Introduction

Nous avons pris la décision d'implémenter une extension **trigonométrique** afin de faciliter les calculs mathématiques que souhaiteront faire les utilisateurs. La trigonométrie est une branche des mathématiques qui s'applique à divers domaines tels que les télécommunications, l'acoustique, la cartographie...

Notre extension apporte un réel intérêt puisque cela étend les domaines d'utilisation de notre compilateur Deca.

Ce document a pour but d'apporter une analyse scientifique et des précisions sur l'extension.

Sommaire

I. Analyse Bibliographique

Nous avons aussi étudié le code Java des fonctions cos et sin de la bibliothèque maths. Cependant le code Java ne répondait pas à notre besoin. Les objets manipulés en Java ne sont pas des flottants mais des doubles ce qui ne s'applique pas au langage Java.

Nous n'avons pas trouvé de solution efficace dans la littérature pour implémenter les méthodes cos et sin avec des valeurs d'entrées éloignées de 0. Ainsi avec l'inspiration du code Java nous avons développé notre propre algorithme qui permet de calculer le modulo de 2π de tous les flottants afin de se ramener à un petit flottant entre 0 et 2π.

Parmi les liens présents en webographie nous nous sommes principalement inspirés des algorithmes donnés. Les sites internet étaient très bien documentés et nous avons pu facilement comparer les différentes solutions proposées pour sélectionner celles qui répondaient le mieux à nos objectifs. Nos objectifs algorithmiques étant dans un premier temps la précision des calculs, puis le faible taux de fluctuation de l'erreur et enfin la complexité de l'algorithme en temps.

Nous nous sommes appuyés sur la méthode babylonienne pour calculer la racine carrée qui donne un résultat à 1 ULP près.

Pour arcsin nous nous sommes référés à l'article du chercheur W. Randolph Franklin afin d'implémenter en algorithme sa méthode de calcul qui fourni un taux d'erreur minimal en utilisant Tchebyshev. Nous obtenons ainsi une interpolation de la fonction arcsin satisfaisante.

II. Spécification de l'extension

Trigonométrie : math.decah

Nous avons implémenté une extension mathématique trigonométrique pour résoudre des calculs avec des fonctions sinusoïdales usuelles:

- sin
- cos
- arcsin
- arctan

La méthode ULP permet de contrôler la précision du résultat lors de la validation. Un ULP d'une valeur flottante est la distance positive entre la valeur donnée et la valeur suivante qui est plus grande en amplitude.

Signatures des méthodes math.deca

```
float sin(float f)
float cos(float f)
float asin(float f)
float atan(float f)
float ulp(float f)
```

Utilisation de math.decah

Les étapes pour utiliser les méthodes trigonométriques :

- Ajouter la bibliothèque math.decah avec : `#include "Math.decah"`
- Instancier la classe Math avec : `Math m = new Math();`
- Appeler les méthodes souhaitées avec les paramètres nécessaires : `m.sin(3.18)`

III. Algorithmes et structures de données

1. Algorithme d'approximation des fonctions cosinus et sinus

Pour calculer des estimations numériques des fonctions trigonométriques usuelles, la série de Taylor (cf. wikipédia) est appliquée pour des paramètres compris entre 0 et π/4. Néanmoins, l'utilisateur est libre de rentrer des valeurs supérieures à π/4 ou inférieures à 0 donc il faut s'assurer de suffisamment **moduler** le paramètre rentré par l'utilisateur.

Modulo

Voici notre algorithme qui permet de moduler un flottant :

Moduler un flottant par 2π

Principe :

Les flottants sont représentés par une somme de 2^k (k étant l'exposant hexadécimal du flottant):
$$f[2\pi] = (2^k[2\pi] + \ldots + 2^{(k-23)}[2\pi])[2\pi]$$

En "stockant" les valeurs de 2^k modulo 2π pour k allant de 1 à 127, il est possible de donner une approximation du modulo de n'importe quel flottant. En sommant les modulos concernant le flottant en paramètre et en soustrayant par 2π à chaque fois que la somme dépasse cette valeur, on obtient le résultat attendu.

Concrêtement, on va décomposer notre paramètre en binaire pour déterminer quels termes seront nécessaires pour calculer le modulo.

- **1.** Si le paramètre est plus petit que 2π, l'algorithme est terminé.
- **2.** On cherche l'exposant hexadécimal du paramètre
- **3.** On ajoute 2^k modulo (2π) à une somme initialisée à 0 au début de l'algorithme. (Le résultat de 2^k modulo (2π) est "stocké" dans des instructions de comparaison.)
- **4.** Si la somme est supérieure à 2π, on soustrait 2π à la somme.
- **5.** On soustrait 2^k au paramètre de l'utilisateur.
- **6.** Si le paramètre est toujours supérieur ou égal à 2π, l'algorithme retourne à l'étape 2.
- **7.** On ajoute ce qui reste de x à la somme et on vérifie une dernière fois s'il est bien inférieur à 2π. Dans le cas contraire on soustrait finalement 2π à la somme.

Cette méthode a certaines limites. Si jamais on somme des modulos qui ne sont pas du même ordre de grandeur (qui ont un exposant hexadécimal différent), on perd à chaque fois en précision. En particulier au moins 2^n ULP (n étant la différence des exposants des 2 termes à additionner).

Il est par ailleurs nécessaire de précalculer les valeurs de 2^k modulo (2π) pour tout k allant de 1 à 127. Nous nous sommes basé sur les résultats de WolframAlpha pour cela puisque Java ne permet pas de calculer des modulos de flottants très élevé.

Moduler un flottant par π , π / 2, π /4

On applique ici les règles trigonométriques pour pouvoir moduler le paramètre jusqu'à une valeur comprise entre 0 et π / 4.

Pour cos(x):

- Si x est négatif, on on affect x en -x.
- Si x est supérieur à π, on affecte (2π - x) à x.
- Si x est supérieur à π/2, on affecte (π - x) à x et on renverra -cos(x).
- Si x est supérieur à π/4, on affect (π/2 - x) à x et on renverra sin(x).

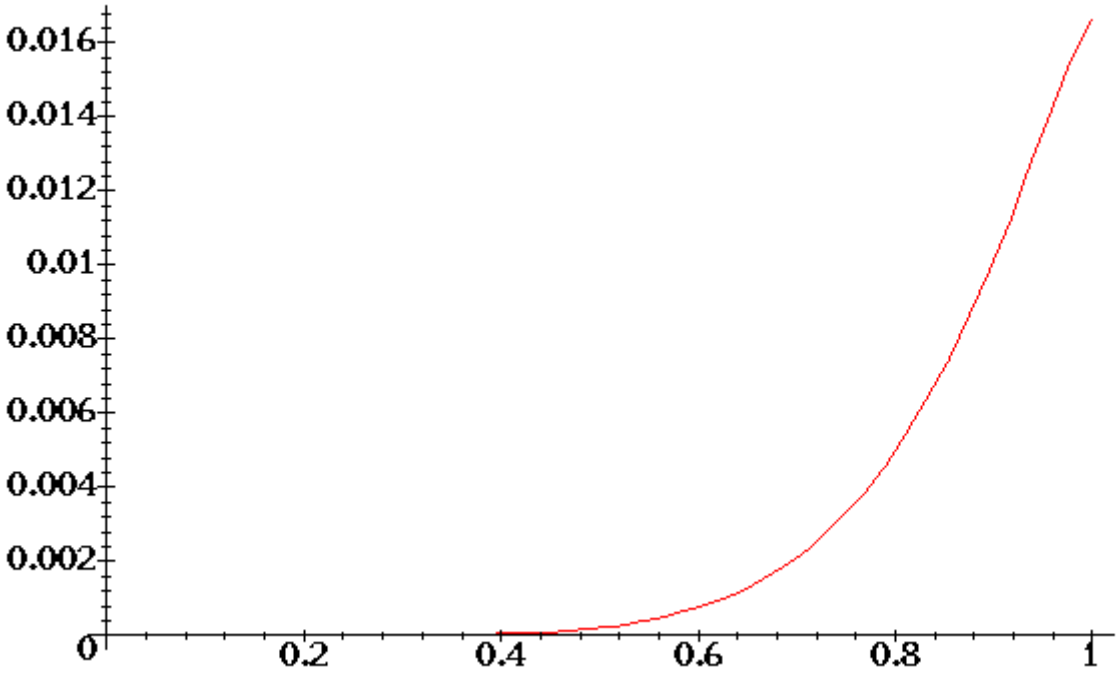
Pour sin(x):

- Si x est négatif, on on affect x en -x et renverra -sin(x).
- Si x est supérieur à π, on affecte (2π - x) à x et on renverra -sin(x).
- Si x est supérieur à π/2, on affecte (π - x) à x.
- Si x est supérieur à π/4, on affect (π/2 - x) à x et on renverra cos(x).

2. Algorithme d'approximation d'arcsin

Pour donner la meilleur approximation d'arcsin possible, nous utilisons la méthode de la série de Taylor sur l'intervalle [-0.3, 0.3], puis la méthode "Fast Sqrt Available" en utilisant le polynôme de Tchebyshev et approximant de Padé pour interpoler l'arcsinus sur les intervalles [-1, -0.3] et [0.3, 1]. Car la méthode de la série de Taylor ne pourrais pas bien interpoler l'arcsinus quand x approche à 1.

Voici le graphe d'erreur d'approximation avec la méthode de Taylor :



On pourrais trouver que l'erreur de la série de Taylor(puissance maximale = 6) augemente de façon polynomiale après la valuer 0.4.

Après rechercher les algorithmes sur Internent, on trouve qu'il y a une méthode "Fast Sqrt Available" pour calculer l'arcsin pour des valeurs proches de 1.

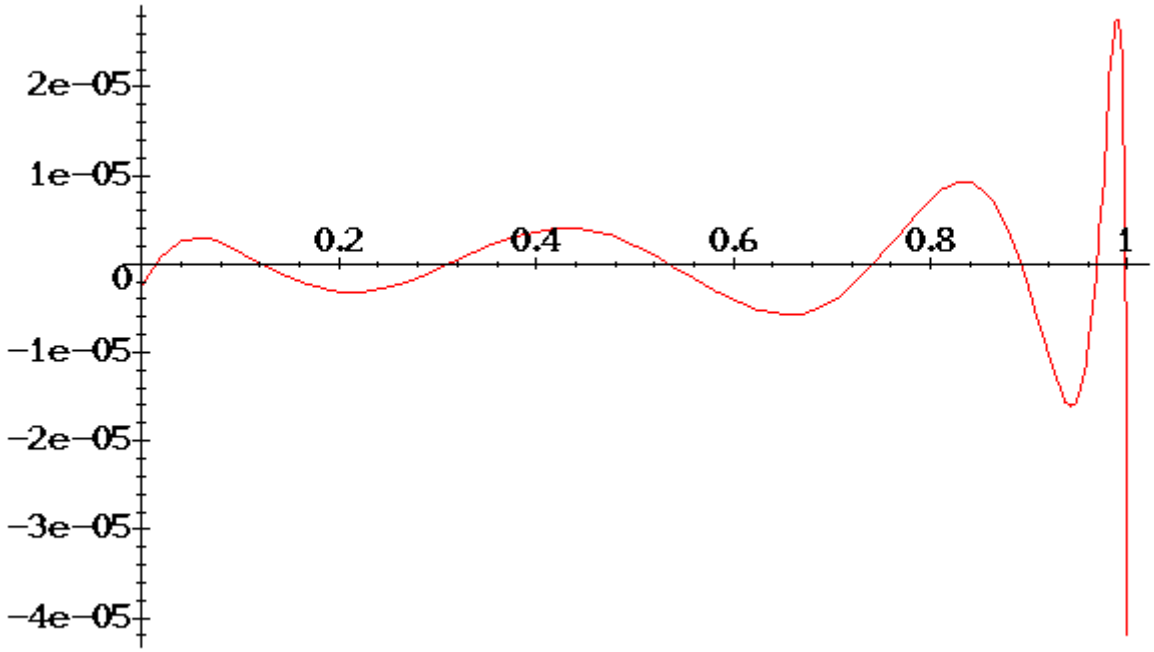
$$arcplussqrt := \arcsin(x) + \sqrt{1 - x^2}$$

Pour choisir l'algorithme pour arcsin(x), le chercheur W. Randolph Franklin a comparé avec différents méthodes. Et on a trouvé que la méthode "Tchebyshev-Pade approximation" est le meuilleur choix pour nous. Car la formule est simple à implémenter. Et il a une meilleur performance après 0.4.

Voici l'équation de Tchebyshev-Pade :

$$(2.420122797 - 2.169670497\,x - .2667407008\,(2.\,x - 1.)^2 + .1783360790\,(2.\,x - 1.)^3) \Big/ (1.941200684 - 1.960601847\,x + .07820047988\,(2.\,x - 1.)^2 + .04436068112\,(2.\,x - 1.)^3)$$

Et voici le graphe obtenu :



3. Algorithme d'approximation d'arctan

Concernant arctan, il est possible d'utiliser la formule suivant si le paramètre est supérieur à 1 pour que la série de Taylor soit exploitable:

$$atan(x) = \frac{\pi}{2} - atan(1/x)$$

Ayant des écarts conséquents entre nos résultats et les valeurs attendues pour des valeurs d'entrée autour de 1 et -1, nous avons opté pour appliquer une seconde série de Taylor centrée en ces points. L'intervalle où est appliquée cette seconde série a été déterminé suite à nos tests et cela nous a permis de réduire les écarts de précision de 600 000 ULP à seulement 200 ULP.

Série de Taylor

La série de Taylor est appliquée pour les fonctions cos, sin, arctan et arcsin. La série de Taylor est applicable seulement pour des petites valeurs de x autour d'un point d'abscisse *a* (souvent 0):

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \cdots$$

Cette méthode nécessite de calculer à l'avance la valeur de la fonction au point *a* ainsi que pour ses dérivées.

4. Algorithme d'approximation d'ULP

Pour obtenir l'ulp (Unit in the Last Place) d'un flottant, il suffit de récupérer l'exposant hexadécimal du flottant, de le soustraire par 23 et de renvoyer "2 puissance le nouvel exposant obtenu".

Le principe est de partir de 1 et de le multiplier par 2 jusqu'à dépasser la valeur du paramètre pour en déterminer son exposant.

IV. Validation de l'extension

La méthode ULP est utilisée lors des tests pour vérifier le bon résultat de l'extension. Elle permet de définir une marge d'erreur en fonction de l'algorithme appelé.

Par exemple pour tester la méthode "cos" :

```
coef = 1500 // la marge d'erreur fixée
res = m.cos(input)
ULP = m.ulp(res)
comparaison(res > expected - coef * ULP && res < expected + coef * ULP)
```

Pour les tests, la partie "expected" qui permet de valider nos tests a été calculé à partir de résultats Java. Le coefficient qui permet modifier la marge de ULP a été ajusté suite à une analyse de nos tests.

Afin d'assurer la précision des résultats de nos fonctions trigonométriques, nous les avons testé en particulier sur les valeurs où la valeur en sortie fluctue le plus. Par exemple, pour arcsin, il était intéressant

de tester des valeurs proches de 0. On s'est rendu compte que les écarts de valeurs étaient assez importants. C'est pour cela que nous avons appliqué par la suite une autre méthode autour de 0 pour approximer le mieux arcsin sur tout son domaine de définition.

Analyse de nos méthodes

La marge d'erreur annoncée est seulement théorique et basée sur le pire cas qu'il est possible de rencontrer pour chacun de nos algorithmes.

1) Sinus

Marge d'erreur théorique : 2048 ULP

Complexité : O(log2(n))

Cette erreur théorique est importante, cela est dû à la façon dont nous avons calculé le modulo 2π. Nous avons une marge d'erreur importante lors de l'appel à la méthode _modulo2PI.

Une piste d'amélioration de la précision serait de sommer les 2^k modulo 2π par ordre de grandeur ce qui permet de réduire la marge d'erreur. Il avait été aussi envisagé de sommer les décimales des 2^k modulo 2π qui ne sont pas représentables en flottant.

La complexité de cet algorithme s'explique aussi par la complexité du modulo 2π. En effet, chercher la valeur de 2^k modulo 2π a pour complexité O(log2(n)).

2) Cosinus

Marge d'erreur théorique : 2048 ULP

Complexité : O(log2(n))

Il est à noté que nos résultats ne concordonnent pas exactement à nos prévisions. La marge d'erreur de cosinus en pratique est assez faible. Cela s'explique par le fait que même si l'erreur du modulo lorsque le résultat avoisine 0, la dérivée de cosinus est nulle et donc cosinus fluctue très peu en ce point contrairement à sinus qui a une dérivée égale à 1. Donc l'erreur du modulo 2π sera directement transférée au résultat du sinus.

Pour se ramener à une complexité en O(1), il serait intéressant d'implémenter des tableaux afin d'accéder directement aux valeurs des 2^k modulo 2π.

3) Arcsinus

Marge d'erreur théorique : 512 ULP

Complexité : O(n^(1/2))

L'erreur provient directement de la méthode de Tchebyshev. Pour améliorer cette précision il serait peut-être possible d'appliquer un développement limité en 1 et -1 pour donner une meilleure approximation de arcsinus en ces points qui posent problème.

La complexité de cet algorithme s'explique par l'utilisation de la méthode _sqrt afin de donner une meilleur approximation.

4) Arctangente

Marge d'erreur théorique : 256 ULP

Complexité : O(1)

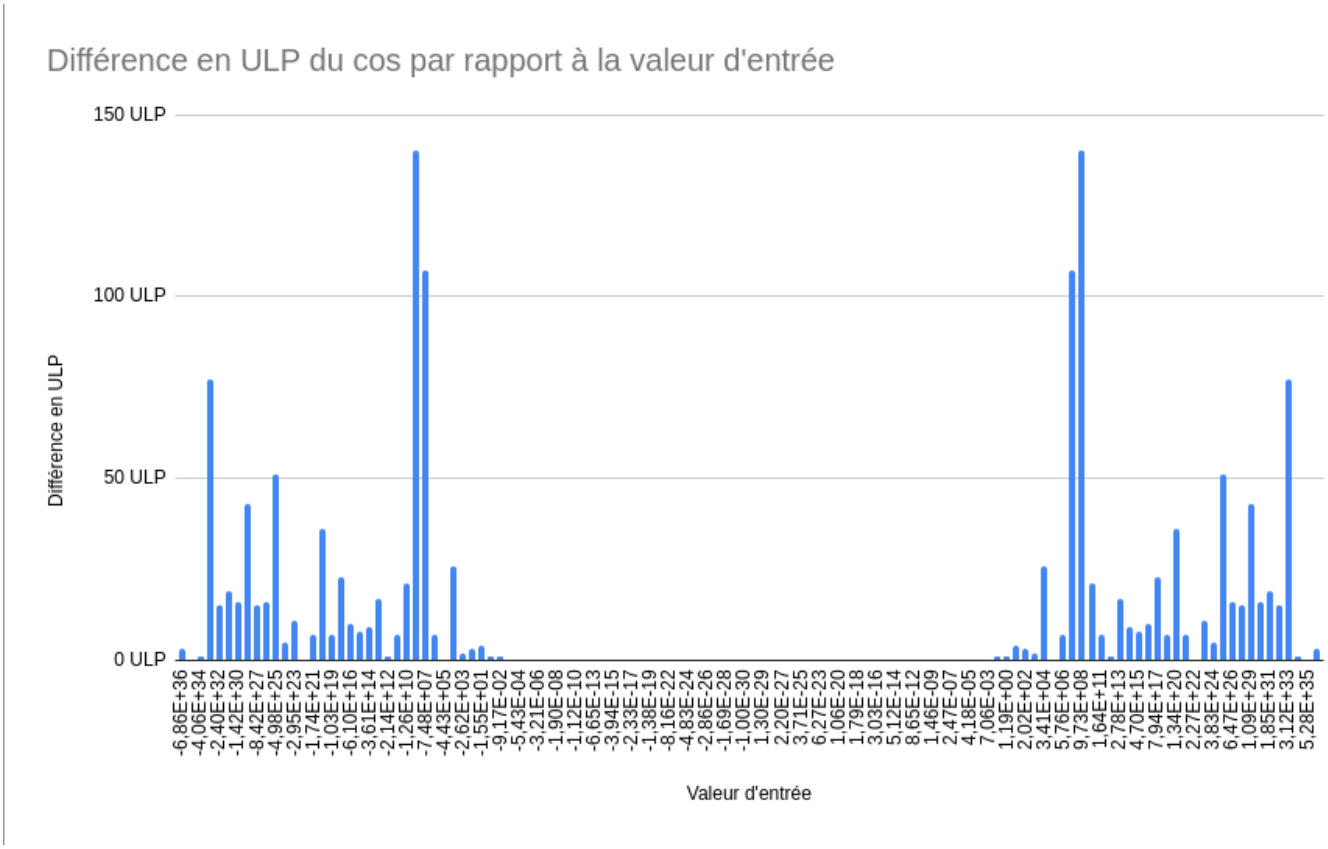
Il a été possible de réduire cette marge d'erreur théorique en appliquant un second développement limité à 1. De manière similaire, il serait possible de réduire à nouveau la marge d'erreur en appliquant 3 développements limités aulieu de 2.

La complexité de cet algorithme est de plus très satisfaisante. On applique uniquement des séries de Taylor à degré fixé.

V. Les résultats de la validation

Pour tous les graphes suivants nous avons utilisé une banque de résultat associé aux méthodes calculées à l'aide de Java. Nous avons comparé les résultats de nos propres méthodes avec la banque de données pour mettre en évidence les différences.

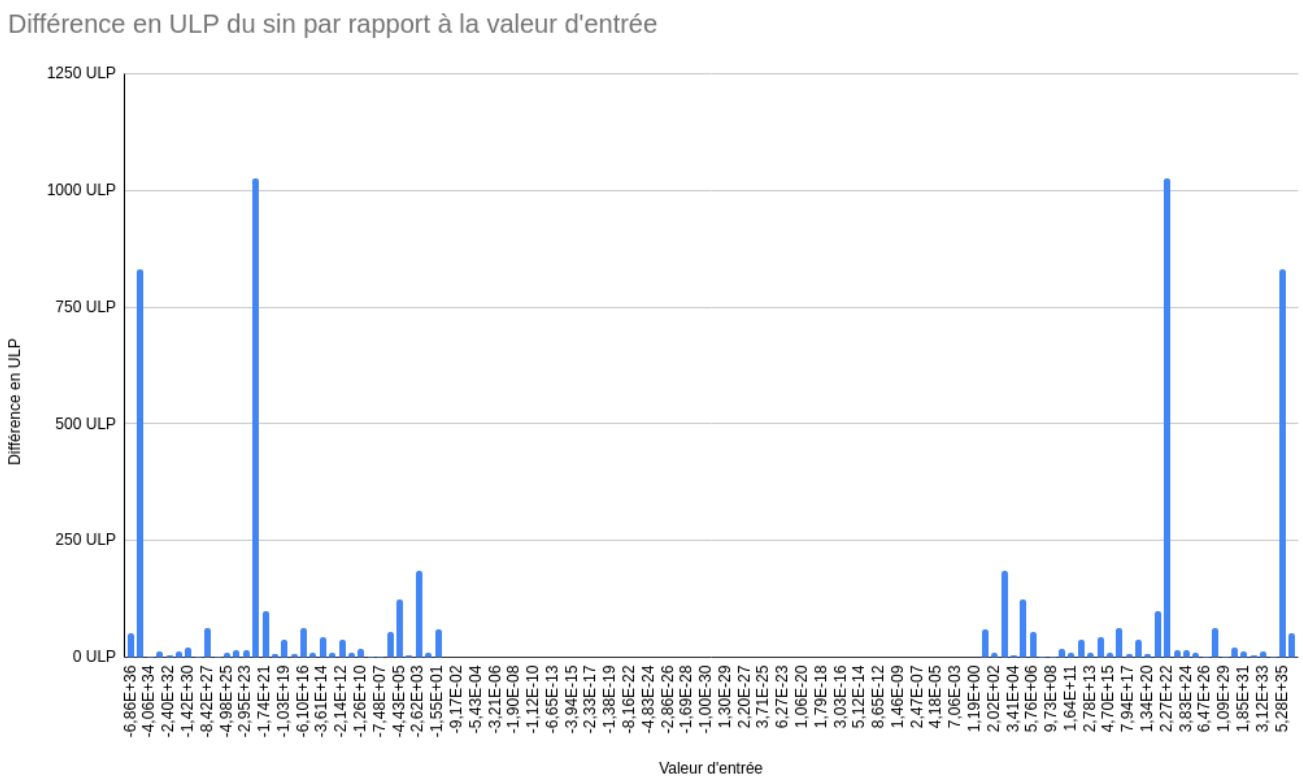
Graphique à colonnes des écarts d'approximation de cos



Nous observons qu'il y a n'y a aucune différence de résultat entre nos valeurs et les valeurs de java pour des valeurs d'entrées comprises entre -1 et 1. Cela s'explique par le fait que nous s'appliquons moins d'opération sur la valeur entrée. D'un autre côté, pour des valeurs entrées en dehors de l'intervalle [-1,1] nous remarquons des différences de l'ordre de quelques dizaines de ULP en moyenne. Cela est dû aux

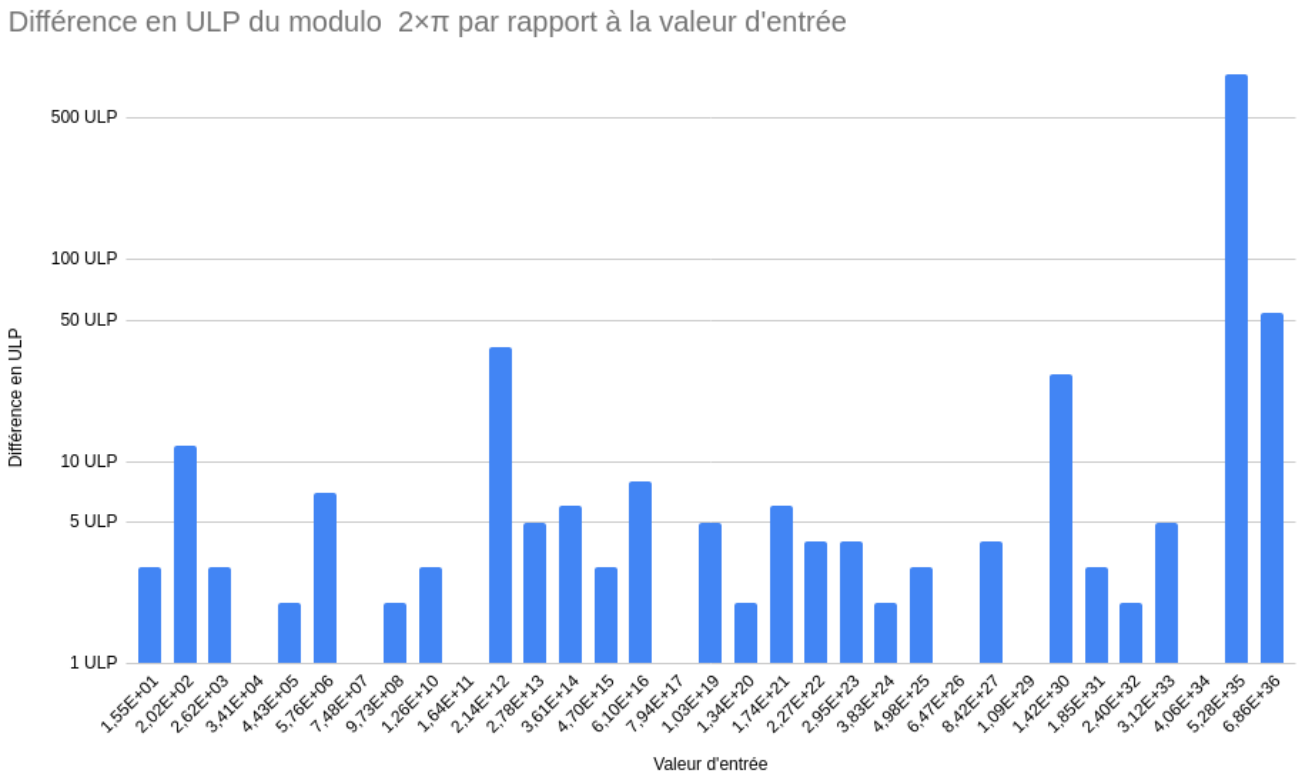
différentes opérations qui ont lieu sur la valeur d'entrée. Le résultat obtenu est satisfaisant contenu de la complexité en temps des algorithmes.

Graphique à colonnes des écarts d'approximation de sin



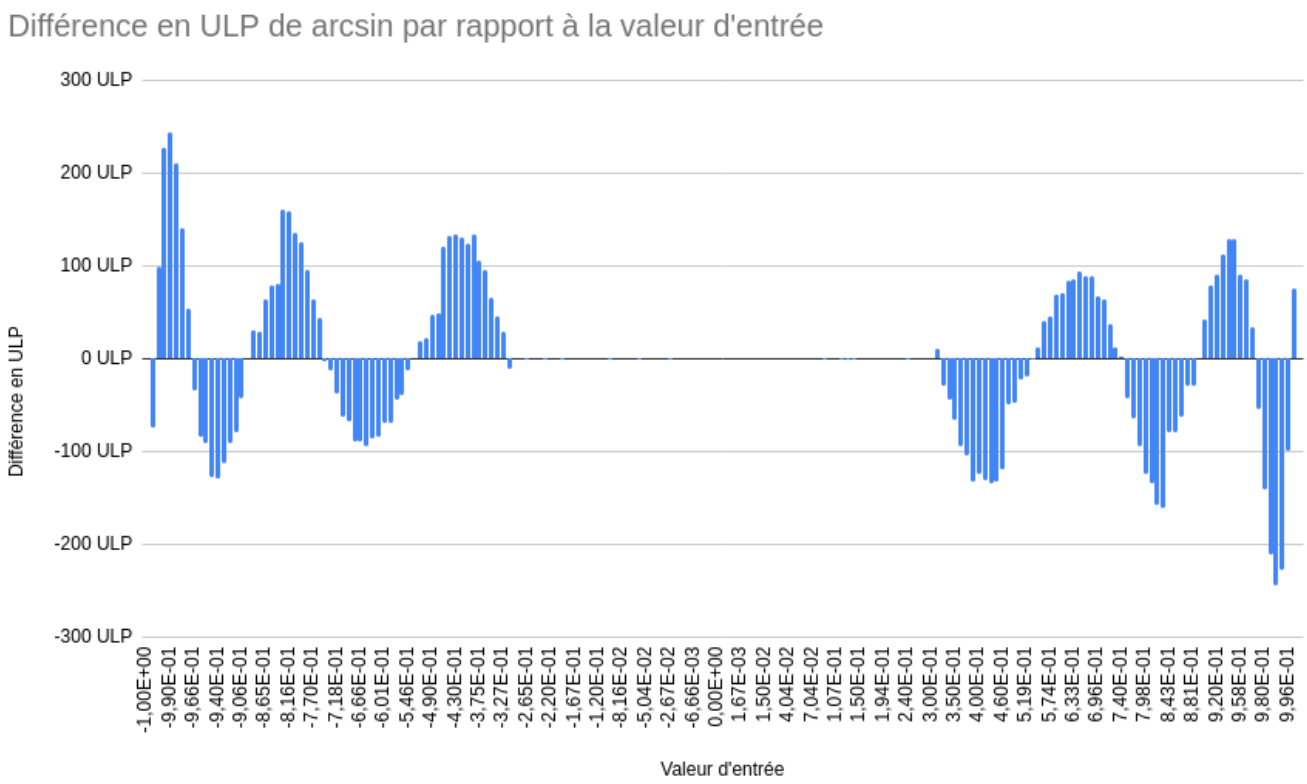
De manière similaire à cos, il n'y a aucune différence entre les valeurs d'entrées -1 et 1. Pour le reste, nous sommes toujours à une différence d'une dizaine d'ULP malgré des anomalies importantes. La raison est que les fonctions sinusoïdales s'appuient sur la méthode modulo 2PI qui présente quelques erreurs de précisions.

Graphique à colonnes des écarts d'approximation de modulo 2π (échelle logarithmique)



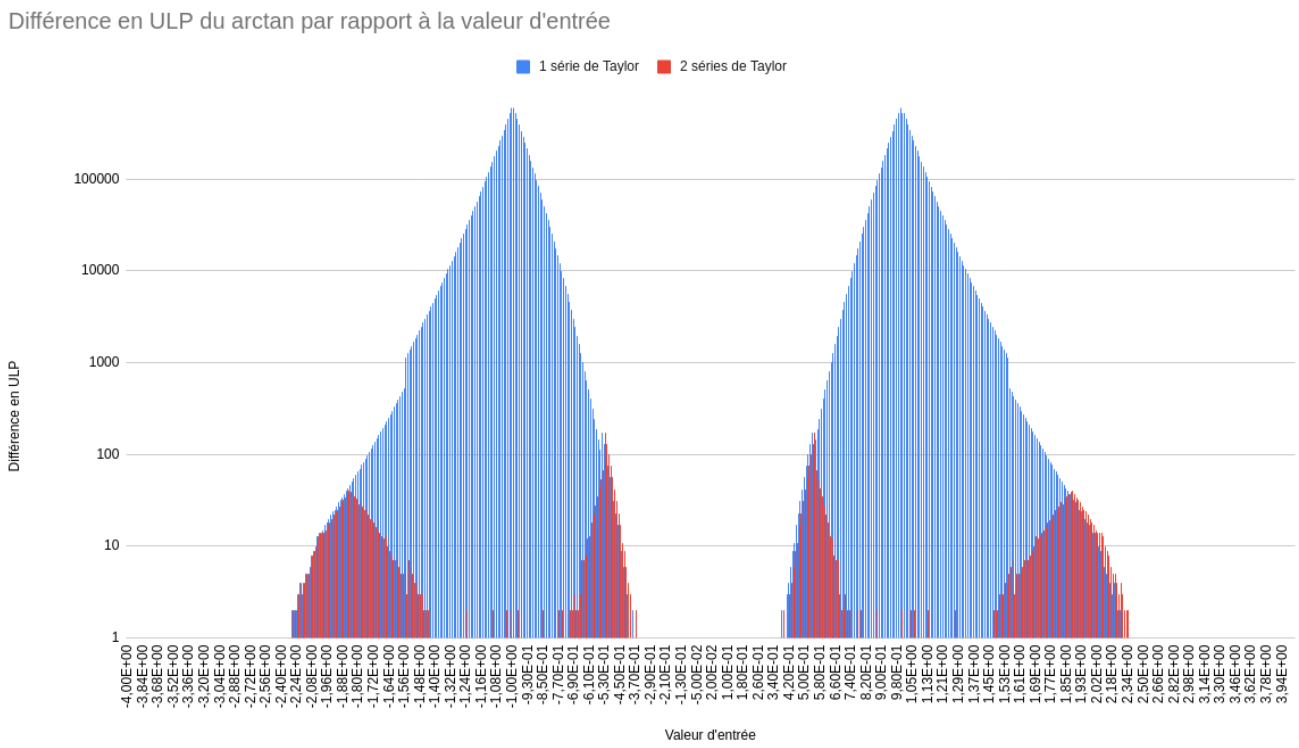
Pour la méthode modulo 2π, la différence de nos résultats avec la banque de données est de l'ordre de la dizaine d'ULP. Des cas particuliers apparaissent approximés les 800 ULP ce qui explique certains écarts pour les fonctions sinusoïdales.

Graphique à colonnes des écarts d'approximation de arcsin



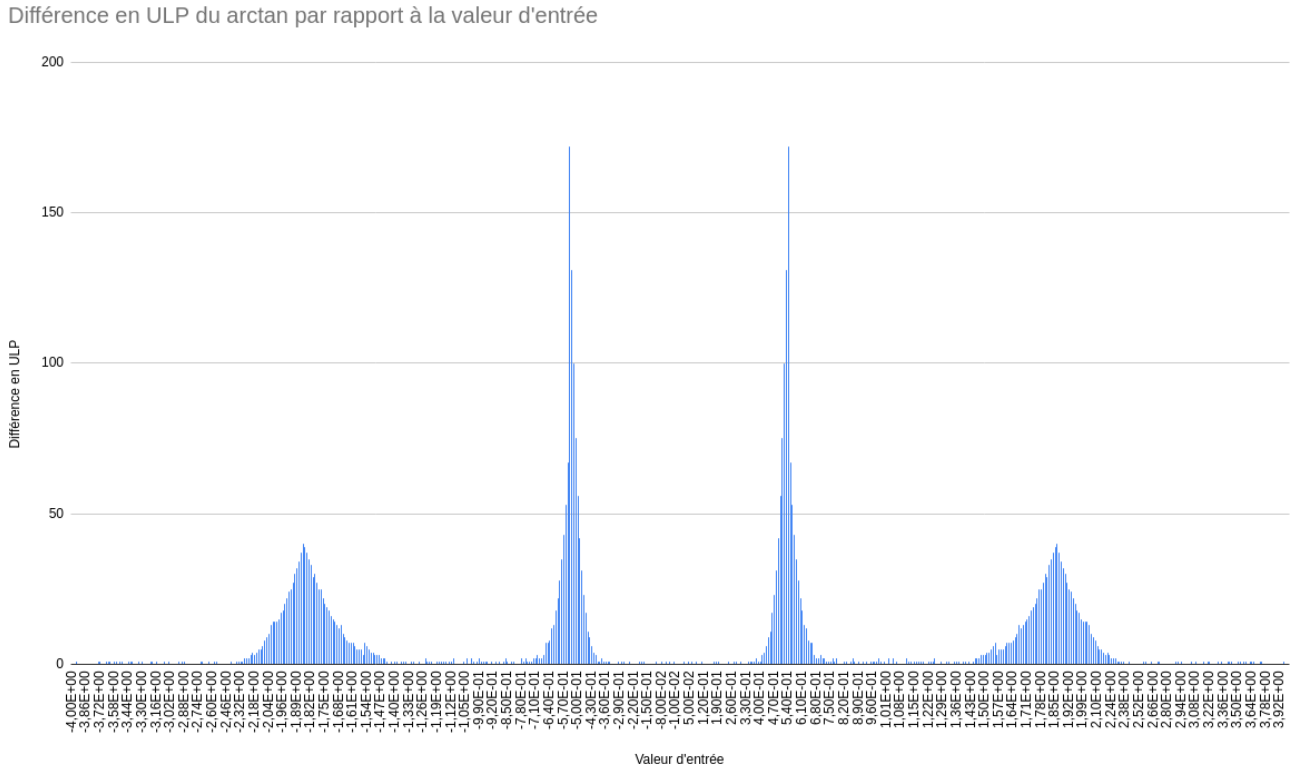
Nous avons utilisé les racines des polynômes de Tchebyshev, c'est pour cela que nous observons des écarts réguliers. Entre -0.3 et 0.3 nous n'avons quasi aucune erreur de résultats puisque nous utilisons la série de Taylor sur ce petit intervalle.

Graphique à colonnes des écarts d'approximation de arctan



On a affiché les résultats de nos deux versions de la méthode d'approximation de arctan. On peut voir ici qu'en rouge les résultats de la méthode basée sur une série de Taylor centrée en 0. Tandis qu'en bleu sont affichés les résultats de la méthode basée sur la série de Taylor centrée en 0 **et** une autre série de Taylor centrée en 1 (et -1). Cette seconde série permet de pallier aux erreurs de précision de la première méthode lorsque l'utilisateur entre une valeur d'entrée proche de 1 ou -1. La deuxième méthode permet de passer de 600 000 ULP à 200 ULP.

On a affiché les résultats de nos deux version de la méthode d'approximation de arctan. On peut voir ici que en rouge les résultats de la méthode basée sur une série de Taylor en centrée en 0. Tandis qu'en bleu sont affichés les résultats de la méthode basée sur la série de Taylor centrée en 0 **et** une autre série de Taylor centrée en 1 (et -1). Cette seconde série permet de palier aux erreurs de précision de la première méthode lorsque l'utilisateur entre une valeur d'entrée proche de 1 ou -1.



On peut voir plus clairement l'influence des 2 séries de Taylor sur ce graphe entre chaque pic d'erreur.

Conclusion

On a rencontré des difficutés à déterminer la meileur méthode pour obtenir la précision qui nous intéressait. Comprendre très clairement le concept de flottant en 32 bit était primordial dans la conception de nos algorithme.

Vis-à-vis des objectifs que l'on s'est fixés, nous sommes satisfaits par la précision de chacune de nos méthodes. Malgré cela, la compilation d'un ficher faisant appel à l'extension trigonométrique pourrait être accélérée. A cause des nombreuses comparaisons de nos algorithmes, générer l'assembleur prend un temps significatif.

Afin d'améliorer cette extension, il serait intéressant d'étendre cette classe à des doubles (flottant en 64 bits). Il serait intéressant d'implémenter les tableaux pour compiler cette extension plus rapidement et optimiser la complexité de certains algorithmes. Appliquer d'autres séries de Taylor à nos algorithmes et étudier d'autres méthodes d'approximation numérique permettrait d'améliorer grandement la précision obtenu.

Références Webographiques

Nous nous sommes principalement appuyés sur trois sites internet :

- https://en.wikipedia.org/wiki/Sine_and_cosine - Méthode d'approximation de sinus et cosinus
- https://fr.wikipedia.org/wiki/Arc_sinus - Méthode d'approximation d'arc sinus
- https://fr.wikipedia.org/wiki/Arc_tangente - Méthode d'approximation d'arc tangente
- https://wrfranklin.org/Research/Short_Notes/arcsin/ - Articles sur les différentes méthodes d'approximation de arcsin par le chercheur W. Randolph Franklin
- https://c-for-dummies.com/blog/?p=5286 - Méthode babylonienne permettant d'approximer la racine d'un nombre
- https://fr.wikipedia.org/wiki/S%C3%A9rie_de_Taylor - Méthode d'approximation des fonctions dans un interval autour d'un point
- https://www.wolframalpha.com - Site de calculs mathématiques nous ayant permis de construire des tests solides