DOCUMENTATION DE L'EXTENSION TRIGO

EQUIPE 20

GROUPE 4

Table des matières

SECTION 1

Introduction

1.1 Objectif de l'extension

La classe Math a été le choix de l'extension pour le compilateur implémenté lors du projet GL. L'objectif de cette extension est d'implémenter les fonctions trigonométriques de base. Le langage Deca utilise les nombres flottants pour représenter un réel. De nombreux algorithmes sont disponibles pour calculer les fonctions trigonométriques. Il est cependant nécessaire de prendre en compte l'imprécision qu'apporte le calcul de nombre flottant. C'est pour cela que les algorithmes ont été choisis sur deux critère : la précision et leur durée d'exécution.

1.2 Manipulation de nombres flottant

1.2.1 Nombres flottant

La représentation en binaire d'un nombre à virgule nécessite d'attribuer au nombre une structure en trois partie. Ce nombre sera constitué d'une partie significative s, d'un exposant e et d'une mantisse m. Tout nombre flottant x aura alors la structure suivante :

$$x = (-1)^s * m * 2^e$$

Nous travaillons sur 32bits. Selon la norme IEEE 754, notre nombre posséderas un bit pour le signe, 8 pour l'exposant et 23 pour la mantisse.

1.2.2 Notion d'erreur et d'arrondi

Une première approximation arrive lorsque l'on divise un flottant et que la mantisse a besoin de plus de bits qu'elle ne possède. Le nombre sera alors tronqué sur 32 bits et la valeur obtenue ne sera pas précisément celle attendue.
La majorité des erreurs vont alors provenir de la propagation des erreurs d'arrondi. Le résultats obtenu à la suite d'opération va être différents de la valeur exacte. Pour borner l'erreur obtenue, on utilise l'unité de mesure Ulp (Unit in the last place) qui permet de quantifié la distance entre un flottant et flottant le plus proche.

1.2.3 Méthodologie suivie

La première partie de notre travail sur l'extension a été la recherche bibliographique. Cela nous a permis de découvrir les différents algorithmes utiles pour implémenter les fonction trigonométriques.
Par la suite, plutôt que des les implémenter directement en Deca, et de devoir attendre la fin du projet pour réussir à exécuter les algorithmes sur notre propre compilateur, nous avons décidé de commencer par implémenter ces algorithmes en Java. Cela permet notamment de tester leurs efficacité sur un compilateur fiable et de ne pas perdre de temps à implémenter un algorithme trop peu efficace en Deca.

2

SECTION

Implementation Java

2.1 Choix des algorithmes

Lors de la recherche pour l'implémentation de fonctions trigonométriques, 3 méthodes ont été trouvées et étudiées. Ces trois méthodes sont les tables de correspondances, l'algorithme Cordic et l'approximation polynomiale. Le site [4] présente ces trois méthodes pour la fonction arc tangente. La conclusion est que l'algorithme Cordic se présente comme un algorithme plutôt décevant vis à vis de sa performance, son temps d'exécution et de sa précision par rapport aux deux autres méthodes. Le choix à donc été entre les tables de correspondance et l'approximation polynomiale. Les tables de correspondances sont très précises mais elles nécessitent une grande capacité de stockage et la récupération de données en est compliquée. Cette extension nécessite un compromis entre précision et rapidité. Le choix à donc été d'implémenter les fonctions trigonométrique par l'approximation polynomiale.
L'approximation polynomiale consiste à approcher une fonction complexe par des fonctions simples. Les fonctions trigonométriques ont des approximations polynomiales connues grâce aux polynômes de Taylor. Elles se caractérisent par des polynômes qui sont relativement simple à coder car ils nécessitent seulement des opérations arithmétiques élémentaire.
Une autre fonction à implémenter dans cette extension est la fonction ulp (unit in the last place) utilisée pour exprimer des erreurs en arithmétique virgule flottante. Pour cette fonction, il faut calculer l'espace entre deux flottants consécutif. Ce calcul se base sur le calcul de l'exposant du flottant.

2.2 Analyse théorique

2.2.1 Fonction cosinus

Le développement de Taylor de la fonction cosinus est disponible ci dessous.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

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

FIGURE 1 – Polynôme de Taylor de la fonction cosinus

Le code implémenté de ce développement de Taylor est donc le suivant. Nous avons décidé d'effectuer le développement à l'ordre 20 pour avoir une bonne approximation de la valeur voulue.

```
public float cos(float x){

    float res = 1f;
    float sign = 1, fact = 1, pow = 1;
    for (float i = 1; i < 20; i++) {
         sign = sign * -1;
        fact = fact * (2 * i - 1) * (2 * i);
        pow = pow * x * x;
        res = res + sign * pow / fact;
    }

    return res;
}
```

2.2.2 Fonction sinus

$$\sin(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1}$$
$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

FIGURE 2 – Polynôme de Taylor de la fonction sinus

Le code implémenté de ce développement de Taylor est donc le suivant. Nous avons décidé d'effectuer le développement à l'ordre 20 pour avoir une bonne approximation de la valeur voulue.

```
public float sin(float a){

    float sinx, pterm;
    float i, sign=-1,n=20;
    sinx = a;
    pterm = a;
    for(i=1;i<=n;i++){
    sinx = sinx + sign*pterm*a*a/(2*i*(2*i+1));
    pterm = pterm * a* a /(2 * i * (2 * i + 1));
    sign = -1 * sign;
    }

    return sinx;
}
```

2.2.3 Fonction arc tangente

Pour la fonction arc tangente, nous avons décidé d'avoir une grande précision sur l'intervalle [-1,1], puis en dehors de cet intervalle, la fonction implémentée est celle du polynôme de Taylor. La fonction sur [-1,1] utilise la fonction fmaf. La fonction fmaf (float a, float b, float c) effectue l'opération a*b+c. Les coefficients utilisé ont été trouvés dans [5] lors de la documentation et permettent d'avoir une précision excellente. Cependant, cette fonction n'est valable que sur [-1,1]. En effet, au delà de cet intervalle, la fonction ne suit pas la tendance de la fonction arc tangente. Nous avons donc du utilisé le développement de Taylor de la fonction arc tangente au dehors de cet intervalle. Cela permet un bon compromis entre précision et rapidité. Cependant, proche de 1, l'algorithme perd en précision.

$$\forall x \in [-1,1] \quad \arctan x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \cdots.$$

FIGURE 3 – Polynôme de Taylor de la fonction arc tangente

```
public float atan(float a) {
    if(a > 1f){
        return FastArctan(a);
    }
    if(a < -1f){
        return FastArctan(a);
    }
    float s = a * a, u = fmaf(a, -a, 0x1.fde90cp-1f);
    float r1 =                0x1.74dfb6p-9f;
    float r2 = fmaf (r1, u,  0x1.3a1c7cp-8f);
```

```
    float r3 = fmaf (r2, s, -0x1.7f24b6p-7f);
    float r4 = fmaf (r3, u, -0x1.eb3900p-7f);
    float r5 = fmaf (r4, s,  0x1.1ab95ap-5f);
    float r6 = fmaf (r5, u,  0x1.80e87cp-5f);
    float r7 = fmaf (r6, s, -0x1.e71aa4p-4f);
    float r8 = fmaf (r7, u, -0x1.b81b44p-3f);
    float r9 = r8 * s;
    float r10 = fmaf (r9, a, a);
    return r10;
}

float FastArctan(float x) {
    float sign = 1;
    if(x<0){
    sign = -1;
    }
    float A = 1.57079632679f;

    return A*sign - 4f*x/fmaf(4f,x*x,1);
}
```

2.2.4 Fonction arc sinus

Une fois la fonction arc tangente implémentée, une relation simple relie la fonction arc tangente et arc sinus (figure 4). Il a falue uniquement implémenté une nouvelle fonction : la fonction racine carrée. En effet, la racine carrée n'est pas disponible en deca, il faut donc créer une fonction pour la calculer. L'avantage de cette méthode est que la fonction arc sinus est définie sur [-1,1]. Or notre fonction arc tangente est très précise sur cet intervalle. On aura donc une très bonne précision pour notre arc sinus.

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

FIGURE 4 – Relation entre arctan et arcsin

Le calcul de la fonction racine carrée est effectué par l'algorithme de Héron [5]. Cet algorithme calcul une approximation de la racine carrée en calculant la suite définie par récurrence suivante :

$$\forall n \in \mathbb{N} \quad x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2},$$

FIGURE 5 – Suite récurrente pour le calcul de la racine carrée

On effectue un calcul à l'ordre 10 pour cette fonction. On peut alors coder la fonction arcsinus à l'aide de cette fonction nommée sqrt(float a).

```
public float arcsin(float x){
    if(x > 1f || x < -1f){
        throw new Error("the valued entered for arcsin must be in [-1,1]");
    }
    else{
        float a = (1f + sqrt(1f-x*x));
        return 2 * atan(x/a);
    }
}
```

2.2.5 Fonction Ulp

Comme expliquée dans l'introduction, la fonction ulp permet de déterminée une unité de mesure pour quantifier l'erreur. Un ulp correspond à la distance entre un flottant et son flottant le plus proche. Pour l'implémentation de la fonction ulp, nous avons tout d'abord besoin de créer une fonction qui serviras à récupérer l'exposant du flottant. Si l'exposant est supérieur à -127 (-127 étant la valeur minimale pour l'exposant) alors l'ulp est égal à 2**(e-23) (ou le nombre 23 correspond aux bits attribués à la mantisse). Il a fallu donc introduire aussi la fonction puissance.

```
public float pow(float a, int b) {
            if (b < 0 ) { return pow(1/a, -b); }
        else if ( b==0 ) {  return 1; }
            else if ( b==1 ) {      return a; }
            else if (b%2== 0) { return pow(a*a, b/2); }
            else { return a*pow(a*a ,(b-1)/2); }
    }
 public float ulp(float x){
    int e;
            if (x ==0 ) {
                    return 0;
            }
            else {
                    e = getExponent(x);
                    if (e >= -127) {
                            return pow(2.0f, e-23);
                    }
                    else {
                            return pow(2.0f, -127-23);
                    }
            }
}
```

2.3 Limite des algorithmes

Les algorithmes implémentés ont pour objectif d'approcher les valeurs réelles des fonctions voulues. Les résultats obtenus sont donc des approximations. On cherche à limiter au maximum ces approximations. Il est possible d'atteindre un niveau de précision important mais cela entraîne souvent l'augmentation de la durée de l'exécution de l'algorithme. L'objectif des algorithmes est donc de trouver un compromis entre rapidité et précision.

2.3.1 Fonction cosinus

La fonction cosinus obtient de bon résultats sur

$$[-\pi, \pi]$$

. On a en général sur cette intervalle une erreur d'au maximum 3 ulp. L'exécution de cette fonction est en moyenne de 10259 ns.

2.3.2 Fonction sinus

La fonction sinus comme la fonction cosinus obtient de bon résultats sur

$$[-\pi, \pi]$$

. On a en général sur cette intervalle aussi une erreur d'au maximum 3 ulp. L'exécution de cette fonction est en moyenne de 13666 ns.

2.3.3 Fonction arc tangente

La fonction arc tangente se présente donc sous deux algorithme selon l'intervalle dans lequel nous nous trouvons. Sur [-1,1], on remarque que notre algorithme est bien très précis. Notre erreur se trouve tout le temps sous la barre du 1 ulp. Néanmoins même si notre arc tangente semble suivre la tendance de l'arc tangente de la classe Math de java, au niveau de -1 et de 1, noter second algorithme est peu précis. Cependant, en s'écartant de ces valeurs, on retrouve une bonne précision. L'exécution de cette fonction est en moyenne de 5891 ns.

2.3.4 Fonction arc sinus

La fonction arc sinus hérite donc de la fonction arc tangente qui a précédemment prouvé sa précision. On remarque cependant que l'on a perdu en précision car notre erreur se trouve autour de 1 ulp. Cependant ce résultat reste tout de même très satisfaisant. L'exécution de cette fonction est en moyenne de 11892 ns.
Cette différence entre la fonction arctan et la fonction arcsin pourrait s'expliquer par l'erreur produite lors du calcul de la racine carrée (cf Figure 4). L'algorithme d'Héron utilisé pour l'approximation de la racine carrée a une erreur croissante. Cependant, dans

l'équation de la figure 4, le coefficient dans la racine carrée est toujours inférieur ou égal à 1. Or, notre fonction racine carrée est très précise sur cet intervalle. L'erreur ne vient donc pas de la fonction racine carrée.

2.3.5 Fonction ulp

La fonction `ulp` ne présente presque aucune erreur sauf proche de 0. Cependant l'erreur est de l'ordre de 10**-49. L'algorithme mis en place est donc extrêmement satisfaisant. L'exécution de cette fonction est en moyenne de 7013 ns.

2.3.6 Graphes

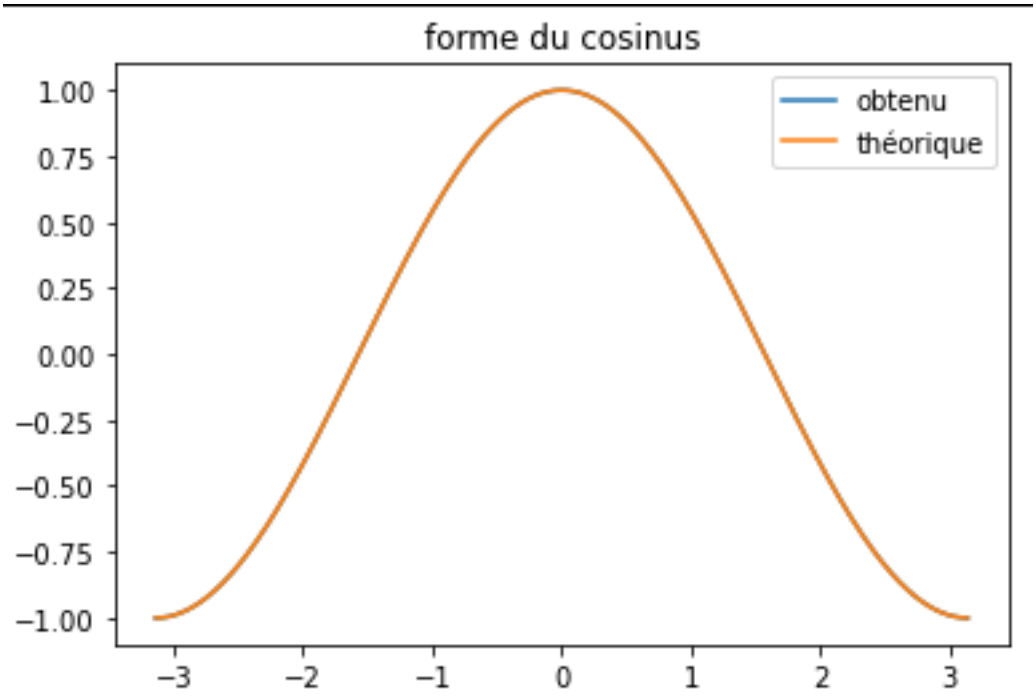

FIGURE 6 – Forme du cosinus sur [-π,π]

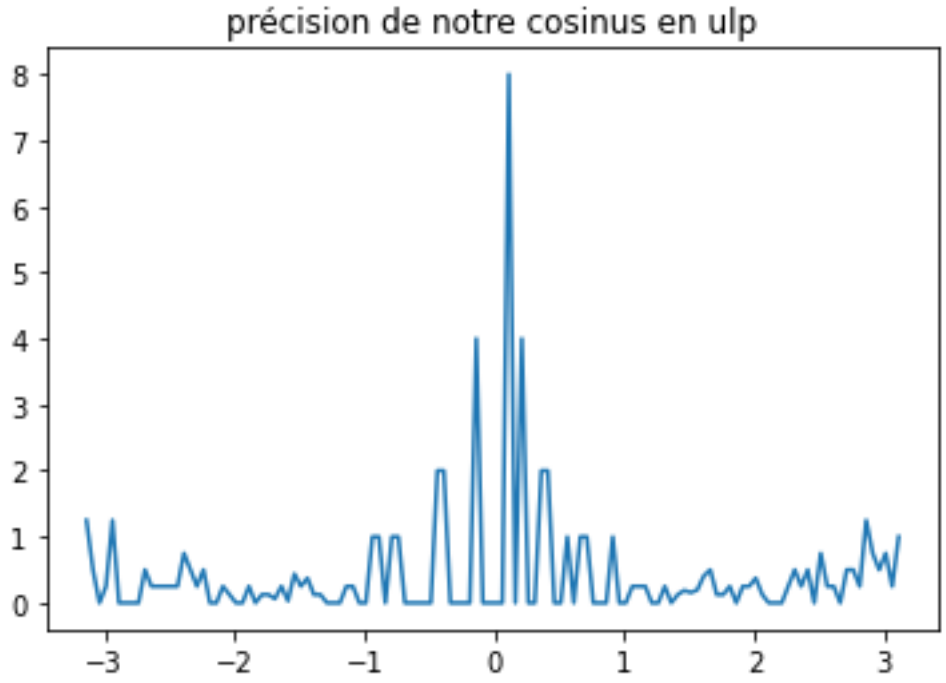

FIGURE 7 – Précision de notre fonction sinus

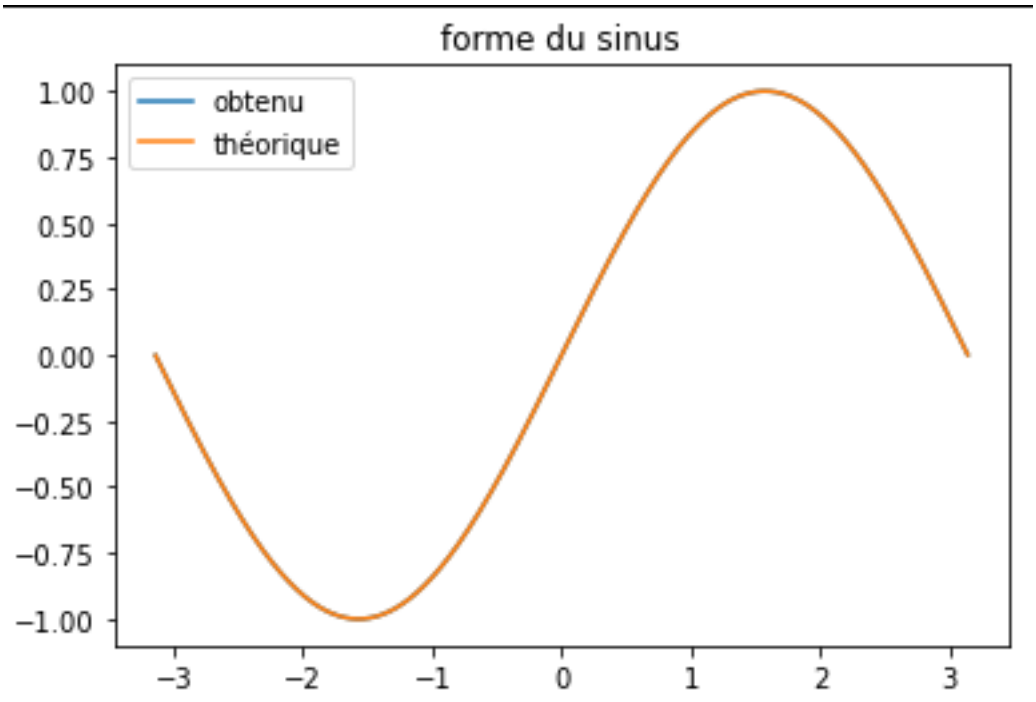

FIGURE 8 – Forme du sinus sur [-π,π]

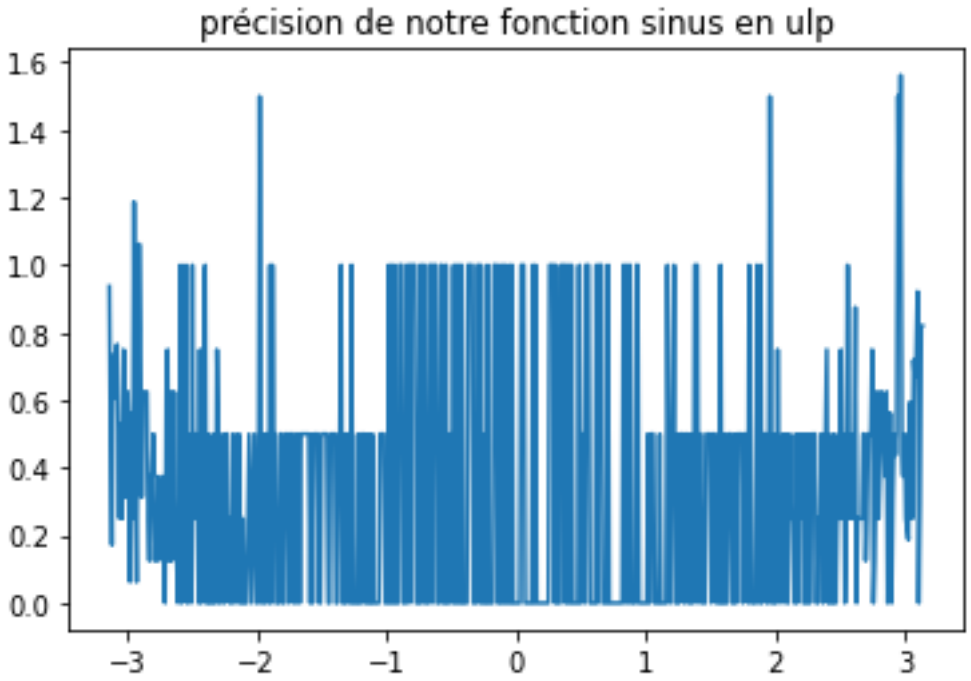

FIGURE 9 – Précision de notre fonction sinus

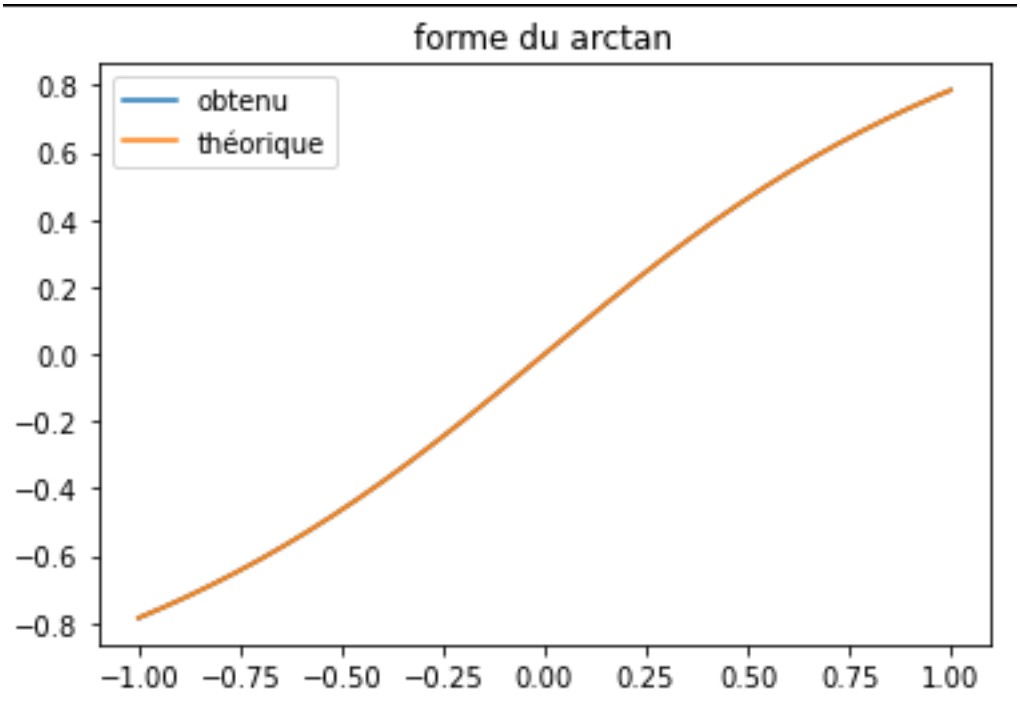

FIGURE 10 – Forme de l'arc tangente sur [-1,1]

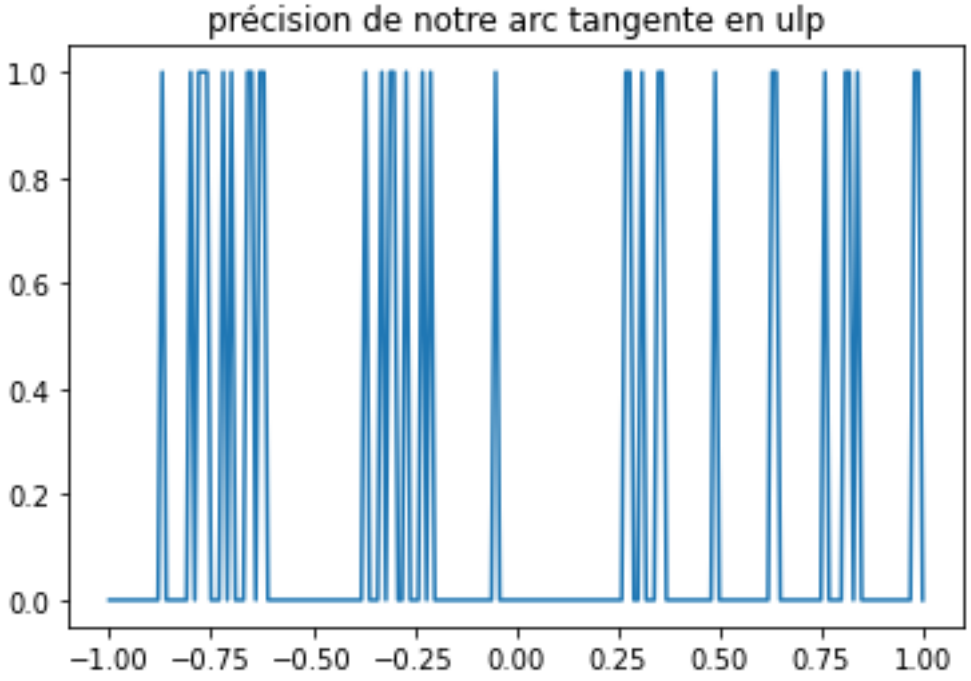

FIGURE 11 – Précision notre fonction arctan sur [-1,1]

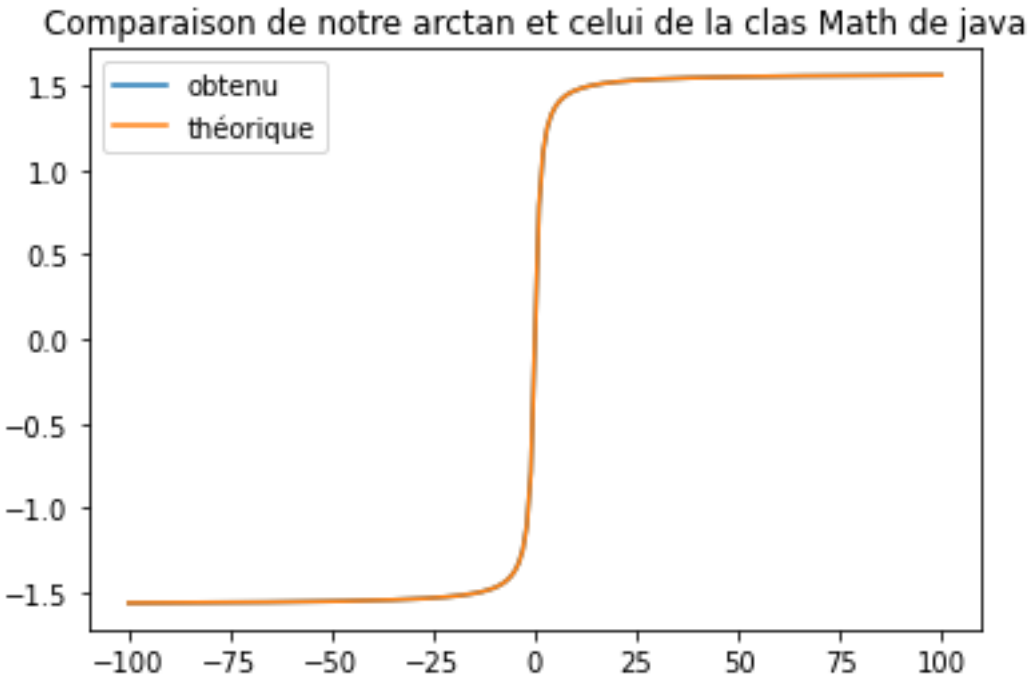

FIGURE 12 – Forme de l'arc tangente sur [-1,1]

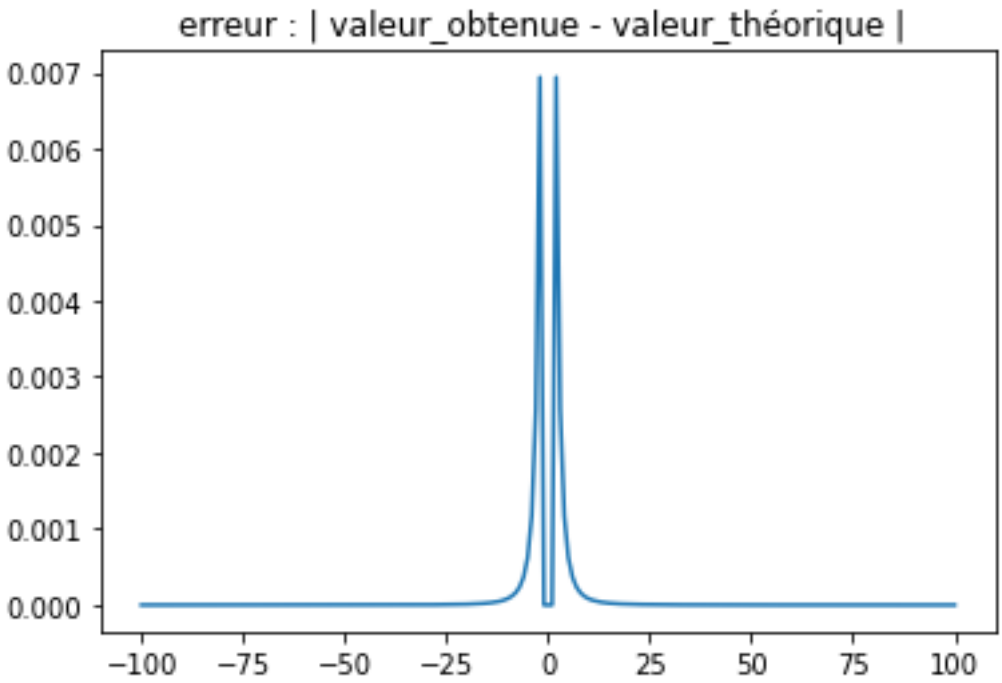

FIGURE 13 – Erreur entre notre fonction actan et celle de la classe java

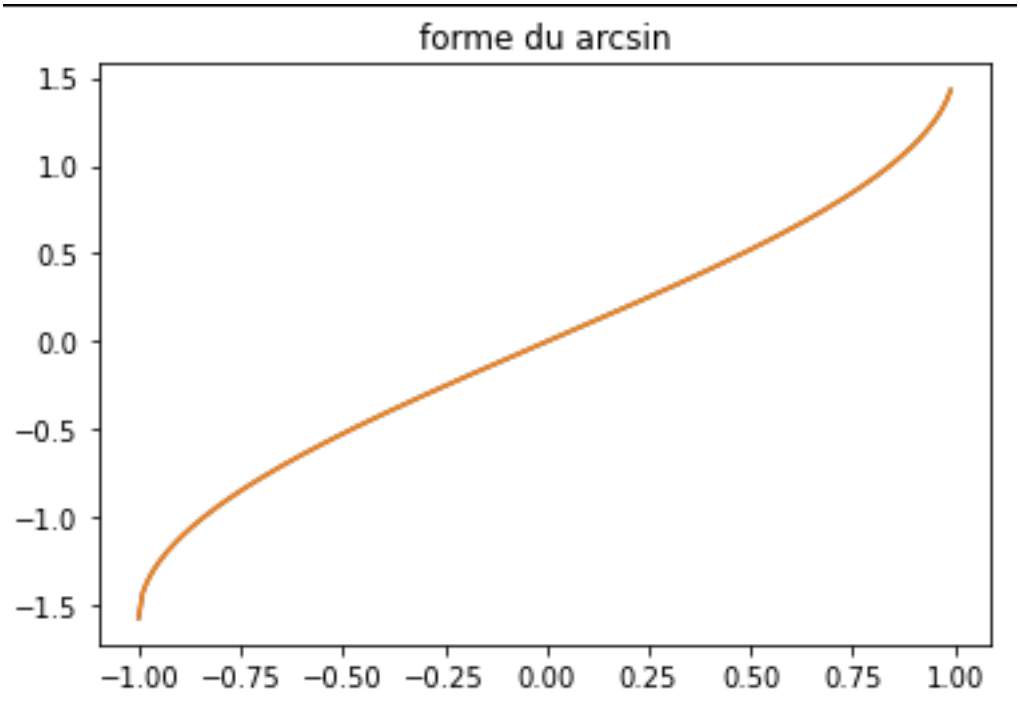

FIGURE 14 – Forme de l'arc sinus sur [-1,1]

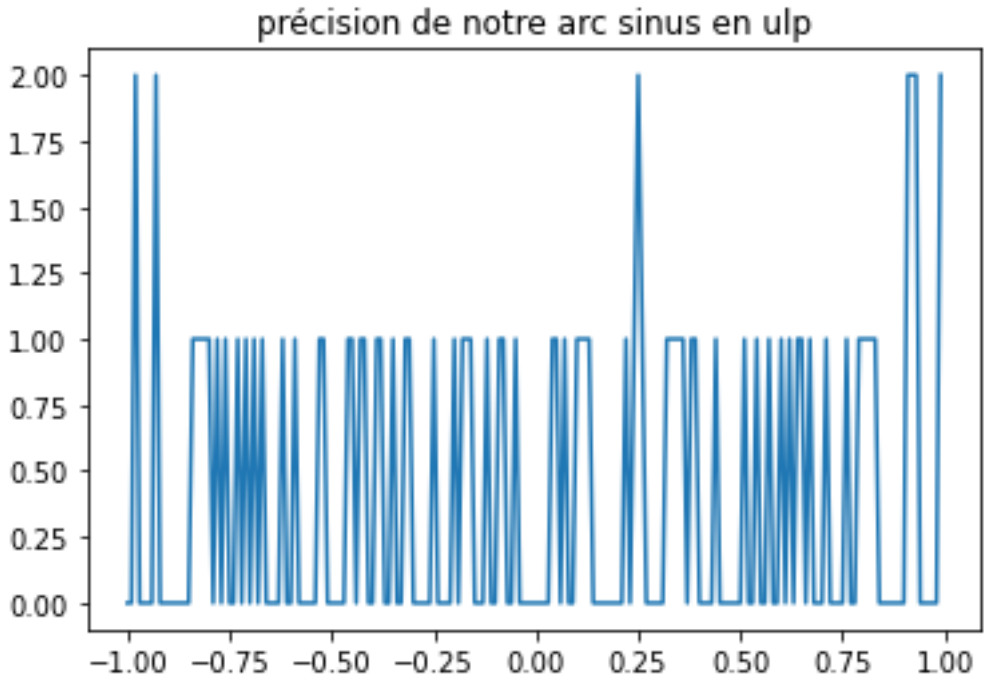

FIGURE 15 – Erreur entre notre fonction arcsin et celle de la classe java

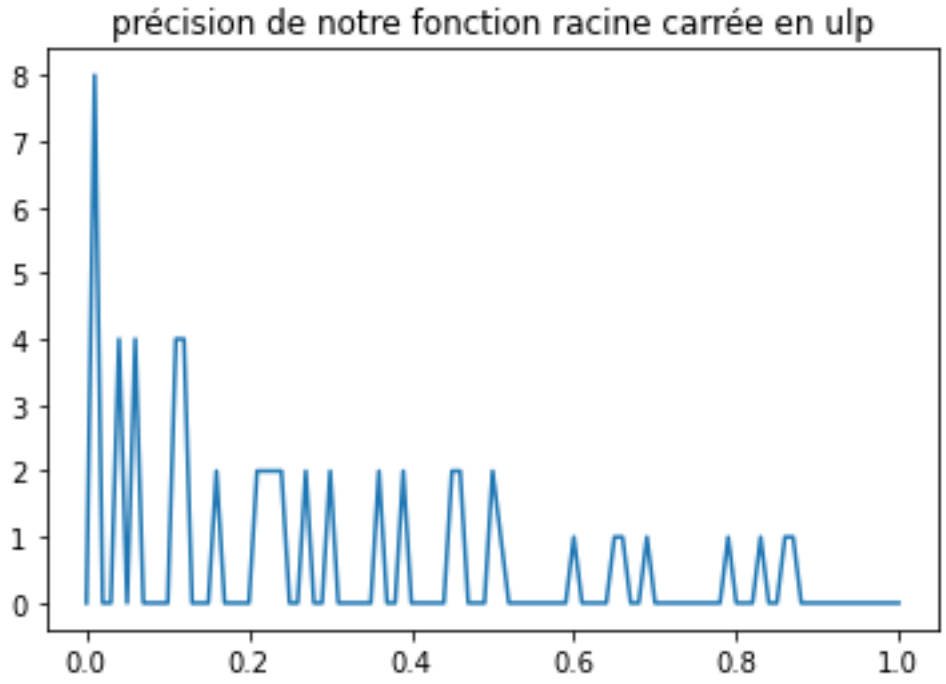

FIGURE 16 – Précision de notre fonction sqrt sur [0,1]

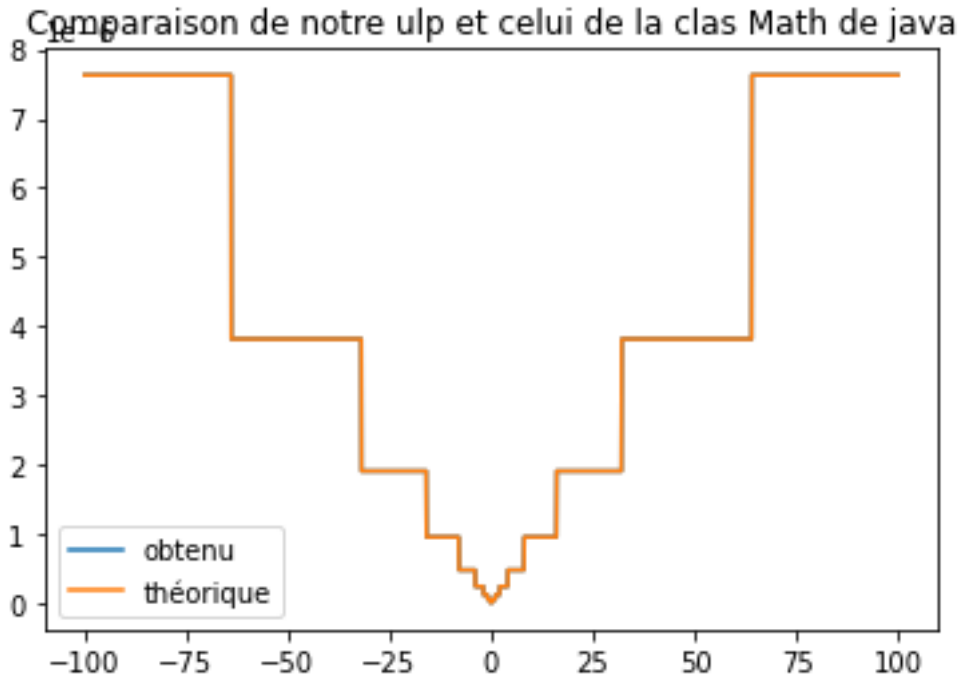

FIGURE 17 – Fonction ulp

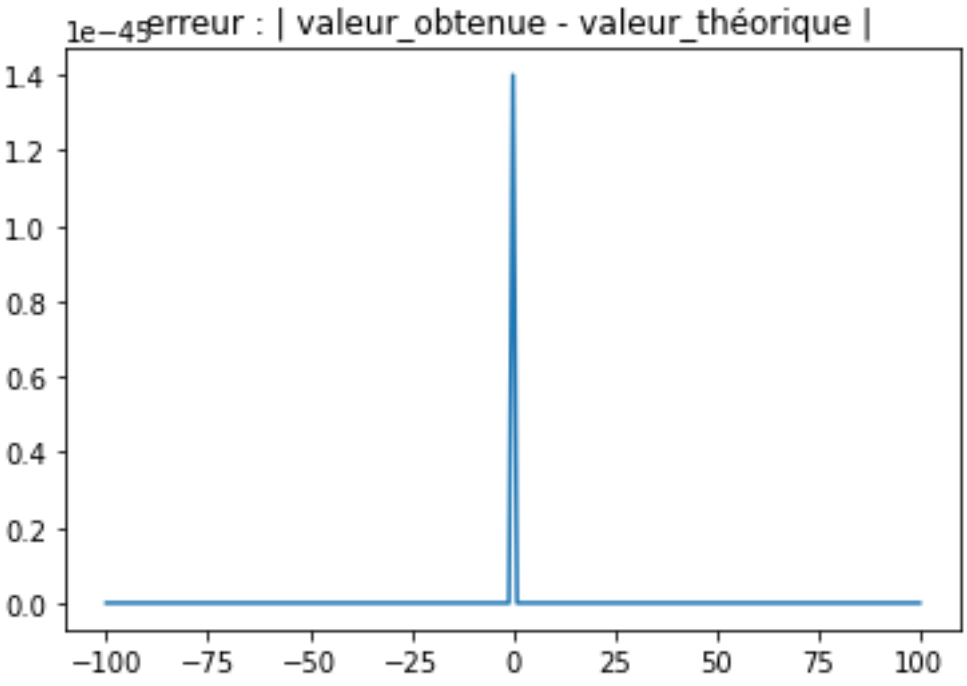

FIGURE 18 – Erreur entre notre fonction ulp et celle de la classe java

3

SECTION

Implémentation Deca

3.1 Traduction

Pour l'implémentation Deca de ces algorithmes, nous avons simplement repris les algorithmes écrits en Java et les avons réadaptés pour qu'ils soient des programmes Deca corrects. Pour cela, il a principalement fallu faire attention de bien définir toutes les variables locales utilisées en début de méthode, et remplacer les boucles `for` par des boucles `while`.
De plus, il est utile de remarquer que certains algorithmes évaluent un polynôme à partir de la méthode de Horner. Celle-ci consiste à évaluer le polynôme suivant en x_0

$$P = a_n X^n + a_{n-1} X^{n-1} + ... + a_0$$

On utilise la factorisation suivante :

$$P(x_0) = ((...((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + ...)x_0 + a_1)x_0 + a_0$$

Celle permet donc de ramener l'évaluation du polynôme à des calculs successifs de la forme $a * b + c$. Or, il existe justement une instruction assembleur nommée `FMA` réalisant ce calcul en n'effectuant qu'un seul arrondi au lieu de deux. Nous avons donc écrit une fonction `fmaf` directement en assembleur intégrer au code de la bibliothèque `Math` à l'aide de la fonctionnalité `asm`.

3.2 Résultats

Les résultats obtenus sont satisfaisant. En effet on obtient une très légère erreur lors du passage de java à deca (de l'ordre du e^-7). La fonction `ulp` n'as pas perdue en précision.

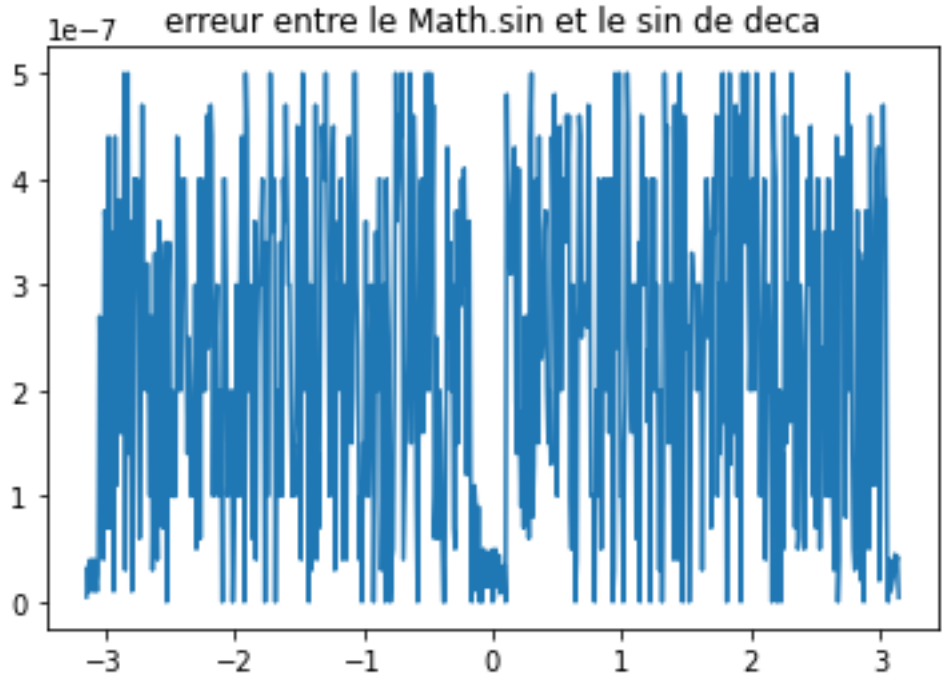

FIGURE 19 – Erreur entre notre sinus (deca) et le sinus de java.lang.Math

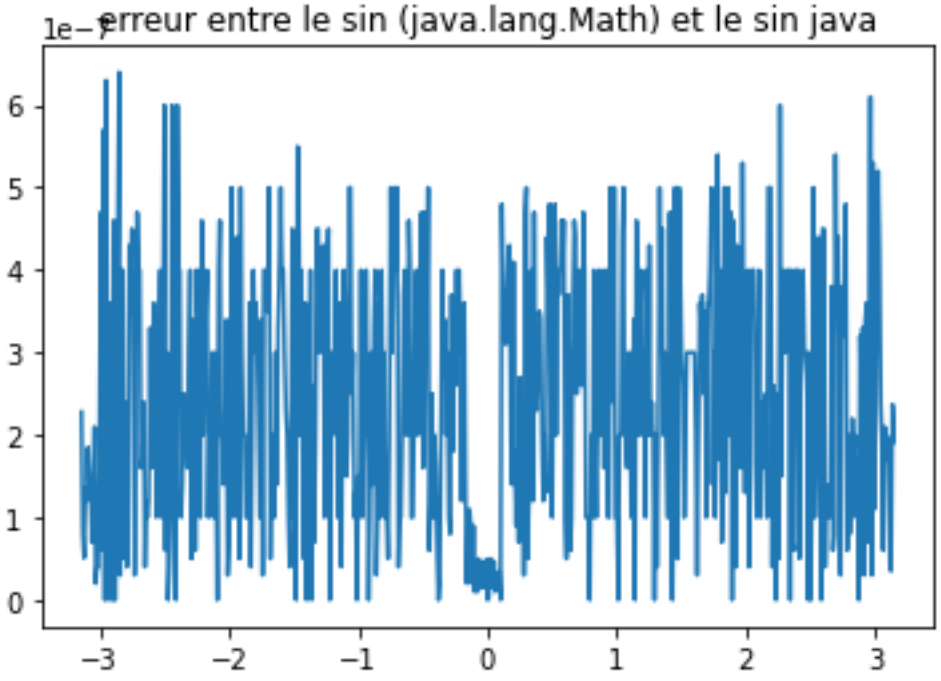

FIGURE 20 – Erreur entre notre sinus (deca) et le sinus implémenté java

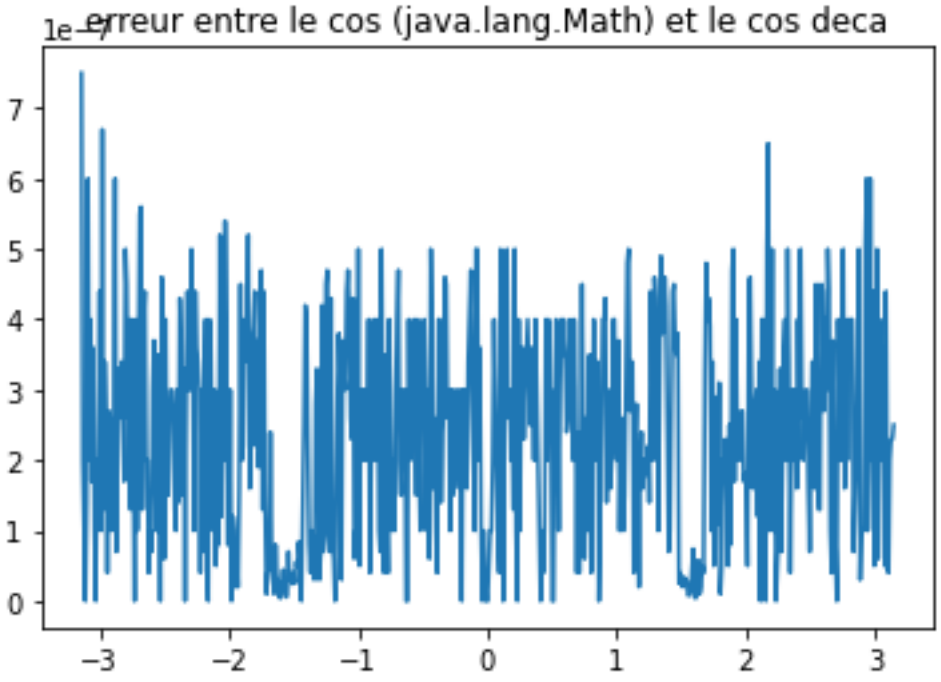

FIGURE 21 – Erreur entre notre cosinus (deca) et le cosinus de java.lang.Math

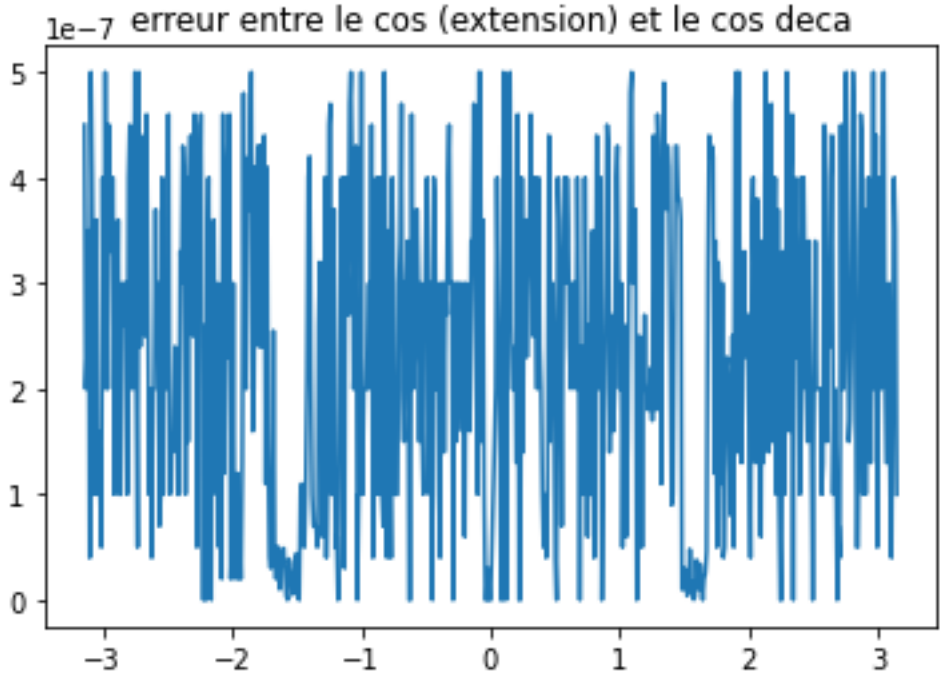

FIGURE 22 – Erreur entre notre cosinus (deca) et le cosinus implémenté java

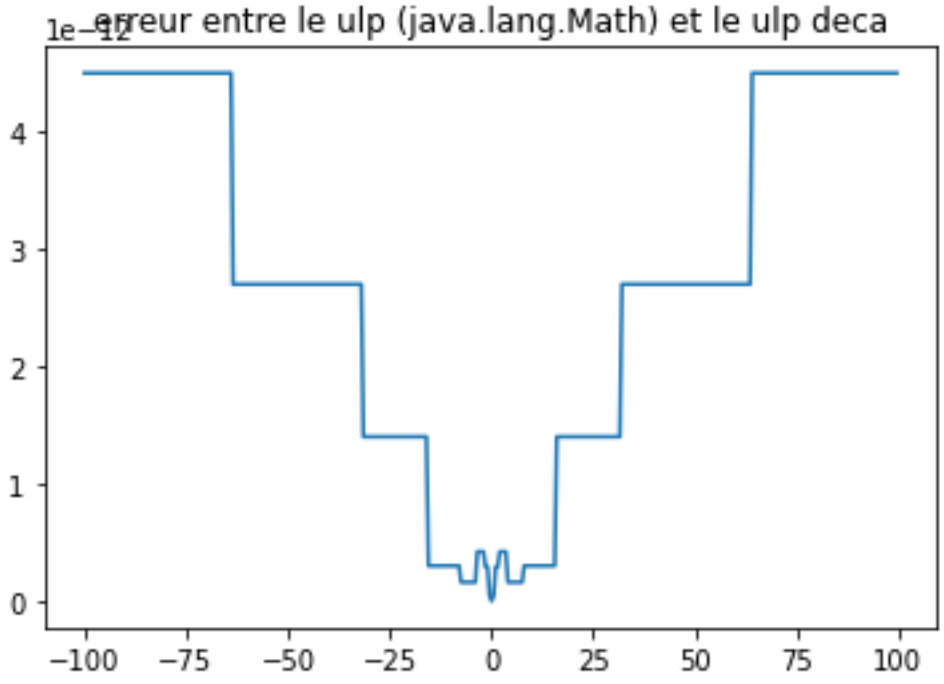

FIGURE 23 – Erreur entre notre cosinus (deca) et le cosinus de java.lang.Math

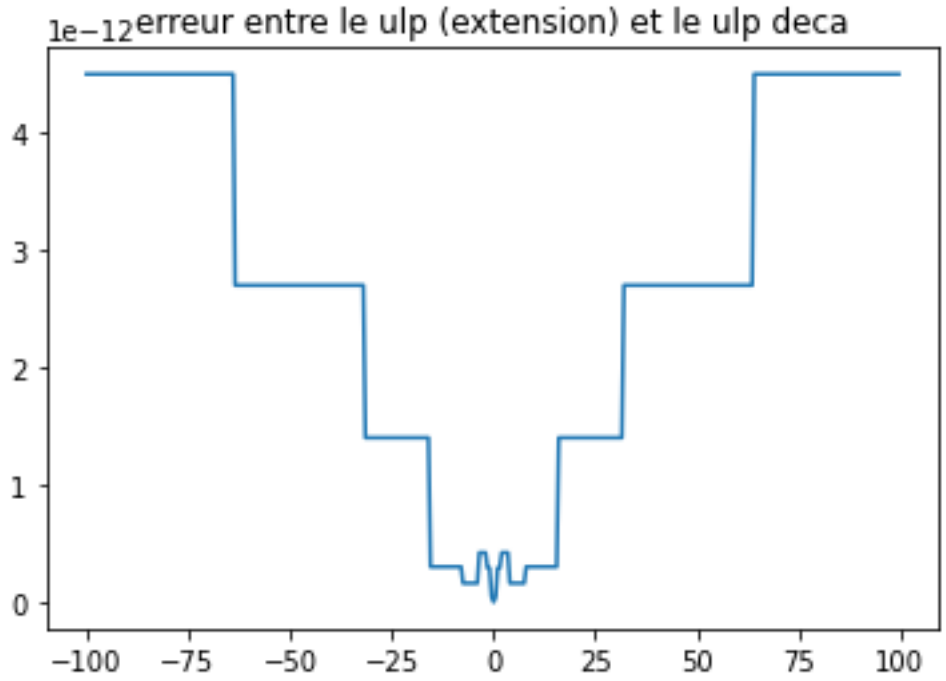

FIGURE 24 – Erreur entre notre cosinus (deca) et le cosinus implémenté java

4

SECTION

Conclusion

L'implémentation de l'extension Math a été très intéressante. La recherche de documentation puis la mise en pratique a été enrichissante. Cela a permis de mieux comprendre le calcul des nombre flottant et de comprendre les erreurs liés a leurs opérations. Cette extension n'as malheureusement pas pu être terminée. Une implémentation intéressante aurait pu être d'exploiter la périodicité de la fonction cosinus et sinus. En effet en ramenant les valeur dans $[-\pi,\pi]$, l'erreur serait minimum car notre erreur sur cet intervalle est moindre. Il aurait était aussi intéressant de réduire l'erreur de la fonction arc tangente au abords de -1 et 1.

SECTION 5

Références

[1] https://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/node48.html

[2] https://www.keil.com/support/man/docs/c51/c51_ap_floatingpt.htm

[3] https://stackoverflow.com/questions/32659336/getting-the-exponent-from-a-floating-point-in-c/32659379

[4] https://codes-sources.commentcamarche.net/source/9917-ainsi-c-calcul-d-une-racine-carree-par-algorithme-d-heron

[5] https://membres-ljk.imag.fr/Bernard.Ycart/mel/dl/node6.html