

Extension Trigo:

Responsable de la partie :

Youssef Benhachem

Tables des matières :

Introduction.....	2
Implémentation des différentes fonctions.....	2
Précision de l'algorithme CORDIC.....	2
Les modes de l'algorithme CORDIC.....	4
Implémentation de l'arcsinus.....	4
Implémentation de la fonction ulp.....	5
Tableau de performances du CORDIC.....	5
Structure double.....	6
Structure float.....	6
Comparaison avec les fonctions de la bibliothèque Math en Java	6
Précision détaillée des différentes fonctions.....	7
Courbe des fonctions.....	9
Evaluation des fonctions auxiliares.....	12
Pistes d'amélioration de la précision.....	13
Dépendances entre les fonctions principales et les fonctions auxiliaires.....	14
Les aspects énergétiques de l'extension :.....	15
Les tests en JUnit 5.....	16
Les tests en Deca.....	17
Instructions utilisateur pour le compilateur:.....	18
Ressources bibliographiques.....	20

1-Introduction :

Pendant la phase de l'analyse et conception du sujet, plusieurs algorithmes ont captés notre attention, le premier, connu par tout le monde, l'approximation par les polynômes de Taylor : c'est un algorithme basé sur le calcul des puissances et factoriels dont complexité dépend de la précision attendue. Néanmoins, cet algorithme présente deux défauts majeurs :

1. La convergence est lente si nous nous trouvons loin du point de développement.
2. Le temps de calcul est grand avec le calcul des puissances, des divisions et des factorielles.

Pour ces raisons, nous avons décidé d'opter pour le CORDIC qui en un nombre fixe d'itérations et un temps minime assure une bonne précision avec le moindre de ressources. Inventé par Volder en 1959, cet algorithme a été utilisé dans plusieurs calculatrices (*HP 35*) et dans des coprocesseurs arithmétiques comme *Intel 8087*.

2 -Implémentation des différentes fonctions :

2.1- Présentation de l'algorithme CORDIC :

L'idée de l'algorithme CORDIC est de décomposer un angle dont on cherche le cos et le sin dans ce qu'on appelle une base discrète dont les valeurs sont stockées en mémoires (voir 6.4 pour plus de détail), cette base dans le cas de l'algorithme CORDIC est :

$$\text{atan}(2^{-i}), i = 0, \dots, 45.$$

Après cette décomposition, il devient très clair comment trouver le cos ou bien le sin, prouver ça théoriquement est très facile et se base sur des relations trigonométriques du niveau lycée, l'idée est qu'on peut toujours regrouper la somme des arctangentes en une seule arctangente et que le cos et le sin d'une arctangente sont des fonctions rationnelles de x. Les relations sont ci-dessous :

$$\arctan(a) + \arctan(b) = \arctan\left(\frac{a+b}{1-ab}\right)$$

$$\cos(\arctan(x)) = \sqrt{1-x^2}$$

$$\sin(\arctan(x)) = \frac{x}{\sqrt{1-x^2}}$$

En pratique, l'algorithme est basé sur une suite d'itérations effectuée sur un système à trois inconnus dont les valeurs initiales déterminent la convergence à la fin. Le système est le suivant :

7.2 The Conventional CORDIC Iteration

Volder's algorithm is based upon the following iteration,

$$\begin{cases} x_{n+1} = x_n - d_n y_n 2^{-n} \\ y_{n+1} = y_n + d_n x_n 2^{-n} \\ z_{n+1} = z_n - d_n \arctan 2^{-n}. \end{cases} \quad (7.1)$$

Type	$d_n = \text{sign} z_n$ (Rotation Mode)	$d_n = -\text{sign} y_n$ (Vectoring Mode)
circular	$x_n \rightarrow K (x_0 \cos z_0 - y_0 \sin z_0)$ $y_n \rightarrow K (y_0 \cos z_0 + x_0 \sin z_0)$ $z_n \rightarrow 0$	$x_n \rightarrow K \sqrt{x_0^2 + y_0^2}$ $y_n \rightarrow 0$ $z_n \rightarrow z_0 + \arctan \frac{y_0}{x_0}$

Extrait du livre : livre « Elementary functions Algorithms Implementations » de Monsieur John Michelle Muller (p134 chapitre 7)

2.2- Les modes de l'algorithme CORDIC :

Il s'avère que l'algorithme CORDIC possède deux modes : mode rotation et mode vecteur.

- D'une part, le mode rotation est celui utilisé pour le calcul du cosinus et sinus, en fait, en prenant $x_0 = \frac{1}{K}$, $y_0 = 0$ et $z_0 = \theta$; où θ est l'angle voulu, le couple (x, y) converge vers $(\cos(\theta), \sin(\theta))$.
- D'autre part, le mode vectoriel permet de calculer la fonction arctangente, en prenant par exemple $x_0 = 1$, $y_0 = \text{val}$ et $z_0 = 0$ la variable z convergera vers $\text{atan}(\text{val})$.

Résultats : A l'aide de ce système, on a implémenté les fonctions : cos, sin et atan.

2.3- Implémentation de l'arcsinus:

La fonction arcsinus a été déduit de la fonction arctangente à l'aide de la relation :

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

En revanche, la fonction arcsinus n'était pas complètement gratuite comme il fallait implémenter la fonction racine. Cette dernière a été implémentée selon la fameuse méthode d'Héron qui pour calculer la racine d'un nombre a donné, on itère selon la récurrence linéaire suivante :

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Au début, le choix de x_0 appelé « seed » était arbitraire, ce choix présentait deux défauts majeures :

- L'algorithme prend beaucoup d'itérations pour donner la précision voulue
- La fonction asin prend beaucoup de temps de calcul.

Ensuite, la décision était d'optimiser l'algorithme ou bien le substituer, l'optimisation de l'algorithme se base sur l'idée suivante : le plus proche la « seed » est de la racine recherchée, le plus rapide l'algorithme serait. La seed choisie est :

$$x_0 = 10 - \frac{190}{a + 20}$$

Ce seul changement a radicalement changé la rapidité de l'algorithme : dans uniquement « trois » itérations on obtient la précision voulue.

1.4- Implémentation de la fonction ulp:

La fonction ulp est implémentée selon la définition de la fonction `ulp(float)` de Java , tout en respectant la norme IEE 754 pour les flottants : une mantisse de 24 bits et un exposant de 8 bits.

L'implémentation de cette fonction nécessitait l'implémentation de la fonction `_getExponent(float f)` qui retourne l'exposant en base binaire d'un flottant donné, et aussi la fonction `_puissanceDeDeux(int i)` qui a été partagé par toutes les fonctions déjà implémentées.

Notre fonction ulp prends compte de tous les cas : Infinie, zéro ou subnormal et le cas normal.

3-Tableau de performances du CORDIC :

Les performances des fonctions implémentées sont mesurées en deux temps , d'abord en utilisant la structure *double* en *Java*, ensuite en utilisant la structure *float*.

Le temps de calcul est mesuré à l'aide de la méthode `Sytem.nanoTime()`. Afin d'être le plus précis possible, on mesure pour un grand nombre d'échantillons (100 000) de 100 entrées le temps nécessaire pour le calcul de l'une des fonctions, et on prend après l'espérance (la moyenne arithmétique), ceci donne le temps pour 100 entrées, ce temps est ensuite divisé par 100 pour obtenir le temps pour une seule entrée.

La précision est assurée à l'aide de la fonction `assertEquals` de la classe *Assertions* .

3.1- Structure double :

Le tableau suivant est pour l'implémentation avec la structure *double*.

Fonction	Temps pour une entrée (en ms)	Temps pour 100 entrées (en ms)	Précision
cos	0,00062 ms	0,062 ms	1E-13
sin	0,00063 ms	0,063 ms	1E-13
atan	0,00068 ms	0,068 ms	1E-13
asin	0,00079 ms	0,079 ms	1E-13 *
ulp	0,00002 ms	0,002ms	1E-5

* La précision E-13 de l'*asin* est pour x tel que $|x| < 0.975$, pour x tel que $|x| > 0.975$ la précision diminue tant que $|x| \rightarrow 1$

3.2- Structure float :

Le tableau qui suit est selon l'implémentation avec la structure *float*.

Fonction	Temps pour une entrée (en ms)	Temps pour 100 entrées (en ms)	Meilleur précision
cos	0,00081 ms	0,081 ms	1E-6
sin	0,00084 ms	0,084 ms	1E-6
atan	0,00085 ms	0,085 ms	1E-7
asin	0,00087 ms	0,087 ms	1E-6
ulp	0,00002 ms	0,002 ms	1E-8

* La précision E-6 de l'*asin* est pour x tel que $|x| < 0.917$, pour x tel que $|x| > 0.917$ la précision diminue en s'approchant de 1. (Plus de détail dans le tableau ci-dessous)

*ulp nous garantit une précision de E-6 dans tous les cas, et une précision de E-8 dès que $|x| > 1$

3.3-Comparaison avec les fonctions de la bibliothèque Math en Java:

La comparaison est faite avec le type double, parce que la bibliothèque math de Java n'implémente pas des fonctions cos et sin pour le type float.

On note aussi que les temps fournis dans le tableau ci-dessous sont en microsecondes.

Fonction	CORDIC en double	Java.lang.Math en double
cos	0,62 μ s	0,043 μ s
sin	0,63 μ s	0,042 μ s
atan	0,68 μ s	0,072 μ s
asin	0,79 μ s	0,330 μ s
ulp	0,02 μ s	0,030 μ s

Commentaire :

- Il s'avère que notre fonction ulp économise 33,33% pourcent du temps que la fonction ulp de java.lang.Math, avec une différence de 1 microseconde pour chaque entrée.
- Notre fonction asin vient avec une différence de 4 microsecondes avec la fonction asin fournie en java.
- Les fonctions cos, sin et atan viennent avec une différence de 5,8 microsecondes avec les fonctions de la bibliothèque java.lang.Math

Transition :

Dans la suite on détaillera la précision des différentes fonctions que nous avons implémentées selon le choix de la valeur de départ, ensuite on verra les courbes des fonctions pour avoir une vue globale de la précision.

4-Précision détaillée des différentes fonctions :

- **cos et sin :**

La précision de ces deux fonctions trigonométriques dépend de l'intervalle de départ :

- : La précision est de 1E-6
- $\mathbb{R} \setminus [-9\pi, 9\pi]$: La précision oscille entre 1E-6 ET 1 E-5

Explication et idées pour amélioration :

Les fonctions cos et sin sont 2π -périodiques, et comme l'implémentation repose sur les égalités trigonométriques entre le cos et le sin pour prolonger les deux fonctions sur tout l'axe réel, l'imprécision vient de la fonction « *_anglePrincipal* » .

Comme piste d'amélioration, on pourra au lieu de faire une suite de soustraction dans une boucle while, d'abord localiser l'angle d'entrée dans un intervalle de la forme $[k\pi, (k+1)\pi]$, et économiser une suite de soustractions à une multiplication et une addition en plus de comparaisons (ces dernières ne représentent pas une source d'imprécision).

⇒ Après implémentation de l'idée ci-dessus, malheureusement ça n'a pas été suffisant pour augmenter la précision en dehors de $[-9\pi, 9\pi]$.

- **atan :**

- $[-0.274, 0.274]$: La précision est de 1E-7
- $\mathbb{R} \setminus [-0.274, 0.274]$: La précision est 1E-6

- **asin :**

La précision l'*asin* pour x tel que $|x| < 0.917$ est 1E-6, ensuite pour x tel que $|x| > 0.917$ la précision diminue en s'approchant de 1. (Plus de détail dans le tableau ci-dessous). On se contente des intervalles positifs ; la précision est pareille pour $[-1, 0]$.

Intervalle	[0,0.915]	[0.915,0.943]	[0.943,0.965]	[0.965,0.983]	[0.983,0.995]	[0.995,1]
Précision	1E-6	1E-5	1E-4	1E-3	1E-2	1E-1

- **ulp :**

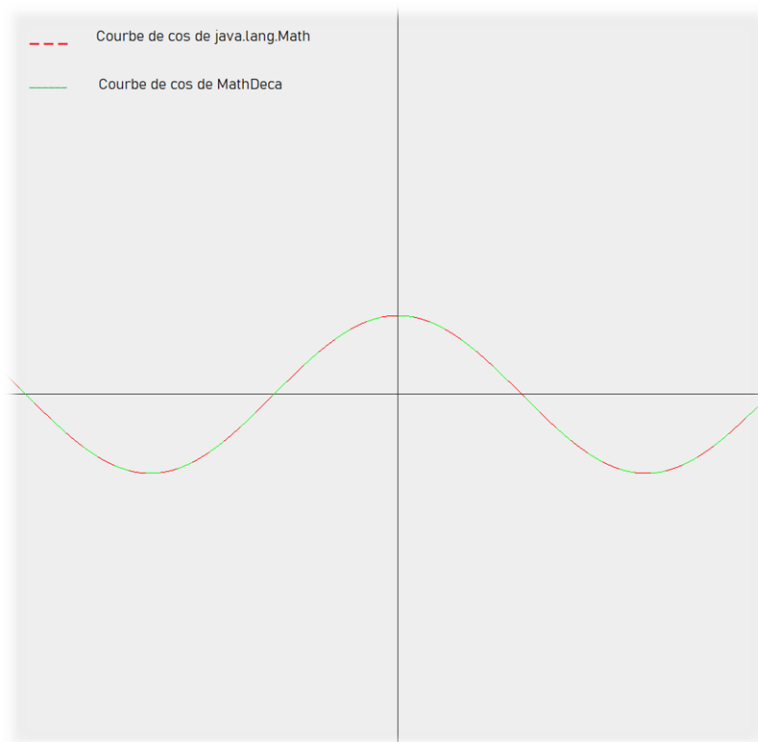
ulp nous garantit une précision de E-8 dès que $|x| \geq 1$, et une précision qui varie entre E-6 et E-7 dans $] -1, 1[$

Intervalle	$] -\infty, -1]$	$] -1, -0.25]$	$[-0.25, 0.25]$	$[0.25, 1[$	$[1, +\infty [$
Précision	1E-8	1E-7	1E-6	1E-7	1E-8

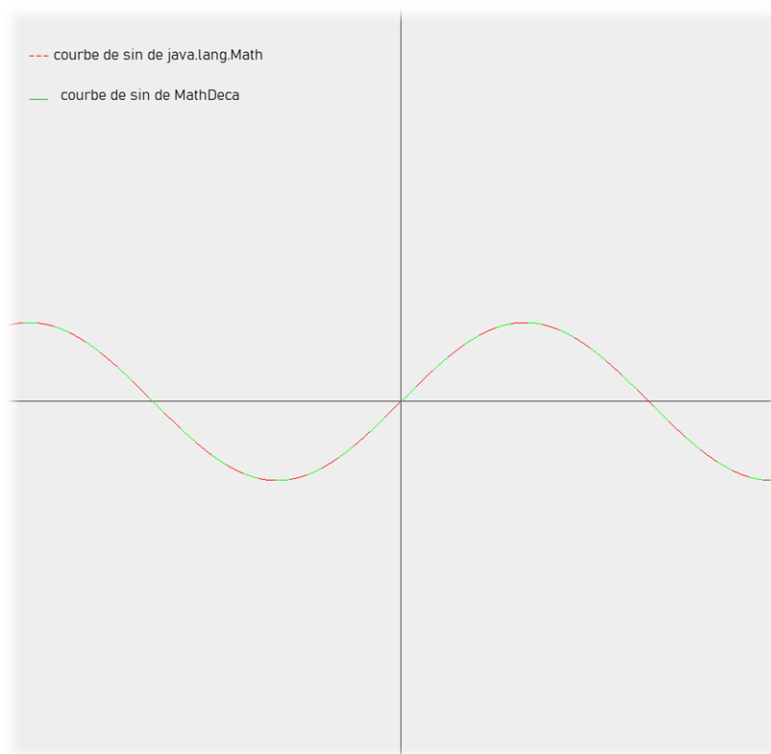
5-Courbe des fonctions:

Les courbes ont été tracé à l'aide des classes de java.awt (abstract window toolkit). On a dû faire notre propre traceur de fonctions, l'unité de ce dernier était de 100 pixels (c'est-à-dire chaque 1 pixel = 0.01). Le traçage est fait sur l'intervalle $[-10,10]$ = 1000 pixels pour les deux axes. Les courbes en - - - rouge tiret sont les fonctions de la bibliothèque Math de Java, et en — vert continu les fonctions de notre bibliothèque.

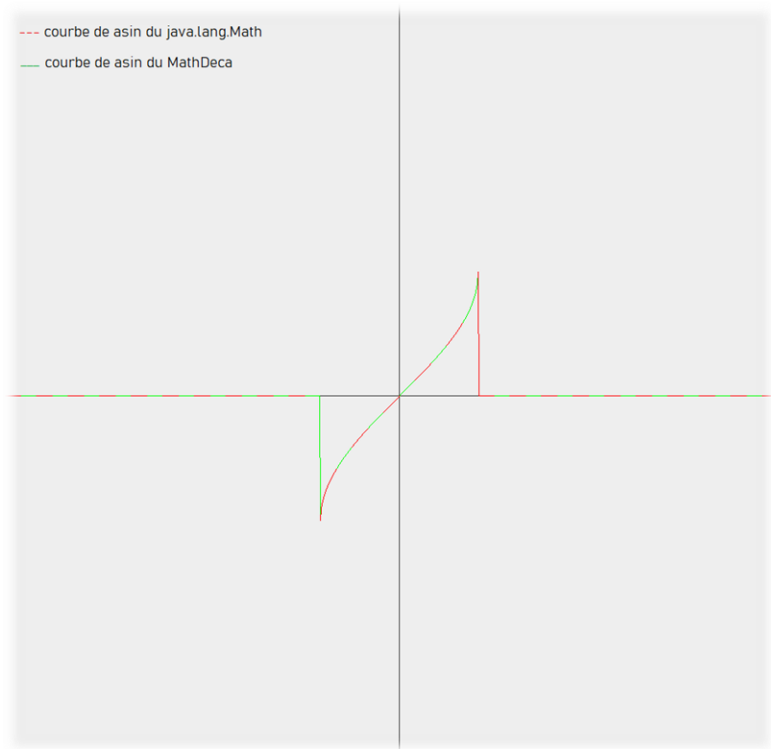
- Fonction cosinus



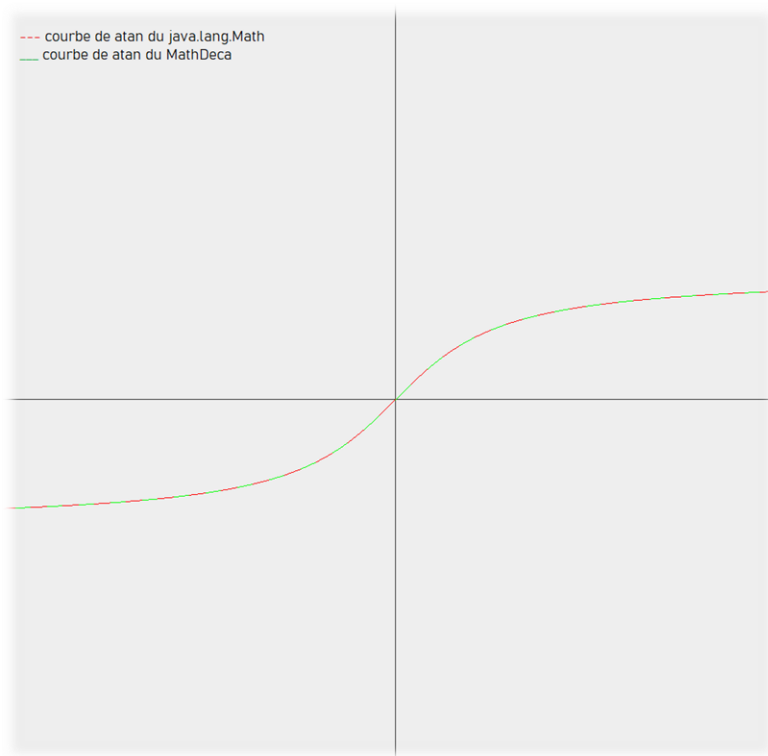
- Fonction sinus



- Fonction arcsinus



- Fonction arctan :



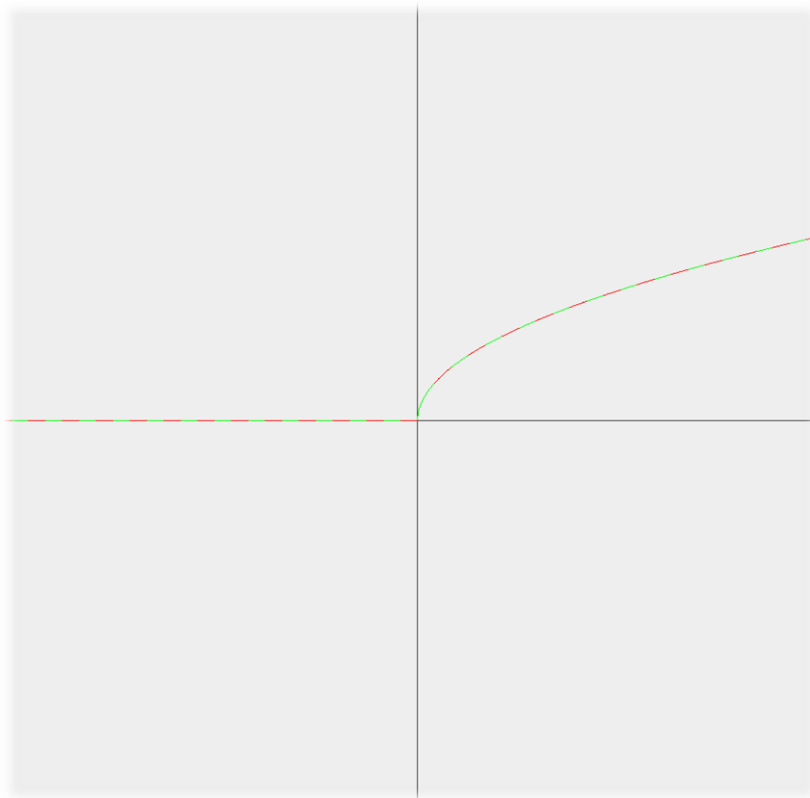
Commentaire :

Les fonctions coïncident parfaitement, et on ne peut distinguer avec l'œil nue la différence entre les deux courbes, c'est d'ailleurs la raison pour laquelle on a décidé de tracer les courbes de la bibliothèque Math de Java en tiret.

6- Evaluation des fonctions auxiliaires:

L'évaluation des fonctions auxiliaires reste indispensable comme il nous permet d'étudier les sources d'imprécisions pour les fonctions principales, et aussi les sources de retard, en terme d'étude temporelle.

6.1- Fonction racine:



La fonction racine se superpose avec la fonction racine de la bibliothèque `java.lang.Math`, après exécution des tests la précision de notre fonction implémenté en float oscille entre E-6 et E-7.

6.2- Fonction puissanceDeDeux :

La fonction `puissanceDeDeux` était très simple à implémenter, et passe tous nos tests de précision, on a été même à vérifier pour une précision de E-200. Ceci nous assure, qu'elle n'est pas une source d'imprécision pour les fonctions principales.

6.3- Fonction getExponent :

Nous avons testé la fonction getExponent en comparant ses valeurs avec la fonction getExponent de la bibliothèque Math de Java. Tous les tests sont passés et aucune erreur n'a été détectée .

6.4- Fonction getAtan :

La fonction getAtan est la fonction qui gère le stockage de la valeur de la base discrète de décomposition : **atan(2^{-i})** , **i = 0,...,45**, c'est une fonction dont l'entrée est un entier, et distingue 46 cas pour retourner la valeur voulue.

7- Pistes d'amélioration de la précision:

- Implémentation de l'instruction **asm** :

Comme en CORDIC, chaque itération nécessite 3 additions-multiplications, en totale 115 additions-multiplications, l'implémentation de l'instruction asm aurait augmenter la précision des résultats du CORDIC.

- Optimiser davantage la fonction **racine** :

Bien qu'elle soit rapide, la fonction racine présente une source d'imprécision et affecte le calcul de la fonction asin, une implémentation plus précise de la racine affectera d'une manière positive la précision de l'arcsinus.

8- Dépendances entre les fonctions principales et les fonctions auxiliaires :

Dans cette partie, on détaille dans le tableau ci-dessous la dépendance entre les cinq fonctions exigées et les fonctions auxiliaires sur lesquelles elles dépendent pour fournir leur résultat.

<u>Fonction Principale</u>	<u>Fonction auxiliaire</u>
Cos	_puissanceDeDeux _getAtan
Sin	_puissanceDeDeux _getAtan
Atan	_puissanceDeDeux _getAtan
Asin	_puissanceDeDeux _getAtan _racine
Ulp	_puissanceDeDeux _getExponent _abs

Remarques :

- Ces tests nous permettent de bien cerner les sources d'imprécisions des fonctions principales et ainsi avoir une idée claire sur l'amélioration de leur précision.
- Certes il y a des dépendances entre les fonctions principales, comme le cos et le sin, l'atan et l'asin, mais ici ce qui importe c'est les fonctions auxiliaires.

9- Les aspects énergétiques de l'extension :

Dans le tableau ci-dessous on détaille les aspects énergétiques de l'extension, on citera le nombre d'opération et de comparaisons faites par chaque fonctions.

Fonction	Nombre d'additions et soustractions	Nombre de multiplication	Nombre de division	Nombre de comparaisons
Cos	181	271	1	93
Sin	181	271	1	93
Atan	181	270	1 ou 0	92
Racine	5	4	4	5
Asin	186	274	4 ou 3	96

Les résultats ci-dessus sont exploités dans le document « Analyse des aspects énergétiques » qui détaille la consommation énergétique de notre compilateur.

9- Les tests en JUnit 5 :

9.1 Modalités de testage:

Les tests ont été effectués en JUnit5 en suivant les modalités **BDD** (Behaviour Driven Development), comme suit :

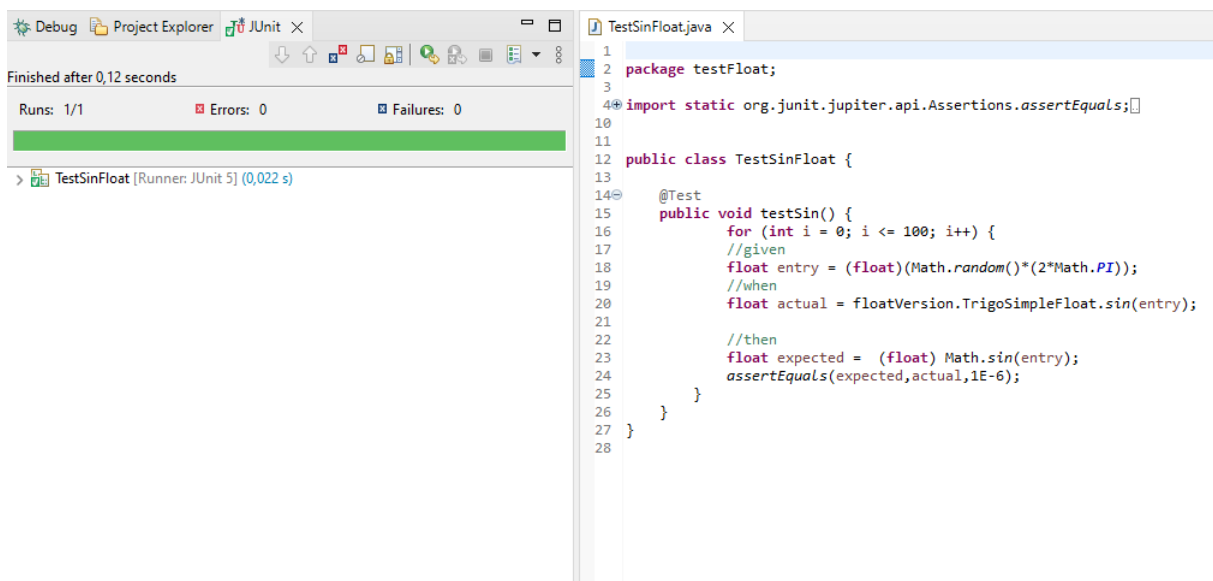
//given: dans cette section on déclare la variable d'entrée de la fonction à tester, nommée **entry**

//when: ici on calcule la valeur de sortie de la fonction qu'on teste, qu'on stocke dans une variable nommée conventionnellement **actual**

//then: dans cette étape, on déclare la variable **expected** qui représente la sortie attendue de la fonction testée.

⇒ Cette méthode de testing rend cette phase très claire, lisible et facile à réutiliser.

9.2- Exemple du test pour la fonction sinus



```
1 package testFloat;
2
3
4 import static org.junit.jupiter.api.Assertions.assertEquals;
5
6
7
8
9
10
11
12 public class TestSinFloat {
13
14     @Test
15     public void testSin() {
16         for (int i = 0; i <= 100; i++) {
17             //given
18             float entry = (float)(Math.random()*(2*Math.PI));
19             //when
20             float actual = floatVersion.TrigoSimpleFloat.sin(entry);
21
22             //then
23             float expected = (float) Math.sin(entry);
24             assertEquals(expected,actual,1E-6);
25         }
26     }
27 }
28
```


10 – Tests en Deca :

En Deca , on a réalisé deux genres de tests, les tests sans objets et les tests objets, on a commencé par les fichiers sans objets comme la partie sans objet était prête avant la partie objet.

Les fichiers tests sans objets contenaient beaucoup de code, et donc ont mérité une attention particulière pour ne rien manquer.

Pour chaque fonction exigée (cos, sin , atan, asin et ulp), on a fait un fichier tests, en fait ces fichiers tests peuvent être vus comme une version sans objet pour l'extension, puisqu'ils sont capables de générer les valeurs des fonctions demandées dans des intervalles précises.

Le tableau ci-dessous détaille les intervalles à lesquelles on doit se limiter pour les tests sans objet :

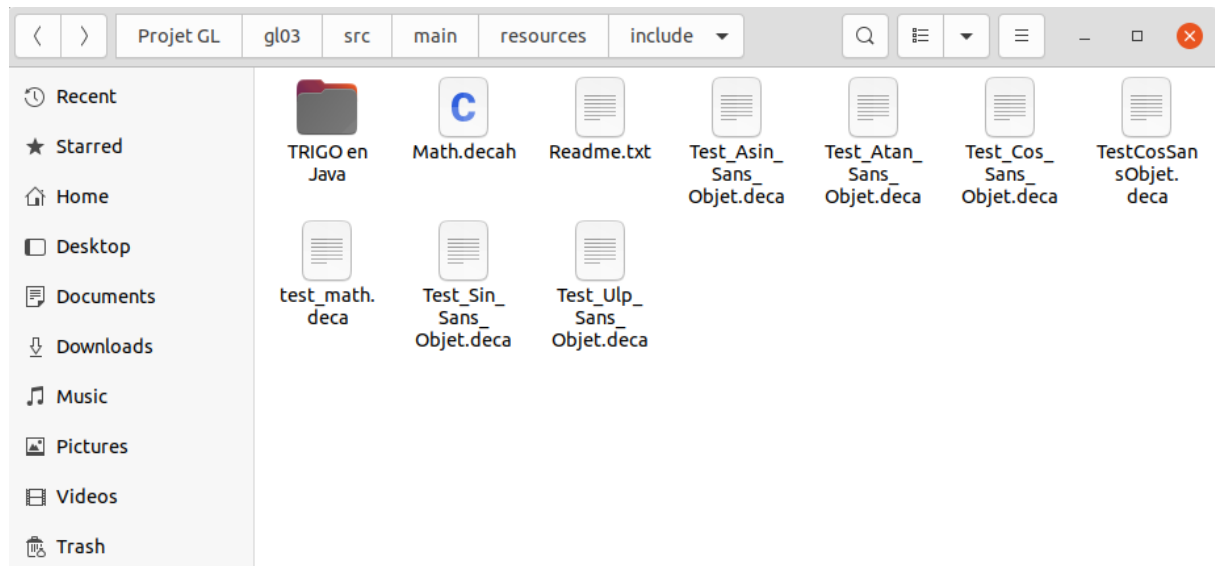
<u>Fonction</u>	<u>Intervalle de départ</u>
Cos	$[-\pi/2, \pi/2]$
Sin	$[-\pi/2, \pi/2]$
Atan	$[-1, 1]$
Asin	$[-0.7071, 0.7071]$
ulp	R

Remarque :

Certes les tests pourraient bien être prolongé à tout l'intervalle de définition de la fonction, mais comme ces fichiers sont considérés des fichiers tests et non pas de fichiers sources, on s'est limité aux intervalles fournies par le CORDIC sans utilisation des égalités trigonométriques pour la prolongation.

11- Instructions utilisateur pour le compilateur:

La partie extension du compilateur se situe sur le répertoire *src/main/resources/include*



Notre compilateur manque de la perfection au niveau objet, et comme le fichier *math.decah* est implémenté en objet, on a décidé de le faire des tests en mode sans objet, comme illustré par l'image ci-dessus, il y a cinq fichiers dont le nom suit la forme suivante : *Test_Fonction_Sans_Objeto.deca*, où *fonction* représente la fonction voulue. Le fichier *readme.txt* détaille les entrées des fonctions des fichiers tests sans objet, et comment les utiliser pour changer la valeur d'entrée, l'utilisateur a donc le choix d'utiliser la version sans objet de l'extension.

Le dossier « *TRIGO en Java* » représente une version Java de la partie extension, ce dossier est en fait un projet Maven qui pourra être exécuter indépendamment de tous les autres fichiers. C'est de ce fichier qu'on a tracé les courbes et mesuré la précision des fonctions.

Remarque :

En Java, on codait en float parce que double n'est pas valable en Deca, et comme ça conserve la précision.

D'une part, ce dossier contient deux fichiers sources Java : **ForMath.java** et **MathDeca.java** ; comme son nom l'indique ForMath.java contient les fonctions pour (For) MathDeca.java qui lui contient les fonctions trigonométriques avec la fonction ulp.

D'autre part, dans le répertoire des tests, on a à l'aide de **Junit5** réalisé des tests pour chaque fonction de **ForMath.java** et **MathDeca.java**

Pour compiler et exécuter tous les tests, il suffit d'avoir Maven installé et exécuter la commande :

mvn test dans le terminal dans dossier ***TRIGO en Java*** .

12- Ressources bibliographiques:

- « Elementary functions Algorithms Implemantions» de Monsieur John Michelle Muller (Directeur de recherche à l'ENS de Lyon)
- fr.wikipedia.org/wiki/Méthode_de_Héron
- La bibliothèque `java.lang.Math`