

Extension Trigo

Equipe 46 : -Ait Taleb Aymane
-Boussemid Youssef
-Hassou Karim
- Jaoudar Reda
-Bendou Safouane

Janvier 2022



Contents

1	Introduction	3
2	Autres algorithmes	3
2.1	Méthode de Taylor	3
2.2	Méthode de Chebyshev	5
2.3	Méthode Cordic	5
3	CORDIC : fonctions trigonométriques	7
3.1	Principe de l'algorithme : cosinus, sinus	7
3.2	Principe de l'algorithme : Arcsin, Arccos, Arc- tan	10
3.2.1	Arcsin et Arcos	10
3.2.2	Arctan	12
3.3	Implémentation	12
4	Précision	16
5	Bibliographie	20

1 Introduction

L'extension choisie étant TRIGO, il s'agit à cet égard d'implémenter les fonctions trigonométriques par le biais des divers algorithmes et méthodes connus, soit par la rotation des vecteurs dans le cercle trigonométrique (CORDIC), ou par le biais des coefficients de Tchebychev. Notre groupe choisi d'opter pour l'algorithme CORDIC, pour des raisons qui seront spécifiées dans les parties ci-dessous, où l'on détaillera le principe de la méthode tout en le comparant à la méthode polynomiale. L'obtention d'une valeur précise et correcte à l'issue des fonctions trigonométriques demeure un aspect d'immense importance. En effet, une erreur en ULP pourrait causer de grandes divergences, ce qui trouble la précision des fonctions trigonométriques ainsi implémentées.

2 Autres algorithmes

2.1 Méthode de Taylor

Les fonctions trigonométriques peuvent être implémentées par le biais de la méthode de Taylor ; en effet, il s'agit de ramener les différentes fonctions à l'écriture d'un polynôme dit de Taylor. Une écriture semblable à celle du développement limité. Ci-dessous, nous présentons l'écriture polynomiale de chaque fonction trigonométrique, ainsi que la méthode d'implémentation dans un langage de programmation de

haut niveau.

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

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

L'implémentation des deux fonctions consiste à réaliser des boucles faisant usage d'un très grand nombre d'itérations, en divisant à chaque étape par le factoriel de l'indice selon la formule plus haut. Une meilleure approche serait de sauvegarder les valeurs générées par le factoriels calculées pour minimiser les calculs, ainsi que la réduction des conditions d'intervalles.

Quant à l'inverse de ces fonctions trigonométriques, la formule est de suite :

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

$$\arccos x = \pi/2 - \arcsin x$$

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

et

$$\arctan x = \pi/2 - \arctan\left(\frac{1}{x}\right)$$

L'avantage de cette écriture polynomiale est la simplicité de ces formules, calculant ainsi des fonctions aussi complexes que les fonctions trigonométriques. D'ailleurs, la dérivation à chaque étape est de plus en plus coûteuse

après augmentation de nombres d’itérations. Le temps de résolution dès lors augmente à chaque reprise.

2.2 Méthode de Chebyshev

Les séquences polynomiales de Chebyshev permettent d’exhiber une approximation à l’aide des fonctions trigonométriques. Une séquence de Chebyshev ne donne pas une approximation de la fonction donnée en soit, mais, la précision n’est pas réduite en augmentant le nombre d’itérations. L’identité trigonométrique s’écrit :

$$T_n(x) = \cos(n \arccos(x))$$

avec

$$\int_{-\pi/2}^{\pi/2} \cos(\cos x \cos nx) dx = \int_{-1}^1 \frac{\cos(x) T_n(x)}{\sqrt{1-x^2}} dx$$

et

$$U_n(\cos \theta) = \frac{\sin(n+1)\theta}{\sin \theta}$$

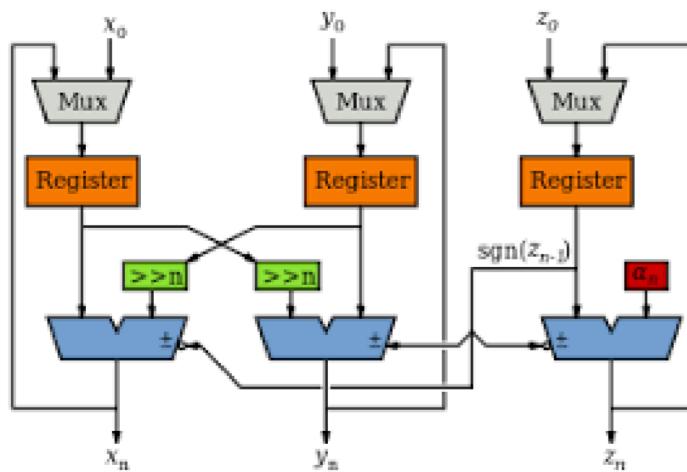
Ces derniers représentent les coefficients de Chebyshev de deuxième type. Sans rentrer dans le détail de l’implémentation :

($T = [1, x, 2x^2 - 1, 4x^3 - 3x, 8x^4 - 8x^2 + 1, 16x^5 - 20x^3 + 5x]$ sont les cinq premiers coefficients du premier type, et $U = [1, 2x, 4x^2 - 1, 8x^3 - 4x, 16x^4 - 12x^2 + 1, 32x^5 - 32x^3 + 6x]$ sont ceux du deuxième type.

2.3 Méthode Cordic

L’algorithme CORDIC (Coordinate rotation Digital Computer), algorithme proposé par Jack E. Volder en 1959,

fait appel à la convergence linéaire, afin d'obtenir une précision à n bits. Il effectue ainsi n itérations d'équations que l'on décrira, n étant d'ordre 32. Au cours de ces itérations, le calcul des équations vectorielles et des angles est effectué. Dès lors, cette méthode est moins coûteuse comparée à l'utilisation des polynômes de Taylor, qui requièrent un grand nombre d'itérations pour aboutir à une correcte précision plus ou moins. Les polynômes de Chebyshev permettent d'avoir une bonne précision, pourtant, elle n'est pas aussi énorme. L'algorithme de Cordic fait usage des méthodes ADD et du SHIFTING en binaire. Les méthodes arithmético-géométriques donnent une meilleure précision, mais font usage des milliers de digits. Ceci justifie ainsi le choix de l'algorithme Cordic pour notre implémentation.



Nous détaillerons nos choix d'implémentation de l'algorithme proposés dans les sections ci-dessous.

3 CORDIC : fonctions trigonométriques

3.1 Principe de l'algorithme : cosinus, sinus

En principe, l'algorithme démarre du vecteur unitaire $(x_0, y_0) = (1, 0)$, crée des vecteurs à chaque itération en effectuant la rotation de celui-ci. Ceci se traduit mathématiquement par une multipliant par la matrice de rotation :

$$\begin{pmatrix} \cos(\alpha_i) & -d_i \sin(\alpha_i) \\ d_i \sin(\alpha_i) & \cos(\alpha_i) \end{pmatrix}$$

celle-ci s'écrit aussi :

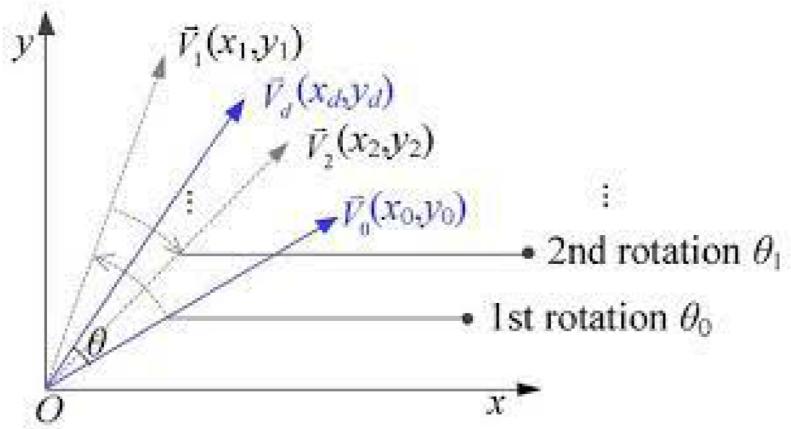
$$\cos(\alpha_i) \times \begin{pmatrix} 1 & -d_i \tan(\alpha_i) \\ d_i \tan(\alpha_i) & 1 \end{pmatrix}$$

où $d_i = -1$ ou $d_i = 1$ pour déterminer le sens de rotation.

Au premier abord, l'entrée de l'algorithme est un réel quelconque. Il sied alors de rapporter cette valeur à l'intervalle $[-\pi, \pi]$ (périodicité des fonctions trigonométriques). Ensuite, puisque nous traînons dans le premier et le dernier quart du cercle trigonométrique, nous rapportons l'angle à l'intervalle $[-\pi/2, \pi/2]$.

Effectivement, la première rotation est de 45 degrés ; on divise à chaque étape notre cercle trigonométrique pour aboutir en fin de compte à notre angle finale, en

effectuant des rotations dans le sens horaire et contre le sens horaire. Il sied ainsi d'invoquer le paramètre d_i qui détermine le sens de rotation de l'angle pas à chaque itération. Le pas choisi n'est autre que $\alpha_i = \arctan(2^{-i})$. Le choix de la division par les pas 2^{-i} est simple en langage machine, puisqu'il s'agit du SHIFTING en binaire.



On obtient ainsi :

$$\begin{aligned} & \cos(\arctan(2^{-i})) \times \begin{pmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{pmatrix} \\ &= \frac{1}{\sqrt{1 + 2^{-2i}}} \times \begin{pmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{pmatrix} \end{aligned}$$

Le vecteur à chaque itération s'écrit ainsi :

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \frac{1}{\sqrt{1 + 2^{-2i}}} \times \begin{pmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{pmatrix} \times \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

Ou encore :

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \frac{1}{\sqrt{1 + 2^{-2i}}} \times \begin{pmatrix} x_i - d_i 2^{-i} y_i \\ y_i + d_i 2^{-i} x_i \end{pmatrix}$$

Les coefficients $K_i = \frac{1}{\sqrt{1 + 2^{-2i}}}$ sont ignorés, puisque le produit après les n itérations donne $K = 0.6072529350088$:

$$\prod_0^{n-1} \frac{1}{\sqrt{1 + 2^{-2i}}} \xrightarrow{\infty} K$$

Au lieu de multiplier à chaque étape nos coordonnées par $\sqrt{1 + 2^{-2i}}$, quitte à démarrer du vecteur

$$(x_0, y_0) = (0.6072529350088, 0)$$

, et effectuer en fin de compte des opérations ADD et SHIFTING en binaire pour retrouver le résultat désirer.

Pour obtenir l'angle, il sied d'orienter correctement les angles dits "pas" pour y aboutir. Ceci se fait grâce aux coefficients d_i . Ces derniers sont déterminés grâce à la sauvegarde de la donnée des angles : on caractérise l'angle totale obtenu à l'itération i par l'équation :

$$z_{i+1} = z_i - d_i \alpha_i$$

La connaissance du sens d'orientation des z_i permettent à cet égard de déterminer d_i , soit ainsi x_i et y_i . Ceci permet de déterminer à chaque étape le sens d'orientation de l'angle "pas", que ce soit dans le sens trigonométrique ou dans le sens horaire. A l'issue de toutes les itérations, le cosinus de l'angle donné n'est autre que l'abscisse du vecteur limite, le sinus étant l'ordonnée de celui-ci. La

tangeante étant la division en ces derniers respectivement.

La connaissance au préalable des vecteurs permet inversement de déterminer les angles ; ceci permet le calcul des fonctions trigonométriques inverses. L'algorithme est similaire, toutefois, il sied d'effectuer des opérations supplémentaire pour éviter la perte des données de calcul.

3.2 Principe de l'algorithme : Arcsin, Arccos, Arctan

3.2.1 Arcsin et Arcos

Sûrement, l'entrée standard de l'algorithme doit effectivement être un réel appartenant à l'intervalle $x \in [-1, 1]$, et de déterminer en fin de compte l'angle θ vérifiant $\sin \theta = x$ ou $\cos \theta = x$. Nous utilisons ici la même séquence de pas, ici $\alpha_i = \arctan(2^{-i})$. L'angle en entrée doit vérifier la condition suivante :

$$\text{theta} \in \left[-\sum_{0}^{\infty} \arctan 2^{-i}, \sum_{0}^{\infty} \arctan 2^{-i} \right]$$

De la même façon, nous démarrons du vecteur unitaire $(x_0, y_0) = (1, 0)$, tout en s'arrêtant lorsque la valeur de l'angle correspondant a été trouvée. Le $i+1$ -ième vecteur s'écrit :

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \frac{1}{\sqrt{1 + 2^{-2i}}} \times \begin{pmatrix} x_i - d_i 2^{-i} y_i \\ y_i + d_i 2^{-i} x_i \end{pmatrix}$$

L'abscisse du vecteur converge vers $K \cos \theta$, tandis que l'ordonnée converge vers $K \sin \theta$

Aussi, la sauvegarde de l'angle précédent permet à cet égard connaître le signe du prochain angle en fonction du sens d'orientation de l'angle précédent.

En fait, la double itération au sein de l'algorithme permet de corriger les erreurs de signe pouvant apparaître après chaque itération, tout en garantissant la convergence. Ceci consiste à garder la même valeur du pas (ici $\alpha_i = \arctan(2^{-i})$, dans une étape donnée, et d'utiliser le même pas dans l'étape suivante. Pourtant, il suffit de réaliser la moitié de ces itérations pour les fonctions cos et sin et arctan

Le facteur devient ainsi

$$K_n = \prod_0^n (1 + 2^{-2i})$$

lors de la n-ième étape. L'angle généré en fonction de celui de l'étape précédente devient :

$$\theta_{i+1} = \theta_i + 2d_i\alpha_i$$

, et on procède aux itérations jusqu'à ce qu'on trouve la valeur de l'angle désiré. Cette double itération s'écrit en lignes de code :

```
xi = x + y*byTwoPower(i);
yi = y - x*byTwoPower(i);
x = xi + yi*byTwoPower(i);
y = yi - xi*byTwoPower(i);
```

ou

```
xi = x - y*byTwoPower(i);  
yi = y + x*byTwoPower(i);  
x = xi - yi*byTwoPower(i);  
y = yi + xi*byTwoPower(i);
```

Selon le signe de l'angle

3.2.2 Arctan

Pour l'arctan, il s'agit de calculer la valeur de l'angle θ vérifiant

$$\tan(\theta) = y/x$$

. Ceci permet ainsi de calculer la valeur $\arctan(y/x)$ où y et x sont les valeurs données à l'entrée, qui sont respectivement eux-mêmes les valeurs de l'ordonnée et de l'abscisse du vecteur utilisé lors du calcul. Sans perte de généralité, poser $x = 1$ permet de calculer directement $\arctan(y)$. On procède au final au même algorithme pour retrouver l'angle correspondant.

Sauf que, au sein de l'algorithme, nous ne multipilions pas par les coefficients K_i pour retrouver la valeur de l'angle, puisque la succession d'addition des pas permet d'y aboutir directement, tout en changeant la valeur de l'ordonnée et de l'abscisse à chaque étape comme illustré précédemment.

3.3 Implémentation

L'implémentation des fonctions trigonométriques en Deca est similaire à celle qu'en java, sauf que, l'utilisation des

tableaux n'est pas possible au sein du compilateur sans l'extension TAB. Ainsi, le défi était de trouver une alternative, permettant de stocker tous les pas mentionnés précédemment, qui sont en effet utilisés lors de chaque étape de nos algorithmes. Un appel de chaque valeur est ainsi primordiale pour déterminer l'orientation de l'angle, selon la formule

$$\theta_{i+1} = \theta_i + 2d_i\alpha_i$$

ou

$$\theta_{i+1} = \theta_i - 2d_i\alpha_i$$

Les pas étant $\alpha_i = \arctan(2^{-i})$

Une première alternative était d'utiliser des listes chaînées, mais, il s'est avéré que l'utilisation des variables de "pas" stockées au préalable et utilisées à chaque itération demeurait une meilleure solution.

Nous avons utilisé les fonctions de puissance de 2, en float :

```
/*Rend la puissance de 2 d'un entier donné*/
float twoPower(int i){
    float result = 1;
    int k = 1;
    if(i == 0){
        return 1;
    }
    else{
        while(k <= i){
            result = result*2;
```

```

        k = k + 1;
    }
    return result;
}
}

//Rend la puissance inverse de 2 d'un entier donné
float byTwoPower(int i){
    if(i == 0){
        return 1;
    }
    else{
        float result = 1;
        int k = 1;
        while(k <= i){
            result = result/2;
            k = k + 1;
        }
        return result;
    }
}

```

Ces deux fonctions permettent de donner une valeur en float de la puissance de deux. La valeur en float se justifie par le fait que la multiplication entre un int et un float pourrait donner des valeurs erronées à chaque étape, soit alors une valeur éronnée en tout.

Ensuite, il s'agit à chaque étape d'effectuer les opérations :

$$x_{i+1} = x_i - d_i y_i 2^{-i}$$

et

$$x_{i+1} = y_i + d_i x_i 2^{-i}$$

à chaque étape pour aboutir au résultat final.

Dans notre implémentation, nous avons rapprocher nos formules à l'écriture : $a+b*c$ par le biais de l'instruction FMA pour réduire les erreurs d'arrondi. Pour ce faire, nous avons séparés nos calculs selon le signe des angles générés lors de chaque étape, en utilisant des conditions if/else, et en évitant les opérations de divisions. En java, le code s'écrit :

```
xi = x + y*byTwoPower(i);  
yi = y - x*byTwoPower(i);
```

et

```
xi = x - y*byTwoPower(i);  
yi = y + x*byTwoPower(i);
```

L'instruction FMA :

```
float multadd(float a, float b, float c)  
    asm(  
        LOAD -3(LB), R1  
        LOAD -4(LB), R2  
        LOAD -5(LB), R3  
        FMA R2, R3  
        LOAD R3, R0  
        RTS");
```

4 Précision

Le calcul de la précision se fait grâce à l'unité de précision élémentaire vis-à-vis aux nombres flottant. Il s'agit de déterminer la distance entre deux flottants, afin de mesurer les erreurs liées aux fonctions trigonométriques implémentées. L'algorithme de notre fonction ULP(x) consiste à calculer l'entier vérifiant $2^i < x < 2^{i+1}$. La valeur de l'ulp n'est autre que 2^{i-23} . Ceci génère un calcul à 2^{-23} en valeur relative, pour obtenir l'erreur en ULP comme unité de précision.

Nous avons utilisé la fonction qui retourne la valeur absolue d'un float, qui s'écrit :

```
float absolute(float f){
    if(f >= 0){
        return(f);
    }
    else{
        return(-f);
    }
}
```

Quant au calcul de la précision de nos fonctions, ceci se fait à l'aide de la formule suivante : en notant l'erreur relatif

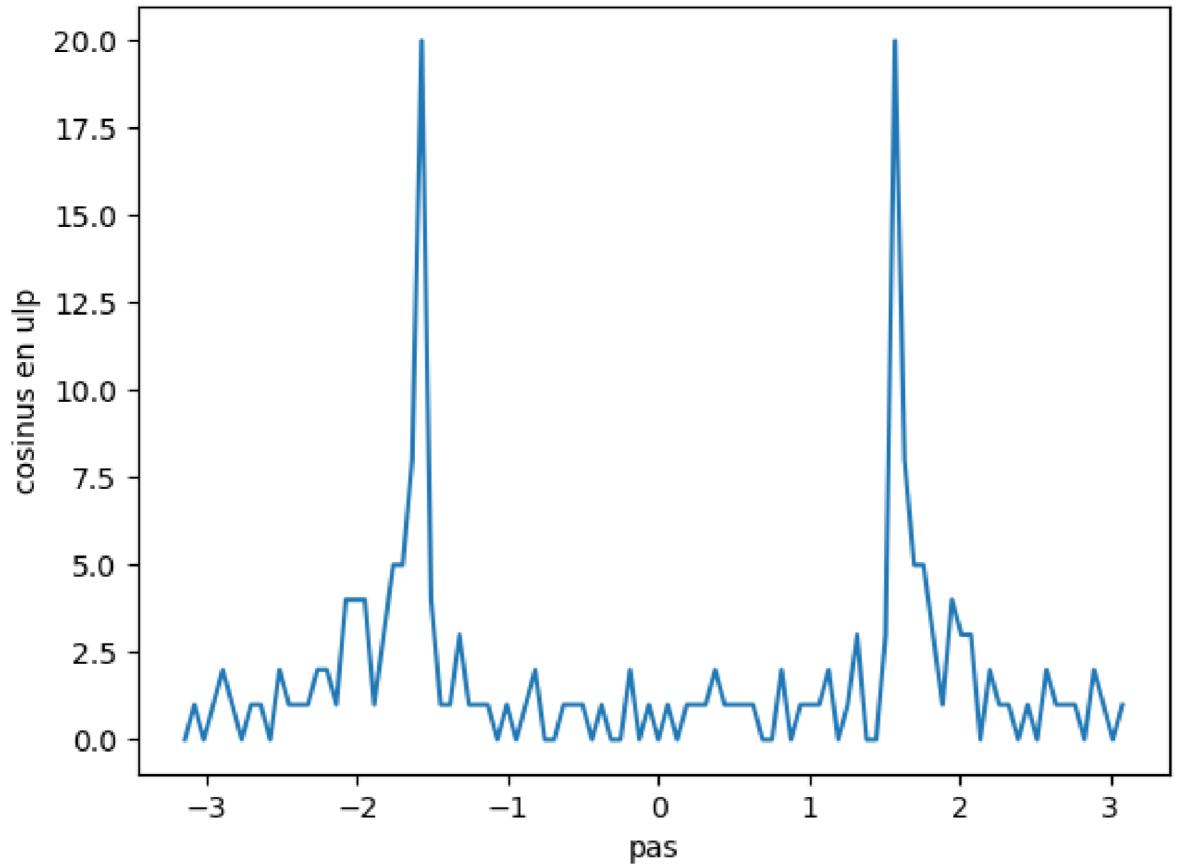
$$\gamma(x) = \frac{|val_{cordic}(x) - val_{java}(x)|}{val_{java}(x)}$$

notre précision s'écrit :

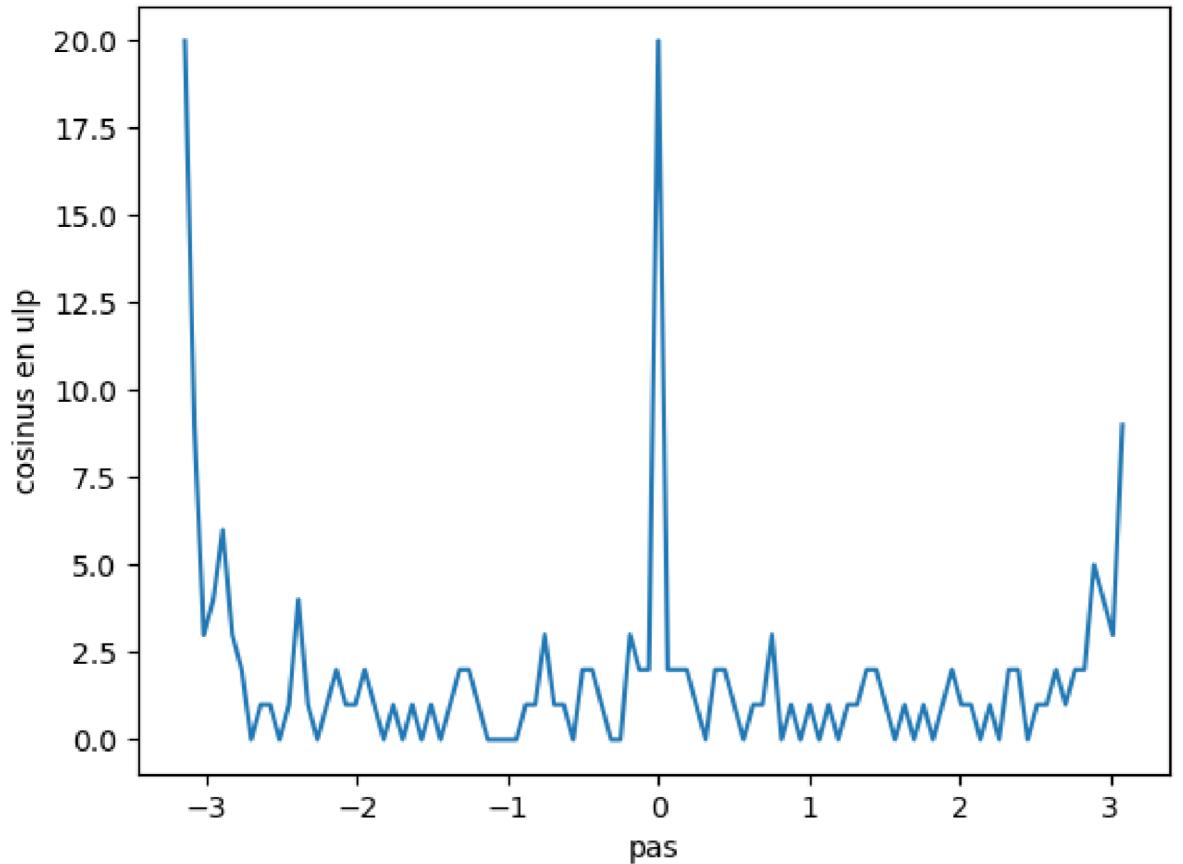
$$p = \frac{\gamma(x)}{ulp(x)}$$

tels que val_{cordic} représente la valeur de la fonction trigonométrique sur un flottant après les tests, et val_{java} représente la valeur de la fonction en java dans la bibliothèque `java.lang.Math` par exemple, soit la valeur théorique de ces fonctions-ci.

Le graphe suivant indique la courbe des erreurs d'arrondis du test pour la fonction cosinus : il s'agit de l'avancement dans le cercle trigonométrique en fonction de γ :



Et pour la fonction sinus :



Pour vérifier les résultats obtenus, nous avons réaliser des tests en java avant decca, qui permettent d'avoir une idée quant à la sûreté de notre implémentation. Le test boucle sur le cercle trigonométrique en entier, et compare les résultats avec ceux de la bibliothèque `java.lang.Math`

```
import java.lang.Math;
System.out.println("cos(90) = " + cos(pi/2) + " compared to " + java.lang.Math.cos(pi/2));
System.out.println("cos(30) = " + cos(30*pi/180) + " compared to " + java.lang.Math.cos(30*pi/180));
System.out.println("cos(45) = " + cos(45*pi/180) + " compared to " + java.lang.Math.cos(45*pi/180));
System.out.println("cos(60) = " + cos(60*pi/180) + " compared to " + java.lang.Math.cos(60*pi/180));
System.out.println("cos(210) = " + cos(210*pi/180) + " compared to " + java.lang.Math.cos(210*pi/180));
System.out.println("cos(-45) = " + cos(-45*pi/180) + " compared to " + java.lang.Math.cos(-45*pi/180));
System.out.println("cos(-90) = " + cos(-90*pi/180) + " compared to " + java.lang.Math.cos(-90*pi/180))
```

```

System.out.println("cos(-210) = " + cos(-210*pi/180) + " compared to " + java.lang.Math.cos(-210*pi/180));
System.out.println("sin(3°) = " + sin(30*pi/180) + " compared to " + java.lang.Math.sin(30*pi/180));
System.out.println("sin(45) = " + sin(45*pi/180) + " compared to " + java.lang.Math.sin(45*pi/180));
System.out.println("sin(60) = " + sin(60*pi/180) + " compared to " + java.lang.Math.sin(60*pi/180));
System.out.println("sin(90) = " + sin(90*pi/180) + " compared to " + java.lang.Math.sin(90*pi/180));
System.out.println("sin(-45) = " + sin(-45*pi/180) + " compared to " + java.lang.Math.sin(-45*pi/180));
System.out.println("sin(-90) = " + sin(-90*pi/180) + " compared to " + java.lang.Math.sin(-90*pi/180));
System.out.println("sin(-210) = " + sin(-210*pi/180) + " compared to " + java.lang.Math.sin(-210*pi/180));
System.out.println("sin(210) = " + sin(210*pi/180) + " compared to " + java.lang.Math.sin(210*pi/180))

ainsi que :

float variable;
int i = 0;
while(i <= 100){
    variable = -pi + i*pi/50;
    System.out.print((cos(variable) - java.lang.Math.cos(variable))/java.lang.Math.ulp(java.lang.Math.abs(cos(variable))));
    i = i + 1;
}

```

5 Bibliographie

Dès les premiers jours du projet, la recherche des algorithmes d'implémentation était une partie primordiale pour garantir l'avancement en extension. Les références consultées sont :

Pour cosinus et sinus :

<http://earsiv.cankaya.edu.tr:8080/xmlui/bitstream/handle/20.500.12416/2066/Alnafutchy>

Pour les méthodes Arcsin et Arcos :

<https://math-python-latex-et.monsite-orange.fr/file/f8223763e06b5feca3774082eb5dba59.pdf?fbclid=IwAR3k7zc5ESGCLDEtFLG24u9oPScAhTW5awwuHDkJtLXACYCX0dPOZk-SIug>

Pour la méthode Arctan :

<https://hal.inria.fr/hal-01091138/document?fbclid=IwAR11CoX-8fYkCgG-fZYFpgwXDJSPlYreZTXrOcOopgT4NFl1HY3LSmRKfvQ>