

Documentation de l'extension  
Extension TRIGO  
Groupe 44

COLIN-SOHY Mathéo

DIMITRIOU Tristan

EL HIT Oumayma

GAUX Antoine

HARRAUD Paul-Louis

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Les flottants simple précision . . . . .	2
1.2	Choix d'implémentation . . . . .	2
1.2.1	CORDIC . . . . .	2
<b>2</b>	<b>Implémentations</b>	<b>4</b>
2.1	CORDIC . . . . .	4
2.2	ulp(x) . . . . .	4
2.3	Fonctions supplémentaires . . . . .	4
<b>3</b>	<b>Résultats</b>	<b>5</b>
3.1	ulp(x) . . . . .	5
3.2	sin(x) et cos(x) . . . . .	6
3.3	arcsin(x) . . . . .	8
3.4	arctan(x) . . . . .	9

# 1 Introduction

L'extension choisie, après négociation, est l'extension TRIGO (Bibliothèque de fonctions trigonométriques et calcul flottant).

L'objectif est d'implémenter des fonctions calculant les valeurs approchées des fonctions suivantes :

- Les fonctions trigonométriques *sin* et *cos*.
- Les fonctions réciproques *arcsin* et *arctan*.
- La fonction de mesure *ulp*.

Ces fonctions opéreront sur des flottants simple précision.

Contraintes :

- Exigences de précision (au presque dernier bit près ou mieux)
- Efficacité des algorithmes (en place mémoire et temps de calcul)

## 1.1 Les flottants simple précision

D'après la norme IEEE754 [3] les nombres flottants sont représentés sous la forme :  $(-1)^s \cdot 2^e \cdot m$

Les flottants simple précision sont stockés dans 32 bits.

avec  $s = \pm 1$ , le signe du nombre représenté (1 bit)

$e \in [-126, 127]$ , l'exposant, codé sur 8 bits

$m$  la mantisse, codée sur 23 bits

Par défaut en deca, les nombres sont arrondis au flottant représentable le plus proche. Nous n'avons pas expérimenté avec d'autres méthodes d'arrondis

## 1.2 Choix d'implémentation

Il existe de nombreuses manières de calculer des fonctions trigonométriques.

Pour une première approximation, on peut utiliser les séries de Taylor correspondant à chaque fonction.

Nous avons choisi d'implémenter l'algorithme CORDIC, pour les fonctions trigonométriques et réciproques. Et une méthode d'encadrement pour *ulp*.

### 1.2.1 CORDIC

Cordic (Coordinate Rotation Digital Computer ou calcul numérique par rotation de coordonnées) est un algorithme permettant le calcul des valeurs approchées des fonctions trigonométriques.

Le principe de calcul est le suivant :

Pour un angle  $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$  correspondant au point  $M(\cos \theta, \sin \theta)$  dans le cercle trigonométrique, on part du point  $M_0(1, 0)$ , on lui applique des rotations, on construit ensuite une suite de points  $M_i = (x_i, y_i)$  dans le cercle trigonométrique et qui converge vers le point  $M$ .

Pour cette raison, on fixe une famille d'angles  $\alpha_i$  qui serviront de tourner le point en question.

On note  $z_i$  l'angle obtenue après  $i$  itérations on a donc :

si  $z_i < \theta$  ; ça signifie qu'on a pas encore atteint l'angle  $\theta$ , on ajoute  $\alpha_i$  :  $z_{i+1} = z_i + \alpha_i$

si  $z_i > \theta$  ; ça signifie cette fois-ci qu'on a dépassé l'angle  $\theta$ , on fait :  $z_{i+1} = z_i - \alpha_i$

On note  $d_i = 1$  si on fait une rotation de  $\alpha_i$ ,  $-1$  sinon.

À chaque itération  $i$  de notre algorithme, on fait tourner le vecteur  $\overrightarrow{OM_i} = (\cos \theta_i, \sin \theta_i)$  en le multipliant par la matrice de rotation :

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

$$\text{Du coup, } \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} \cos(\alpha_i d_i) & -\sin(\alpha_i d_i) \\ \sin(\alpha_i d_i) & \cos(\alpha_i d_i) \end{pmatrix} \begin{pmatrix} x_{i-1} \\ y_{i-1} \end{pmatrix}$$

On factorise par  $\cos(\alpha_i)$

$$\text{On obtient donc, } \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \cos(\alpha_i) \begin{pmatrix} 1 & -d_i \tan(\alpha_i) \\ d_i \tan(\alpha_i) & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ y_{i-1} \end{pmatrix}$$

On fixe la famille  $(\alpha_i)$  de manière à ce que  $\tan(\alpha_i) = 2^{-i}$ , multiplier par la tangente n'est rien qu'une multiplication par une puissance de 2 ou, en binaire, ce n'est rien d'autre qu'un décalage de bits.

On note  $K_i = \cos(\alpha_i) = \cos(\arctan(2^{-i}))$

Pour  $n = 32$

$$K_n = \prod_{i=0}^{n-1} k_i = 0.6072529350088812561694$$

On note  $(x'_i)$  et  $(y'_i)$ , les deux suites définies par :  $x'_0 = 1$  et  $y'_0 = 0$  et

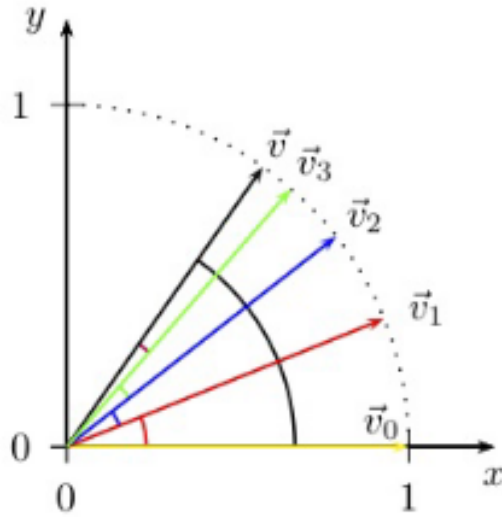
$$\begin{pmatrix} x'_{i+1} \\ y'_{i+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x'_i \\ y'_i \end{pmatrix}$$

Et donc,  $\begin{pmatrix} x_n \\ y_n \end{pmatrix} = K_n \begin{pmatrix} x'_n \\ y'_n \end{pmatrix}$

Une valeur approchée de  $x_n$  et  $y_n$  permet de déterminer une valeur approchée de  $\cos(\theta)$  et  $\sin(\theta)$  respectivement.

L'algorithme se résume donc au système S suivant

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



## 2 Implémentations

### 2.1 CORDIC

CORDIC est un algorithme qu'on peut assimiler à une recherche dichotomique sur un vecteur dans le cercle unitaire.

On peut l'utiliser de plein de manières simplement en changeant les conditions initiales et les conditions de comparaison pour le choix de la direction de rotation.

Pour réutiliser le code on a écrit une seule boucle CORDIC principale, appelée `loop()`, cela évite donc la redondance mais introduit des comparaisons qui seront exécutées à chaque itération, pour toutes les fonctions, ce qui augmente le temps d'exécution.

Après étude [4], on a admis qu'il serait compliqué d'obtenir une précision suffisante sur l'ensemble des réels pour `sin` et `cos`. En effet, le modulo  $2\pi$  aurait fait perdre toute précision sur des grandes valeurs d'entrée. La méthode proposée dans ce même papier était également impossible à implémenter en deca, du moins selon la spécification. Nous n'avons pas eu le temps de tester le `bitshift`, ou encore l'utilisation de l'instruction `FMA`. On a donc décidé de se limiter à une période de `sin` et `cos`, soit entre  $[-\pi; +\pi]$ . C'est une décision importante mais nous avons donc pu nous concentrer sur la minimisation de l'erreur dans ce domaine.

En fait, CORDIC ne fonctionne qu'entre  $-\pi/2$  et  $+\pi/2$ , donc dans les quadrants de gauche. Pour se ramener à ce domaine, on vérifie si le nombre est entre  $-\pi$  et  $+\pi$  et on lui soustrait  $\pi/2$ . Grâce aux propriétés des sinus et cosinus, on renvoie l'opposé de ceux-ci selon les cas. un exemple de code est donné en annexe.

Si le nombre fourni est hors du domaine, le programme renvoie une erreur et s'arrête.

La boucle `cordic` a un nombre d'itérations constant égal à 32, égal au nombre de bits représentant les flottants sur lesquels on opère.

### 2.2 `ulp(x)`

Notre implémentation de `ulp` est une version de l'algorithme donné dans [2] en deca.

Il consiste à minorer un encadrement de `x` en puissances de 2 tel que  $2^i \leq x \leq 2^{i+1}$ .

On choisit les deux bornes de départ comme les puissances maximales en flottant signés 32bits, soient -126 et 127 [3].

On divise par deux un curseur qui est le milieu des deux bornes, en fonction de sa position par rapport à l'exposant du nombre cible, les bornes sont déplacées jusqu'à ce qu'elles encadrent le nombre.

Alors `ulp(x)` est la puissance de l'exposant minimal moins le nombre de bits de mantisse (23).

### 2.3 Fonctions supplémentaires

Pour l'implémentation de nos algorithmes nous avons eu besoin de fonctions et constantes supplémentaires.

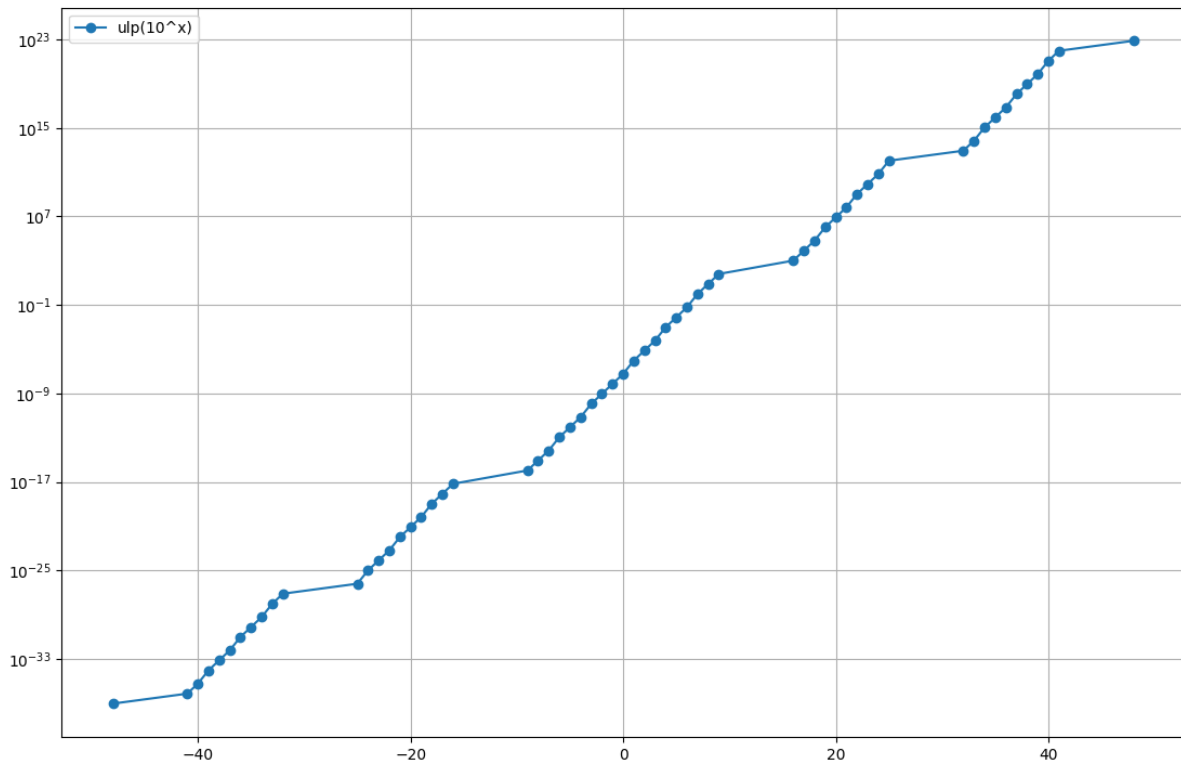
- `pow(a, b) = ab`
- `sign(x) = sign(x)`
- `abs(x) = |x|`
- `pi =  $\pi$`
- `halfpi =  $\pi/2$`

Celles-ci pouvant être utiles pour l'utilisateur, on les a laissées dans la classe `Math`.

### 3 Résultats

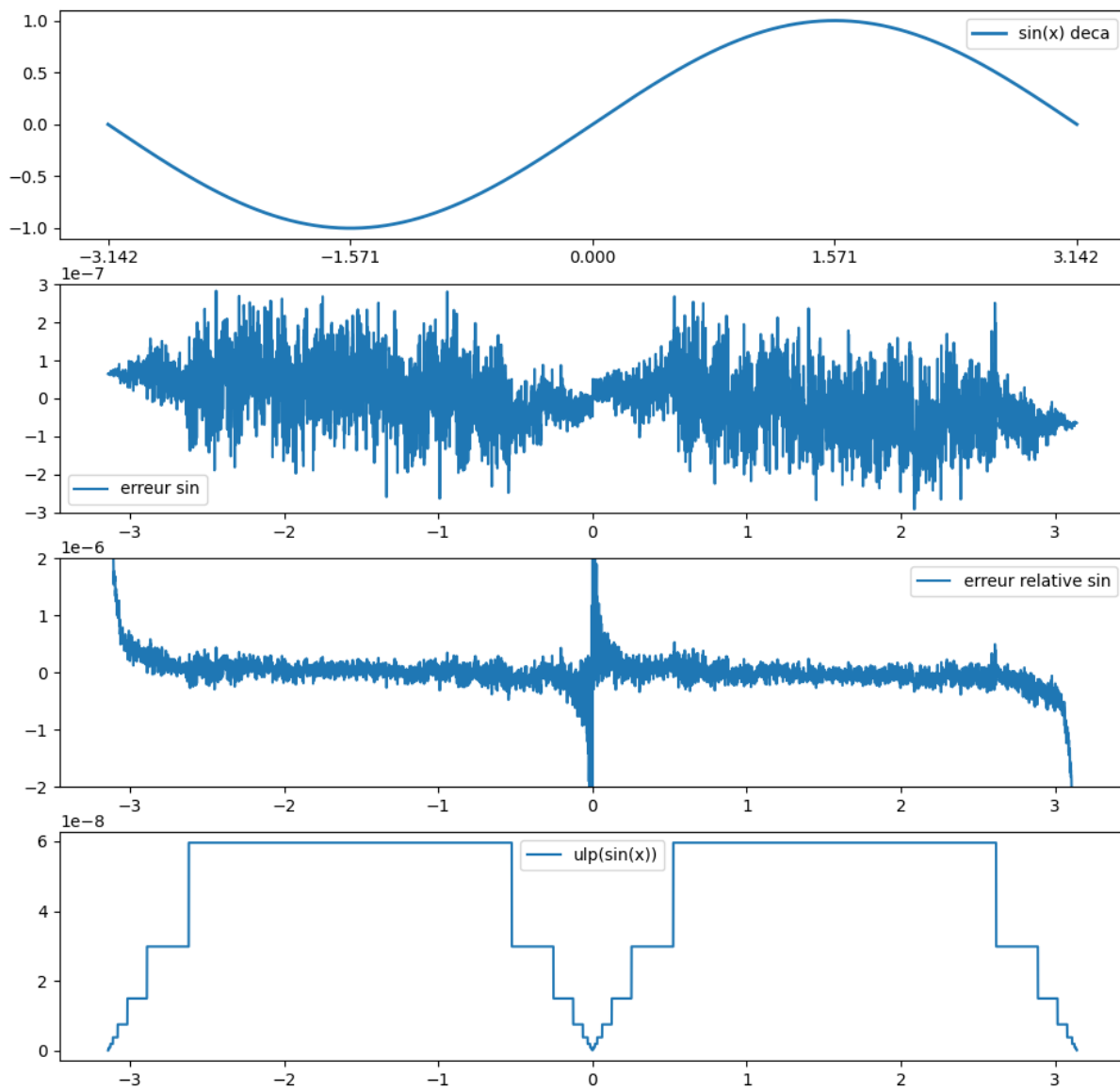
#### 3.1 $\text{ulp}(x)$

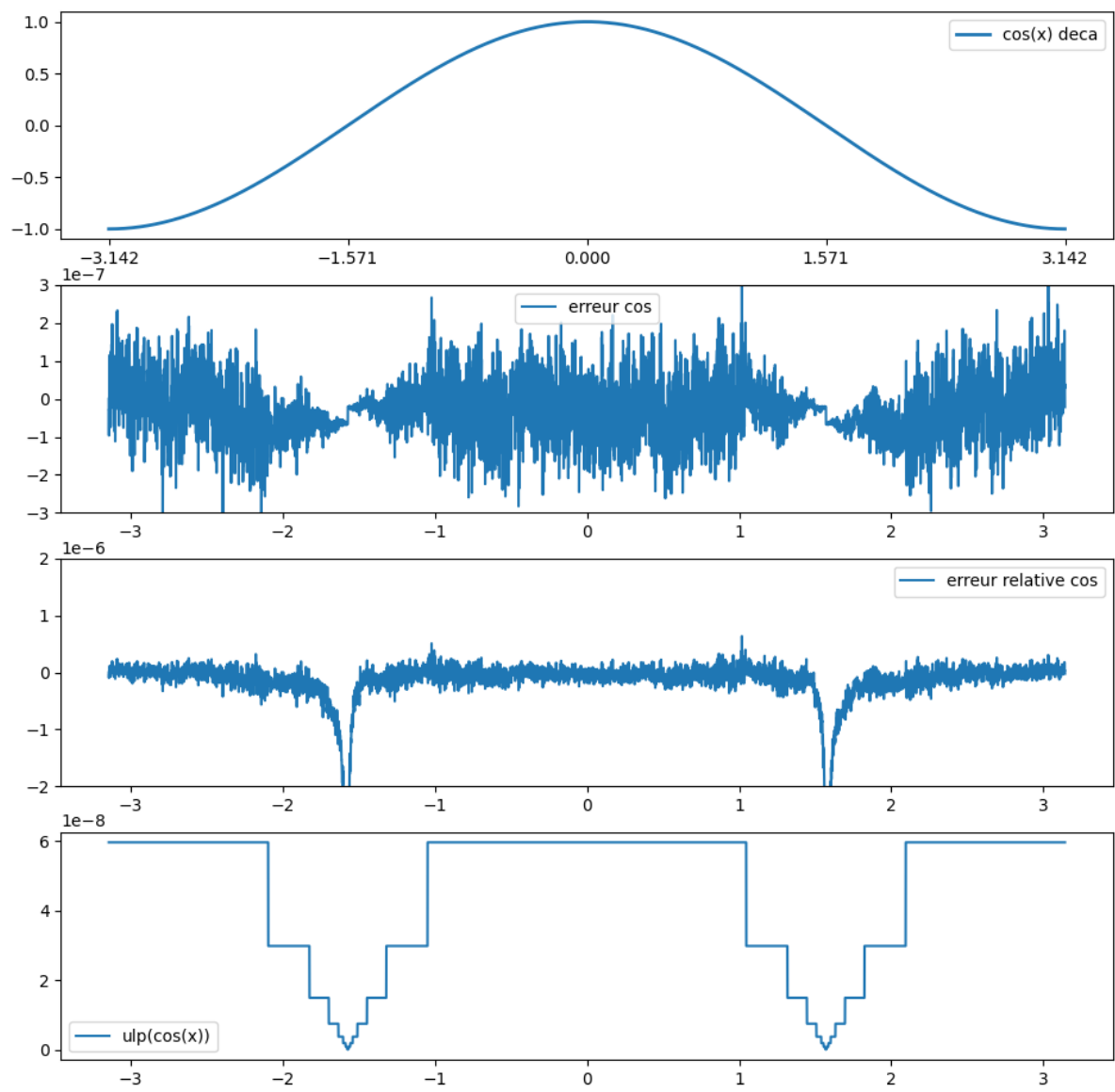
$\text{ulp}(x)$  représente la puissance du dernier bit de  $x$ . Sur ce graphe on valide le fonctionnement en traçant l'ulp dans tout le domaine représentable en flottants 32bits.



### 3.2 $\sin(x)$ et $\cos(x)$

On a calculé sur 2000 points entre  $-\pi$  et  $+\pi$  la valeur de  $\sin(x)$  et  $\cos(x)$  et on a mesuré l'erreur en comparant à la valeur donnée par Python. Avec notre implémentation on arrive à une précision de  $3 \cdot 10^{-7}$  au maximum, mais en erreur relative, on remarque de gros écarts autour de 0, car cordic renvoie une valeur approchée.





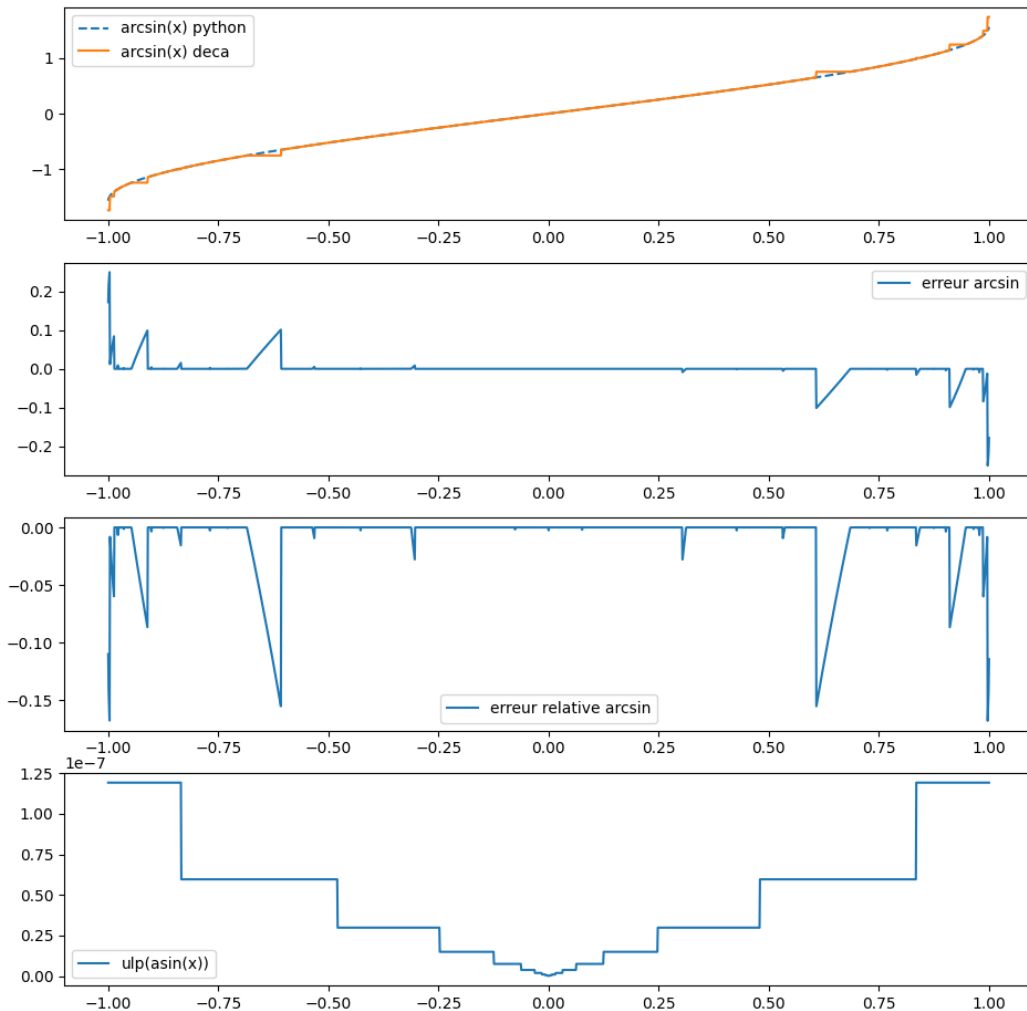
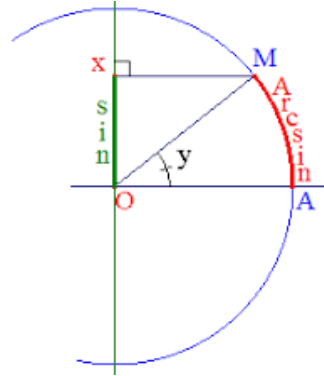


### 3.3 arcsin(x)

Pour calculer  $\arcsin(A)$  pour un  $A$  réel compris entre  $-1$  et  $1$ , on reprend le système S en modifiant la valeur de  $z_0$  qui vaut  $A$  cette fois-ci ainsi que la valeur de  $d_i, d_i = \text{signe}(y_i - z_0)$  et de  $z_{i+1}$  qui vaut maintenant  $z_{i+1} = z_i - d_i \alpha_i$

Le système devient donc :

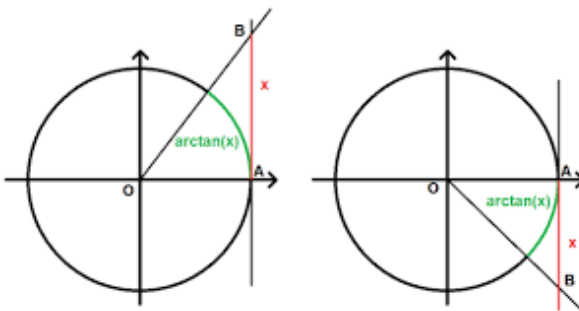
$$\begin{cases} x_0 = K, y_0 = 0, z_0 = A \\ d_i = \text{signe}(y_i - z) \\ x_{i+1} = x_i - d_i y_i 2^{-i} \\ y_{i+1} = y_i + d_i x_i 2^{-i} \\ z_{i+1} = z_i + d_i \alpha_i \end{cases}$$



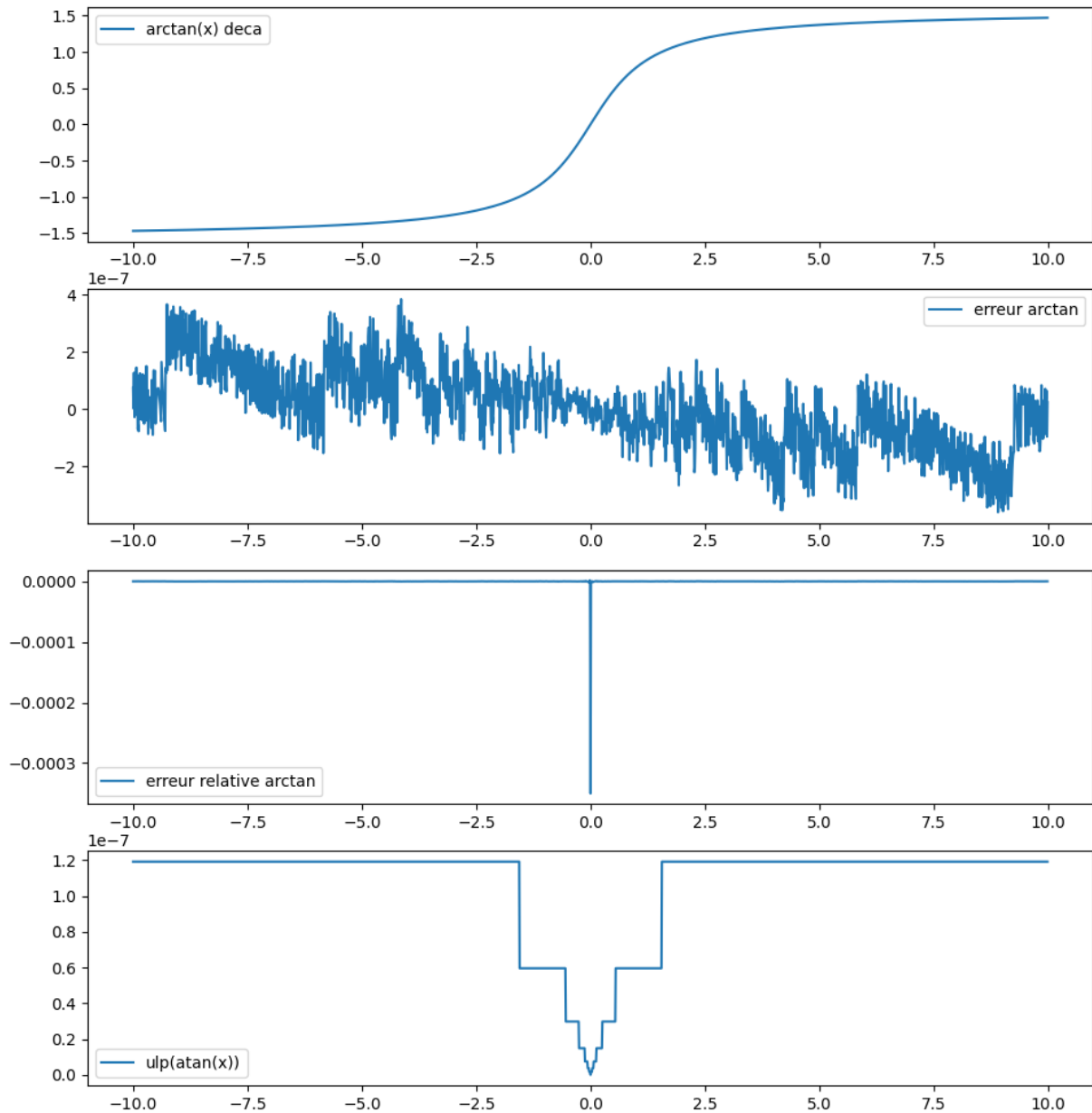
Cependant il est évident sur ce graphe que notre algorithme présente des erreurs que nous ne parvenons pas à expliquer. C'est très gênant pour l'utilisateur, surtout alentour de 1 et -1. On arrive à 15% d'erreur relative au maximum alors que sur le reste de la courbe, la précision est d'environ  $10^{-7}$  comme sur les autres fonctions.

### 3.4 $\arctan(x)$

Pour calculer  $\arctan$  d'un nombre réel  $x$ , on reprend notre système Cordic  $S$ , on change uniquement la valeur de  $d_i$  qui vaut cette fois-ci :  $-\text{signe}(y_i)$ . Le système  $S$  devient donc :

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


De la même manière, on obtient une précision d'au maximum  $4.10^{-7}$  et même en 0 l'erreur est remarquablement faible.



## Références

- [1] Christophe Devalland *L'algorithme CORDIC*, APMEP.  
<https://www.apmep.fr/IMG/pdf/cordic.pdf>
- [2] Jean-Michel Muller (2005) *On the definition of  $ulp(x)$* , ENS Lyon.  
<http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2005/RR2005-09.pdf>
- [3] IEEE *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society.  
<https://irem.univ-reunion.fr/IMG/pdf/ieee-754-2008.pdf>
- [4] Brisebarre, Defour, Kornerup, J.-M Muller and N. Revol *A New Range-Reduction Algorithm*, IEEE  
<http://perso.ens-lyon.fr/nathalie.revol/publis/BDKMR05.pdf>
- [5] *ARGUMENT REDUCTION FOR HUGE ARGUMENTS : Good to the Last Bit*  
<https://www.csee.umbc.edu/~phatak/645/supl/Ng-ArgReduction.pdf>
- [6] Arctan using CORDIC  
<https://www.convict.lu/Jeunes/Math/arctan.htm>

```

1 float ulp(float f) {
2     int     emin, emax, emid;
3     float   pmin, pmax, pmid;
4
5     f = this.fabs(f);
6     if (f > 1.70141e+38) {
7         return this.fpow(2, 127-24);
8     }
9     emin = -126;
10    emax = 127;
11    pmin = this.fpow(2, emin);
12    pmax = this.fpow(2, emax);
13
14    while (emax - emin > 1) {
15        emid = (emax + emin) / 2;
16        pmid = this.fpow(2, emid);
17
18        if (f >= pmid) {
19            pmin = pmid;
20            emin = emid;
21        } else {
22            pmax = pmid;
23            emax = emid;
24        }
25    }
26    if (f == pmin) {
27        return this.fpow(2, emin - 24); // nb de bits mantisse
28    }
29
30    return this.fpow(2, emin - 23);
31 }

```

Listing 1 – Fonction ulp

```

1 float sin(float f) {
2     if (f >= this.halfpi && f <= this.pi) { // II -> I
3         this.loop(1, 0, f-this.halfpi, 1);
4         return this.x;
5     } else if (f <= -this.halfpi && f >= -this.pi) { // III -> IV
6         this.loop(1, 0, f+this.halfpi, 1);
7         return -this.x;
8     } else if (f >= -this.halfpi && f <= this.halfpi) { // I ou IV
9         this.loop(1, 0, f, 1);
10        return this.y;
11    } else {
12        println("sin: domain error");
13        println(f, " is not in [-pi;pi]");
14        this.crash();
15    }
16 }

```

Listing 2 – Exemple de réduction d'argument dans la fonction sin

```

1 void loop(float x0, float y0, float z0, int select) { // CORDIC loop
2     float x = x0 * K;
3     float y = y0 * K;
4     float z, xt, d = 1, e;
5     int i = 0;
6     if (select == 2) { // asin
7         z = 0;
8     } else {
9         z = z0;
10    }
11
12    while (i < 32) {
13        e = fpow(2.0, -i);
14        if (select == 0) { // atan
15            if (y >= 0) {
16                d = -1;
17            } else {
18                d = 1;
19            }
20        } else if (select == 1) { // sin & cos
21            if (z >= 0) {
22                d = 1;
23            } else {
24                d = -1;
25            }
26        } else if (select == 2) { // asin
27            if (y < z0) {
28                d = 1;
29            } else {
30                d = -1;
31            }
32        }
33
34        xt = x - d * y * e;
35        y = y + d * x * e;
36        x = xt;
37
38        if (select == 2) {
39            z = z + d * arctable(i);
40        } else {
41            z = z - d * arctable(i);
42        }
43
44        i = i+1;
45    }
46
47    this.x = x;
48    this.y = y;
49    this.z = z;
50 }

```

Listing 3 – Boucle CORDIC