Documentation technique de l'extension TRIGO

❖ Introduction

Ce document présente les spécificités de la classe Math implémenté, ainsi que la conception choisie et les résultats et précision obtenus. La partie Objet de l'étape C n'ayant pu être terminé, l'exécution en Deca avec notre compilateur de la classe Math fut impossible. Les résultats suivants sont ainsi produits à partir des méthodes de Math.decah transférées dans fichier java. Ils seront donc présentés de manière à mettre en valeur l'apport de chaque algorithme par rapport à la version de base.

❖ Algorithmes intermédiaires de calcul

Ici sont détaillés les méthodes qui ont été implémentées pour alléger les algorithmes choisies pour calculer les fonctions trigonométriques. Elles proposent une version plus aboutie (plus de précision) que les calculs intuitifs.

<u>Méthode de Héron :</u>

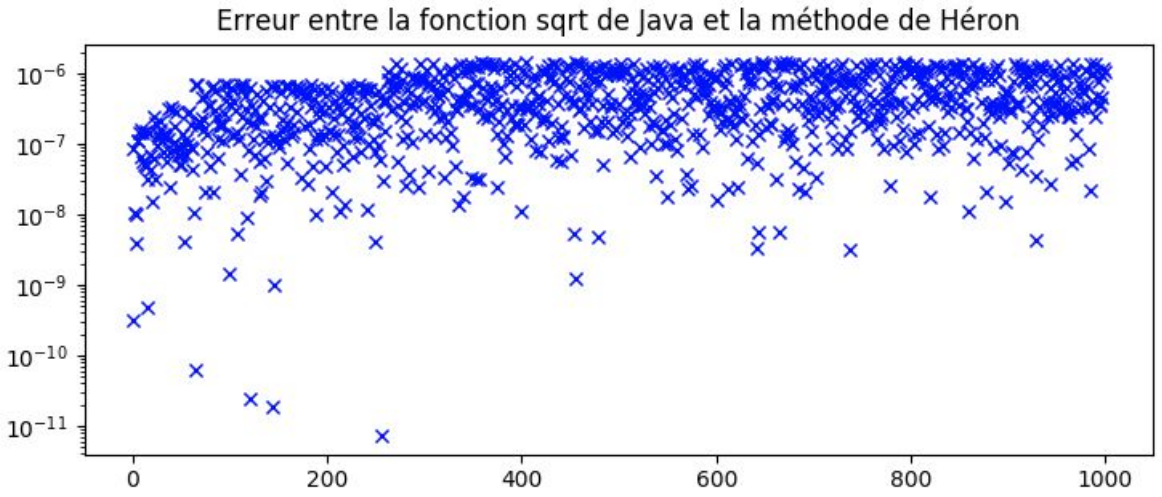
C'est un algorithme d'approximation de la racine carré, opération non disponible en Deca.

Il consiste, pour $a \in \mathfrak{R}^+$, à chercher le carré d'aire a (le côté de ce carré donne alors $\sqrt{a}$).

Pour cela, on se donne un rectangle de largeur $x \in \mathfrak{R}^+$ fixé arbitrairement, si possible proche de $\sqrt{a}$ pour accélérer la précision de l'algorithme.

La longueur du rectangle est alors a/x.

Il suffit alors de rapprocher ce rectangle du carré solution itération après itération en choisissant cette fois un rectangle de largeur (x + a/x)/2, la moyenne des côté du rectangle précédent.
On obtient ainsi une suite définie par $x_{n+1} = (x_n + a/x_n)/2$ qui converge vers $\sqrt{a}$.



Nombre d'itérations : 40
Erreur maximale : 1.4291367e-06
Erreur moyenne : 5.137575e-07
Ecart-type : 4.169555e-07

Nous admettons donc une marge d'erreur de l'ordre de 10^{-7} sur la méthode de Héron, cette erreur sera cependant amortie par les algorithmes qui l'utilisent (l'approximant de Padé par exemple).

<u>Algorithme d'exponentiation rapide :</u>

Cet algorithme consiste à diminuer le nombre de multiplications lors du calcul de la puissance entière d'un flottant a^n.
La fonction intuitive multiplie a n-1 fois par lui-même, utilisant donc O(n) multiplications.
La méthode implémentée est récursive, pour chaque appel si n est :

- pair , il suffit alors de calculer $a^{n/2}$ et de mettre le résultat au carré.

- impair, il suffit alors de calculer $a^{(n-1)/2}$, de mettre au carré puis de multiplier le résultat par a.

Le nombre de multiplications utilisé est alors en O(log(n)).

La précision glanée avec cet algorithme n'est cependant que visible sur Deca puisque le nombre de décimales lors du calcul flottant est plus limité.

Méthode de Horner :

Comme l'algorithme d'exponentiation rapide, l'objectif de la méthode de Horner est de limiter le nombre de multiplications effectuées, cette fois-ci lors du calcul d'un polynôme.

Pour $P(x) = \square^{n}_{i=1} a_i x^i$, le calcul de base réalise n(n+1)/2 multiplications, soit $O(n^2)$ multiplications.

On passe à O(n) multiplications en effectuant la modification suivante :

$P(x) = a_0 + x(a_1 + x(...(a_{n+1} + a_n x)...))$.

Pour déterminer l'efficacité de cette méthode, voir les sections Sinus et Cosinus où elle a été implémentée. La méthode de Horner n'ayant pas présenté une précision satisfaisante, elle n'a finalement pas été utilisé dans le calcul de sin, cos, asin et atan.

Nous avons en plus réalisé l'extension de l'opération modulo pour les flottants, utile à de nombreuses reprises pour se déplacer sur un intervalle de calcul plus efficient.

❖ ULP (Unit in the last place)

L'ULP représente l'intervalle entre le flottant passé en argument et le flottant directement supérieur à celui-ci codable dans le langage.

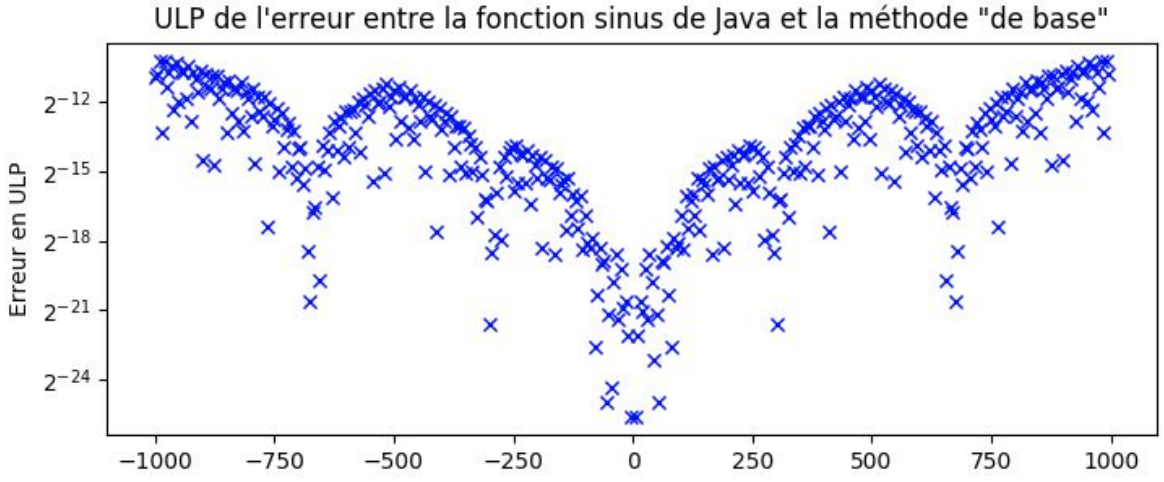
Il est calculé de la manière suivante :

```
float ulp(float f) {
		float res = _pow_opti(2, - 23);
		if (f < 0.0) {
			return ulp(- f);
		}
		if (f == 0.0) {
			return _pow_opti(2, - 149);
		}
		while (f < 1.0) {
			f = f * 2;
			res = res / 2;
		}
		while (f >= 2.0) {
			f = f / 2;
			res = res * 2;
		}
		return res;
	}
```

Il n'y a pas d'approximations sur l'ULP, à part la gestion des exceptions, la méthode est sensiblement identique à celle implémenté dans la classe Math de Java.

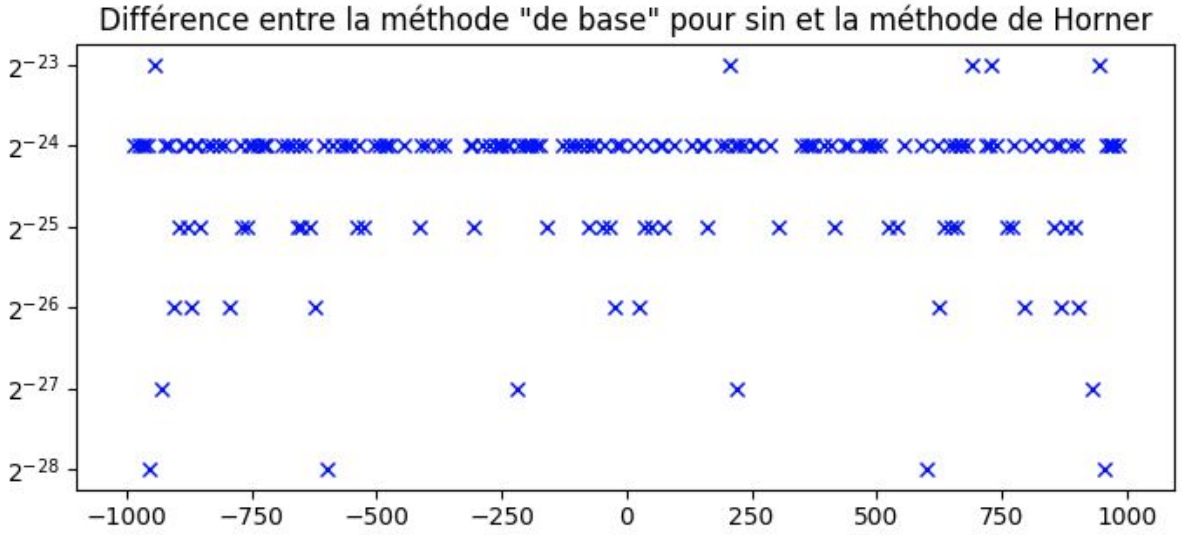
❖ Sinus

Pour les fonctions sin, cos, asin et atan, nous avons commencé par implémenter des méthodes dites “de base”, utilisant le développement limité des fonctions en 0. L’approximation étant impossible éloigné de 0, nous avons immédiatement implémenté une méthode déplaçant le calcul sur $[0; 2\pi]$ (utilisable uniquement sur sin et cos dû à leur périodicité) puis de le diviser en 4 sous-intervalles sur lesquels des transformations trigonométriques permettaient déjà d’obtenir des résultats intéressants.



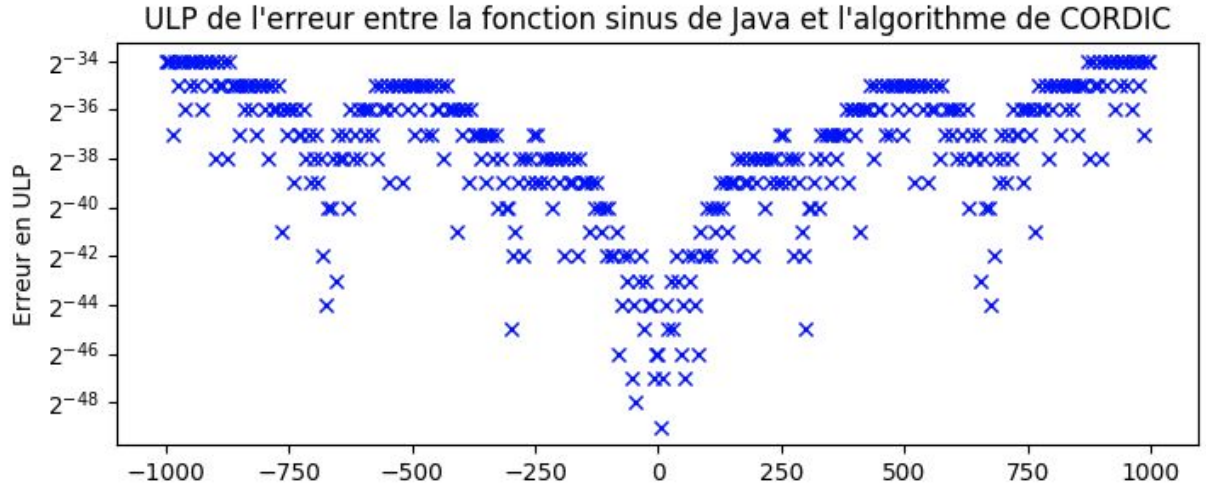
Erreur maximale : 0x1.c1a284p-11
Erreur moyenne : 0x1.469fc2p-13
Ecart-type : 0x1.8b3572p-13

Sans se lancer dans plus de documentations pour le moment, nous voulions voir quelle précision nous pouvions obtenir en utilisant uniquement des algorithmes que nous connaissions.

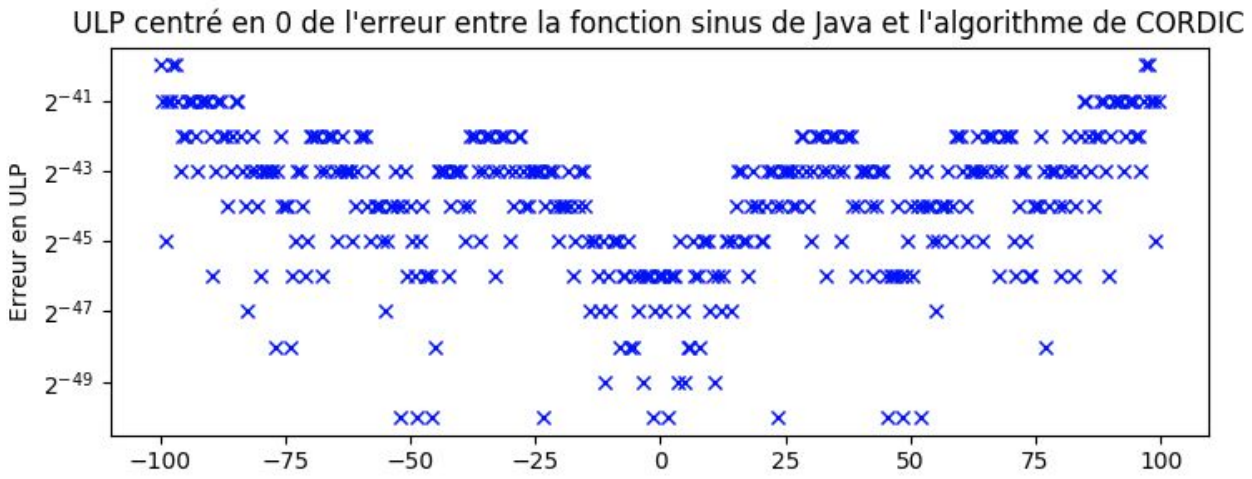


La méthode de Horner offrit finalement très peu d'améliorations par rapport à la méthode "de base" au regard de l'ordre de grandeur préalablement obtenu.

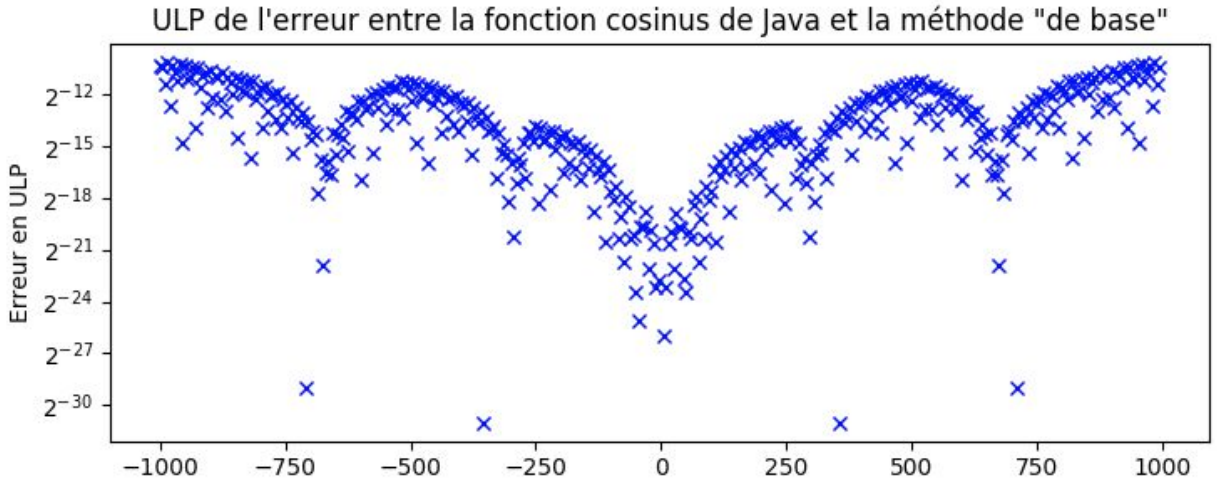
La méthode finalement conservée pour calculer la fonction sinus utilise l'algorithme de CORDIC, son implémentation propose en effet une erreur maximale de 2^{-34} :



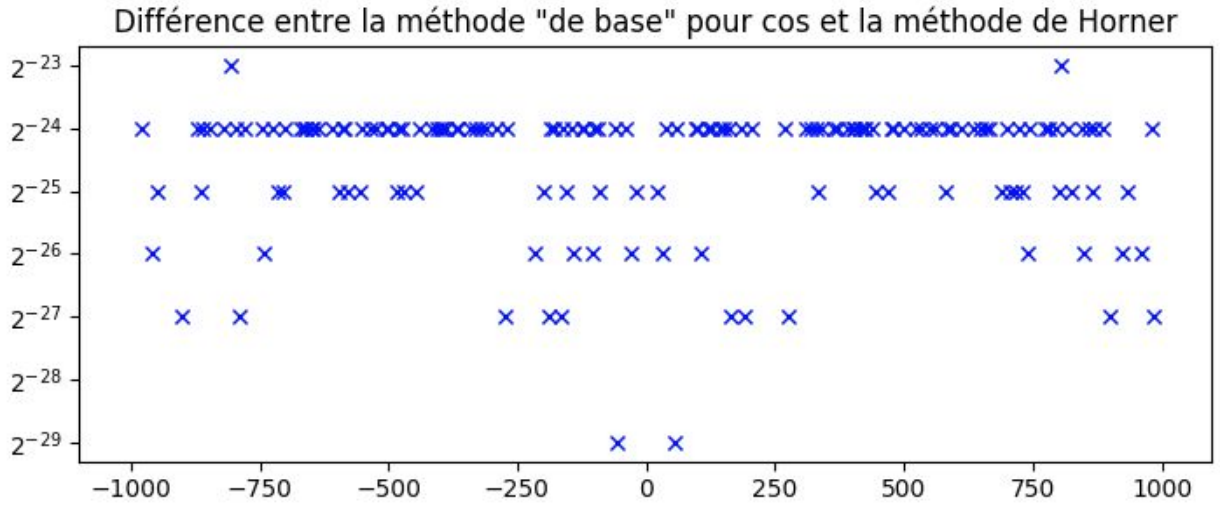
Erreur maximale : 0x1.0p-34
Erreur moyenne : 0x1.d8e81ep-37
Ecart-type : 0x1.204aecp-36



❖ Cosinus

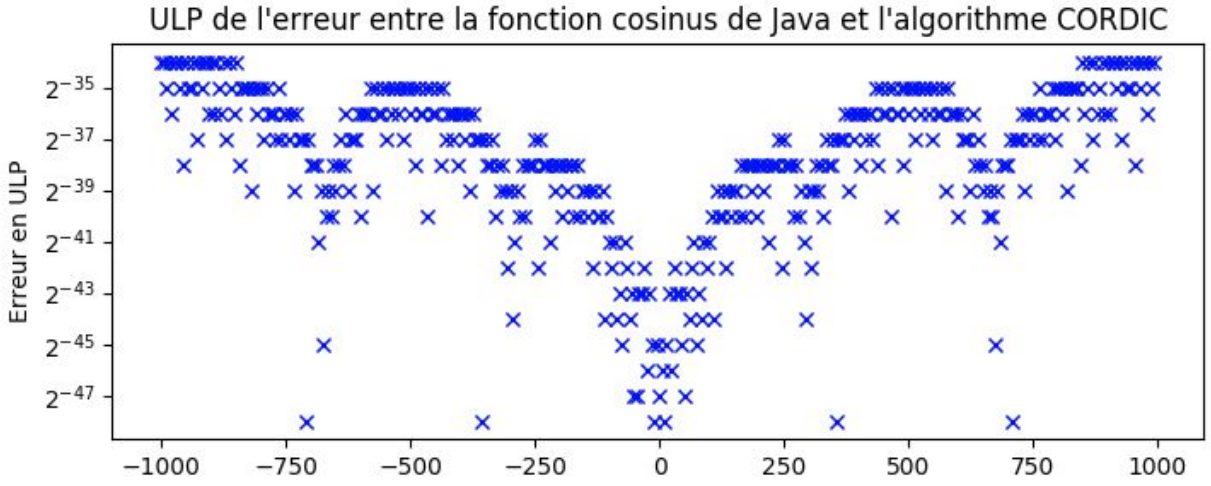


Erreur maximale : 0x1.cf498cp-11
Erreur moyenne : 0x1.46f6dap-13
Ecart-type : 0x1.8dee76p-13

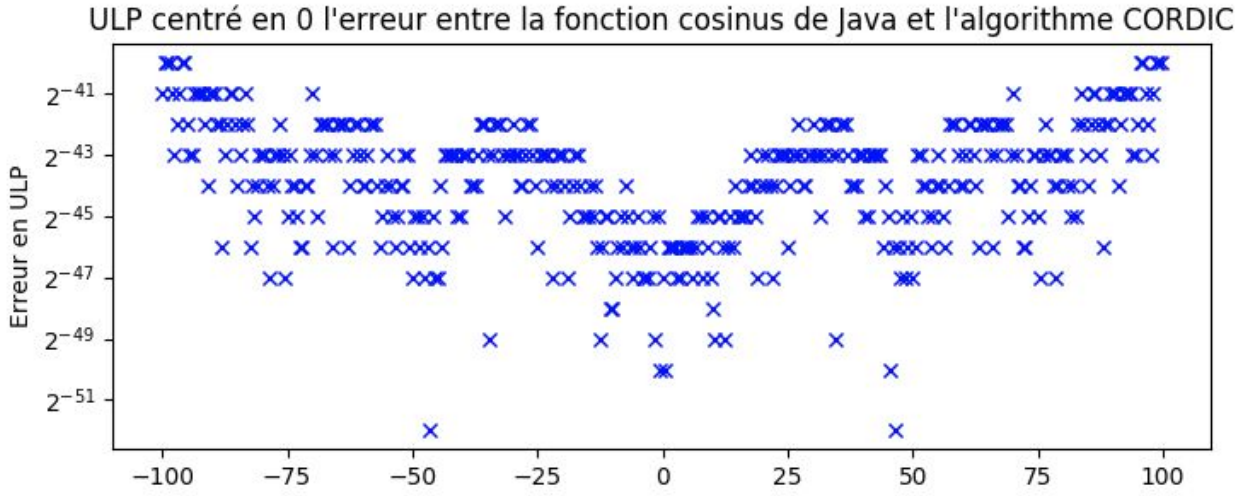


La méthode de Horner offrit finalement très peu d'améliorations par rapport à la méthode "de base" au regard de l'ordre de grandeur préalablement obtenu.

La méthode finalement conservée pour calculer la fonction cosinus utilise l'algorithme de CORDIC, son implémentation propose en effet une erreur maximale de 2^{-34} :



Erreur maximale : 0x1.0p-34
Erreur moyenne : 0x1.d18732p-37
Ecart-type : 0x1.1e30e6p-36



❖ Algorithme de CORDIC

Cet algorithme approxime les fonctions trigonométriques et hyperboliques, c'est celui qui nous a fournit la meilleure précision à ce jour. Il n'est cependant pas utilisable pour les fonctions Arctan et Arcsin.

CORDIC, pour COordinate Rotation DIgital Computer, effectue des rotations d'angles de plus en plus petits pour se rapprocher de l'angle recherché.

Dans le cas trigonométrique de cosinus et sinus on se fixe un angle $\theta \in [0; \pi/2]$. En effet, on s'y ramène facilement en utilisant les formules $\sin(\pi/2 - \theta) = \cos(\theta)$ et $\cos(\pi/2 - \theta) = \sin(\theta)$.

Il faut choisir une suite d'angles $(\theta_i)_{i\in\mathbb{N}}$ telle que $\square^{\infty}_{i=0}\theta_i = \theta$.

Notons R_i la matrice de rotation d'angle θ_i, alors on définit la suite $(v_i)_{i\in\mathbb{N}}$ de vecteurs unitaires par rotation de R_i :

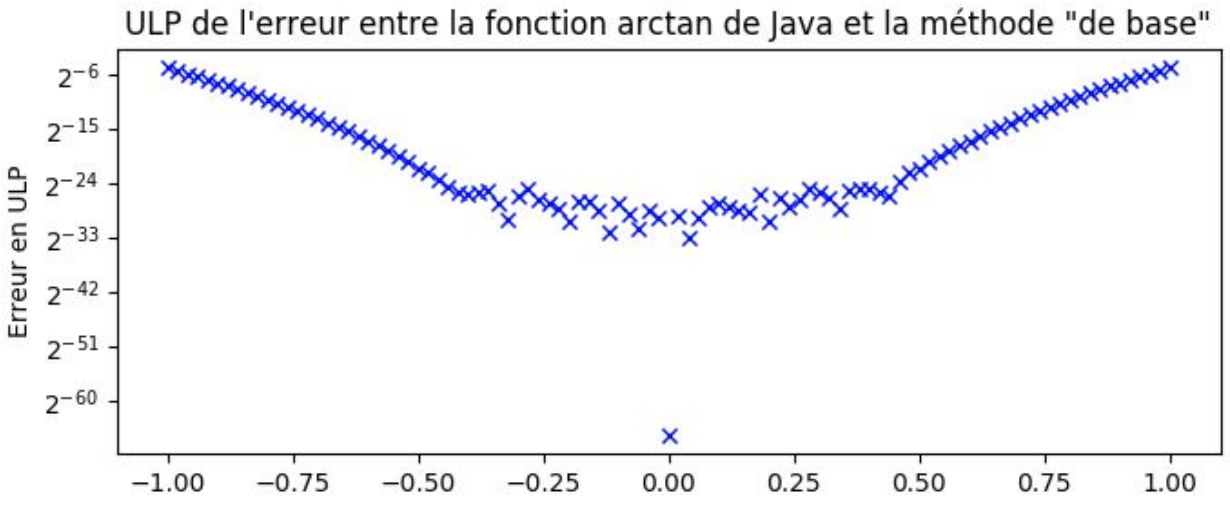
- $v_0 = (x_0 = 1, y_0 = 0)$
- $v_{i+1} = R_i v_i = \cos(\theta_i)\,(x_i - sgn_i \times \tan(\theta_i) \times y_i\,,\, sgn_i \times \tan(\theta_i) \times x_i + y_i)$ avec $sgn_m = 1$ si $\square^{m}_{i=0}\theta_i < \theta$, -1 sinon

La suite $(v_i)_{i\in\mathbb{N}}$ converge ainsi vers le vecteur v unitaire d'angle θ dont on souhaite les coordonnées $(\cos(\theta), \sin(\theta))$.

Il ne reste plus qu'à choisir la suite $(\theta_i)_{i\in\mathbb{N}}$ de manière à ce que $\tan(\theta_i)$ et $\cos(\theta_i)$ soient faciles à calculer. En prenant les θ_i tels que $\tan(\theta_i) = 2^{-i}$, soit $\theta_i = \arctan(2^{-i})$, alors $\cos(\theta_i) = 1 / \sqrt{(1 + 2^{-2i})}$. La suite $(u_i = 2^{-i})_i$ converge rapidement vers 0 et le calcul de

arctan(2^{-i}) reste très précis en utilisant la méthode “de base” ou même en utilisant l’approximant de Padé sur arctan comme décrit ci-dessous.

❖ Arctangente



Les résultats sont ici très décevants avec la méthode “de base”, même proche de 0, mais finalement peu surprenant; contrairement au sinus et au cosinus arctan n’est pas périodique, on ne peut donc que difficilement se placer sur un intervalle intéressant.

La méthode d’Horner ayant montré des résultats peu concluant (par rapport à ce qui avait été préalablement obtenu), nous n’avons pas même tenter de l’implémenter pour arctan, sachant qu’il nous fallait quoiqu’il arrive trouver une méthode plus performante.

<u>Approximation par les fractions continues de Gauss :</u>

Une fraction continue de Gauss est une fonction de la forme :
$f(x) = k + 1/(b_0 + a_1x/(b_1 + a_2x/...))$. A vrai dire, cette forme sera toujours l’approximation d’une fonction, plus le développement au dénominateur sera long, plus l’approximation sera bonne.

La fonction f(x) = arctan(x)/x est une fraction continue de Gauss. En effet, f(x) vérifie une identitée que l'on ne montrera pas ici. Les coefficients k, $(a_i)_{i\in\mathbb{N}}$ et $(b_i)_{i\in\mathbb{N}}$ valent alors :

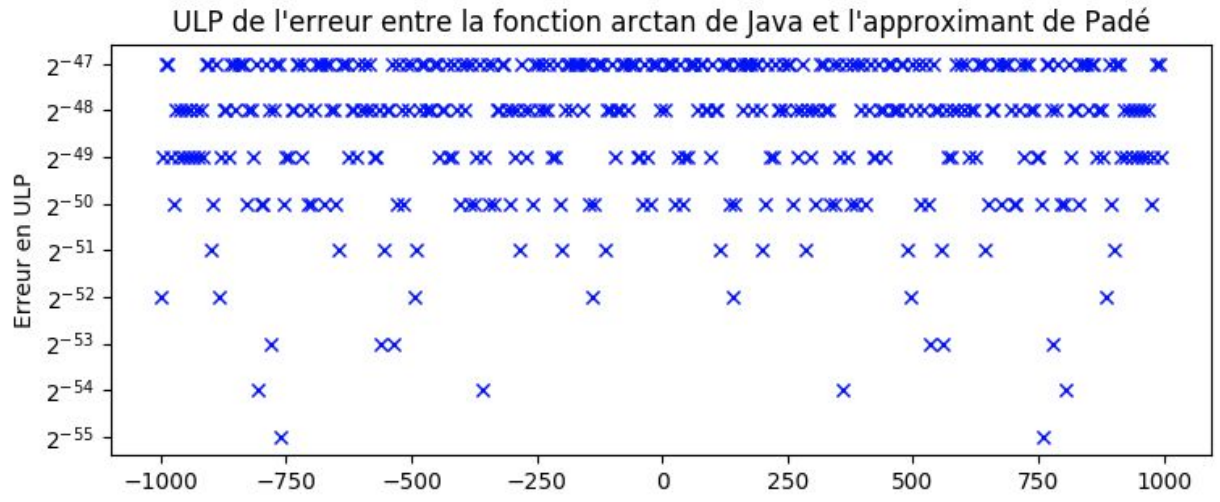
- k = 0
- $a_i = i^2$
- $b_i = 2i + 1$

On obtient alors $\arctan(x) = x/(1 + x^2/(3 + 2^2x^2/(...)))$.

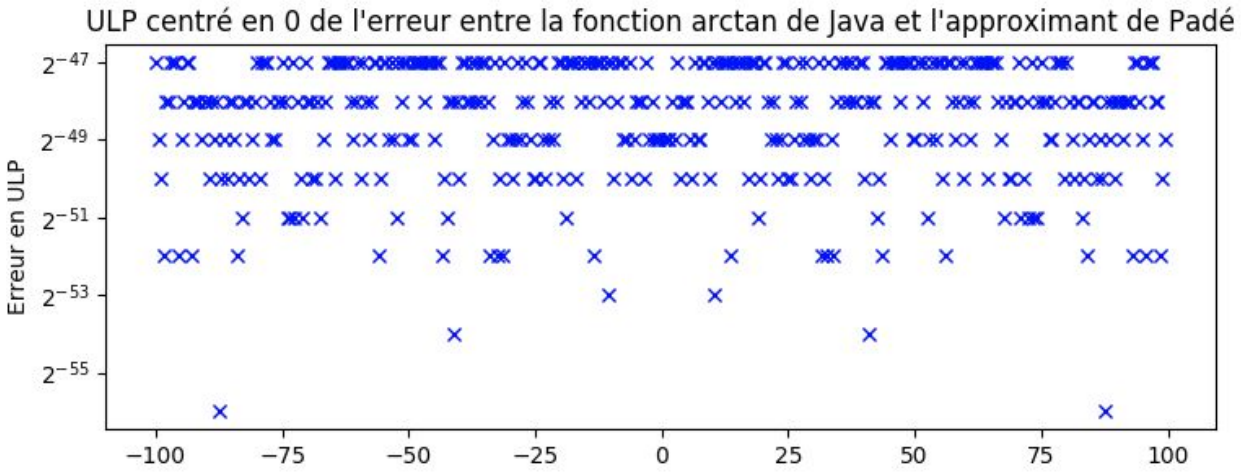
Malheureusement l'approximation par fractions continues de Gauss ne converge que sur [-1; 1].

Approximant de Padé :

La méthode conservée pour calculer l'Arc tangente d'un réel réalise l'approximant de Padé. Le fonctionnement de cet algorithme est décrit plus bas, puisqu'il convient aussi au calcul de l'Arc sinus.

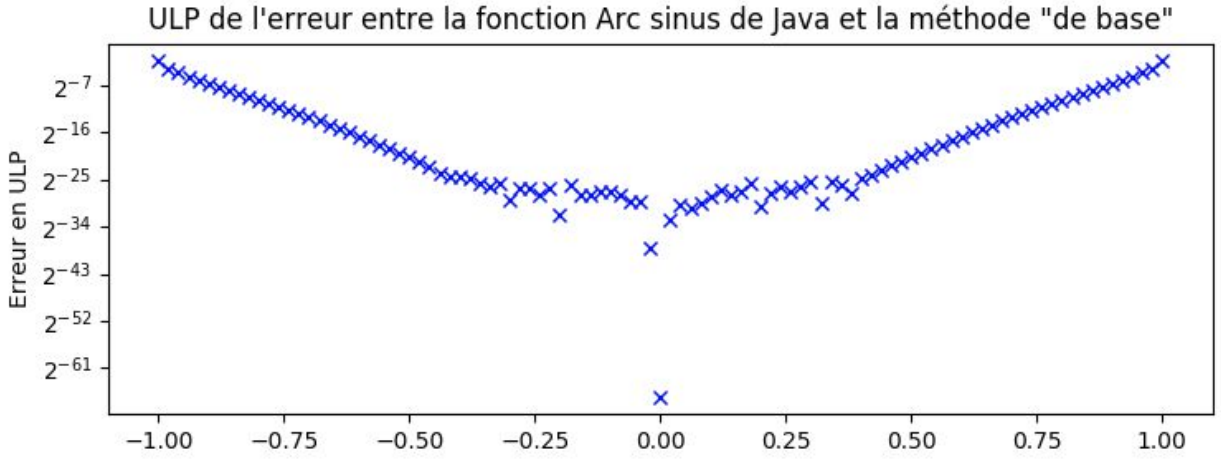


Erreur maximale : 0x1.0p-47
Erreur moyenne : 0x1.1dcdf4p-48
Ecart-type : 0x1.7369a6p-49



Comme on peut le voir sur les courbes ci-dessus, l'approximant permet d'approcher la fonction Arc tangente à 2^{-47} dans le pire cas, et cette marge d'erreur reste la même où que l'on se place.

❖ Arcsinus

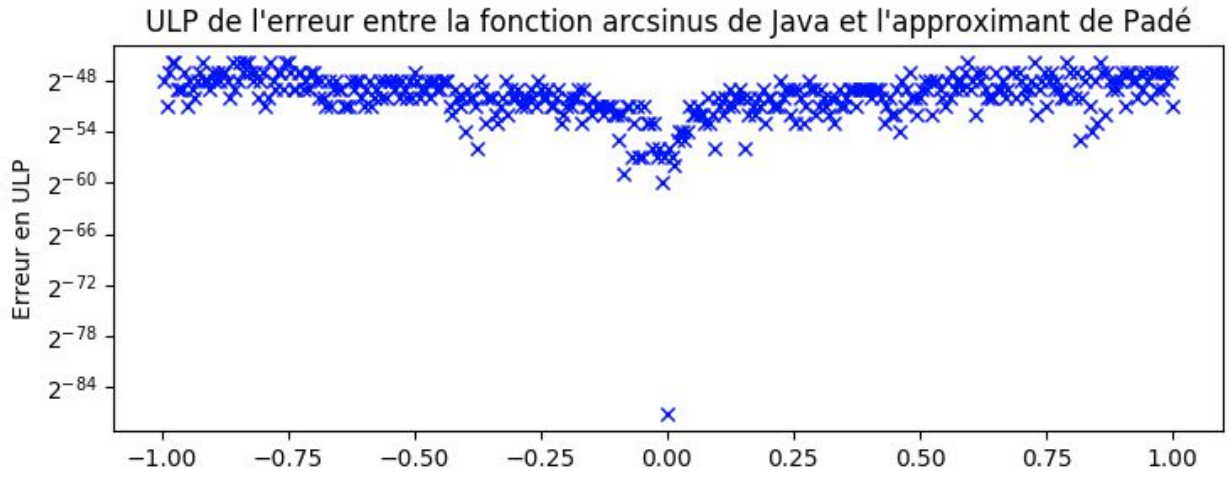


Erreur maximale : 0x1.b6a16ap-3
Erreur moyenne : 0x1.f75f0ap-8
Ecart-type : 0x1.0323aep-5

La méthode “de base” ne représente évidemment pas une méthode de calcul fiable sur [-1; 1], mais elle propose toutefois une approximation intéressante sur [-⅓; ⅓], avec un ulp de 2^{-27} en moyenne.

Comme Arc tangente, Arc sinus peut s’approcher par une fonction rationnelle, Arcsin(x)/x étant une fraction continue de Gauss. Cependant, ici aussi l’implémentation n’a pas été suffisamment concluante, avec une précision même plus faible qu’avec la méthode “de base”.

L’algorithme retenu pour calculer Arc sinus est l’approximant de Padé. Celui-ci mettant en pratique des fonctions polynomiales dont les coefficients sont compliqués à calculer comme à trouver dans les documentations appropriées, nous avons repris la méthode utilisée pour Arc tangente au moyen de la formule suivante : $\arcsin(x) = 2\arctan(x/(1 + \sqrt{(1 - x^2)}))$. Ce fut finalement une solution tout à fait satisfaisante.



Erreur maximale : 0x1.0p-46
Erreur moyenne : 0x1.63f79ap-49
Ecart-type : 0x1.bcc39cp-49

Comme pour Arc tangente, les résultats sont satisfaisants. Etrangement, l’erreur moyenne sur le calcul est même plus faible (on passe de 2^{-48} pour arctan à 2^{-49} pour arcsin) malgré les quelques opérations supplémentaires effectuées, opérations

qui ont dû amortir une partie des erreurs de précision de l'approximant de Padé et des méthodes intermédiaires utilisées.

❖ Approximant de Padé

La méthode de l'approximant de Padé approche elle aussi des fonctions analytiques (elle couvre donc un domaine plus large) par des fonctions rationnelles. Les deux méthodes sont très proches et utilisent les mêmes concepts. En fait, celle de l'approximant de Padé est même à l'origine des approximations par développement limité.

Il s'agit de l'algorithme retenu pour calculer atan ainsi que asin, il admet en effet une bonne précision au voisinage de 0 mais aussi à infini, bien que la justesse du calcul ait perdu quelques décimales.

Soit (p, q) un couple d'entiers, f une fonction analytique en 0. Un approximant de Padé d'ordres (p, q) de f en 0 est une fonction rationnelle P/Q, avec P (respectivement Q) un polynôme de degré inférieur ou égal à p (resp q), telle que le développement limité de P/Q en 0 coïncide avec celui de f jusqu'à l'ordre p + q. Il est démontré que pour (p, q) et f donnés, il existe un approximant de Padé d'ordres (p, q) de f en 0 et qu'il est unique.

Le calcul des coefficients des polynômes P et Q demandent cependant un temps et des connaissances mathématiques que nous ne possédions pas. Nous avons ainsi profité des calculs trouvés lors de la documentation sans parvenir à les comprendre.

Remarquons toutefois la puissance de l'approximation réalisée; c'est en 0 que le développement limité de P/Q est calculé, et notre implémentation montre une erreur moyenne très faible et quasiment constante sur l'intervalle considéré, alors que l'approximant de Padé n'a été calculé qu'aux ordres (4, 4).

❖ Bibliographie

- ➢ Algorithme d'exponentiation rapide : http://polaris.imag.fr/jean-marc.vincent/index.html/ALGO5/ALGO5-Exponentiation.pdf
- ➢ Méthode de Héron : https://fr.wikipedia.org/wiki/M%C3%A9thode_de_H%C3%A9ron
- ➢ Méthode de Horner : https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Ruffini-Horner
- ➢ Algorithme de CORDIC : https://fr.wikipedia.org/wiki/CORDIC
- ➢ Algorithme de la fraction continue (Gauss) : https://books.google.fr/books?id=qy83gXoRps8C&pg=PA347&lpg=PA347&dq=compute+atan+pade&source=bl&ots=9hPIlnLW85&sig=ACfU3U0MOGRVvjkNd24QnWiFUNdfoIyrVQ&hl=fr&sa=X&ved=2ahUKEwjWza_OypfnAhW0DWMBHU0yDZ8Q6AEwAXoECAkQAQ#v=onepage&q=compute%20atan%20pade&f=false https://books.google.fr/books?id=PewJAAAAQBAJ&pg=PA67&lpg=PA67&dq=compute+atan+pade&source=bl&ots=JiLCJkPo4A&sig=ACfU3U1vOdFtq5y9RRACbpuA4fFpxjrHVg&hl=fr&sa=X&ved=2ahUKEwjD96LoypfnAhWB3eAKHTbzBNsQ6AEwBHoECAgQAQ#v=onepage&q=compute%20atan%20pade&f=false
- ➢ Algorithme de Padé : https://www.developpez.net/forums/d749954/general-developpement/algorithme-mathematiques/algorithmes-structures-donnees/algorithm-fonction-atan/ https://fr.wikipedia.org/wiki/Approximant_de_Pad%C3%A9