Projet Génie logiciel

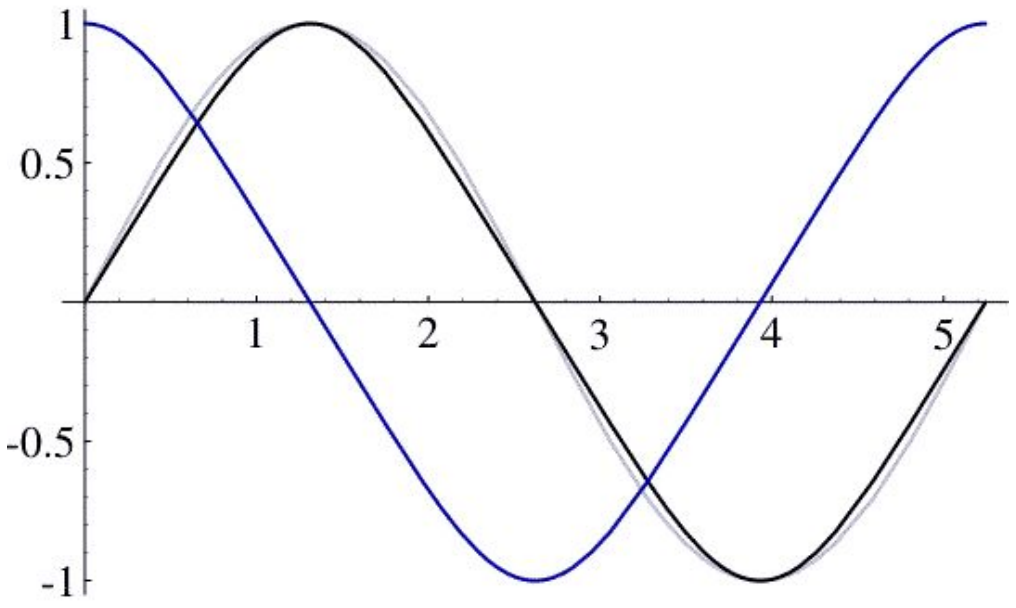
Documentation de l'extension - TRIGO

Groupe GL 23:

Hicham Darouach, Oussama Allali
Najète Benmansour, Rémi Youssef
Alexian Serrano

Encadré par:

Serwe Wendelin
Buthod-garcon Tony



Année Universitaire 2019/2020

Table des matières

I - Introduction

Trigo est une extension qui est basé sur les mathématique et comme toute les autres extensions elle représente une grande partie du projet Génie logiciel, une partie qui représente la continuité de notre compilateur et qui nous permettra d'avoir un compilateur qui contient des fonctions trigonométrique(Sinus, Cosinus, arc tangente ...) et apte á faire des calculs grâce á ces fonctions mathématiques et tout ca sans oublier l'implémentation de la fonction ULP (unit in the last place) qui est identique à la fonction Math.ulp(float) en Java et qui nous permettra de calculer l'écart avec le prochain flottant et ce pour pouvoir calculer la précision des fonction implémenté et de travailler sur leurs précision, mais cette précision est relative au fait que notre langage Deca ne contient pas les doubles, et d'autre part le nombre PI est irrationnel du coup il n'est pas présentable sur machine.

L'implémentation des fonctions trigonométrique doivent chercher la meilleure approximation possible permise par la représentation des flottants simple précision, à l'intérieur du codomaine de la fonction mathématique. Typiquement, la méthode asin doit retourner un flottant dans [- pi/2, pi/2].

Convention obligatoire: Le fichier Math.decah peut éventuellement fournir d'autres classes que la classe Math, et cette classe peut fournir des méthodes et des champs qui ne sont pas dans la liste ci-dessus, pourvu que ces nouveaux identificateurs commençant par le caractère _. Ceci permet à l'utilisateur de pouvoir écrire du code utilisant Math.decah sans risque de conflit de noms pourvu que ses propres identificateurs ne commencent pas par _.

II - Fonctions intermédiaires :

1- Racine carré :

Après les premières analyses il s'est avéré nécessaire d'implémenter une fonction racine carré qui permettra de calculer arc sinus et arc cosinus en utilisant arc tangente cf (partie …..). On a choisi d'implémenter la racine carrée par la méthode d'Héron (Algorithme de Babylone) :

Mathématiquement la suite Xn définie par récurrence tend vers la racine carrée de a.

De plus sa vitesse de convergence est quadratique comme elle vérifie la relation :

Ainsi quitte à utiliser un algorithme itératif dont la condition d'arrêt est l'égalité entre 2 termes consécutifs de la suite permet d'obtenir une très bonne approximation de la racine carrée d'un flottant cf figures 1 et 2

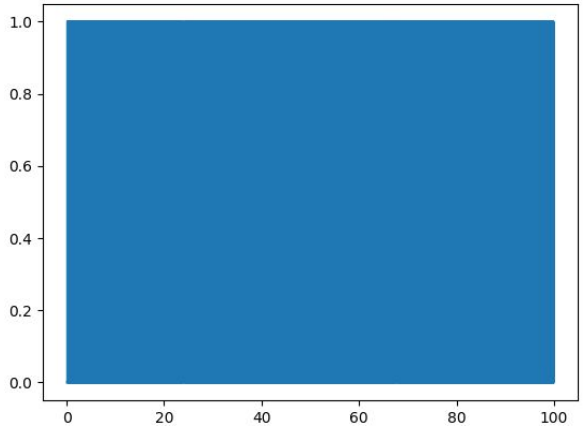


fig 1: ULP de notre racine comparer à celle du Java

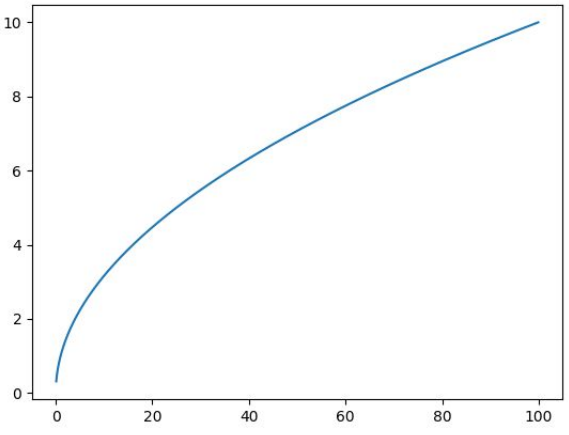


fig 2: Notre fonction sur [0,100]

2- Puissance :

Comme on cherche une approximation polynomiale de nos fonction l'implémentation d'une fonction puissance assez performante s'avère nécessaire pour réduire considérablement le coût des approximations ainsi on a choisi l'exponentiation rapide définit comme suit:

$$\text{Puissance}(x,n)= \begin{cases} 1 \text{ si } n = 0 ; \\ \text{Puissance } (x^2, n/2) \text{ si } n \text{ est pair } ; \\ \text{Puissance } (x^2, (n-1)/2) \text{ si } n \text{ est impair } ; \end{cases}$$

Cette méthode inspirée des multiplications de deux nombre chiffre par chiffre. De complexité O(log(n)) cette méthode augmente d'une façon spectaculaire la vitesse de calcul des puissance des grands nombres entiers cf figures(3,4).

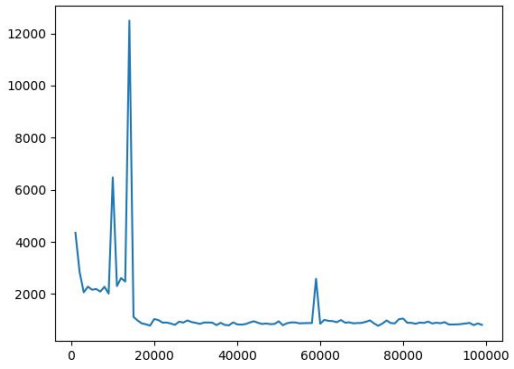


figure 3: exponentiation naïve

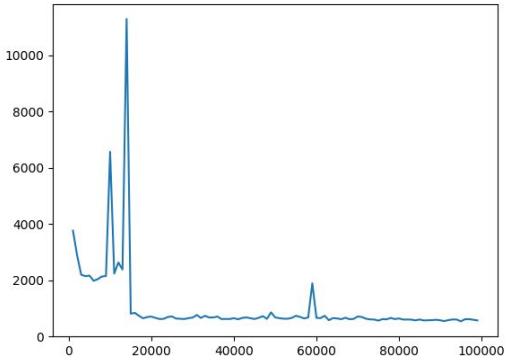


figure 4: exponentiation rapide

III-Fonctions Demandés

1- Fonction ULP :

Définition :

ULP (unit in the last place) représente l'espacement entre les nombre à virgule flottante à savoir les chiffre le moins significatif représenté.

Utilité :

Elle est utilisée pour calculer la précision dans les calculs numériques ainsi elle sera un des moyens d'évaluation de la précision de nos fonctions mathématiques.

Calcul :

Son calcul sauf dans des cas particuliers qui seront exposé ci-dessous ce fait par la formule :

ULP(x) = 2^(n-23) ;
Avec n = Log2(x) ;

Cas particuliers:

-Si f est plus petit en valeur absolue (cas du 0 inclus) que 2^149 alors on renvoie 2^149

-Si f est plus grand que le plus grand flottant représentable alors en retourne INFINITY et vu son absence en déca alors on a choisi de retourner le flottant maximal.

Implémentation :

Comme le logarithme de base 2 n'est pas utilisé pour les autre fonction et pour ulp il suffit de comparer 2^k avec x pour l'obtenir on a opter vers une version naïve du logarithme à l'intérieur de la fonction ulp cependant pour utilisation plus avancé du logarithme il serait convenable de l'implémenter séparément en utilisant le principe de la dichotomie.

Résultats:

Ainsi on a obtenu une fonction ulp identique à celle de la bibliothèque Math du langage java dans tous les cas abordés par nos tests.

b - Fonction Sinus :

a- Réduction de l'intervalle d'approximation:

La fonction sinus est périodique de période 2π ainsi on réduit notre étude à l'intervalle [-π, π] et comme cette dernière est aussi impaire l'intervalle est réduit à [0, π] et enfin on sait que sin(π-x) = sin(x) et donc il suffit d'approximer le sinus sur [0, π/2] et réduire l'angle pour calculer les grandes valeurs.

b - Approximation sur [0, π/2]

Formule Taylor :

Comme en travail sur un intervalle relativement petit et sur sinus qui est indéfiniment dérivable une première intuition suggère l'approximation de Taylor en 0 vu que les coefficients d'ordre n de Taylor en valeur absolue sont des inverses de factorielle(n):

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots + \frac{(-1)^n x^{2n+1}}{(2n+1)!} + o(x^{2n+1}) \ .$$

L'algorithme de Cordic:

CORDIC abréviation de calcul numérique par rotation de coordonnées est un algorithme de calcul des fonctions trigonométriques utilisé dans les calculatrices.Ce dernier offre une possibilité d'approximer sinus sur les partie de l'intervalle [0, π/2] où le polynôme de Taylor en 0 n'est pas assez précis vu que Taylor est une approximation locale.

Cet algorithme repose sur le principe de calculer le sinus en passant par des rotation consécutives d'un vecteur dont l'origine est l'origine du repère et de norme 1 on procédant par des petites rotations le sinus d'un angle θ est l'ordonnées du vecteur quand l'angle entre le vecteur et l'axe des abscisses est égal à θ.

Tests et validation :

On implémente l'approximation par les polynômes de Taylor de sinus entre [0,π/2] et On compare avec le sinus du java.On constate donc des ULP entre 0 et 2 ce qui est implique une bonne précision on a donc une bonne approximation du sinus sur [0, π/2]. (cf figures)

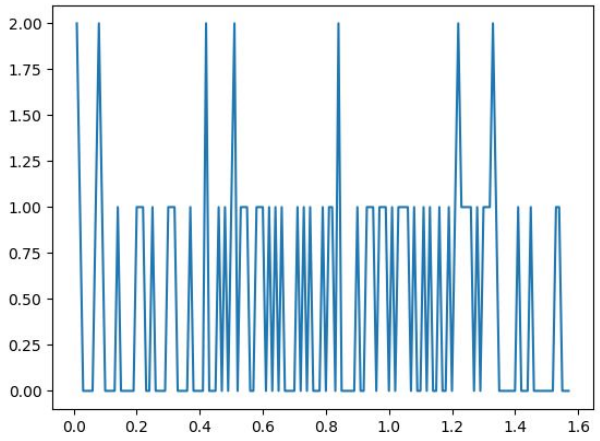


figure 5: ULP de notre sinus comparer à celui du java

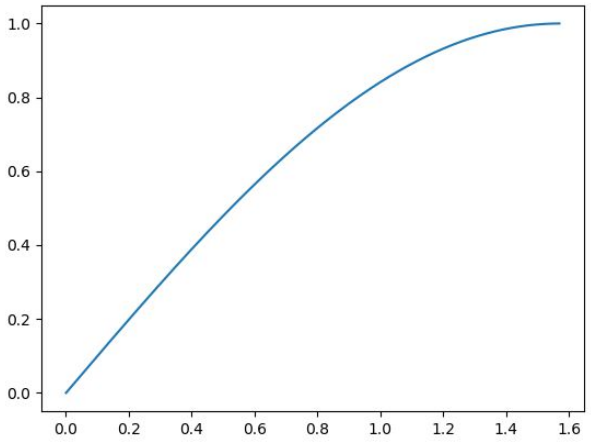


figure 6: Notre fonction sur [0, π/2]

c - Résultat sur [-π, π] :

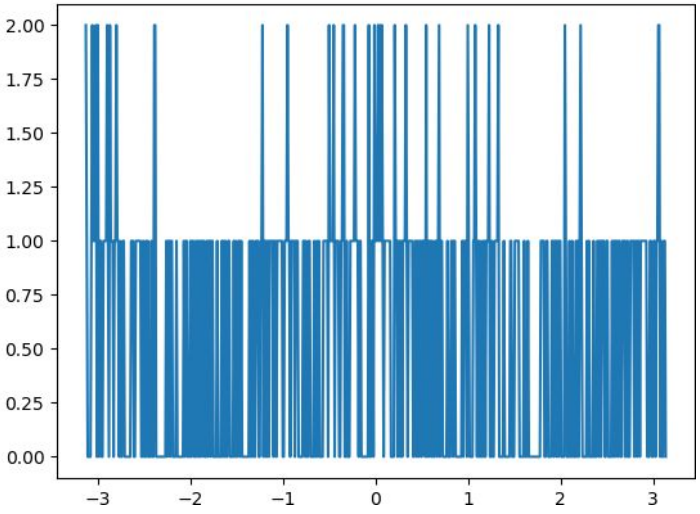


figure 7: ULP de notre sinus comparer à celui du java

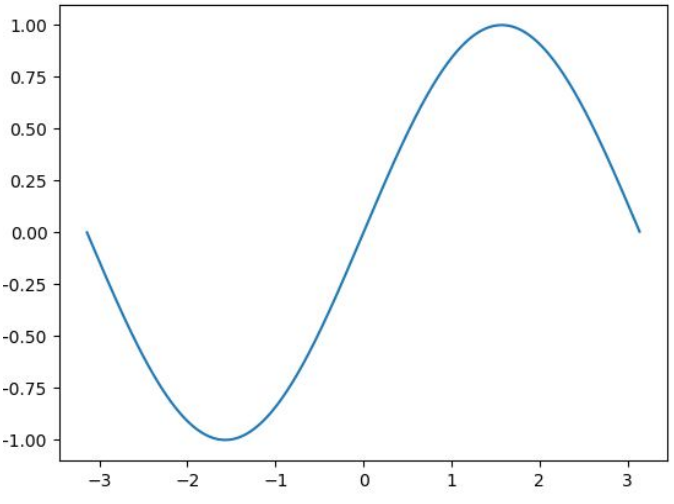


figure 8: Notre fonction sur [-π, π]

On utilisant des formules de transformation trigonométriques on a abouti à un résultat satisfaisant sur cet intervalle cf figures 7-8

Fonction cosinus

a - Réduction de l'intervalle d'approximation:

La fonction cosinus est périodique de période 2π ainsi on réduit notre étude à l'intervalle [-π, π] et comme cette dernière est aussi paire l'intervalle est réduit à [0, π] et enfin on sait que cos(π-x) = -cos(x) et donc il suffit d'approximer le cosinus sur [0, π/2] et réduire l'angle pour calculer les grandes valeurs.:

* Implémenter la série de taylor du cos définie par :

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots + \frac{(-1)^n x^{2n}}{(2n)!} + o(x^{2n}) \,.$$

* Utiliser les propriétés trigonométriques pour passer du sinus au cosinus et comme on a implémenter un bon sinus on aura par la suite un bon cosinus :

Comme les deux méthodes ont quasiment la même complexité on a choisi de tester la deuxième méthode qui fonctionnelle permettra la factorisation du code.

b - Test et validation :

On implémente l'approximation du cosinus de x par le sinus de π/2 - x et on comparant avec le sinus du java.On constate donc des ULP entre 0 et 2 ce qui est implique une bonne précision on a donc une bonne approximation du cosinus sur [0, π/2] et un code factorisé. (cf figures)

c - Résultat sur [0, π/2] :

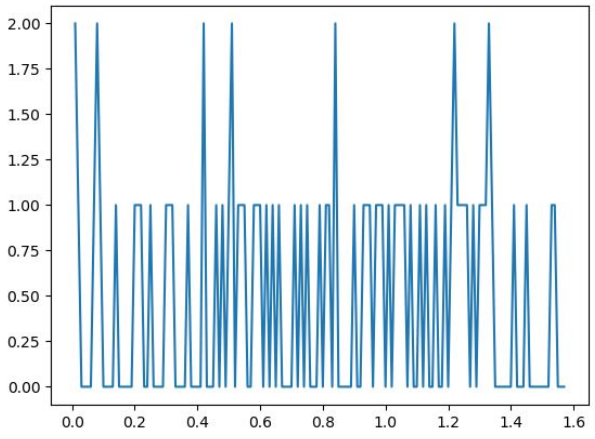


Fig 9: Les ulp de notre cos comparé a celui du java

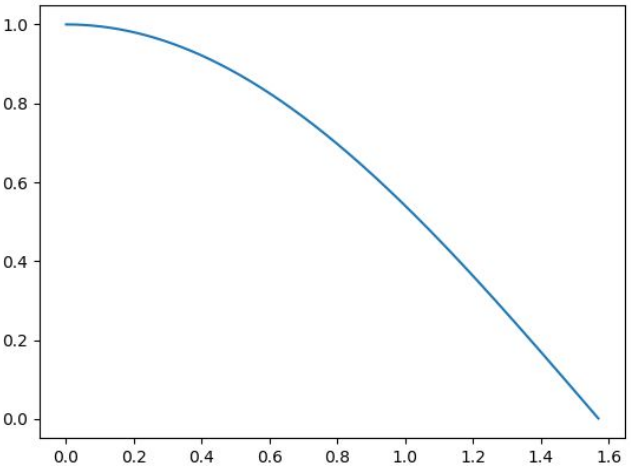


Fig 10: Notre cos sur [0,π/2]

d - Résultat sur [-π, π] :

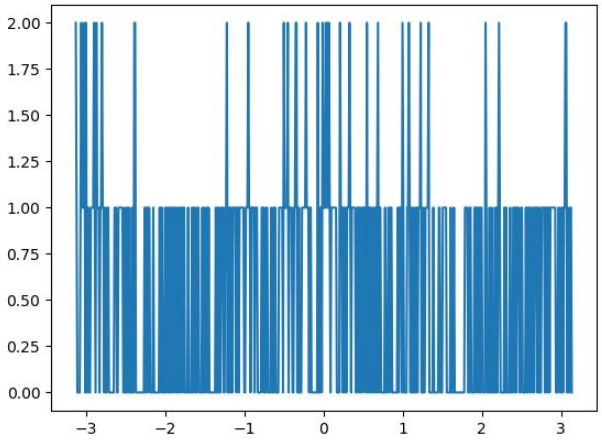


Fig 10: Les ulp de notre cos comparé a celui du java

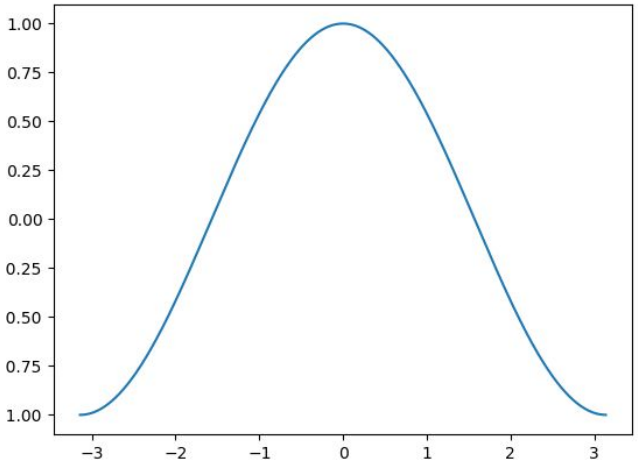


Fig 11: Notre cos sur [-π,π]

Fonction Arc Tangente :

a- Réduction de l'intervalle :

On sait que arctangente est une fonction impaire on peut donc restreindre notre approximation au valeur positive de plus une simple dérivation prouve que pour x>0 on a :

$$\text{Arctan}(x) + \text{Arctan}(1/x) = \pi/2;$$

Et par conséquent on peut réduire l'intervalle d'étude a [0,1].

b - Approximations:

- Polynômes de taylor:

On est de nouveau devant une fonction indéfiniment dérivable à approximer sur un un intervalle centré en 0 et de plus les coefficients du polynôme de taylor d'ordre n est en valeur absolue 1/n dérivable une des possibilités à implémenter est l'approximation de Taylor :

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} + \cdots + \frac{(-1)^n x^{2n+1}}{2n+1} + o(x^{2n+1})$$

- Somme de Riemann:

les **sommes de Riemann** sont des sommes finies approchant des intégrales.Et plus précisément la somme de Riemann d'une fonction sur une intervalle I=[a,b] converge vers son intégrale sur I cf figure ... ainsi la somme de Riemann de $1/(1+x^2)$ permet d'approximer l'arctangente.

$$\frac{1}{n}\sum_0^n f\left(a+\frac{b-a}{n}k\right) \xrightarrow{n\infty} \frac{1}{b-a}\int_a^b f(t)d(t)$$

L'algorithme de Cordic:

De la même maniére expliqué ci-dessus l'approximation de Cordic permet d'obtenir l'arctangente d'un nombre avec la même implémentation juste en changeant les coefficient d'initialisation.

Comme les fonctions étudié précédemment on commence par tester la précision de la méthode de Taylor cette dernière a produit une assez bonne précision sur [0,1] avec quelques problèmes au voisinage de 1 (l'erreur en ULP monte de 2 vers 7) cependant l'algorithme de cordic ou la série de Riemann n'ont pas donné un résultat plus précis et pour le passage vers ℝ aucun problème n'est à signaler.

c- Résultat sur [0, 1] :

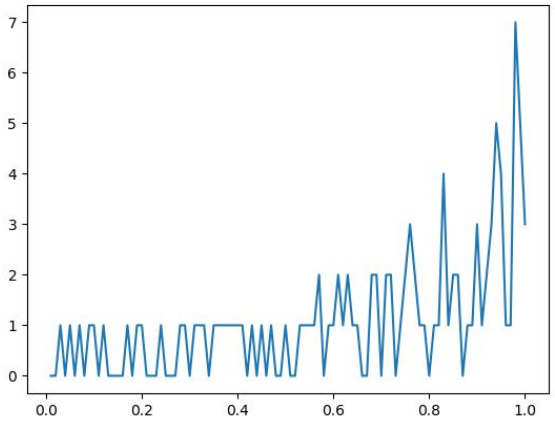


Fig 12: Les ulp de notre arctan comparé a celui du java

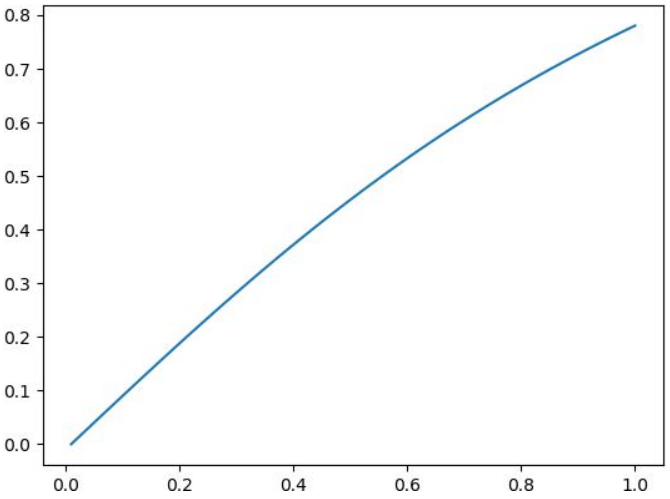


Fig 13: Notre cos sur [0,1]

c- Résultat sur ℝ :

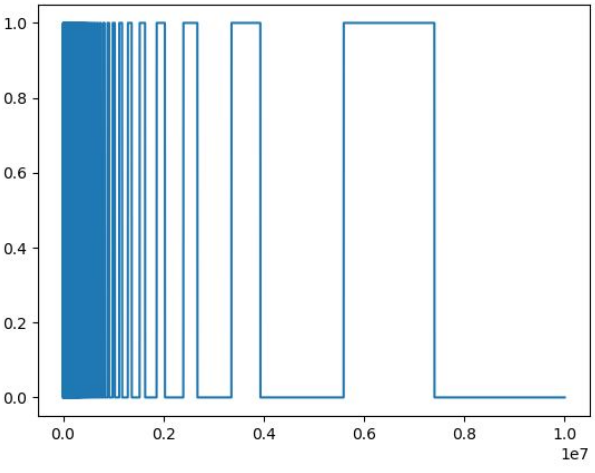


Fig 14: Les ulp de notre atan comparé à celui du java

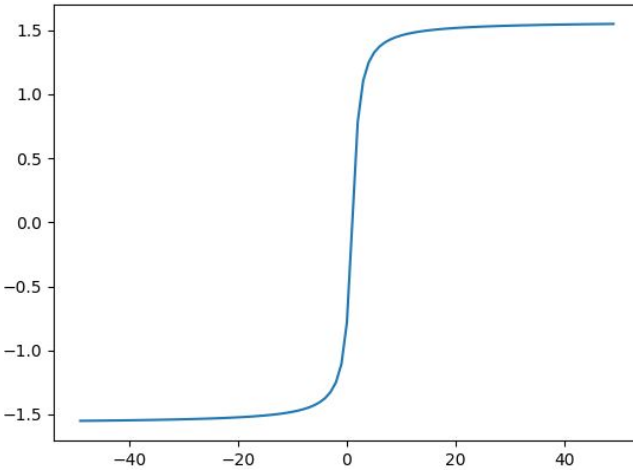


Fig 15: Notre atan sur [-40,40]

Fonction Arcsinus

Le même dilemme de la partie concernant les fonction trigonométrique se pose pour l'implémentation de leurs réciproque.

Principalement deux approche sans possible:

- Approximer l'arcsinus par les série de taylor selon la relation:

$$\operatorname{Arcsin} x = x + \frac{1}{2}\frac{x^3}{3} + \cdots + \frac{1 \times 3 \times \cdots (2n-1)}{2 \times 4 \times \cdots \times 2n}\frac{x^{2n+1}}{2n+1} + O\left(x^{2n+3}\right)$$

- Se ramener au cas de l'arctangente par la relation:

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

Cependant comme on a implémenté une bonne arctangente et le calcul des coefficient du polynôme de taylor est plus complexe alors on a opté vers la deuxième méthode.Et comme précédement on l'a comparé avec l'arcsinus du java

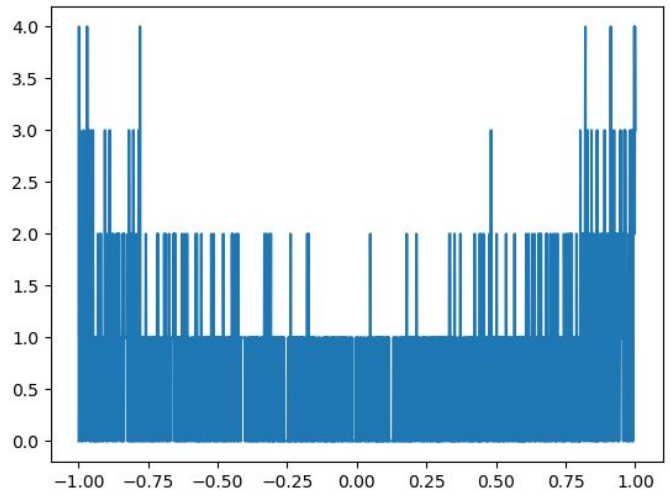


Fig 16: Les ulp de notre asin comparé a celui du java

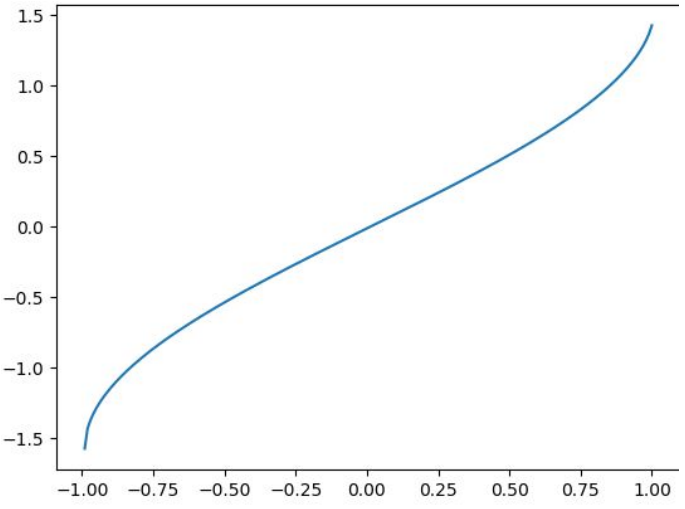


Fig 17: Notre asin sur [-1,1]

Fonction Arccosinus

1 - Réduction de l'intervalle:

On a la relation :

arccos(x) + arccos(-x) = π

On peut donc réduire notre intervalle à [0,1]

De même que l'arcsinus deux approche sont possibles pour implémenter l'arccosinus:

- Se ramener au cas de l'arcsinus par la relation:

  arcsin(x) = π/2 - arccos(x)

- Se ramener au cas de l'arctangente par la relation:

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

Cependant comme on a implémenté une bonne arctangente et le calcul de l'arcsinus passe par l'arc tangente on a opté pour la deuxième méthode afin de minimiser l'erreur.

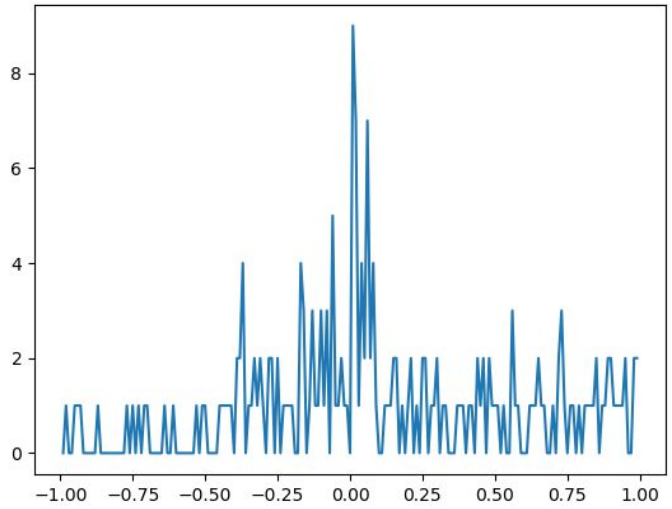


Fig 18: Les ulp de notre acos comparé a celui du java

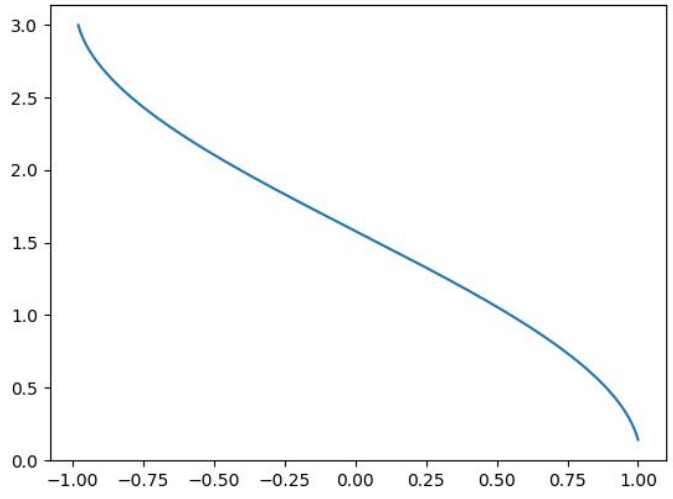


Fig 19: Notre acos sur [-1,1]

<u>La réduction d'angle:</u>

a-nécessité:

Cette partie représente la partie la plus difficile de l'extension Math. En effet, Les fonction trigonométriques implémentées ne sont fonctionnelle que entre [-π,π] il faut donc ramener un angle quelconque à cet intervalle pour pouvoir calculer son sinus.

b-Validation:

Pour la validation en va procéder en deux manières calculer la valeur absolue de l'erreur entre le sinus de l'angle réduit et le sinus du vrai angle en java pour des grands nombres et calculer la différence en ulps entre notre sinus et le sinus du java.

c-Algorithmes:

- Version naïve:

  La première idée était d'effectuer des soustractions ou bien des additions consécutives pour se ramener à l'intervalle [-π,π] cette version est très simple à implémenter cependant sa complexité est élevée pour les grands nombre et les erreurs s'accumulent à cause de l'erreurs causé par la différence entre notre π flottant et le π mathématique.

- Amélioration pour la complexité:

  On implémente une version plus rapide en utilisant la partie entière ainsi on réduit l'angle x par la relation :

  $x' = x - floor(x/\pi)$

  ou floor est la fonction qui renvoie la partie entière.

  Cette version est plus rapide mais on constate le problème du voisinage des multiples de π ou l'erreur explose.

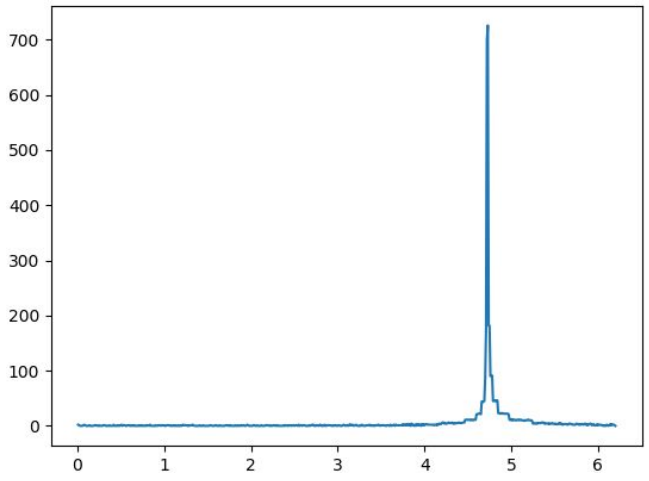


Fig 20: Les ulp de notre sin entre 0 et 2π

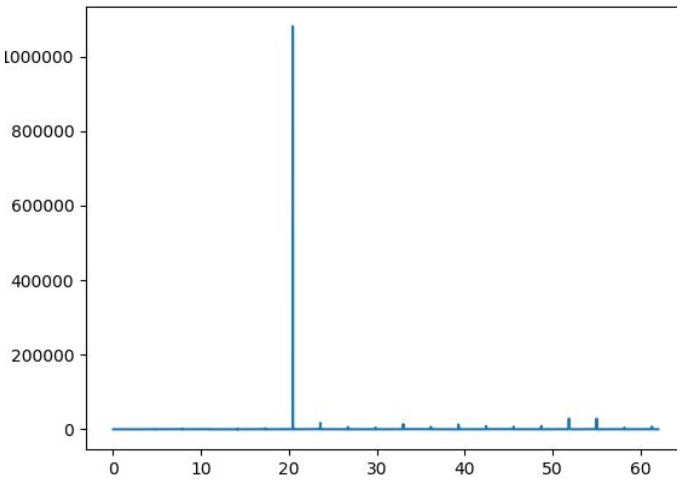


Fig 21: ULP entre 0 et 20π on note que les pics correspondent a des voisinages de π

.

- Méthode Cody and Wait (amélioration du π flottant):

  Cette méthode consiste à corriger l'erreur entre le π réel et le π flottant par un flottant qu'on notera par la suite π' où

  π' = (float) (π - πf);

  avec πf : Le flottant le plus proche de π.

  Cette méthode réduit considérablement l'erreur cependant l'explosion de l'erreur continue cette dernière est dû au grands nombre de flottant autour du zéro (>40% des flottants sont inférieurs en valeur absolue à 2^-23) ainsi une petite erreurs implique une explosion en ulp.

  Faute du temps ce problème n'a malheureusement pas pu être réglé cependant sa gestion pourrait être plus simple si on disposait de tableau de valeur (extension tab) où on pouvait stocker un grand nombre de chiffre de la représentation décimale de π.

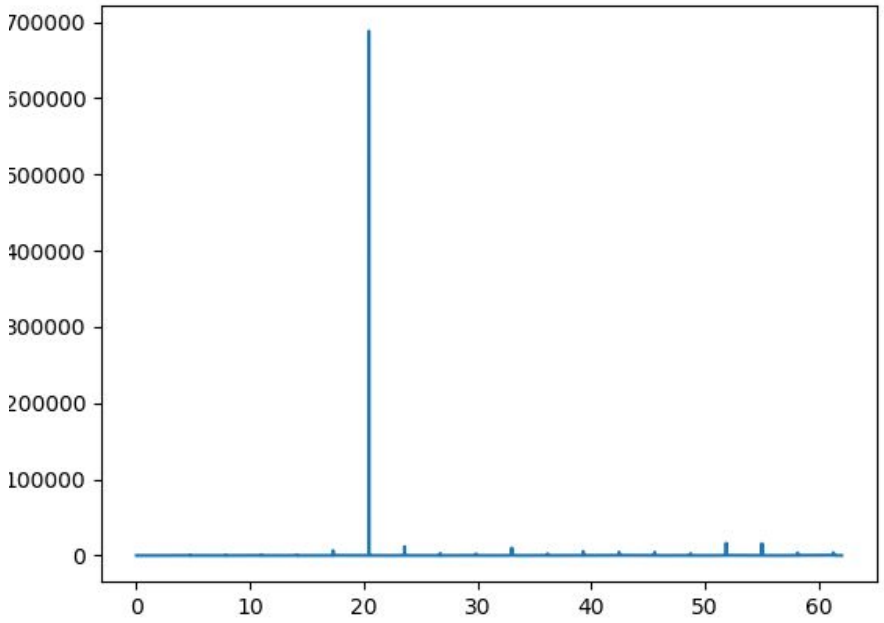


Fig 22 : Réduction de l'ULP grâce à la méthode cody and wait

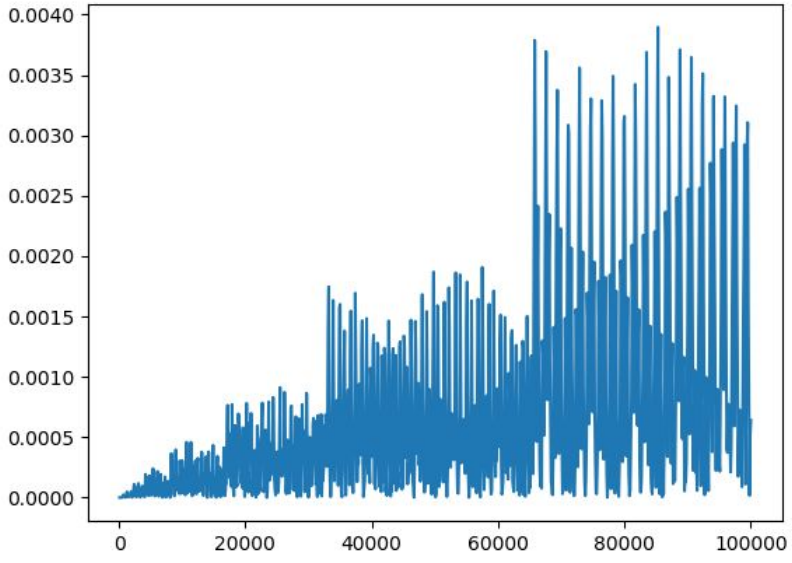


Fig 23 : Erreur entre notre sinus et le sinus du java

Conclusion

Globalement, l'implémentation des fonctions trigonométrique arc tangente, cosinus, , arc sinus et sinus ont été réalisé avec succès avec une précision correcte, malheureusement on pas eu assez de temps à la fin pour pouvoir tester et déboguer l'extension vu que la partie C posé quelques problèmes lors de l'intégration de l'extension.

Concernant la réduction d'angle et comme cité dans la documentation une méthode s'est avéré bonne mais qui malheureusement nécessite l'implémentation des tableaux et qui n'était pas envisageable au moment oú on l'a découverte vu le peu de temps qui nous restait à ce moment là.

La réalisation de projet et plus précisément cette partie nous a permis de comprendre des subtilités du calcul informatique et que ceci peut engendrer des imprécisions immense pour des grands nombres.

Bibliographie:

[1]: https://ljk.imag.fr/membres/Carine.Lucas/TPScilab/JMMuller/ulp-toms.pdf

[2]:https://www.math.univ-toulouse.fr/~jgillibe/enseignement/MHT204_chap4.pdf

[3]:https://www.apmep.fr/IMG/pdf/cordic.pdf

[4]:https://fr.wikipedia.org/wiki/Exponentiation_rapide

[5]:http://perso.ens-lyon.fr/jean-michel.muller/Daumas_M.pdf