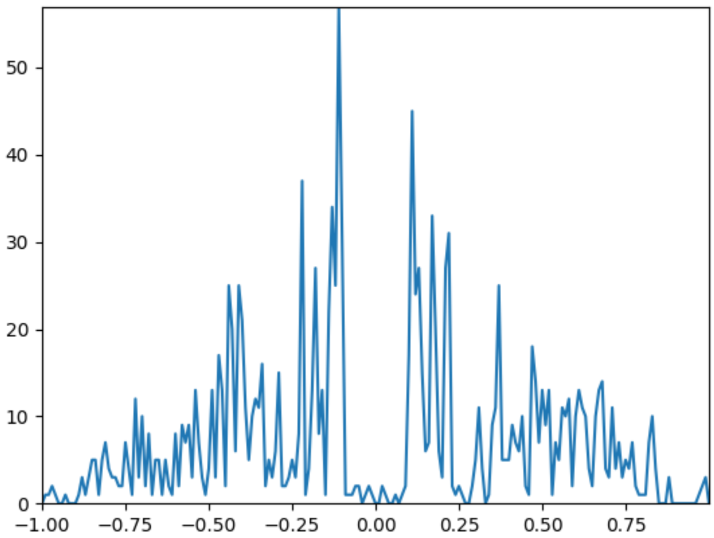

FIGURE 16 – Erreur en Ulp de Asin entre -1 et 1 par Chebychev

On remarque sur la figure 17 que le tracé de la fonction asin est très proche de celui de la réalité.

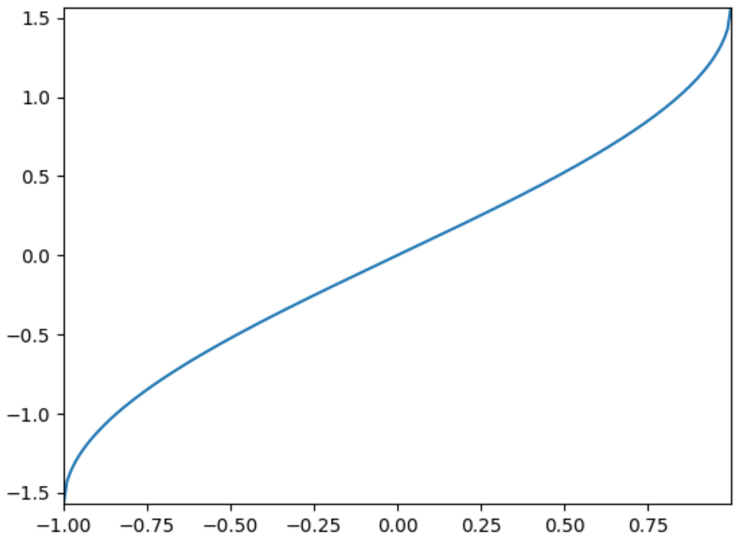

Figure 17 – Asin entre -1 et 1

6 Conclusion

On conclut que pour toutes les fonctions qu'on a implémenté, la méthode de Cordic n'apportait pas des valeurs très précises, le mieux était donc de combiner entre l'approximation de Chebychev et le développement limité de Taylor dans les points critiques. Les résultats était satisfaisants pour Arctan, Sinus et Arcsin, et un peu moins pour Cosinus. On pourrait améliorer encore la réduction du rang en utilisant la méthode de Payne & Hanek, mais elle paraît un peu difficile à implémenter en Deca vu qu'elle utilise des tableau de grande taille.

Références

[1] Jean Michel Muller. Elementary Functions Algorithms and Implementation.

[2] Christophe Devalland L'ALGORITHME CORDIC", https ://www.apmep.fr/IMG/pdf/cordic.pdf

[3] trigonometry For Enthusiasts : The CORDIC Algorithm,https ://en.wikibooks.org/wiki/Trigonometry/For-Enthusiasts/The-CORDIC-Algorithm.

[4]JP. Zanotti,ALGORITHME CORDIC,http ://zanotti.univ-tln.fr/ALGO/I31/Cordic.html

[5] J.F.Hart, Computer Approximations.

[6] Jean Michel Muller, ACM Transactions on Mathematical Software.