

Documentation extension

GL37

Thomas Avare, Quentin Candaële, Helena Cazals, Germain Vu, Andry Rakotonjatovo

22 janvier 2022

Cette documentation a pour but de présenter l'implémentation de l'extension [TRIGO] en deca. Elle comprend les fonctions `cos`, `sin`, `arcsin`, `arctan` ainsi que la fonction `ULP` (Unit in the Last Place) regroupées dans la bibliothèque `Math.decah`.

Table des matières

| | |
|--|----------|
| 1 Consignes | 3 |
| 1.1 travail demandé | 3 |
| 1.2 Exigences de la classe <code>Math.decah</code> | 3 |
| 1.3 Les autres méthodes présentes | 4 |
| 2 Analyse bibliographique | 4 |
| 2.1 approximation par une série de Taylor | 5 |
| 2.2 algorithme de type CORDIC authentique respectant les contraintes matérielles de l'époque | 5 |
| 2.3 algorithme CORDIC amélioré adapté à l'IMA | 5 |
| 2.4 approximation de l'arctan (pour en déduire acos et asin) | 6 |
| 3 Choix de conception, d'architecture, et d'algorithmes. | 7 |
| 3.1 COS SIN : Justification du choix | 7 |
| 3.2 COS SIN : Fonctionnement de l'algorithme | 7 |
| 3.3 ATAN : Justification du choix | 8 |
| 3.4 ACOS ASIN : Justification du choix | 9 |
| 3.5 Calcul de π | 9 |
| 3.6 Temps de calcul | 11 |

| | | |
|----------|---|-----------|
| 3.7 | Analyse théorique de la précision | 12 |
| 4 | Test, validation et résultat | 14 |
| 4.1 | Test Graphique de nos fonctions | 14 |
| 4.2 | Flottant et calcul de précision | 16 |
| 4.3 | Test de précision | 17 |
| 5 | Améliorations | 21 |
| 5.1 | Amélioration cosinus et sinus | 21 |
| 5.2 | Amélioration arctan et arccos | 21 |
| 6 | Conclusion | 22 |

1 Consignes

1.1 travail demandé

Objectif : Implémenter quatre fonctions trigonométriques sin, cos, asin et atan, ainsi que la fonction ulp (unit in the last place) dans une bibliothèque Math.decah qui devra être utilisable par le client.

Difficulté :

- Comprendre comment les flottants sont stockés en mémoire.
- Obtenir la meilleure précision possible pour chacune de ces fonctions trigonométriques (précision à quelques bits près).
- Analyser la précision et la rapidité d'exécution de nos fonctions .

Compétences acquises :

- Comprendre les subtilités du calcul flottant.
- Comprendre et implémenter des algorithmes de calcul des fonctions trigonométriques.
- Comprendre et implémenter la fonction ULP.

1.2 Exigences de la classe Math.decah

L'utilisateur doit être capable d'utiliser les fonctions trigonométriques tout en ayant des valeurs les plus proches possibles du résultat réel. Pour cela il est important de comprendre comment la machine stocke et interprète les floats grâce à la norme IEEE 754. La classe Math.decah doit pouvoir être ajouter dans n'importe quel fichier deca grâce à l'instruction :
`#include "Math.decah".`

Ce fichier définit une classe Math avec les méthodes suivantes :

- cos :** Retourne le **cosinus** trigonométrie du flottant f (en radians).
- sin :** Retourne le **sinus** trigonométrie du flottant f (en radians).
- acos :** Retourne l'**arccosinus** trigonométrie du flottant f (en radians).
- asin :** Retourne l'**arcsinus** trigonométrie du flottant f (en radians).
- ulp :** Unit in the Last Place, c'est-à-dire “poids du dernier chiffre”.

Pour les 4 dernières méthodes, les calculs devraient dans l'idéal, renvoyer l'arrondi au float représentable le plus proche de la valeur exacte. Cependant, atteindre ce niveau de précision est très difficile, nous allons donc nous contenter d'une précision légèrement moins bonne (cf Choix de conception, d'architecture, et d'algorithmes). Vous retrouverez toutes les limitations de notre implémentation et les améliorations de celle-ci dans la partie Test,

validation et résultat.

1.3 Les autres méthodes présentes

Certaines méthodes vont nous servir pour l'implémentation de la classe Math.decah, elles sont également utilisables par l'utilisateur important une bibliothèque. Nous pouvons retrouver les méthodes suivantes :

La valeur absolue d'un flottant :

`float _abs(float val)` : Retourne la valeur absolue d'une valeur flottante.

La factoriel d'un entier :

`int _factoriel(int val)` : Retourne le factoriel d'une valeur entière.

La racine carrée d'un flottant :

`float _sqrt(float x)` : Renvoie la racine carrée positive correctement arrondie d'une valeur (`float` calculé à partir de l'algorithme de Heron).

puissance d'un entier :

`float puiss(int a, int b)` : Renvoie la valeur du premier argument élevé à la puissance du deuxième argument.

2 Analyse bibliographique

Cette partie consiste en l'étude des documentations existant traitant du sujet. Dans notre cas, il s'agit d'analyser et d'apporter une critique aux algorithmes permettant d'implémenter une fonction trigonométrique. L'objectif étant de trouver les algorithmes dont nous allons nous servir par la suite.

Nombreux sont les algorithmes permettant ces calculs. Nous en avons sélectionné quelques-uns afin d'apporter une analyse de la vitesse d'exécution, de la complexité ainsi que de la précision de chacun de ces algorithmes.

Pour la fonction Cosinus et Sinus :

1. Grâce à une approximation par une série de Taylor.
2. Grâce à un algorithme de type CORDIC authentique respectant les contraintes matérielles de l'époque
3. Grâce à un algorithme CORDIC amélioré adapté à l'IMA.
4. Pour une précision dynamique, il peut être pertinent d'utiliser le développement de Taylor.

Pour la fonction Acos, Asin :

5. Grâce à une approximation de l'arctan (pour en déduire acos et asin).

Nous allons lister les avantages et inconvénients de certaines de ces méthodes.

2.1 approximation par une série de Taylor

Source : Les algorithmes cachés dans les calculatrices : TRIGONOMETRIE (<https://www.youtube.com/watch?v=11LNbe6n9Ek>)

| Avantages | Inconvénients |
|--|--|
| Simplicité de mise en place La précision peut être améliorée grâce à la méthode de Horner. | Convergence lente, le temps de calcul est grand avec le calcul des puissances et des divisions. Très vite imprécis pour les grands nombres |

2.2 algorithme de type CORDIC authentique respectant les contraintes matérielles de l'époque

Source :

- L'algorithme CORDIC par l'association mathématique du Quebec
<https://www.amq.math.ca/wp-content/uploads/bulletin/vol55/no4/09-maitre-CORDIC.pdf>
- Hardware implementation of the elementary functions by digit-by-digit (CORDIC) technique, Vladimir Baykov
<http://baykov.de/CORDIC1972.htm>

| Avantages | Inconvénients |
|---|--|
| Peut être mise en place même sur des micro-contrôleurs simples Flexible, il permet de calculer plusieurs fonctions avec quasiment le même code. Permet d'obtenir une précision déterminée à l'avance en effectuant un nombre d'itération donné. | Exécution longue (pas la meilleure solution pour un micro-contrôleur plus complexe comme le nôtre) |

2.3 algorithme CORDIC amélioré adapté à l'IMA

Source : L'ALGORITHME CORDIC (Christophe.Devalland@ac-rouen.fr)

<https://www.apmep.fr/IMG/pdf/cordic.pdf>

| Avantages | Inconvénients |
|--|---|
| Rapidité de convergence Précision grande pour les petits nombres La précision peut être choisie selon la manière de construire l'algorithme, voir la partie Choix de conception, d'architecture, et d'algorithmes. | Précision moyenne pour les grands nombres voir partie concernant les perspectives d'amélioration. |

2.4 approximation de l'arctan (pour en déduire acos et asin)

Détail de l'algorithme à ATAN : Justification du choix

| Avantages | Inconvénients |
|--|--|
| Convergence rapide → Bonne précision (cf Test de précision) | Demande de coder la fonction <code>arctan</code> pour en déduire <code>arccos</code> . |

3 Choix de conception, d'architecture, et d'algorithmes.

Cette partie détaille les choix techniques et les architectures que nous avons choisis d'implémenter pour notre classe Math.decah. Il se décompose en trois parties

- COS SIN : Justification du choix
- ATAN : Justification du choix et ACOS ASIN : Justification du choix
- Justification et fonctionnement de l'algo ulp

3.1 COS SIN : Justification du choix

Notre choix s'est naturellement tourné vers l'algorithme 2.3 algorithme CORDIC amélioré adapté à l'IMA, car étant la plus adaptée à notre projet. Plus rapide que les deux autres, il converge plus rapidement. Il demande un micro-processeur un minimum complexe. Ça tombe bien.. nous avons ça sous la main !

Cette méthode est implantable dans notre langage DECA car celle-ci n'utilise que les opérations simples (+, -, ×, /, **while**).

Nous allons détailler les spécificités de l'algorithme, son fonctionnement ainsi que les différentes sources d'imprécision dans la partie suivante.

3.2 COS SIN : Fonctionnement de l'algorithme

L'objectif étant de calculer le cosinus (ou le sinus) d'un angle ' a ' quelconque. On décompose l'angle en une somme de plusieurs angles de plus en plus petits (afin d'assurer la convergence).

On a alors : $a = a_1 + a_2 + \dots + a_n$

On converge alors vers l'angle a de la façon suivante (voir fig 1) :

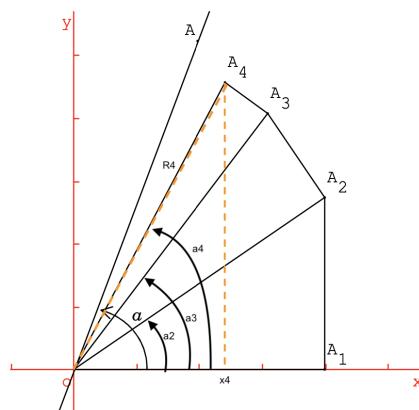


FIGURE 1 – Convergence de la somme d'angle vers l'angle

En construisant notre angle de cette manière on remarque que

$$\cos(a) = \frac{X_n}{R_N} \text{ et } \cos(a) = \frac{Y_n}{R_N}$$

On démontre ensuite les relations suivantes :

$$\begin{aligned}\forall i \in \{1, \dots, n\} x_{i+1} &= x_i - \tan(a_i) y_i \\ y_{i+1} &= y_i - \tan(a_i) x_i \\ R_{i+1} &= R_i \sqrt{1 + \tan(a_i)^2}\end{aligned}$$

Il nous suffit de connaître x_1 , y_1 et R_1 ainsi que l'ensemble des valeurs de $\tan(a_i)$ afin de calculer X_n et R_n et en déduire la valeur de \cos (il en est de même pour le sinus avec Y_n et R_n).

On prend donc les angles dans l'intervalle $[0; \pi/2[$ tels que $\tan(a_i) = 10 - i$. Cela simplifiera les calculs par la suite.

Pour a_i assez petit ($a_i < 10^{-4}$) on peut légitimement approximer $\tan(a_i)$ par a_i (déduction faite du développement limité de la fonction tangente), on rappel :

$$\tan(x) \underset{x \rightarrow 0}{=} x + o(x^3)$$

x^3 devient vite négligeable devant x sur un voisinage de 0.

Pour $i > 5$ on peut donc aisément considérer que $\tan(a_i) = a_i$ si nous souhaitons une précision à 10^{-8} près.

Pour le calcul de notre cosinus, il suffit de stocker en dur les valeurs de $\tan(a_i)$ dans notre classe `Math.decah` (légère difficulté qu'il faut contourner, puisqu'il n'est pas possible de stocker des valeurs dans un tableau en déca).

Ensuite, pour un angle donné a , il faut soustraire autant de fois qu'il est possible les angles a_1, a_2, a_3 , etc... jusqu'à ce que le reste soit inférieur à la précision désirée.

3.3 ATAN : Justification du choix

Nous avons choisi d'implémenter une fonction permettant de calculer l'`arctan` puis de déduire de celui-ci l'`arcos` et l'`arcsin`.

Solution mise de côté :

La fonction `arctan` est facile à décrire avec un développement en série entière mais contrairement aux autres fonctions trigonométriques précédentes, la suite des termes de la série ne

convergent pas aussi vite.

En effet les coefficients ne comportent pas de termes en factoriels aux dénominateurs. Cela va s'amplifier dès que x s'approche de 1 en valeur absolue.

Solution retenue :

Nous sommes partie d'une des multiples applications de la moyenne arithmétique-géométrique. Elle est une valeur intermédiaire obtenue comme limite de deux suites adjacentes satisfaisant une relation de récurrence qui reprend les formules de moyennes arithmétique et géométrique.

Cela va nous permettre de déduire l'arctangente d'un nombre avec très peu d'itérations.

$$\begin{cases} a_{i+1} = \frac{a_i + b_i}{2} \\ b_{i+1} = \sqrt{a_i b_i} \end{cases} \quad \begin{cases} a_0 = \frac{1}{\sqrt{1+x^2}} \\ b_0 = 1 \end{cases}$$

$$a_{AGmean} = \lim_{n \rightarrow +\infty} a_n = \frac{x}{\sqrt{1+x^2} \arctan(x)}$$

Il s'agit donc de choisir le nombre d'itération en fonction de la précision souhaitée.

L'imparité de la fonction arctan nous permet de ramener la résolution sur \mathbb{R}^+ .

3.4 ACOS ASIN : Justification du choix

La fonction atan implémentée étant d'une bonne précision (Cf Test de précision), et racine carré également (algorithme de Héron converge rapidement), nous allons nous contenter de déduire **acos** (et **asin**) grâce à la simple formule géométrique ci-dessous.

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

$$\arccos(x) = 2 \arctan \left(\frac{\sqrt{1-x^2}}{1+x} \right), \quad -1 \leq x \leq 1$$

detail : <http://www.panamaths.net/Documents/Exercices/SolutionsPDF/26/TRIGOC00005.pdf>

3.5 Calcul de π

Pour le calcul de pi, on utilise 2 méthodes de calculs différents.

Méthode 1 : On stocke de pi dans 2 flottants pour simuler un double de java. On stock une première de pi et on stock le reste des décimales de pi sur un autre flottant.

L'avantage du 2e flottant par rapport à un int qui nous aurait permis de stocker plus décimales est on l'ordre de la première décimale stockée dans ce float.

Méthode 2 : On calcul directement la valeur de $n\pi$ à l'aide d'une suite convergente, la suite de Gregory Leibniz un peut adapté. Celle-ci converge plutôt lentement mais est très simple à implémenter. Pour calculer $n\pi$ on multiplie simplement chaque terme de la suite par n .

En pratique, on utilise plus souvent la première méthode car on a pas besoin de plus de décimale mais la deuxième méthode nous permet en théorie d'avoir autant de décimale que l'on veut dans certaines limites techniques et physiques.

3.6 Temps de calcul

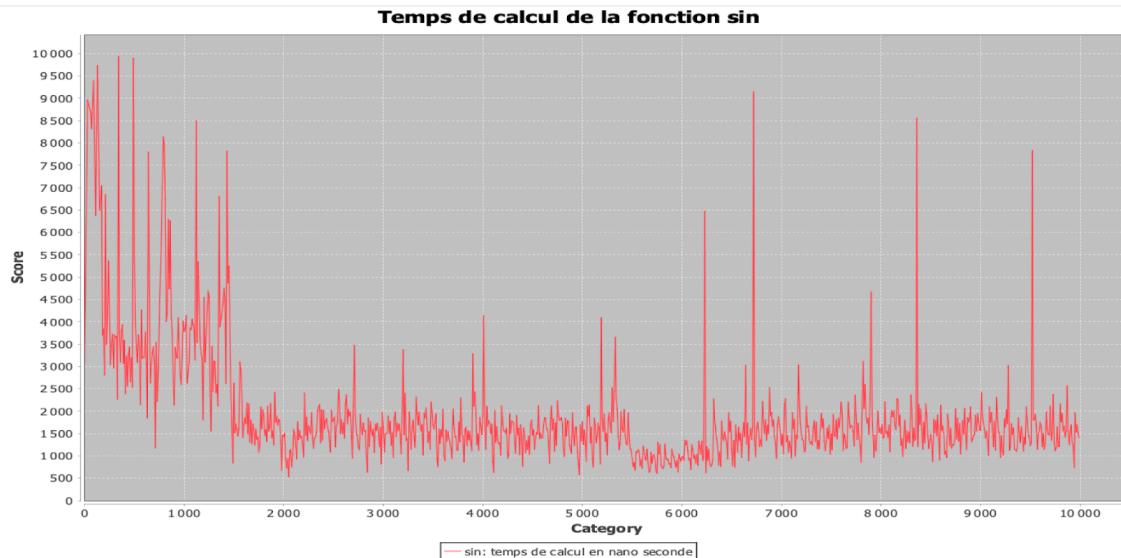


FIGURE 2 – temps de calcul pour la fonction sinus sur l'intervalle $[0, 10000]$ avec un pas de 10

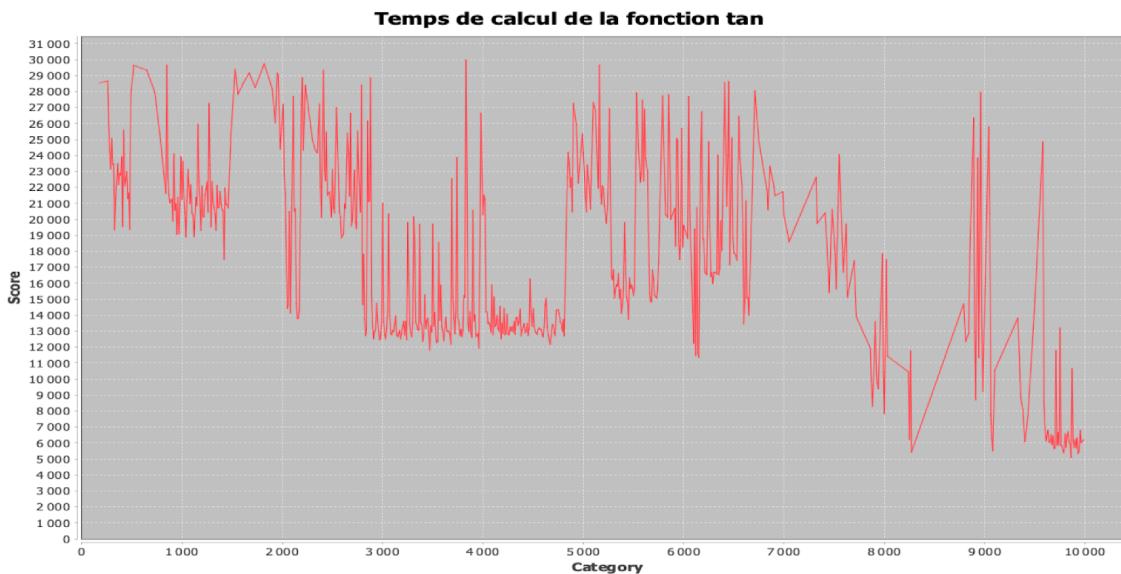


FIGURE 3 – temps de calcul pour la fonction tangente sur l'intervalle $[0, 10000]$ avec un pas de 10

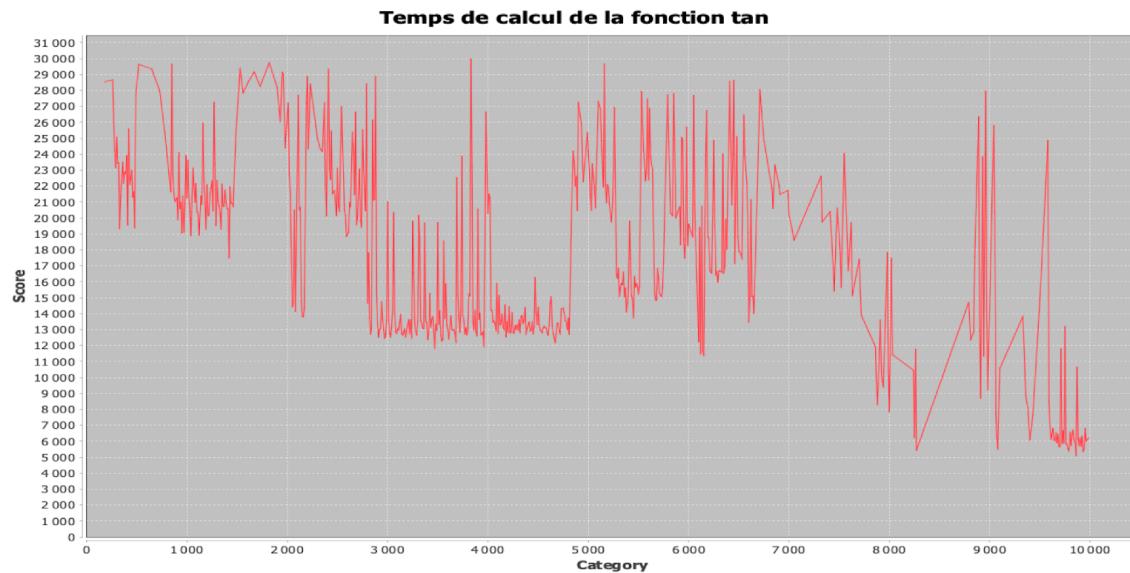


FIGURE 4 – temps de calcul pour la fonction arctangeante sur l'intervalle $[0, 10000]$ avec un pas de 10

On voit que le temps de calcul est relativement stable et assez court. C'est une bonne chose et montre bien que la complexité de nos algorithme ne dépend pas de l'angle à calculer.

3.7 Analyse théorique de la précision

L'analyse théorique de la précision n'a malheureusement pas été faite. avec plus de temps, cette partie peut s'avérer cruciale car elle permet de grandement améliorer les performances des différents algorithmes car ce si seront alors dynamiques plutôt que statique (comme le calcul de $n\pi$). Voici cependant une source pour le calcul de la précision du calcul de l'`arctan`.

```

var
  x,a,b,atg,pi:extended;
  n:integer;
begin
  write('x=');readln(x);
  a:=1/sqrt(1+x*x);
  b:=1;
  for n:=1 to 11 do           {precision relative 2.E-7}
  begin
    a:=(a+b)/2;
    b:=sqrt(a*b);
  end;
  atg:=x/ (sqrt(1+x*x) *a);
  writeln('arctg(x)=' ,atg);
  goto 1;
  readln;
end.

```

La précision relative est environ $0,8 \cdot 10^{-\frac{3}{5}n_{\max}}$

FIGURE 5 – precision de l'algorithme

detail : https://les-mathematiques.net/vanilla/index.php?p=discussion/279172#Comment_279172

(La preuve est laissé en exercice au lecteur.)

4 Test, validation et résultat

Après avoir implémenté les différents algorithmes de calcul, il est pertinent de valider les algorithmes et cela en passant par des tests empiriques.

Les fonctions ont été implémentées en java afin de pouvoir réaliser une multitude de tests (graphique, de précision...).

4.1 Test Graphique de nos fonctions

Un premier test, approximatif, consiste à tracer graphiquement nos fonctions en java à l'aide de la librairie JFreeChart. L'avantage d'un test graphique est qu'il nous permet de visualiser graphiquement si nos fonctions semblent fonctionner.

Ils nous ont permis également de nous rendre rapidement qu'avec l'algorithme CORDIC, les calculs de grands nombres deviennent trop imprécis (Cf ??).

Des captures d'écran du tracé des 4 fonctions trigonométriques écrites en java et tracé sont dispo sur les figure 6, figure 7, figure 8 et figure 9.

Les résultats semblent se superposer parfaitement avec ceux de la classe Math de java.

Mais cela est loin d'être suffisant, il faut désormais étudier la ?? de nos résultats. Mais avant cela, un petit point sur les flottants.

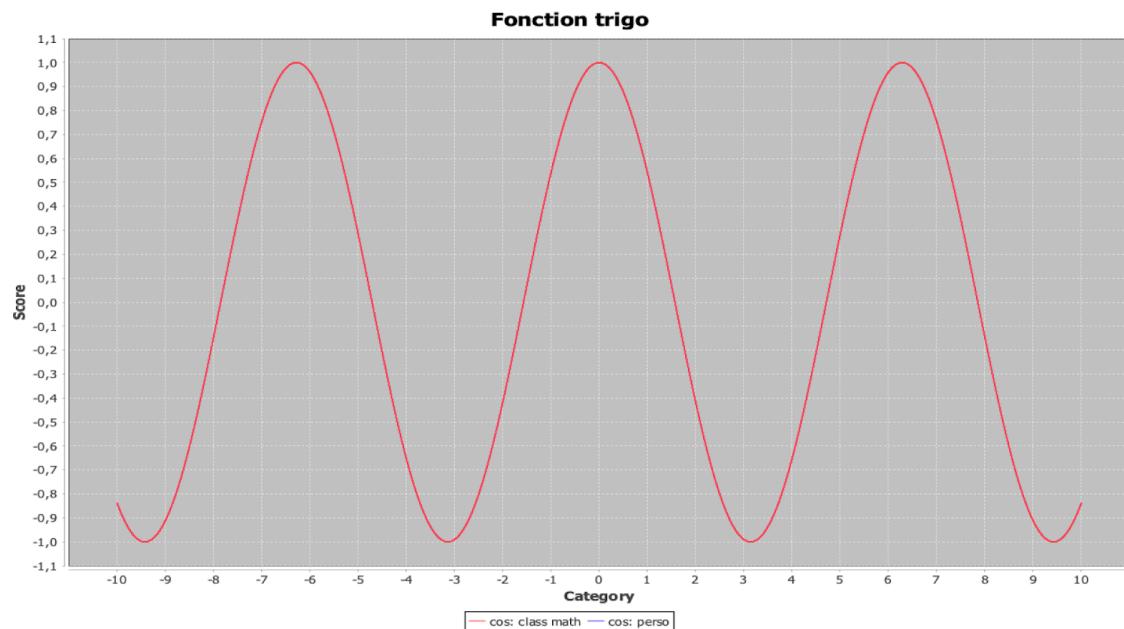
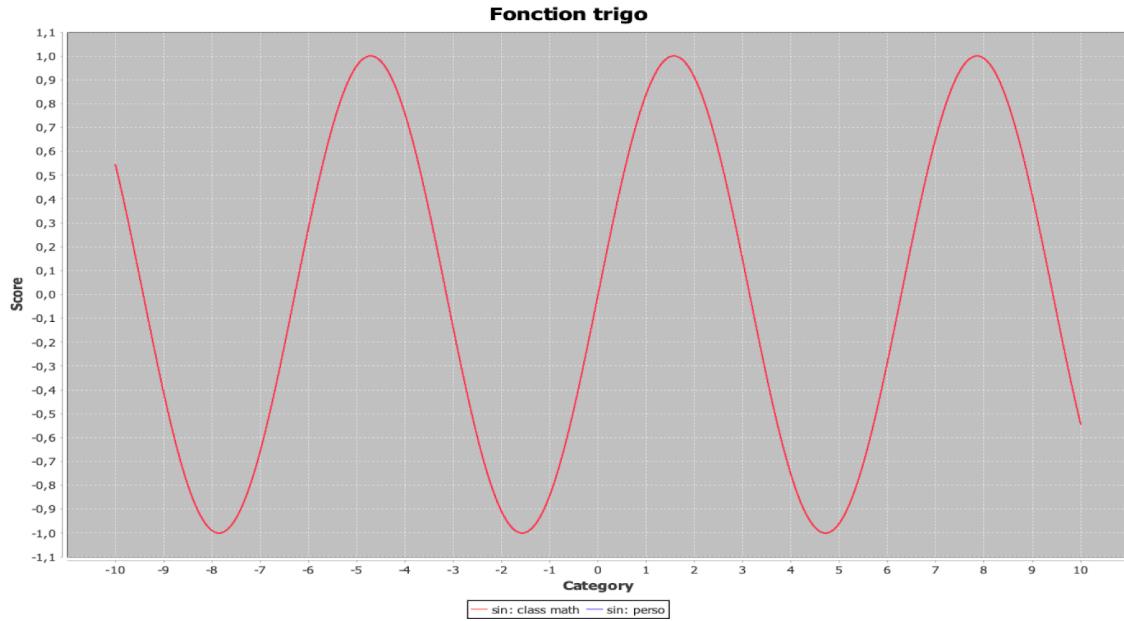


FIGURE 6 – tracé du cosinus entre -10 et 10 avec l'algorithme de CORDIC superposé avec celui de java



!p.

FIGURE 7 – tracé du sinus entre -10 et 10 avec l'algorithme de CORDIC superposé avec celui de java

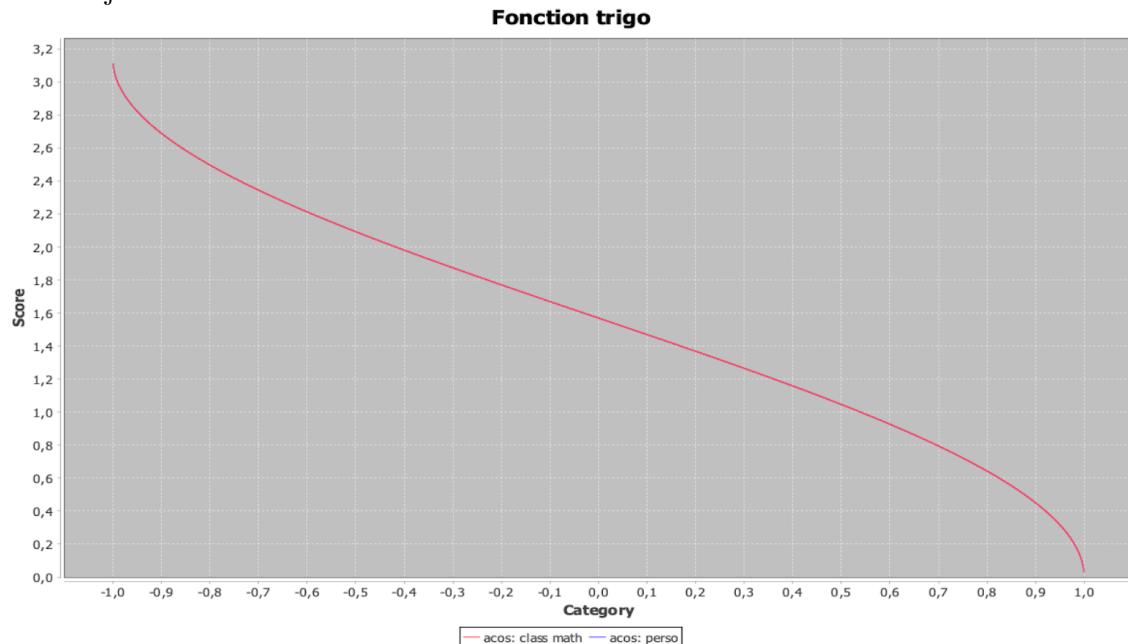


FIGURE 8 – tracé du arccosinus entre -10 et 10 avec l'algorithme de CORDIC superposé avec celui de java

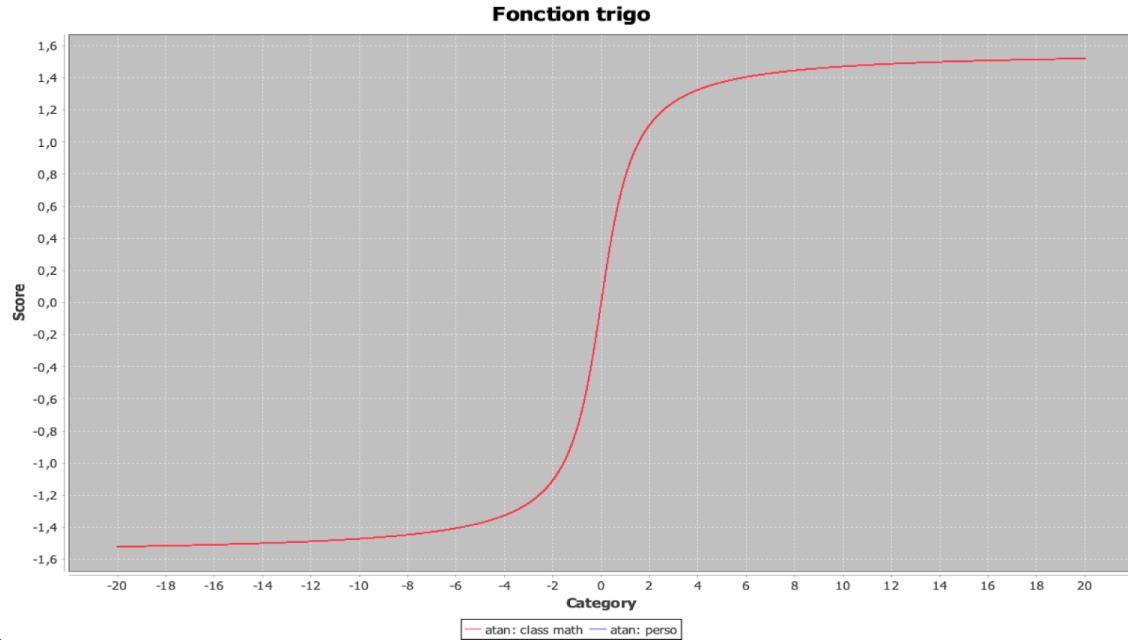


FIGURE 9 – tracé du arctan entre -10 et 10 avec l'algorithme de CORDIC superposé avec celui de java

4.2 Flottant et calcul de précision

Afin de pouvoir tester la précision de nos algorithmes, il est essentiel de pouvoir afficher les résultats en hexadécimal.

En effet l'affichage d'un flottant en décimal fait un arrondi, et n'affiche pas la valeur exacte contenue dans le flottant. Par exemple, 0x1.FFFFEP0 et 0x1.0p1 sont deux flottants différents, mais tous deux affichent 2.00000e+00 en décimal. Inversement, la lecture d'un littéral flottant décimal se fait avec un arrondi par le compilateur, alors que les littéraux hexadécimaux peuvent représenter tous les flottants possibles.

Nous avons donc réalisé un algorithme permettant de mesurer le nombre de bits d'écart entre la valeur réelle (celle de la classe math de java) et la valeur que nous trouvons avec nos algorithmes. Avant cela il faut comprendre comment les flottants sont stockés en mémoire.

Exemple du stockage du flottant 6.25 en mémoire :

Admettons que l'on souhaite stocker le flottant 6.25 en suivant la norme IEE 754. Alors la valeur stockée en mémoire sera la valeur en bit correspondant à l'hexa 0x40c80000.

En effet on a :

$$6.25 = 1.5625 \times 2^2$$

⇒ Donc $2 + 127 = 129 \rightarrow$ La valeur 129 sera stockée dans l'exposant.

⇒ 1.5625 est la valeur qui sera encodé dans la mantisse (0.5625 plus exactement car le 1 devant est sous entendu) $0.5625 = 2^{-1} + 2^{-4}$

On obtient donc stocké en mémoire la valeur 01000000110010000000000000000000.

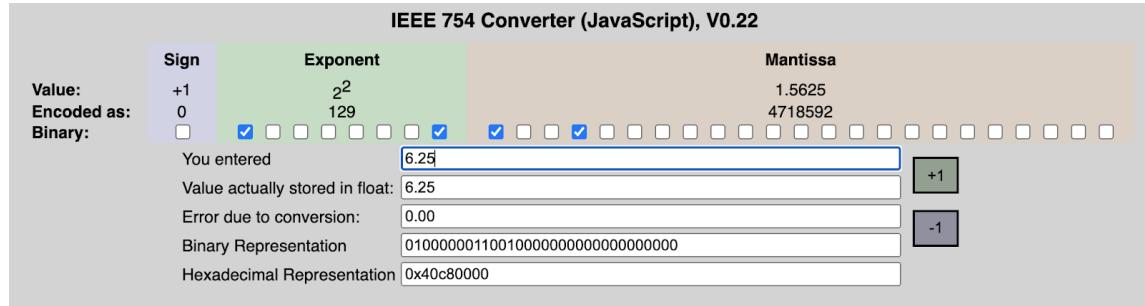


FIGURE 10 – Représentation du flottant 6.25 en mémoire

4.3 Test de précision

Afin de tester la précision, il suffit de regarder le nombre de bits d'écart en mémoire entre la valeur attendue et celle obtenue.

Nous allons étudier la précision pour la fonction `sinus` et `arctan` car :

- La fonction `cosinus` est construite de la même manière que Sinus (Algo Cordic amélioré + décalage de $n * \pi$)
- Notre fonction Acos est déduite de l'Atan (cf partie calcul acos 3.4)

Précision fonction Sinus et arctan sur l'intervalle $[-\pi, \pi]$ avec un pas de 2^{-22}

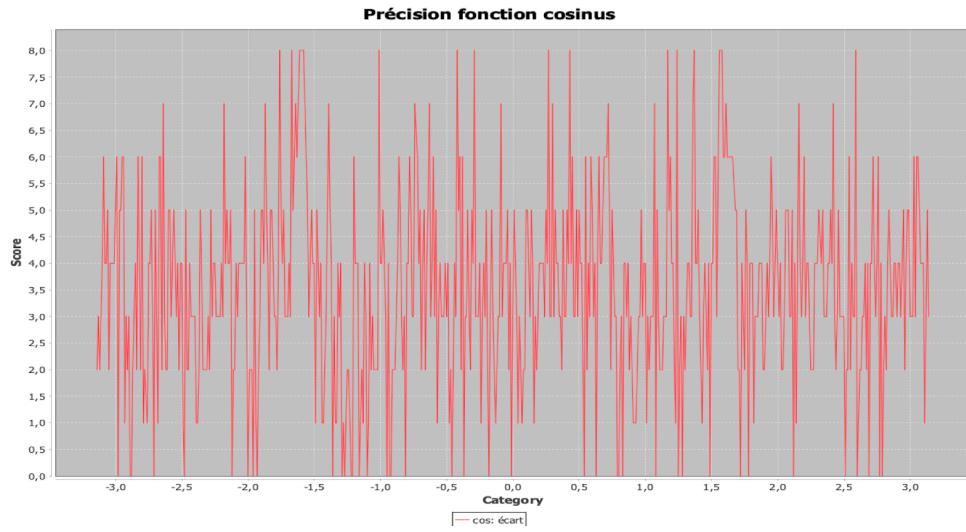


FIGURE 11 – Nombre de Bits d'écart entre le résultat réel du calcul de $\cos(x)$, $x \in [-\pi, \pi]$ et celui obtenu avec notre algorithme.

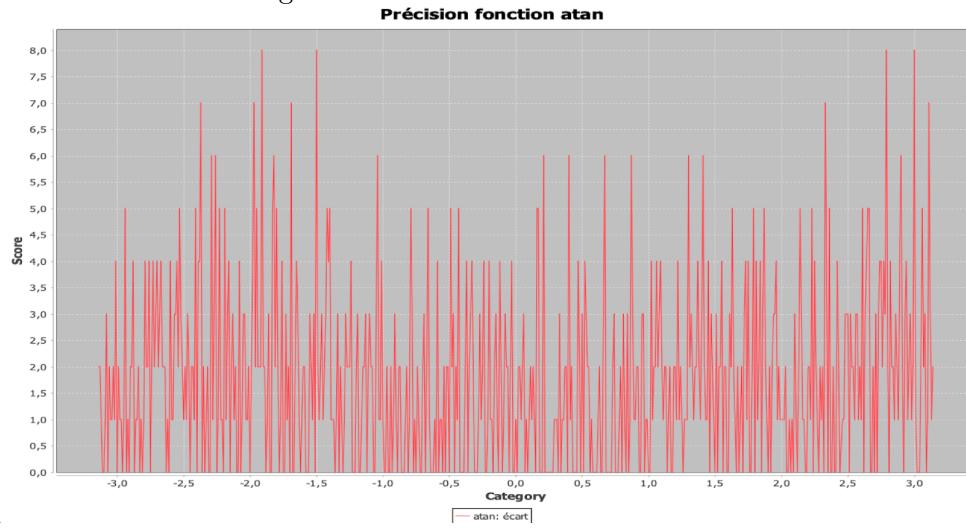


FIGURE 12 – Nombre de Bits d'écart entre le résultat réel du calcul de $\text{atan}(x)$, $x \in [-\pi, \pi]$ et celui obtenu avec notre algorithme.

Précision moyenne de la fonction Sinus et arctan sur l'intervalle $[-\pi, \pi]$

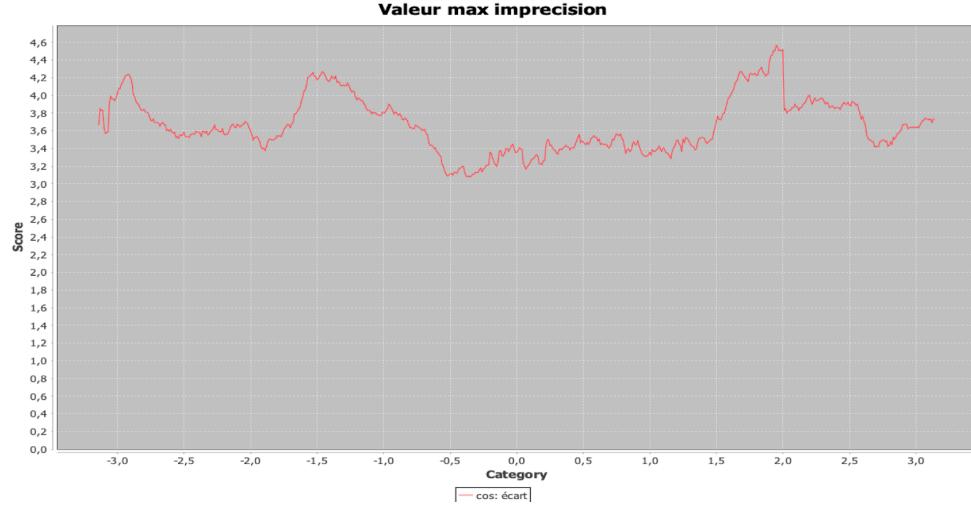


FIGURE 13 – Moyenne du nombre de bits d'écart entre le résultat réel du calcul de $\cos(x)$, $x \in [-\pi, \pi]$, et celui obtenu avec notre algorithme.

Précision moyenne de la fonction Sinus et Atan sur l'intervalle $[0; 100\pi]$

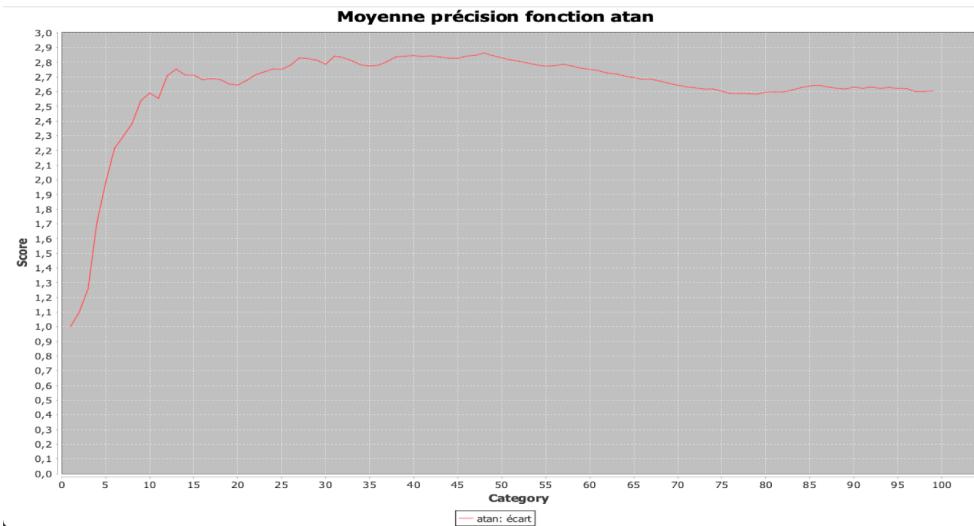


FIGURE 14 – Moyenne du nombre de bits d'écart entre le résultat réel du calcul de atan(x), $x \in [0, 100]$ et celui obtenu avec notre algorithme.

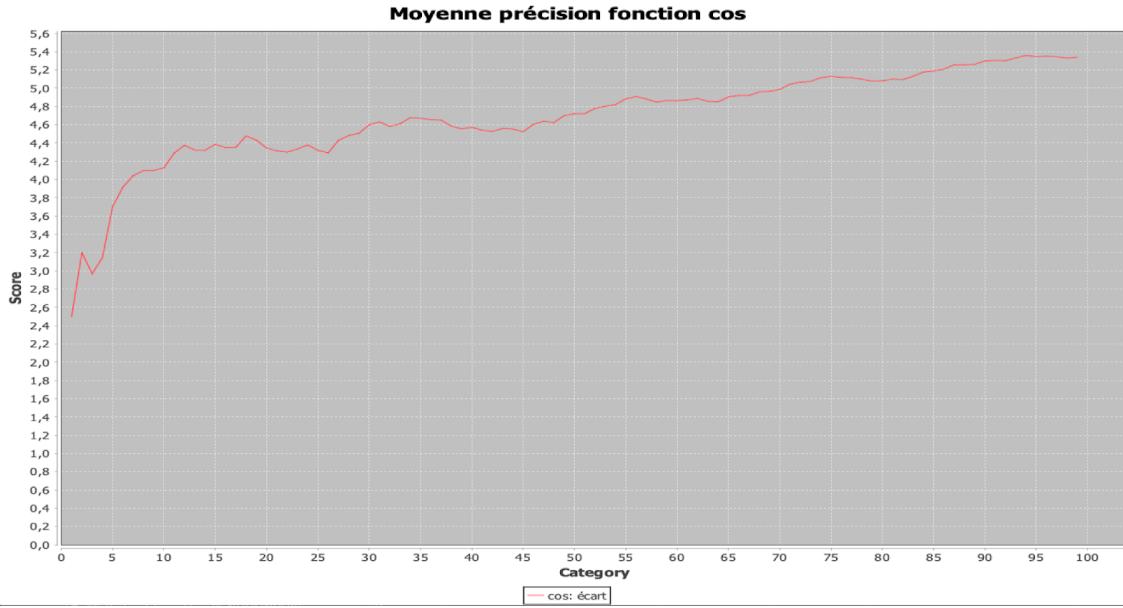


FIGURE 15 – Moyenne du nombre de bits d'écart entre le résultat réel du calcul de $\cos(x)$, $[0, 100]$ et celui obtenu avec notre algorithme.

remarque. *figure 11 : Gros écart de précision d'une valeur à l'autre. Cela se comprend lorsque l'on comprend comment est stocké un flottant dans notre machine. L'erreur est en partie dû à l'approximation lors du stockage de notre valeur, expliquant pourquoi un calcul avec un double (sur 64 bits) est beaucoup plus précis.*

Conclusion :

- Nous réalisons que la fonction CORDIC utilisée à une bien meilleure précision que la méthode d'approximation par une série de Taylor conçue à partir d'un simple développement de Taylor, mais qu'elle est encore perfectible.
- Pour certaines valeurs bien précises, nous pouvons avoir jusqu'à deux octets d'écart ce qui n'est pas négligeable.

Amélioration possible (Cf Améliorations)

remarque. *figure 12 : Pour certaines valeurs bien précises le nombres de bits de différence peut monter jusqu'à 2 octets d'écart ce qui n'est pas négligeable.*

On remarque pourtant dans la figure 12 que le nombre de bits d'écart moyen est plutôt correct entre $[-\pi, \pi]$, avec un moyenne autour des 1,5 bits d'écart.

L'algorithme se montre imprécis pour certaines valeurs très précises.

Amélioration possible (Cf Améliorations)

remarque. *figure 14 : La précision de la fonction cos est de moins en moins bonne lorsque l'on s'éloigne de 0.*

→ Nous réalisons un décalage de $n * \pi$ pour ramener le calcul vers des plus petites valeurs

C'est lors de ce décalage que nous perdons de la précision

Conclusion :

- Approximation dû à la limite du nombre de décimales de π pouvant être stocké. La multiplication de l'approximation de π un grand nombre de fois rend le résultat final bien trop approximatif.
- Un algorithme calculant directement la valeur de $n\pi$ (sans faire n fois π) est une solution envisageable mais qui n'a pas changé la précision de manière significative. On reste limité par le nombre de décimales pouvant être stocké pour les grands nombres rendant une approximation parfaite impossible.

Amélioration possible (Cf Améliorations)

5 Améliorations

5.1 Amélioration cosinus et sinus

Voilà une liste non exhaustive des améliorations pouvant être mises en place afin d'augmenter les performances de nos algorithmes.

- Rester sur l'algorithme CORDIC et l'améliorer :
 - Augmenter le nombre d'itération pendant l'algorithme pour augmenter la précision (attention cela entraîne une baisse des performances de l'algorithme)
- Modifier intrinsèquement notre algorithme
 - Nous pouvons imaginer l'utilisation du développement en série entière de sinus couplé avec une réduction de Cody and Waite sur $[-\pi/8; \pi/8]$. Cette méthode pourrait permettre la réduction à quelques d'écart au maximum. Pour des petits angles ($< \pi/8$), la précision serait de 1 ULP dans le pire cas.
- Modifier la manière de stocker nos flottants
 - Nous pourrions imaginer une valeur stockée sur 64 bits permettant un gain de précision non négligeable.

5.2 Amélioration arctan et arccos

- Utilisation du polynôme de l'Hermite :
- Les polynômes de hermite seraient également une bonne solution pour implémenter la méthode arctan tout en gardant une bonne précision. Ils sont définies par la formule de récurrence suivante

$$p_1(x) = 4 - 4x^2 + 5x^4 - 4x^5 + x^6 p_m(x) = (1 - x)^4 x^4 p_m(x) + (-4)^{m-1} p_1(x)$$

On arrive à trouver l'encadrement de la fonction arctan par les polynômes de Hermite, nous permettant d'implémenter une méthode bien plus précise que le développement en série entière. Pour donner un ordre de grandeur le polynôme d'ordre 2 permet une précision de l'ordre $(4^{-\frac{5}{8}})^{16}$. Pour donner une idée, pour atteindre une précision à 3 décimales avec un développement en série entière, il faudrait faire utiliser un polynôme de degré 1000.

- Pour améliorer la précision, on pourrait également imaginer stocker les valeurs sur 64 bits (double en java)

6 Conclusion

Nous avons obtenu une extension trigonométrique optionnelle, qui donne le résultat des résultats avec de très bonnes approximations. Cependant comme nous avons pu le voir, il existe de nombreuses améliorations possibles. Nous sommes conscient que la précision d'une fonction Cordic est inférieure à ce qu'on aurait pu obtenir avec l'utilisation du développement en série entière de sinus ajouté d'une réduction de Cody and Waite sur $[-\pi/8; \pi/8]$. Nous obtenons tout de même une précision correcte, puisque nous avons un écart moyen de quelques bits, même pour les très grandes valeurs.

Pour ce qui est côté utilisateur, la classe math est facilement utilisable, et l'utilisateur pourra très facilement faire appel aux fonctions trigonométriques de base (et à la fonction ULP). Implémenter cette classe nous a permis de mieux comprendre comment les flottants étaient gérés par un ordinateur.

Nous conclurons sur le fait que cette extension était riche en apprentissage, et que nous n'hésiterions pas à la reprendre si c'était à refaire.