Extension GL : **Trigonométrie**

Groupe GL 35

Janvier 2023

Contents

1 Introduction

L'objectif de cette partie est de créer une classe Maths dans laquelle on implémente des méthodes trigonométriques, tout en obtenant une bonne précision pour chaque fonction, en différenciant selon les sous-domaines du domaine de définition suivantes :

- float *ulp*(float f)
- float *sin*(float f)
- float *cos*(float f)
- float *asin*(float f)
- float *atan*(float f)

Nous avons du implémenté d'autres fonctions qu'on a pensées nécessaire pour l'implémentation des méthodes demandées, à savoir :

- float *puiss*(float a, int b)
- float *abs*(float f)
- float $fact$(float f)
- float *sqrt*(float f)

Cette partie d'extension sera abordé comme suit, d'abord une implémentation de la fonction ULP, qui sera nécessaire pour le calcul de précision tout au long du travail sur l'extension. Ensuite un choix justifié d'algorithmes pour implémenter les différentes méthodes trigonométriques, en cherchant le bon compromis entre précision et quantité de calculs de la machine.

1.1 Les flottants Deca: flottant simple précision

D'après la norme IEEE 754, un nombre flottant simple précision est stocké dans un mot de 32 bits : 1 bit de signe, 8 bits pour l'exposant et 23 pour la mantisse.

Cette représentation impose que l'exposant d'un nombre normalisé va donc de -126 à $+127$.

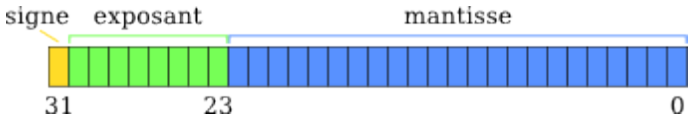


Figure 1: Format simple précision 32bits

1.2 Stratégie adoptée

Notre stratégie consiste à implémenter dans un premier temps nos algorithmes dans java, afin de les tester et d'avoir des idées concrètes sur les précisions attendues, tout en respectant les limitations de l'Ima. Et aussi pour ne pas avoir des retards concernant les délais de rendus, car attendre que le compilateur soit prêt ,ne va pas nous laisser beaucoup de temps pour tester.

Nous avons pensé aussi à faire une deuxième couche de tests après la fin de la partie essentielle du compilateur avec objet. En effet, dés que le compilateur était prêt, nous avons implémenté les algorithmes des fonctions trigonométriques sur Deca, tout en les adaptant pour qu'ils restent fonctionnels.

Tout au long des tests, nous avons priorisé le traçage des courbes, car l'analyse des graphes permet de mieux détecter les problèmes. Les courbes qu'on a tracées ont été crées par Latex, après création de fichiers (.csv) sous java, dans lesquelles on a stockés nos résultats.

Remarque:

L'écriture flottante de java est limitée à 8 chiffres après la virgule, contrairement à l'écriture hexadécimale du Deca qui n'est pas limité. Ce qui donne une meilleure évaluation de la précision sur Deca par les mêmes algorithmes.

1.3 Méthode de validation

Nous tenons à préciser que notre objectif est d'avoir une fonction ULP qui marche parfaitement, pour ne pas influencer le calcul des précisions qui vient après.

Nous visons aussi des précisions de quelques ULP pour les fonctions trigonométriques.

On a aussi tenu compte dans nos implémentations de réduire les calculs au maximum, en faisant par exemple beaucoup de tests qui nous ont permis de choisir le bon intervalle de réduction, ou encore en diminuant au maximum les degrés de polynômes utilisés en développement en série entière.

Pour le calcul d'erreur des fonctions implémentés, nous avons calculé la différence en valeur absolue entre la valeur calculé par notre méthode et celle de la classe Math du Java, le tout divisé par Ulp de la valeur réelle (celle calculée par la classe du Java).

2 ULP

2.1 Principe

ULP: Unit in the Last Place, est l'écart entre deux nombres consécutifs à virgule flottante, c'est-à-dire la valeur que représente le chiffre le moins significatif (chiffre le plus à droite) s'il est égal à 1.
Elle est utilisée comme mesure de précision dans les calculs numériques.

La méthode ULP implémenté dans ce projet, est inspirée de l'algorithme donné pour un double dans la thèse de Monsieur Jean-Michel Muller:[these] qu'on a adapté aux flottants, et auquel on a ajouté des modifications, après de nombreux tests, pour avoir les résultats attendus.

Cet algorithme vérifie d'abord que le flottant est bien dans l'intervalle $[\ 2^{-126}\ ;\ 2^{127}\ ]$, sinon, s'il est plus grand il retourne la valeur max 2^{103}, et s'il est plus petit, il retourne la valeur du nombre dé-normalisé non nul le plus proche de zéro qui est : 2^{-149}.

Ensuite, il utilise une recherche binaire afin d'encadrer le flottant comme suit : $2^i \leq x \leq 2^{i+1}$ pour trouver l'exposant le plus proche du nombre donné, et finalement il renvoie 2 à la puissance l'exposant trouvé.

Notre compilateur ne supporte pas la déclaration de variables flottantes plus grandes que du $2^{31} - 1$ en écriture de 2^x, ,donc on a du écrire en dur toutes les valeurs qui dépassent cette puissance à l'aide de leurs écriture scientifiques.

2.2 Résultats

Les résultats obtenus avec la fonction ULP sont parfaitement précis, vu que l'erreur avec l' ULP de la classe Math de Java est nulle partout.

Le tableau suivant résume ces résultats.

Intervalle	Pas	Nombre d'échantillons	Erreur entre notre ulp et Math.ulp
$[0; 10]$	$3.8146973 * 10^{-6}$	2621440	0
$[2^{-40}; 2^{-7}]$	2^{-28}	2097152	0
$[2^{-7}; 2^{12}]$	$9.765625 * 10^{-4}$	4194296	0
$[2^{12}; 2^{40}]$	$9.765625 * 10^{-4}$	4194296	0

2.3 Graphes

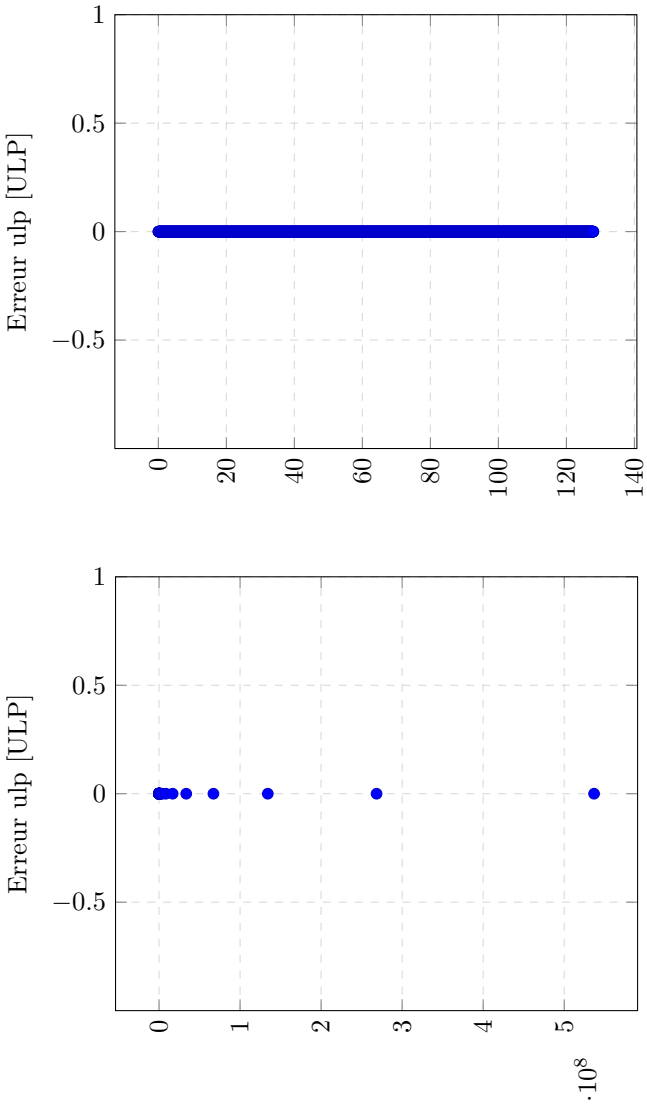


Figure 2: ULP plots

3 Cos et Sin

3.1 Choix d'implémentation

Pour ces deux fonctions trigonométriques, on a choisi de les implémenter à l'aide de séries de Taylor. La définition de la série de Taylor est comme suit : Soit f une fonction d'une variable réelle ou complexe, indéfiniment dérivable en un point a. La série de Taylor de f en a est la série de fonctions :

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

Choix du nombre de coefficients : La courbe de la fonction cos implémentée par 8 coefficients de la série de Taylor coïncide parfaitement avec la courbe de cos sur l'intervalle de $[-\pi;\pi]$ (et ça va de même pour la courbe de sin). Ce qui nous permet de travailler dans un premier temps sur cet intervalle, et de l'élargir à $[-2\pi;2\pi]$ à l'aide des propriétés de cos et de sin suivantes :

- cos(x+π) = -cos(x)
- sin(x+π) = -sin(x)

On élargit l'intervalle après sur R, en retranchant à chaque fois la valeur 2π des nombres hors notre intervalle.

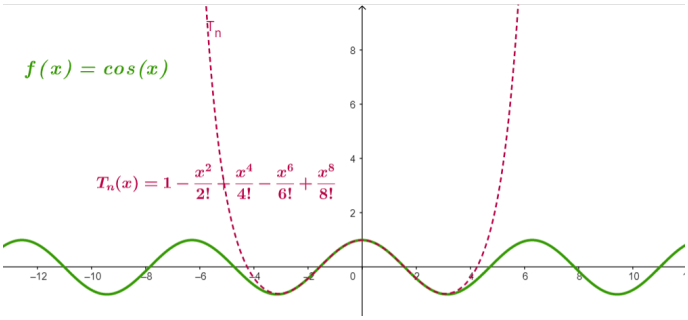


Figure 3: Graphe représentant la fonction : cos et sa série de Taylor à l'ordre 8

3.2 Difficultés et amélioration

Après cette première implémentation, on s'est avéré que prendre un intervalle de départ de $[-\pi;\pi]$, qui est assez grand entraine des ULP d'une valeur de 2 de temps en temps. Alors on a réduit l'intervalle encore plus pour ne travailler que sur $[0;\pi/4]$, et déduire les valeurs du reste de l'intervalle $[-2\pi;2\pi]$ à l'aide des relations trigonométrique:

- cos(π/2-x) = sin(x)
- sin(π/2-x) = cos(x)
- cos(π-x) = -cos(x)
- sin(π-x) = sin(x)

Alors que pour le reste de l'intervalle, on fait la même chose qu'à la première démarche.

3.3 Résultats sin

D'après les tests effectués pour la fonction sin sur Java, on trouve:

Intervalle	step	erreur en ULP	max erreur	nombres d'éléments
$[pi/4; \pi/2]$	2^{-23}	0.43120888	3.0	6588398
$[pi/4; \pi/2]$	2^{-18}	0.42268613	2.0	205888
$[pi/2; \pi]$	2^{-10}	0.22364366	0.75	1609
$[pi; 2\pi]$	2^{-10}	0.22142512	0.75	3217
$[2\pi; 4\pi]$	2^{-10}	0.20667103	0.5	6434
$[50; 50 + 2\pi]$	2^{-10}	0.14266211	0.328125	6434

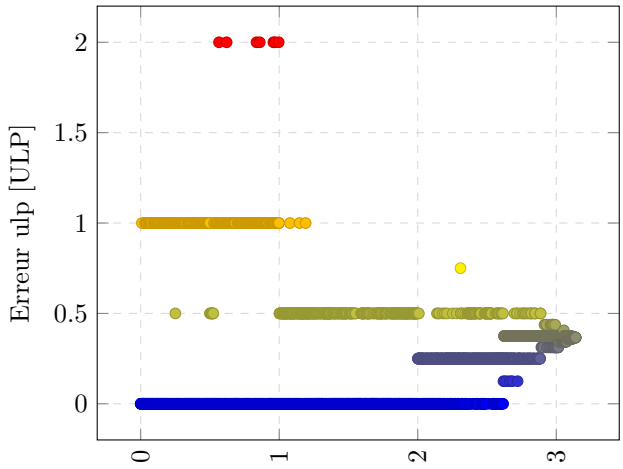
3.4 Résultats cos

D'après les tests effectués pour la fonction cos sur Java, on trouve:

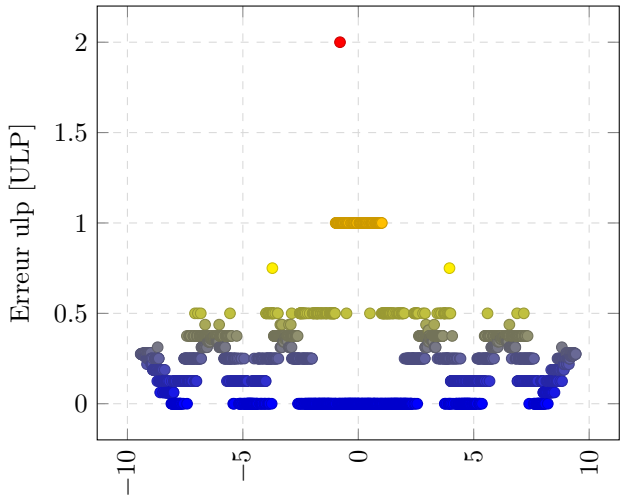
Intervalle	step	erreur en ULP	max erreur	nombres d'éléments
$[0; \pi/4]$	2^{-18}	0.33533767	2.0	205888
$[pi/4; \pi/2]$	2^{-23}	0.43120888	3.0	6588398
$[pi/2; \pi]$	2^{-10}	0.20397955	0.75	1609
$[pi; 2\pi]$	2^{-10}	0.20091482	0.75	3217
$[2\pi; 4\pi]$	2^{-10}	0.21499515	0.625	6434
$[50; 50 + 2\pi]$	2^{-10}	0.16506171	0.37304688	6434

3.5 Graphes

Ci-dessous, les graphes représentants les résultats des deux fonctions Cos et Sin, avant et après optimisations.

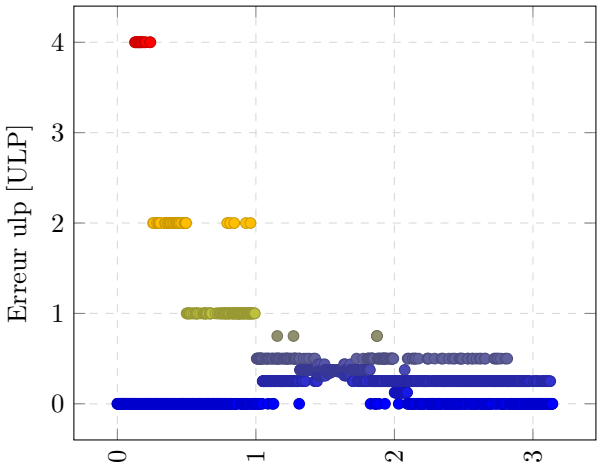


(a) Sin Plot avant correction

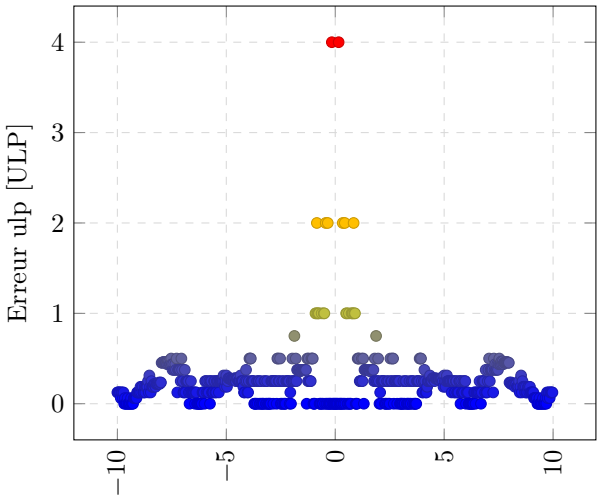


(b) Sin Plot Final

Figure 4: Sin Plots



(a) Cos Plot avant correction



(b) Cos Plot Final

Figure 5: Cos Plots

4 Asin

4.1 Choix d'implémentation

Le choix d'algorithme pour cette fonction, a demandé beaucoup de tests de différents algorithmes. On en cite un qui était le plus performant entre eux et qu'on a changé à la fin.

Pour cette première méthode, on a utilisé 2 implémentations différentes selon l'intervalle de départ. Sur l'intervalle $[0; 0.6]$ on a utilisé un développement en série entière à l'ordre 21 de la fonction *arcsin*, alors que sur l'intervalle restant de $[0.6; 1]$, on a opté pour une approximation polynomiale, en s'aidant aussi du polynôme de Horner pour réduire le cout des opérations. Et on profite de l'imparité de l'arcsin, pour déduire les valeurs de l'arcsin sur l'intervalle $[-1; 0]$:

$$\mathbf{arcsin}(\boldsymbol{x}) = -\mathbf{arcsin}(-\boldsymbol{x});$$

Mais on s'est aperçu que cette méthode donnait des erreurs qui s'élevaient à du $\mathbf{10^{-5}}$ autour de $\mathbf{1/\sqrt{2}}$, ce qui représente un très grand nombre d'ULP(6a).

On a donc décidé de procéder autrement, en n'utilisant qu'un développement en série de Taylor sur l'intervalle $\mathbf{[0; 1/\sqrt{2}]}$ et de profiter de la relation trigonométrique de l'**arcsin**, pour déduire les valeurs sur l'intervalle $\mathbf{[1/\sqrt{2}; 1]}$, et ainsi ne calculer que des valeurs proches de 0 (6b):

$$\mathbf{arcsin}(\boldsymbol{x}) + \mathbf{arcsin}(\sqrt{\mathbf{1} - \boldsymbol{x}^{\mathbf{2}}}) = \boldsymbol{\pi}/\mathbf{2};$$

Ce qui donne des meilleurs résultats, mais encore une fois, le problème persiste un peu au alentour de $\mathbf{1/\sqrt{2}}$.

4.2 Difficultés et amélioration

Ce problème a été du au manque de termes dans notre développement en série entière, qui s'élevait à l'ordre **21**, mais qu'on a trouvé insuffisant lors du traçage du graphe, vu qu'on remarquait une divergence de l'erreur autour de $\mathbf{1/\sqrt{2}}$.

Afin de le corriger, on a poussé un peu plus le développement en série entière, vers un polynôme de degré 31, bien que cela ajoute plus de calcul (7). Et on a laissé l'intervalle $\mathbf{[0; 1/\sqrt{2}]}$ comme intervalle d'étude initial.

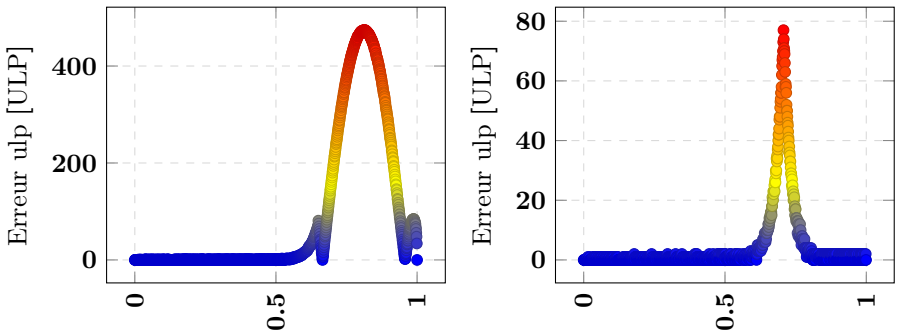
4.3 Résultats

Ci-dessous, les résultats obtenus par les tests effectués sur Java:

Intervalle	step	erreur en ULP	max erreur	nombres d'éléments
$\mathbf{[0; 1/\sqrt{2}]}$	$\mathbf{2^{-23}}$	**0.55730605**	**5.0**	**5931642**
$\mathbf{[1/\sqrt{2}; 1]}$	$\mathbf{2^{-23}}$	**0.5891706**	**7.0**	**3439330**

4.4 Graphes

Les courbes ci-après résument les résultats de la fonction Asin avant et après optimisations.



(a) Asin Plot (1ère implémentation) (b) Asin Plot(Taylor 10 coeff)

Figure 6: Asin Plots

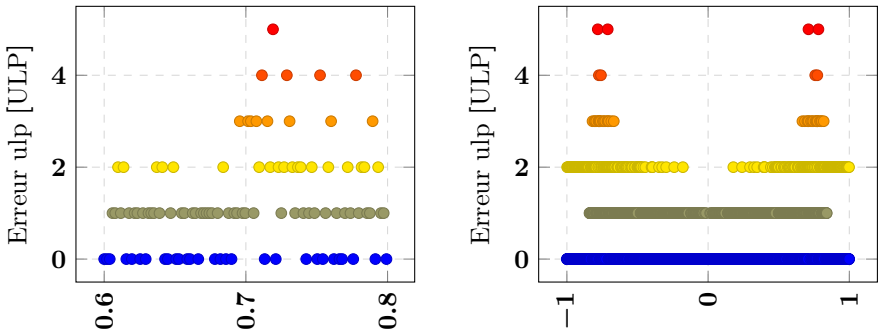


Figure 7: Asin Plots finals(15 coefficients Taylor)

5 Atan

5.1 Choix d'implémentation

Pour la méthode ***atan***, on a choisi comme implémentation les polynômes d'Hermites. C'est une autre séquence à décrire des polynômes dans Q[x] qui s'approche d'**arctan**(x) uniformément sur $[\mathbf{0,1}]$ et qui le fait beaucoup plus rapidement que la séquence de polynômes de Taylor centrée sur 0. Nous notons qu'une telle séquence d'approximation fournit une approximation sur tout R, grâce aux identités suivantes :

arctan(x) = -**arctan**(-x) = $\boldsymbol{\pi/2}$ - **arctan**(1/x)

On a utilisé le polynôme d'Hermite h3(x) qui est un polynôme d'ordre 15, et qui d'après l'article [**5**],donne une erreur de l'ordre de $\mathbf{10^{-7}}$.

Afin de justifier notre choix d'implémentation, on a effectué de nombreux tests sur l'intervalle [**0**;**1**] pour comparer l'implémentation de l'**arctan** avec Séries de Taylor à l'ordre **9** et l'implémentation avec polynômes d'Hermites. Et on a remarqué qu'au voisinage de **0**, les deux méthodes ont une précision de $\mathbf{10^{-8}}$, mais plus on s'approche de **1**, et précisément après $\mathbf{0,6}$, l'erreur de la méthode avec Taylor commence à augmenter pour atteindre du $\mathbf{10^{-4}}$ dans l'intervalle de [$\mathbf{0,65}$; $\mathbf{0,7}$].

Exemple:

Intervalle	Pas	Nombre d'échantillons	Résultats par Hermite	Résultats par Taylor
$[\mathbf{0,65;0,7}]$	$\mathbf{2^{-23}}$	**419431**	$\mathbf{3,2338175 * 10^{-8}}$	$\mathbf{8,9091645 * 10^{-4}}$

5.2 Difficultés et amélioration

L'implémentation avec les polynômes d'Hermites nous a donné de bons résultats (7), mais on a quelques grands valeurs de l'ULP après $\mathbf{0,6}$.

Pour régler ce problème, on s'aide de la relation trigonométrique d' **arctan**:

arctan($\boldsymbol{x}$) + **arctan** (y) = **arctan**($(\boldsymbol{x+y})/(\mathbf{1}-\boldsymbol{xy})$) <=> $\boldsymbol{xy} < \mathbf{1}$.

Et donc si on prend $\boldsymbol{y} = \mathbf{1}$, on peut choisir l'intervalle qu'on veut pour x. Ici c'est $[\mathbf{0,6;1}[$ qui nous pose problème.

5.3 Résultats

D'après les tests effectués sur Java on trouve les résultats suivants:

Intervalle	step	erreur en ULP	max erreur	nombres d'éléments
$[\mathbf{0.0; 0.59}]$	$\mathbf{2^{-23}}$	**1.434709**	**8.0**	**4949279**
$[\mathbf{0.59; 1}]$	$\mathbf{2^{-23}}$	**0.5891706**	**7.0**	**3439330**
$[\mathbf{1; 1000}]$	$\mathbf{2^{-7}}$	**0.36701545**	**3.0**	**127872**

Quelques valeurs sur Deca:

Après implémentation de la fonction **arctan** sur Deca, on a refait quelques tests pour s'assurer des précisions trouvées.

On trouve les mêmes ordre de grandeurs. Avec des fois des précisions meilleures; comme on l'a indiqué au début de la documentation, cela est du au restriction de l'écriture flottante de Java qui est limité à 8 chiffres après la virgule, contrairement à l'écriture hexadécimale du Deca qui n'est pas limitée.

valeur	Résultats sur Deca(hex)	résultat atan java(hex))	erreur	erreur en ulp
1	$\mathbf{0x1.921fb6p-1}$	$\mathbf{0x1.921fb6p-1}$	**0.0**	**0.0**
0.0001	$\mathbf{0x1.a36e2ep-14}$	$\mathbf{0x1.a36e26p-14}$	$\mathbf{6 * 10^{-14}}$	$\mathbf{8.24634e-03}$
5656	$\mathbf{0x1.92142p+0}$	$\mathbf{0x1.92141ep+0}$	$\mathbf{10^{-9}}$	$\mathbf{8.38861e-03}$
$\mathbf{0,005}$	$\mathbf{0x1.47ad62p-8}$	$\mathbf{0x1.47ad62p-8}$	**0.0**	**0.0**
0.0000009	$\mathbf{0x1.e32f0ep-21}$	$\mathbf{0x1.e32f0ep-21}$	**0.0**	**0.0**

5.4 Graphes

Les courbes ci-après résument les résultats de la fonction Atan avant et après optimisations.

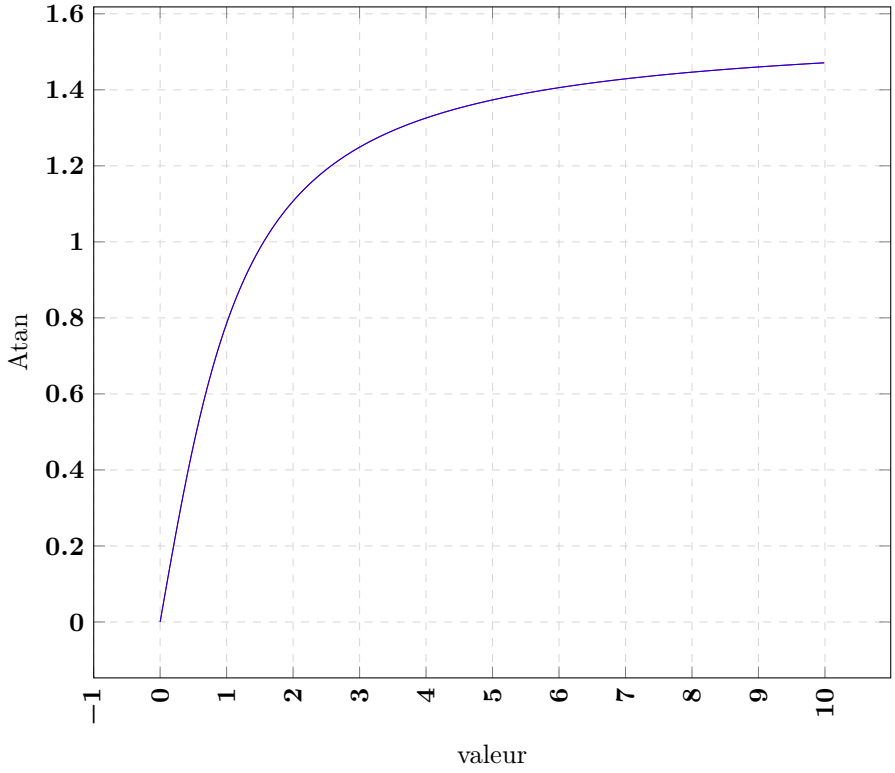


Figure 8: Atan Deca (en bleue)sur Atan Java (en rouge)

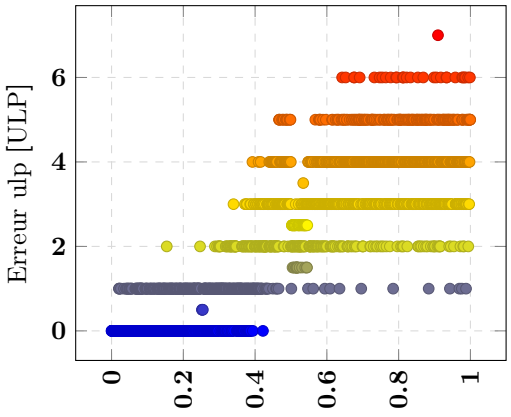
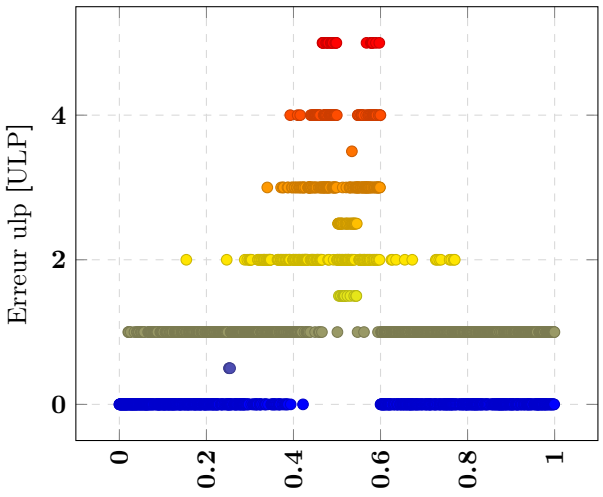
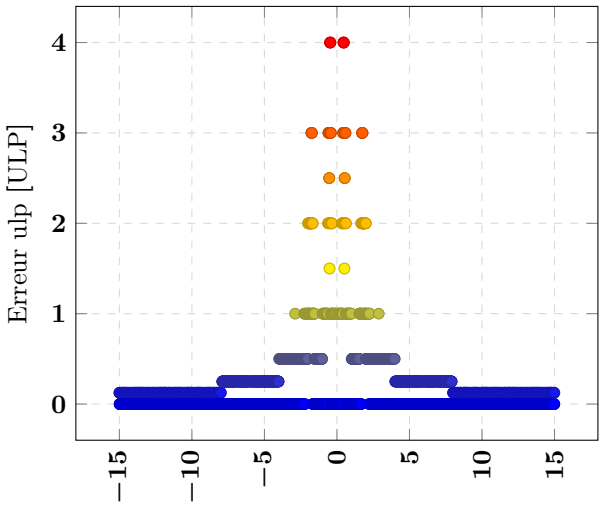


Figure 9: atan avant correction



(a) atan corrigé



(b) atan corrigé sur un grand intervalle

Figure 10: Atan corrigé

References

[1] https://en.wikipedia.org/wiki/Unit_in_the_last_place

[2] https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=bc07e8d52dbbac5bef9551bbc111b95d82dc4766

[3] https://fr.wikipedia.org/wiki/S%C3%A9rie_de_Taylor

[4] https://www.geogebra.org/m/s9SkCsvC

[5] https://www.academia.edu/5490106/A_SEQUENCE_OF_HERMITE_INTERPOLATING_LIKE_POLYNOMIALS_FOR_APPROXIMATING_ARCTANGENT

[6] https://stackoverflow.com/questions/15393573/square-root-with-babylonian-or-herons-method