Documentation de l'extension TRIGO

Samy Ahjaou Hamza Bouihi Omar Guessous
Grégoire Sename Mamadou Thiongane

Groupe 9 – Équipe 42

26 janvier 2024

Table des matières

1 Introduction

Le développement de la classe Math constitue une composante cruciale du projet, mettant en lumière la nécessité de concevoir des algorithmes efficaces pour des opérations mathématiques complexes.

Nous plongerons également dans une analyse théorique approfondie de la précision que nous pouvons attendre des algorithmes choisis. Comprendre les implications de nos choix sur la précision des calculs constitue un aspect essentiel de cette documentation.

La validation de la classe Math en Deca se révèle être un défi distinct par rapport à la validation du compilateur de base. Ici, la notion de précision des calculs prend une importance particulière, ajoutant une dimension supplémentaire à l'évaluation de la robustesse de nos algorithmes et de leur implémentation.

2 Analyse théorique

2.1 Représentation IEEE 754 simple précision

La représentation IEEE 754 simple précision, également connue sous le nom de flottant simple, est une norme de codage binaire pour la représentation des nombres à virgule flottante. Cette norme est définie par l'Institute of Electrical and Electronics Engineers (IEEE). La représentation IEEE 754 simple précision utilise 32 bits pour encoder un nombre à virgule flottante, avec une précision significative d'environ 7 chiffres décimaux.

La structure d'un nombre à virgule flottante en simple précision selon la norme IEEE 754 est la suivante :

1 bit pour le signe (0 pour positif, 1 pour négatif), 8 bits pour l'exposant, 23 bits pour la mantisse (fraction). La formule générale pour calculer la valeur réelle d'un nombre à virgule flottante en simple précision est la suivante :

$$(-1)^{\text{signe}} \times 2^{\text{exposant - décalage}} \times (1 + \text{mantisse})$$

Le bit de signe indique si le nombre est positif (0) ou négatif (1). L'exposant est représenté avec un décalage fixe de 127 (c'est-à-dire, l'exposant réel est exposant - 127 exposant - 127). La mantisse est une fraction binaire normalisée, avec une partie entière implicite de 1. Cette représentation permet de représenter une gamme de nombres réels, mais elle a des limites de précision en raison du nombre fini de bits alloués à la mantisse. Elle est souvent utilisée dans les calculs informatiques, en particulier dans les applications où la précision n'est pas aussi critique que dans les représentations à double précision.

2.2 L'ULP

L'ULP (*Unit of Least Precision*) est une mesure de la précision d'un nombre flottant sur un ordinateur. Elle quantifie la plus petite différence possible entre deux flottants consécutifs autour de ce nombre. En d'autres termes, l'ULP précision maximale qu'un nombre peut avoir.

On la calcule comme cela :

$$\text{ulp(x)} = 2^{\text{e(x)}} \times 2^{-23}$$

où `e(x)` est l'exposant de la représentation scientifique binaire

Illustrons cela avec les nombres 0 et π en utilisant une représentation en virgule flottante simple précision (32 bits).

$$\text{ulp}(\pi) = 2^{-22} \approx 2,4 \times 10^{-7}$$

La représentation la plus précise en IEEE 754 simple précision de π est donc 3,1415927.

0 est un cas particulier car le plus petit nombre représentable est celui d'exposant décalé 0 et de mantisse 000 0000 0000 0000 0000 0001.

$$\text{ulp}(0) = 2^{-149}$$

En résumé, l'ULP est une mesure de la précision relative entre deux nombres consécutifs dans une représentation en virgule flottante. Plus la valeur de l'ULP est petite, plus la précision du système numérique est élevée. L'ULP est une considération cruciale lors de la manipulation de nombres flottants pour comprendre la précision et les erreurs potentielles associées à ces représentations.

2.3 Pourquoi raisonner en binaire

Lors du développement de cette extension, il a fallu raisonner en binaire, étant donné que la représentation interne des nombres sur les ordinateurs est basée sur le système binaire.

Un exemple concret de cette compréhension binaire est la manipulation de nombres en virgule flottante selon les normes IEEE 754. Par exemple, l'addition de 1,1 et 1,1, représentée en IEEE 754, donne un résultat légèrement différent de 2,2. La représentation binaire de 1,1 en IEEE 754 est : 0 01111111 00011001100110011001101, équivalant à

$$1.100000023841858 \times 2^0$$

Cette observation nous a permis de reconnaître les subtilités associées à cette égalité apparemment déroutante.

En revanche, l'addition de 1,25 et 1,25, qui sont des sommes parfaites de puissances de deux, donne le résultat attendu de 2,5.

Les opérations utilisant les puissances de deux ont donc été favorisées dans l'élaboration de tous nos programmes, profitant de cette propriété particulière pour garantir des résultats plus cohérents.

3 Implémentation

3.1 Algorithmes utilisés

3.1.1 Développement en série entière

Les fonctions que nous avions à implémenter sont toutes développables en série entière. Cela a donc été notre premier choix d'implémentation.

Pour chaque fonction, son développement est :

Cosinus

Le cosinus est développable en série entière sur $\mathbb{R}$, et son développement est :

$$\forall x \in \mathbb{R} \quad \cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n}}{(2n)!}$$

Sinus

Le sinus est développable en série entière sur $\mathbb{R}$, et son développement est :

$$\forall x \in \mathbb{R} \quad \sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{(2n+1)!}$$

Arc sinus

Arcsinus est développable en série entière sur]-1, 1[. Son ensemble de définition étant [-1, 1] il faut simplement créer 2 cas particuliers pour -1 et 1.

Son développement est :

$$\forall x \in]-1, 1[\quad \arcsin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{(2n+1) \cdot (2n)!}$$

et

$$\arcsin(1) = \pi/2$$
$$\arcsin(-1) = -\pi/2$$

Arc tangente

Arctangente est développable en série entière sur]-1, 1[. Son ensemble de définition étant $\mathbb{R}$, nous allons utiliser la formule :

$$\forall x \in \mathbb{R} \quad \arctan(x) = \pi/2 - \arctan\left(\frac{1}{x}\right)$$

Son développement est :

$$\forall x \in]-1,1[\quad \arctan(x) = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{2n+1}$$

En conclusion, bien que nos fonctions à implémenter soient toutes développables en série entière, nous avons constaté que le développement en série peut parfois manquer de précision, surtout aux extrémités des intervalles de convergence. Pour pallier ce problème, nous avons opté pour une approche alternative dans le cas de l'arc tangente.

En résumé, l'exploration de différentes méthodes s'est avérée cruciale pour garantir une précision optimale dans le calcul de l'arc tangente notamment. Ce choix judicieux d'algorithmes adaptés à des intervalles spécifiques a contribué à améliorer la fiabilité de nos implémentations trigonométriques dans des situations délicates de convergence.

3.2 Validation des algorithmes et implémentation en Deca

3.2.1 Conception de la classe

La classe utilisée pour l'implémentation de la biliothèque `math` a donc cette structure :

Constante : La classe définit une constante pi qui approxime la valeur de π. Cette constante est utilisée dans les calculs des fonctions trigonométriques.

```
        float pi = 3.141592741012573242187 5;
```

Fonctions Trigonométriques (sin, cos, tan) : Les fonctions trigonométriques, telles que sin, cos et tan, sont implémentées en utilisant des séries de Taylor. Ces séries permettent d'obtenir des approximations précises de ces fonctions en effectuant des itérations.

```
float sin(float f) {
// Implementation de la serie de Taylor pour sin
}

float cos(float f) {
    // Implementation de la serie de Taylor pour cos
}

float tan(float f) {
    // Implementation de la serie de Taylor pour tan
}
```

Fonctions Inverses Trigonométriques (asin, atan) : De manière similaire, les fonctions inverses trigonométriques, arcsin et arctan, sont également implémentées à l'aide de séries de Taylor.

```
float asin(float f) {
// Implementation de la serie de Taylor pour arcsin
}
```

```
float atan(float f) {
    // Implementation de la serie de Taylor pour arctan
}
```

Fonctions Auxiliaires Mathématiques : La classe comprend également diverses fonctions auxiliaires, telles que

```
float _pow(float f, float e);
float _abs(float f);
float _sqrt(float);
float _fact(float f);
float _pow32(float f);
float _floor(float f);
```

qui sont utilisées dans les calculs des fonctions principales.

Elles permettent de calculer :

- `_pow` : f^e
- `_abs` : $|f|$
- `_sqrt` : $\sqrt{f}$
- `_fact` : $f!$
- `_pow32` : exposant de la représentation scientifique binaire du floatant (utilisé dans l'ulp)
- `_floor` : arrondi de f à l'inférieur

3.2.2 Ajustement des algorithmes

Recentrage de sinus et cosinus sur un intervalle plus petit

Nous sommes désormais capables de calculer chaque fonction avec un niveau de précision considérable. Cependant, la précision de nos calculs dépend de l'ulp, laquelle est proportionnelle à la valeur absolue du nombre. Pour optimiser cette précision, nous exploitons la périodicité des fonctions *sinus* et *cosinus* en recentrant les valeurs dans l'intervalle [-π, π].

De plus, nous avons la possibilité de nous recentrer dans l'intervalle [0, π] en exploitant les relations trigonométriques suivantes :

$$\cos(-x) = \cos(x)$$

$$\sin(-x) = -\sin(x)$$

Enfin, en utilisant les relations trigonométriques suivantes, nous avons la possibilité de recentrer à nouveau les valeurs dans l'intervalle [0, $\pi/2$].

$$\cos(x) = \sin\left(\frac{\pi}{2} + x\right)$$

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

Ainsi, en utilisant la périodicité et les relations de trigonométriques de `cosinus` et `sinus` nous avons donc pu recentrer les valeurs de $\mathbb{R}$ sur $[0, \pi/2]$.

Calcul de tangente

Nous avons fait le choix d'implémenter la fonction tangente grâce à :

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

Performance

Dans une optique de performance, davantage que de précision, nous avons cherché à minimiser l'utilisation des fonctions telles que `_pow` et `_fact`. Au lieu de cela, nous avons calculé directement ces valeurs au moyen des itérations des développements en séries entières, évitant ainsi plusieurs calculs redondants.

Prenons l'exemple du développement limité du cosinus qui nécessite des termes comme x^{2n}. Si nous employons la fonction `_pow`, cela implique une boucle de taille $2n$ à chaque itération du développement. Cependant, une approche plus efficace consiste à stocker un flottant `x_pow` et à le multiplier par $x * x$ à chaque itération. Cette optimisation réduit la complexité algorithmique de nos implémentations, contribuant ainsi à améliorer les performances globales.

3.2.3 Test de la classe

Malheuresement, nous n'avons pas pu obtenir un compilateur avec objet fonctionnel lors de ce projet. Nous avons donc du réfléchir à un moyen de tester notre classe sans devoir compiler des objets. Nous avons donc pensé à créer un fichier `test_class_math.deca` implémentant les fonctions trigonométriques de la classe math de manière impérative.

Nous avons ainsi écrit un fichier comme celui en Annexes.

Nous pouvons donc voir dans la boucle principale ce que ferais l'appel à la fonction math.cos si nous avions pu compiler un langage avec objet. Nous avons donc du calculer impérativement les puissances, factorielles et autres valeurs nécessaires au calcul du cosinus.

Nous avons donc opéré comme cela pour toutes les fonctions trigonométriques pour pouvoir les étudier le plus précisemment possible avec les moyens que nous avions.

Une fois ce programme implémenté, nous avons du estimer la précision de ces valeurs.

Pour cela, nous les avons comparées avec la bibliothèque `math` du langage `Python` qui utilise le standard *IEEE 754*. Pour les fonctions `cos`, `sin`, `tan` et `atan`, nous avons mesuré ces erreurs entre -4π et 4π. `asin` n'est défini que sur [-1, 1], nous n'avons donc étudié les erreurs que sur cet intervalle.

Nous avons d'abors pris la décision de calculer l'erreur absolue avec la formule suivante :

$$\text{erreur} = |\text{valeur deca} - \text{valeur python}|$$

Nous avons donc obtenu ces résultats, avec en abscisse l'erreur décrite ci-dessus et en ordonnée l'angle en radians.

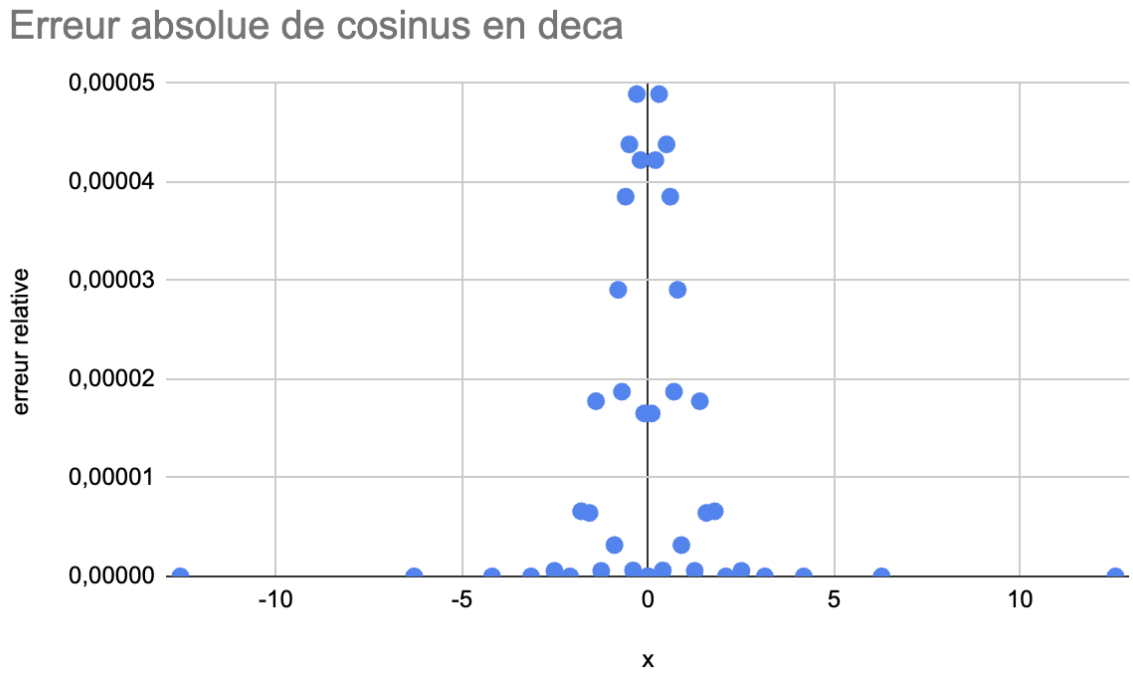

FIGURE 1 – Erreur absolue de la fonction cosinus

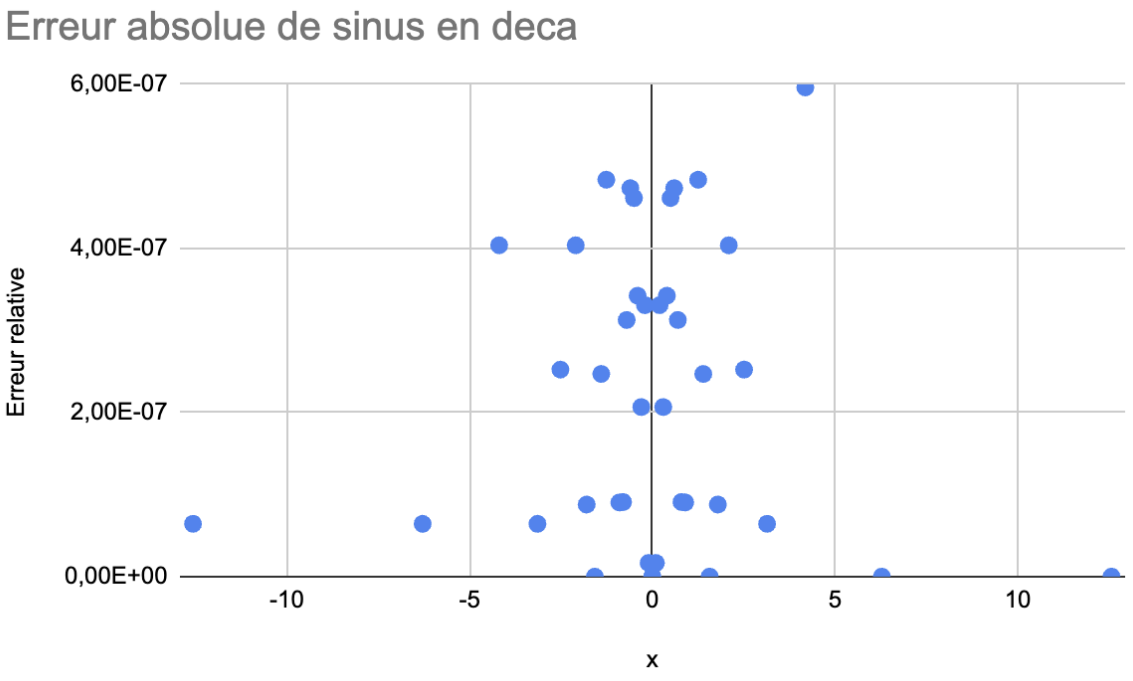

FIGURE 2 – Erreur absolue de la fonction sinus

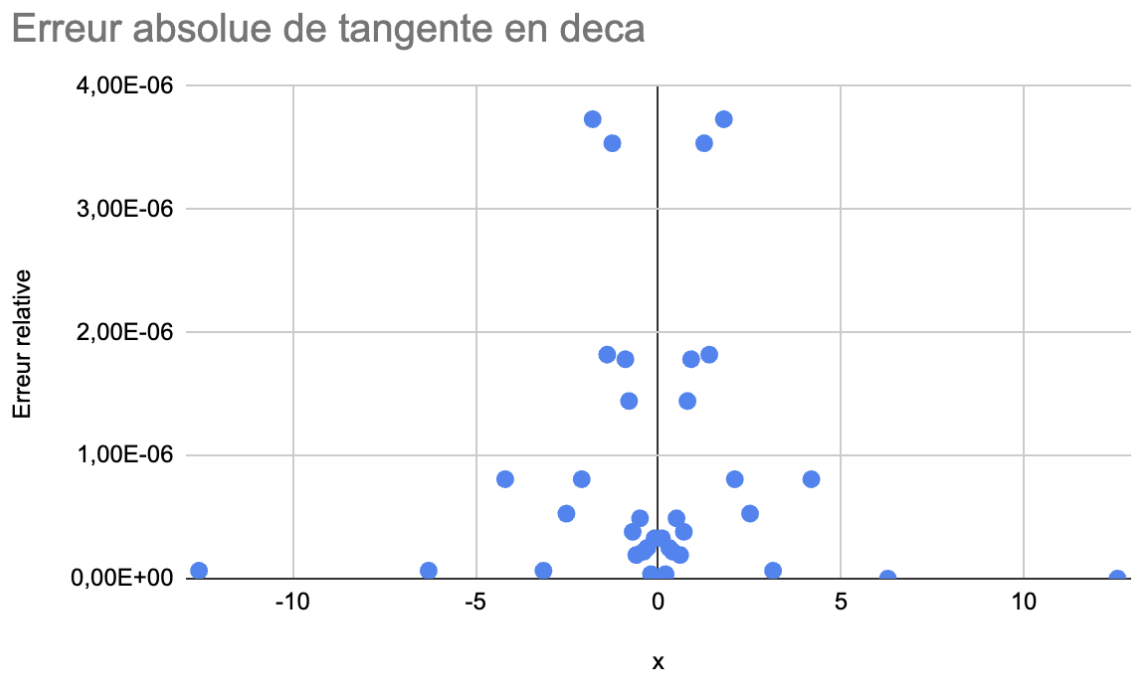

FIGURE 3 – Erreur absolue de la fonction tangente

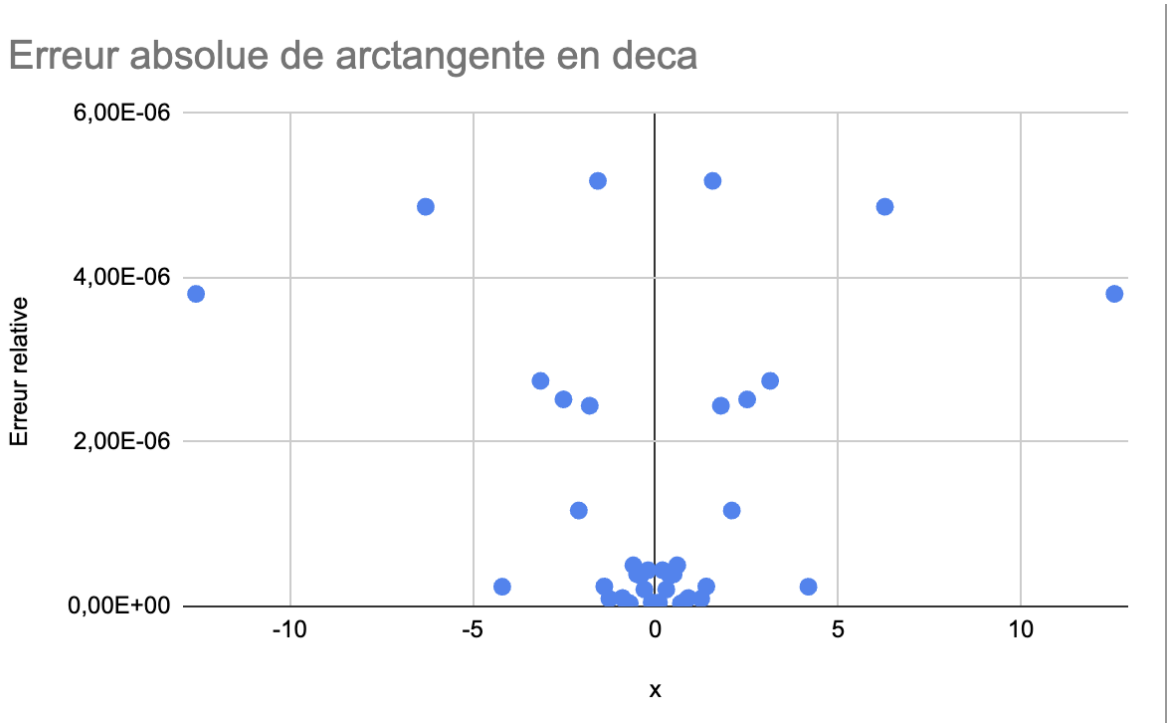

FIGURE 4 – Erreur absolue de la fonction arc tangente

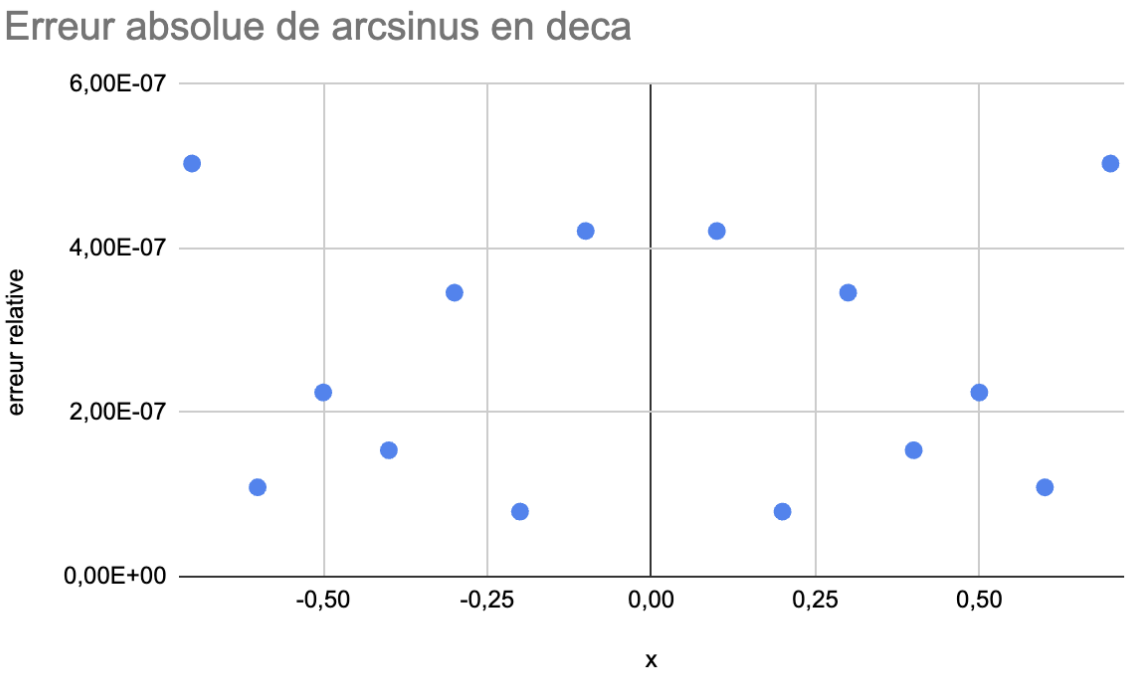

FIGURE 5 – Erreur absolue de la fonction arc sinus

Nous avons ensuite cherché à obtenir des informations plus exploitables sur ces erreurs. Cependant, nous n'avons pas pris la décision de calculer l'erreur relative avec la formule habituelle

$$\text{erreur} = \frac{|\text{valeur mesurée} - \text{valeur théorique}|}{\text{valeur théorique}}$$

car dans les cas des valeurs qui tendent de 0, par exemple `cos($\pi/2$`, on obtient comme valeurs :
En Deca avec notre bibliothèque :

$$\cos(\pi/2) = -6.43447e-08$$

Avec la bibliothèque Python :

$$\cos(\pi/2) = 6.123233995736766e^{-17}$$

Et donc :

$$\text{erreur\_absolue} = 6,43e^{-08}$$

$$\text{erreur\_relative} = \frac{6,43e^{-08}}{6.123233995736766e^{-17}} \approx 1,05e^{+09}$$

Nous voyons donc que l'erreur relative n'est pas représentative, sachant que l'ULP de $\pi/2$ est de l'ordre de 10^{-7}, et que la précision du nombre final sera limitée par cela.

Une erreur plus représentative serait donc une erreur relative par rapport à l'ULP :

$$\text{erreur} = \frac{|\text{valeur deca} - \text{valeur python}|}{ulp}$$

Voici donc les résultats obtenus, avec en abscisse l'erreur relative par rapport l'ulp et en ordonnée l'angle en radians.

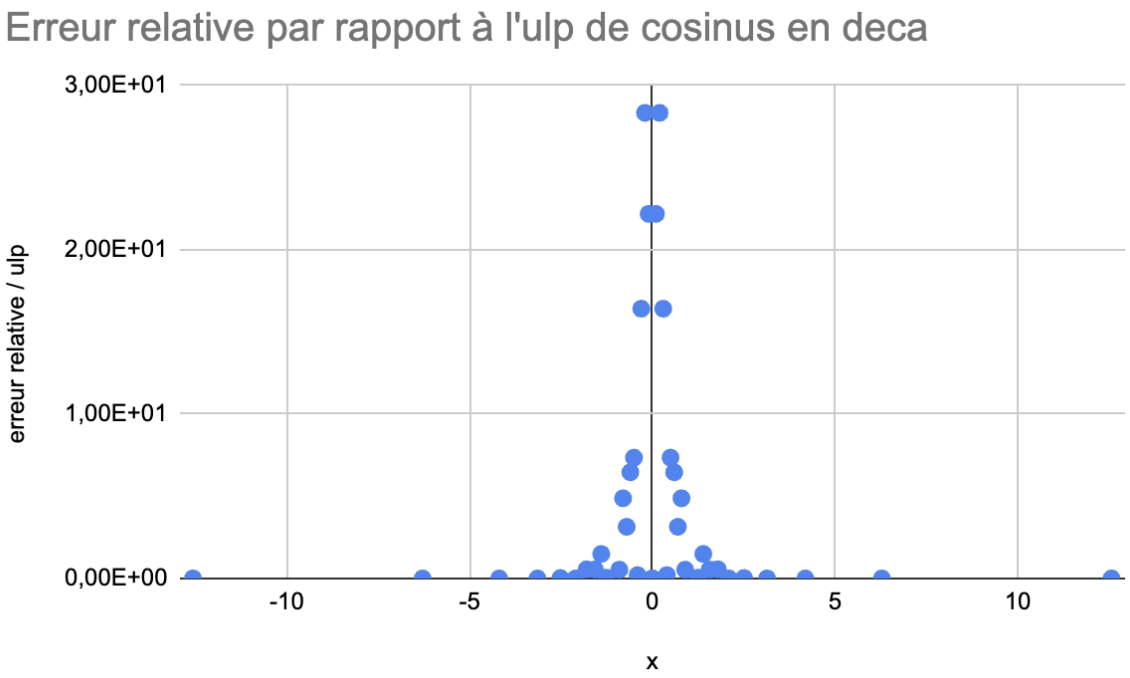

FIGURE 6 – Erreur relative de la fonction cosinus

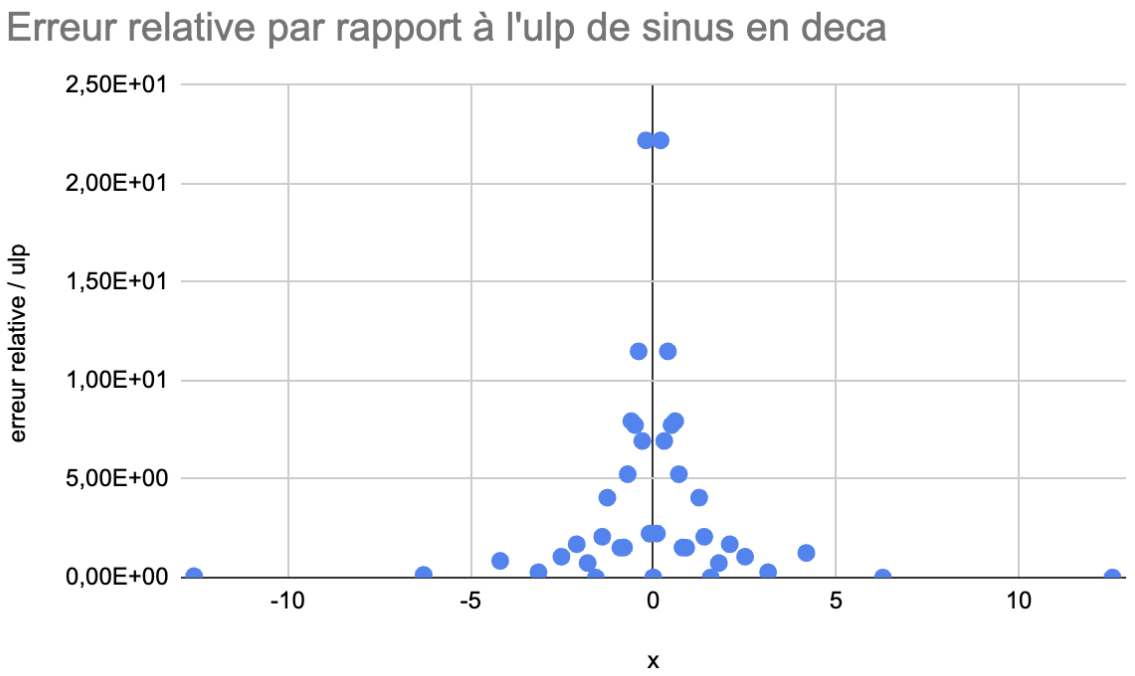

FIGURE 7 – Erreur relative de la fonction sinus

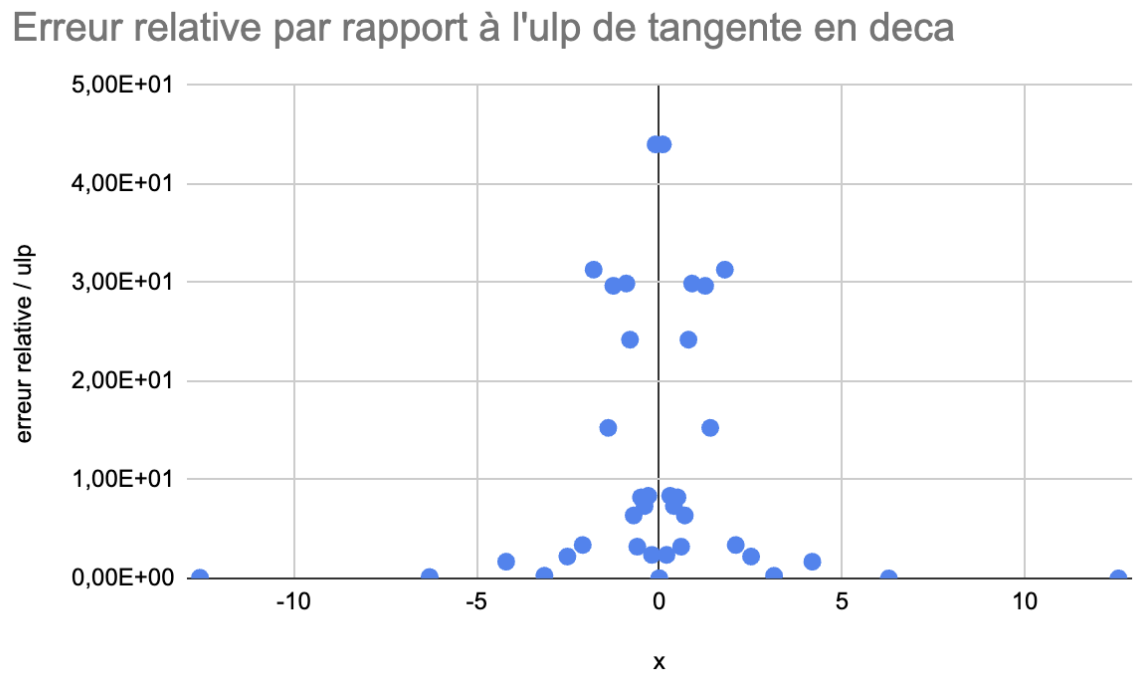

FIGURE 8 – Erreur relative de la fonction tangente

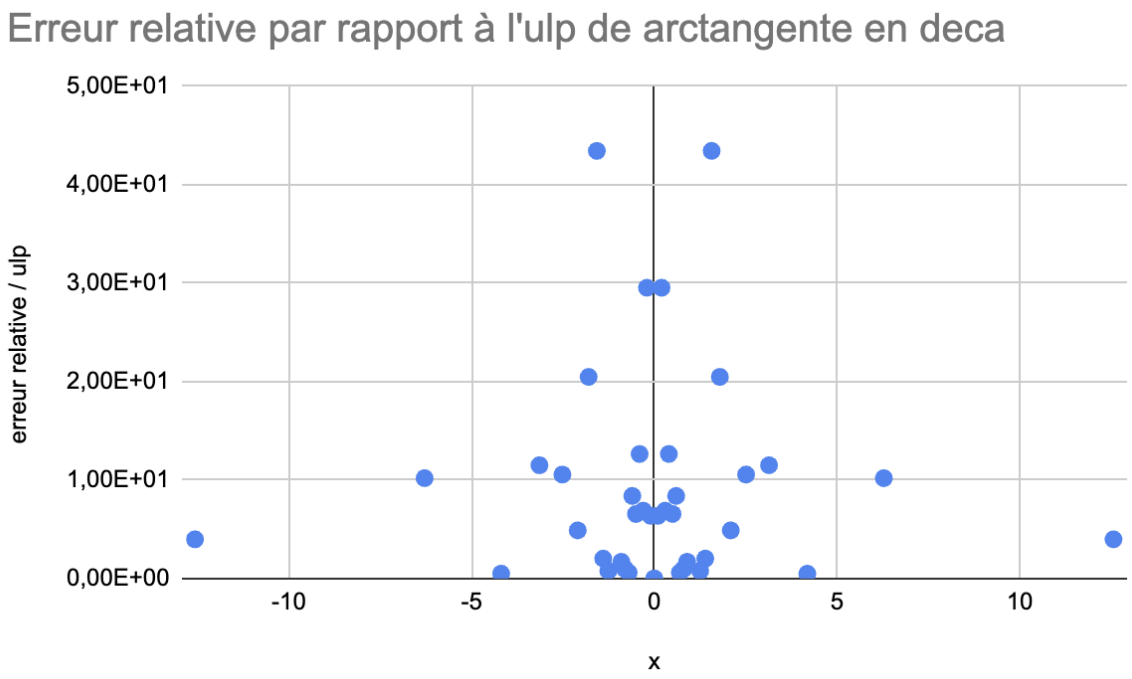

FIGURE 9 – Erreur relative de la fonction arc tangente

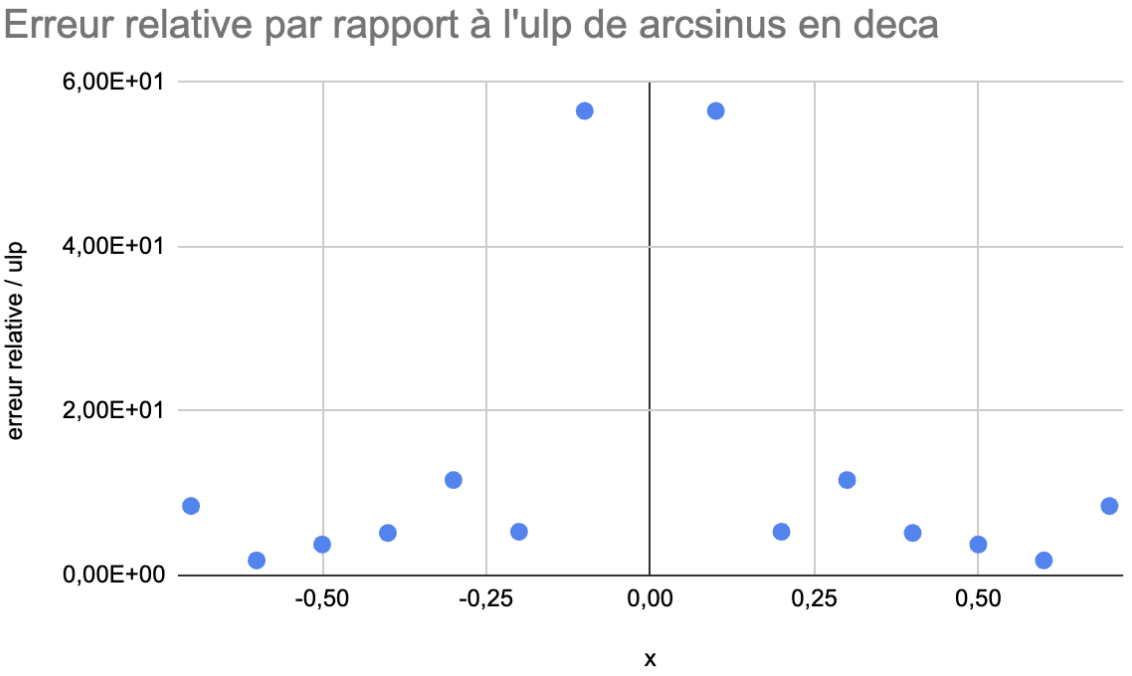

FIGURE 10 – Erreur relative de la fonction arc sinus

Cette représentation des erreurs nous permet d'observer que notre implémentation atteint une précision similaire à celle de l'ULP, une conclusion que nous trouvons cohérente et justifiée.

4 Utilisation de la Bibliothèque

Voici un exemple d'utilisation de la bibliothèque maths de `Deca` :

```
#include "Math.decah"

{
    Math math = new Math();
    println(math.cos(2.5 * math.pi));

}
```

5 Conclusion

En conclusion, le développement de la classe Math dans le cadre de ce projet a constitué une étape cruciale, mettant en évidence la nécessité de concevoir des algorithmes efficaces pour des opérations mathématiques complexes. Nous avons exploré diverses méthodes, en mettant en œuvre des développements en série entière pour les fonctions trigonométriques.

L'analyse théorique a souligné l'importance de la représentation IEEE 754 simple précision et de la mesure de l'ULP pour évaluer la précision des calculs sur les nombres flottants. La manipulation en binaire s'est avérée indispensable pour comprendre les subtilités liées à cette représentation.

L'implémentation a été réalisée en utilisant des séries de Taylor pour les fonctions trigonométriques, et des optimisations ont été apportées pour améliorer les performances, en évitant notamment les calculs redondants.

Le test de la classe a été effectué de manière impérative, en comparant les résultats avec la bibliothèque mathématique de Python. Les erreurs absolues et relatives par rapport à l'ULP ont été étudiées, montrant que notre implémentation atteint une précision similaire à celle de l'ULP, ce qui est cohérent compte tenu des limitations intrinsèques des nombres flottants.

En définitive, ce projet a permis d'approfondir nos connaissances en mathématiques, algorithmique et représentation des nombres flottants, et a souligné l'importance de l'optimisation pour garantir des performances optimales. Cette classe Math constitue une base solide pour des développements futurs dans le domaine de la manipulation mathématique en Deca.

6 Bibliographie

Cours d'Informatique pour Tous 2021 – Jules Svartz Lycée Masséna

David Defour. Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision. Modélisation et simulation. Ecole normale supérieure de lyon - ENS LYON, 2003. Français. ffNNT : ff. fftel-00006022f

Wikipedia IEEE754

7 Annexes

Implémentation impérative de la fonction cosinus dans le fichier de test

```
// Description:
//    test de l'inclusion de la classe Math de maniere imperative
{
    float pi = 3.1415927410125732421875;
    float f = pi/2;
    float ulp;
    float res = 0.0;
    float x = 1.0;
    int i = 0;
    float fact = 1.0;
    float pow = 1.0;
    float sign = -1.0;
    float pow32;
    float fpow32 = f;
    float pow2 = 1.0;
    float pow23 = 1.0;
    float i_for_fact;
    float absx = 1.0;

    int NB_CALC = 0;

    while(NB_CALC < 10){
        pow = 1.0;
        fact = 1.0;
        sign = -1.0;
        i_for_fact = 0.0;
        absx = 1.0;
        res = 0.0;
        x = 1.0;
        i = 0;
        pow2 = 1.0;
        pow23 = 1.0;
        f = 4.0 * pi / (NB_CALC + 1.0);
        pow32 = 4.0 * pi / (NB_CALC + 1.0);

        // RECENTRAGE ENTRE -PI ET PI
        while(f > pi){
            f = f - 2*pi;
        }
        while(f < -pi){
            f = f + 2*pi;
        }
        if(f < 0){
            f = -f;
        }

        // CALCUL DE ULP
        pow32 = 0.0;
        if (fpow32 < 0) {
            fpow32 = -fpow32;
        }
```

```
        while (fpow32 < 1) {
            fpow32 = fpow32 * 2.0;
            pow32 = pow32 - 1;
        }
        while (fpow32 >= 2) {
            fpow32 = fpow32 / 2.0;
            pow32 = pow32 + 1;
        }
        pow2 = 1.0;
        while(pow32 > 0){
            pow2 = pow2 * 2.0;
            pow32 = pow32 - 1;
        }
        i = 0;
        while(i<23){
            pow23 = pow23 / 2.0;
            i = i + 1;
        }
        ulp = pow2 * pow23;
        // FIN ULP

        // CALCUL DE COSINUS EN SERIE ENTIERE
        i = 0;

        while (absx > ulp) {
            pow = pow * f * f;
            if(i == 0){
                pow = 1.0;
            }
            fact = fact * 2.0 * i * (2.0 * i - 1.0);
            if(i == 0){
                fact = 1.0;
            }
            if(i == 1){
                fact = 2.0;
            }
            sign = sign * -1.0;
            x = sign * pow / fact;

            res = res + x;
            i = i + 1;
            absx = 0.0 + x;
            if(x < 0.0){
                absx = -x;
            }
        }

        println(res);

        NB_CALC = NB_CALC + 1;
    }
}
```

Implémentation du sinus en séries entières

```
{
// Debut similaire au cosinus
     while (absx > ulp) {
        pow = pow * f * f;
        if(i == 0){
            pow = 1.0;
        }
        fact = fact * (2.0 * i - 1.0) * 2.0 * i;
        if(i == 0){
            fact = 1.0;
        }
        if(i == 1){
            fact = 2.0;
        }
        sign = sign * -1.0;
        x = sign * (pow * f) / (fact * (2.0 * i + 1.0));
        res = res + x;
        i = i + 1;
        absx = 0.0 +x;
        if (absx < 0.0) {
            absx = -absx;
        }
    }
// fin similaire au cosinus
}
```

Script comparant les valeurs deca avec la bibliothèque math python

```
import subprocess
import math
def lire_valeurs_de_sortie_deca():
    process = subprocess.Popen(["ima",
    ↪  "./src/test/deca/codegen/valid/notimpl/test_include_math.ass"],
    ↪  stdout=subprocess.PIPE)
    output, _ = process.communicate()
    output = output.decode("utf-8")  # Decodez les bytes en str
    valeurs = [float(ligne.strip()) for ligne in output.split('\n') if
    ↪  ligne.strip()]
    return valeurs

def comparer_valeurs(valeurs_deca):
    valeurs_math = []
    for i in range(10):
        valeurs_math.append(math.cos(4 * math.pi /(i+1)));
    # Comparer les valeurs
    for i in range(10):
        print(abs(valeurs_deca[i] - valeurs_math[i]))

if __name__ == "__main__":
    valeurs_deca = lire_valeurs_de_sortie_deca()
    comparer_valeurs(valeurs_deca)
```