Documentation Extension TRIGO

gl25

Janvier 2024

Contents

1 Introduction

Cette documentation fait référence au projet Génie Logiciel de l'ENSIMAG. Dans ce cadre nous avons réalisé un compilateur pour le language *decah*. Cette extension à pour but d'implémenter les fonctions trigonométriques dans une bibliothèque en *decah*.

Cette documentation abordera les aspects suivant :

- Nos choix de conception, d'architecture, et d'algorithmes.
- Nos méthodes de validation.
- Nos résultats de la validation de l'extension.
- Une analyse bibliographique.
- D'autres aspects techniques de l'extension TRIGO.

Plusieurs fois dans l'implémentation de cette extension, nous avons du choisir entre performance et précision, et en vertu du sujet du projet, nous avons opté pour des résultats en faveur de la précision.

2 Brève description des fonction implémentées dans la bibliothèque Math

2.1 sin(float f)

Cette fonction est définie sur R.
Elle calcule une approximation de la valeur du sinus d'un flottant.

2.2 cos(float f)

Cette fonction est définie sur R.
Elle calcule une approximation de la valeur du cosinus d'un flottant.

2.3 asin(float f)

Cette fonction est définie sur]-1, 1[.
Elle calcule une approximation de la valeur de l'arc sinus d'un flottant.

2.4 atan(float f)

Cette fonction est définie sur]-1, 1[.
Elle calcule une approximation de la valeur de l'arc tangente d'un flottant.

2.5 ulp(float f)

Cette fonction est défnie sur R.
Elle calcule la valeur de l'ulp d'un flottant.

2.6 power(float indice, int exposant)

Cette fonction est définie sur R * Z.
Elle calcule la valeur d'un indice puissance son exposant.

2.7 factorial(int facteur)

Cette fonction est définie sur N.
Elle calcule la valeur de la factorielle d'un facteur.

2.8 absVal(float f)

Cette fonction est définie sur R.
Elle calcule la valeur absolue d'un flottant.

2.9 getExposant(float f)

Cette fonction est définie sur R.
Elle calcule la valeur de l'exposant dans la représentation IEEE754 d'un flottant par un parcour dichotomique.

2.10 falseSinus(float f)

Cette fonction est définie sur [-1, 1].
Elle calcule une approximation de la valeur du sinus d'un flottant.

2.11 falseCosinus(float f)

Cette fonction est définie sur [-1, 1].
Elle calcule une approximation de la valeur du cosinus d'un flottant.

2.12 pi()

Cette fonction renvoi l'approximation sur 32bits de pi (soit 3,1415927).

2.13 modulo(float value, float range)

Cette fonction est définie sur R * R.
Elle renvoie le modulo de value par rapport à range.

3 Fonction ULP

3.1 Calculs flottants

En decah, les flottants sont codés sur 32 bits avec la norme **IEEE 754**, donc on a

- 1 bit pour le signe **s** (valeur dans $0, 1$).
- 8 bits pour l'exposant **e** (valeur dans $[0, 255]$).
- 23 bits pour la mantisse **m** (valeur dans $[0, 8388608]$).

Puis on applique la formule suivante pour calculer le flottant **n** :

$$n = (-1)^s \times 2^{e-127} \times (1 + m \times 2^{-23})$$

3.2 Calcul de la formule de l'ULP

Soit un flottant x dont l'on cherche à determiner l'ULP.
On a x_0, x_1 flottants que l'on peut écrire sous forme binaire suivant la norme IEEE 754 tel que :

- x_0 est le flottant le plus proche de x.
- x_1 est le flottant supérieur le plus proche de x_0.

Pour montrer la formule du calcul de l'ULP nous faisons **une disjonction de cas**, selon la valeur de la **mantisse**.

Tout d'abord, **si la mantisse de x_0 est inférieure à 8 388 608** (qui est la valeur max de la mantisse) : Alors on a,
x_0 a 1 bit de signe s, une valeur d'exposant e, et une valeur de mantisse m.
Et x_1 a comme bit de signe s, comme valeur d'exposant e, et comme valeur de mantisse $n = m + 1$.
Et on a,

$$ulp(x) = |x_1 - x_0|$$

Donc,

$$ulp(x) = |(-1)^s \times 2^{e-127} \times (1 + n \times 2^{-23}) - (-1)^s \times 2^{e-127} \times (1 + m \times 2^{-23})|$$

Donc,

$$ulp(x) = |(-1)^s \times 2^{e-127} \times (1+(m+1) \times 2^{-23}) - (-1)^s \times 2^{e-127} \times (1+m \times 2^{-23})|$$

Donc,

$$ulp(x) = 2^{e-127} \times (1 \times 2^{-23})$$

Donc,

$$ulp(x) = 2^{e-150}$$

Maintenant **si la mantisse de x_0 est égale à 8 388 608** (qui est la valeur max de la mantisse) : Alors on a,
x_0 a 1 bit de signe s, une valeur d'exposant e, et une valeur de mantisse $m = 2^{23} - 1$.
Et x_1 a comme bit de signe s, comme valeur d'exposant f = e + 1, et comme valeur de mantisse 0.
Et on a,

$$ulp(x) = |x_1 - x_0|$$

Donc,

$$ulp(x) = |(-1)^s \times 2^{f-127} \times (1 + m \times 2^{-23}) - (-1)^s \times 2^{e-127} \times (1 + 0 \times 2^{-23})|$$

Donc,

$$ulp(x) = |(-1)^s \times 2^{e-126} \times (1+0 \times 2^{-23}) - (-1)^s \times 2^{e-127} \times (1+(2^{23}-1) \times 2^{-23})|$$

Donc,

$$ulp(x) = |(-1)^s \times 2^{e-126} - (-1)^s \times 2^{e-127} \times (2 - 1 \times 2^{-23})|$$

Donc,

$$ulp(x) = 2^{e-126} - 2^{e-126} + 2^{e-127} \times 2^{-23}$$

Donc,

$$ulp(x) = 2^{e-127} \times 2^{-23}$$

Donc,

$$ulp(x) = 2^{e-150}$$

Ainsi on trouve que dans tous les cas, on a :

$$ulp(x) = 2^{e-150}$$

3.3 Encadrement du flottant

Nous savons d'après la partie précédente que l'ulp d'un flottant x **dépend uniquement de la valeur de l'exposant e** dans son écriture suivant la norme IEEE754.

Montrons maintenant que l'on peut encadrer la valeur d'un flottant x, on a : Soit x un flottant, alors on a x_0 flottants que l'on peut écrire sous forme binaire suivant la norme IEEE 754 le plus proche de x, et on a que x_0 a 1 bit de signe s, une valeur d'exposant e, et une valeur de mantisse m, avec ,

$$x_0 = (-1)^s \times 2^{e-127} \times (1 + m \times 2^{-23})$$

Or comme on a que,

$$0 \leq m < 2^{23}$$

Alors,

$$1 \leq 1 + m \times 2^{-23} < 2$$

Donc,

$$2^{e-127} \leq |x_0| < 2^{e-126}(1)$$

Donc, on en déduit que pour trouver la valeur de l'exposant e, il suffit de **faire un parcourt avec les valeurs de e possible** et trouver l'intervalle (1).

Pour faire ce parcourt nous utiliserons un **algorithme de recherche dichotomique** car la fonction $f : x \longrightarrow 2^{x-127}$ est monotone, et la recherche dichotomique est performante avec ce genre de fonctions.

3.4 Codage de l'ULP

Voici le code en Java de l'implémentation du calcul de l'ulp pour un flottant :

Listing 1: Implémentation en Java de l'ulp

```
1    public float getUlp(){
2        int exposant = this.getExposant();//Il s'agit d'un
             parcour dichotomique des valeurs possible de l'
             exposant afin de trouver l'encadrement du
             flottant, et donc la valeur de l'exposant.
3        float valeurUlp = (float) Math.pow(2, (exposant -
             150));//Mise    la puissance e - 150 de 2.
4        return valeurUlp;
5    }
```

Ainsi

3.5 Test performances fonction ULP

Pour tester nôtre fonction ULP, on compare les ULP calculés avec la fonction *ulp* de la bibliothèque *Math* de Java. On tire les valeurs testés **aléatoirement entre 0 et** 10^9 on test sur 10^6 valeurs et on obtient que **l'on a** 0 **erreurs**.

On conclue donc que nôtre **fonction ULP est bien codée et parfaitement précise**.

4 Fonction Sin/Cos

4.1 Différentes méthodes utilisables

1 - Série de Taylor :

Une première manière de réaliser les calculs d'un sin/cos informatiquement est d'utiliser **le développement en série entières des fonctions.**

En effet le développement en série entière du sinus est :

$$\forall x \in]-\frac{\pi}{2}, \frac{\pi}{2}[, sin(x) = \sum_{n=0}^{+\infty} \frac{(-1)^n \times x^{2n+1}}{(2n+1)!}$$

Ainsi en calculant les termes jusqu'à un certain degrés de précision, on peut approximer la valeur du sinus et du cosinus facilement.

Cependant **cette méthode pose problème** :

- Bien que le degré de précision soit très bon, cette méthode demande beaucoup de calculs.
- Cette méthode demande beaucoup de sous fonctions (la puissance, la factorielle, ...) et chaque terme est très coûteux en puissance de calcul.

2 - Lookup Table :

Une deuxième méthode consiste à **utiliser une Lookup Table**, dans laquelle on stock des valeurs précalculée des fonctions sinus et cosinus. Si la valeur que nous recherchons n'est pas stockée dans la table, nous réalisons une interpolation afin de déterminer le résultat.

Cependant **cette méthode pose problème** :

- La précision de la méthode dépend beaucoup du nombre de valeurs stockées dans la Lookup table.
- Pour améliorer la précision des valeurs calculées par interpolation, il est possible d'utiliser des fonctions (plus complexe qu'une simple régression linéaire) mais dans ce cas cela augmente considérablement la complexité de l'algorithme, qui perd alors de son utilité.

3 - CORDIC algorithme :

Une troisième méthode conciste à **utiliser l'agorithme CORDIC** (COordinate Rotation DIgital Computer) inventé en 1959 par *Jack Volder*.

Cette algorithme réalise **une recherche binaire** en effectuant des **rotations circulaires** sur un vecteur initial dans le sens horaire ou trigonométrique selon l'angle dont on cherche à calculer le sin/cos. Ainsi en recalculant la valeur des **coordonnées** (x, y) à chaque itération du vecteur, ce dernier **approche de plus en plus les valeurs de cos/sin** de notre angle. Cependant, afin de réaliser ces calculs sur les coordonnées, **il faut stoker** (dans une lookup table) **les valeurs** des cosinus et sinus (ou cosinus et tangente, ou puissances de 2) **de certains angles particulier**.

C'est pourquoi lorsque l'on travail avec des **nombres en virgule fixe** cette méthode est efficace (il suffit de stocker environ 30 valeurs (15 sin et 15 cos) pour pouvoir estimer à à deux décimale prêt le sinus et le cosinus de notre angle). Cependant lorsque l'on travail avec **des virgules flottantes**, il y a beaucoup trop de précision nécessaire (pouvant dépasser les 50 décimales). Il est donc **impossible d'utiliser cette méthode dans nôtre cas**.

4 - Autres méthodes :

D'autres méthodes peuvent être utilisés pour calculer informatiquement la fonction sinus/cosinus, comme **l'approximation/ interpolation polynomiale**, telle que la méthode de Padé ou de Chebyshev, mais on se rend très vite compte qu'elle ne sont **pas réalisable** car elle demandent de réaliser des calculs complexes comme des calculs d'intégrales ou de connaître au préalable la fonction approchée.

4.2 Méthode utilisée

On déduit d'après l'étude des différentes méthodes listées précédemment, que **la meilleur manière de calculer informatiquement les fonctions sinus et cosinus** dans nôtre cas est d'utiliser **les séries de Taylor** car nous cherchons à implémenter la méthode avec la meilleur approximation possible.

Ainsi il nous suffit de calculer $\forall x \in]-\frac{\pi}{2}, \frac{\pi}{2}[$ la valeur du développement en série entière du sinus définie par :

$$sin(x) = \sum_{n=0}^{+\infty} \frac{(-1)^n \times x^{2n+1}}{(2n+1)!}$$

Or cette somme est infini, ce qui pose un problème. Il nous faut trouver **un arrêt** dans le calcul de la somme tout en ayant **la précision maximale**.

Montrons **à partir d'un certain rang, il est inutile d'ajouter les prochains termes à la somme pour calculer le sinus** avec les séries de Taylor.

En effet si on pose

- $S_N = \sum_{n=0}^{N} \frac{(-1)^n \times x^{2n+1}}{(2n+1)!}$ la somme partielle de la série entière du sinus.
- $\forall n \in N, a_n = \frac{x^{2n+1}}{(2n+1)!}$ la valeur absolue du terme général de la série entière du sinus.
- $\forall n \in N, U_n = ulp(S_n)$ la suite des ulp de S_n.

Alors, comme $\forall n \in N, U_n$ est un ulp, alors on a que $\forall n \in N, U_n > 0$ et on a que $U_n \to ulp(sin(x)) > 0$. De plus on a que $a_n \to_{n\to 0} 0$ par croissance comparée.
Donc

$$\exists m \in N, \forall n > m, a_n < U_n$$

Or d'après **le critère sur les séries alternées**, on a que,

$$\forall n \in N, a_n > |R_n|$$

avec

$$R_n = \sum_{k=n+1}^{+\infty} a_k$$

Donc on a que

$$\exists m \in N, \forall n > m, U_n > |R_n|$$

Donc on a que si **on trouve le rang m à patir duquel le terme de la somme est inférieur à l'ulp de la somme partielle**, on peut **arreter le calcul de la somme car le reste es inférieur à l'ulp et donc ne changera pas le résultat**.

4.3 Première implémentation poposée

Voici le code en Java de l'implémentation du calcul du sinus pour un flottant :

Listing 2: Implémentation en Java du Sinus

```
1 public float getSinus(){
2         //Fonction qui renvoie le sinus approxime du
              flottant.
3         float somme = 0.0f;//On initialise la somme a 0.
4         float terme = (float) this.x;
5         int compteur = 0; // On d finit un compteur.
6         //On commence avec le terme pour n = 0
7         //dans le developpement en serie entiere du sinus.
```

```
8           Flottant sommeFlottant = new Flottant(somme);//
                typage en Flottant obligatoire pour la methode
                getUlp.
9           while(sommeFlottant.getUlp() <= (float) Math.abs(
                terme)){
10          //Tant que U_n <= |a_n |...
11              somme += terme; //on ajoute le prochain terme de
                     la somme dans le calcul du sinus.
12              compteur += 1; //on incr mente de 1 la valeur
                    du compteur.
13              float terme1 = (float) Math.pow(-1, compteur);
14              float terme2 = (float) Math.pow(this.x, 2 *
                    compteur + 1)/this.factorielle(2 * compteur +
                     1);
15              terme = terme1 * terme2; //On calcul le prochain
                     terme de la somme
16              sommeFlottant.setX(somme); //On met a jour le
                    sinus.
17          }
18          return somme //On renvoi la somme, qui est l'
                aproximation du sinus.
19      }
```

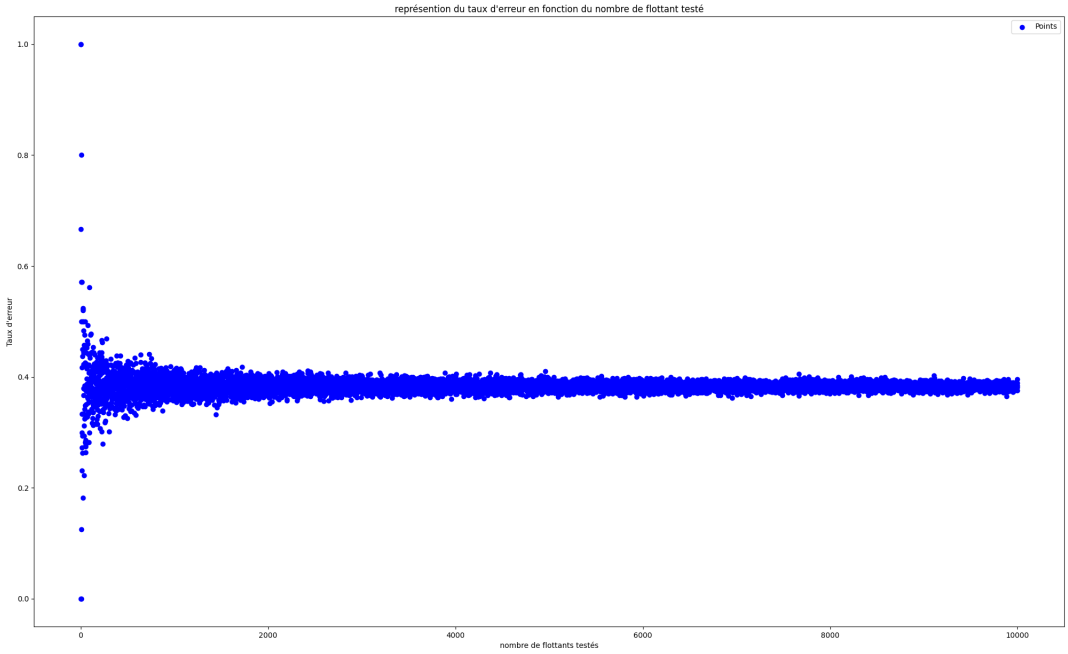
4.4 Liste des fonctions à implémenter en décah

Nous voyons dans notre implémentation du sinus décrite plus haut qu'il nous faut définir **certaines fonctions mathématiques pas déjà implémentées en decah** afin de pouvoir écrire ce code en décah. Ces fonctions sont les suivantes :

- La fonction puissance : $f : (x, n) \to x^n$.
- La fonction factorielle : $f : x \to x! = x \times (x-1) \times ... \times 2 \times 1$.
- La fonction valeur absolue : $f : x \to |x|$.

4.5 Premier problème rencontrée

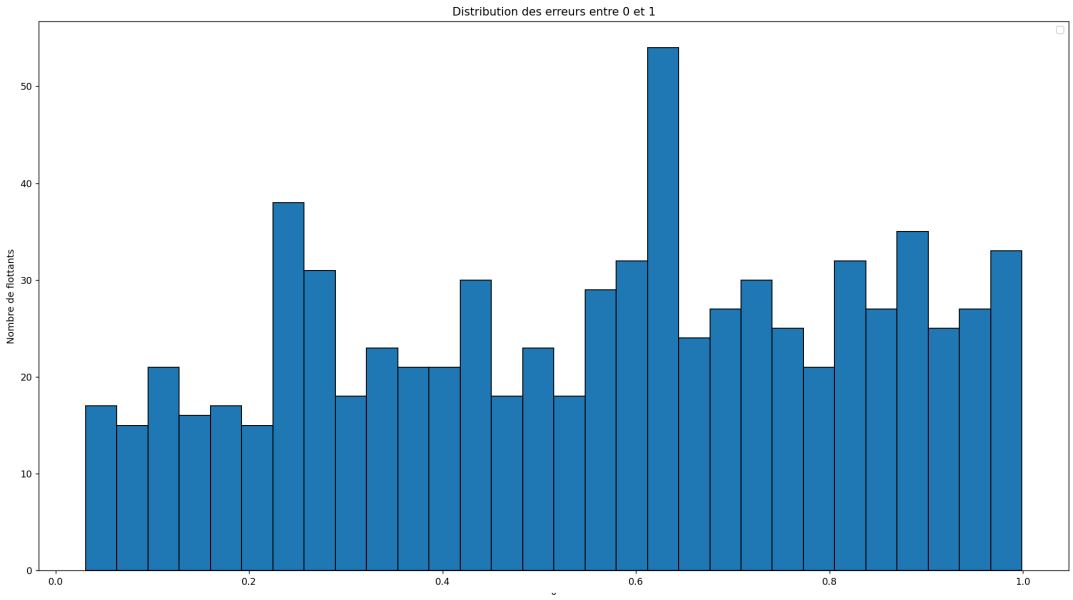
Pour tester la précision de notre algorithme nous comparons nos résultats avec la fonction *sin* du module *Math* en Java. Nous choisissons de tester nos valeurs en prenant n flottants aléatoirement, voici le résultat du taux d'erreur de notre sinus :



On voit bien que on a environ **40 pourcent des valeurs sont sources d'erreurs**.
Le point particulier est que si l'on cherche à quantifier la valeur de l'erreur, on trouve que une grande partie des sinus ont une erreur égale à leur ulp. Si on test la quantité d'erreur parfaitement égale à l'ulp on trouve que : **97 pourcent des erreurs sont des erreurs dans ce cas**.

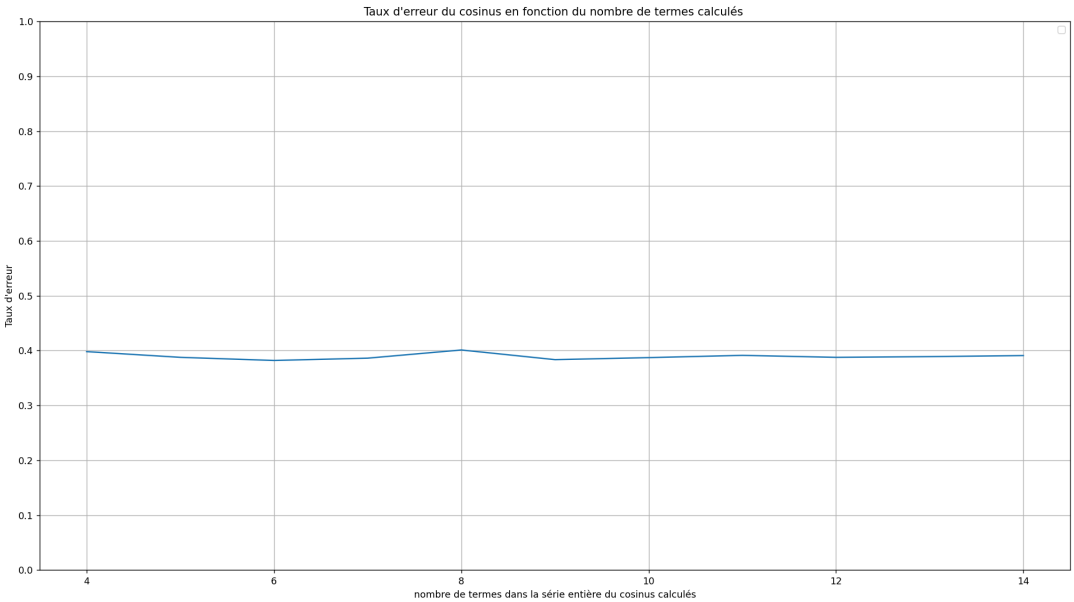
Nous avons aussi codé le Cosinus avec la même méthode que le sinus (on utilise le développement en série entière du cosinus, et on arrête de calculer les prochains termes dans la somme lorsque l'on a que le prochain terme est inférieur à l'ulp de la somme partielle), et l'on **retrouve les mêmes 40 pourcent d'erreur, à 1 ULP près**.

On essai de trouver d'où vient cette erreur. Pour cela on commence par regarder la répartition des valeurs générant une erreur. On obtient l'histogramme suivant :



Ainsi on se rend compte que les erreurs apparaissent uniformément entre 0 et 1. Donc **on ne peut pas déduire que l'erreur est lié à l'intervalle dans lequel est choisit le flottant**.

On s'est ensuite dit que le problème pouvait venir de la condition d'arret. Ainsi pour cela nous avons forcé l'arrêt du calcul des termes dans nôtre somme au n-ième terme, avec n allant de 4 à 14 (4 est le nombre moyen de termes calculés avec la condition d'arrêt précédente). On obtient le graph suivant :



On en déduit donc que le nombre de termes calculés dans la somme n'influe pas sur le taux d'erreur.

Une autre hypothèses est que, **comme nous comparons la valeur calculé de nôtre cosinus avec la valeur du cosinus de la bibliothèque *Math* de Java, qui renvoi un résultat en double, que nous transtypons en float ensuite**, il se peut que le résultat renvoyé avec la précision en double soit différente.

Ainsi pour savoir **si nôtre hyptohèse est bonne**, nous devons calculer nos valeurs sur 64 bits et vérifier si nous avons le même résultat que la bibliothèque *Math* de Java.

Sur 64 bits, les flottants sont codés selon la norme IEEE754 et ont donc :

- 1 bit de signe **s** (allant de 0 à 1).
- 11 bits d'exposants **e** (allant de 0 à 2047)
- 52 bits de mantisse **m** (allant de 0 à 4 503 599 627 370 495)

Puis on applique la formule suivante pour calculer le flottant **n**,

$$n = (-1)^s \times 2^{e-1023} \times (1 + m \times 2^{-52})$$

Ainsi avec un raisonnement similaire à celui employé pour démontrer la formule de l'ULP dans la représentation en 32 bits, on prouve que, pour un flottant x codé en 64 bits,

$$ULP(x) = 2^{e-1075}$$

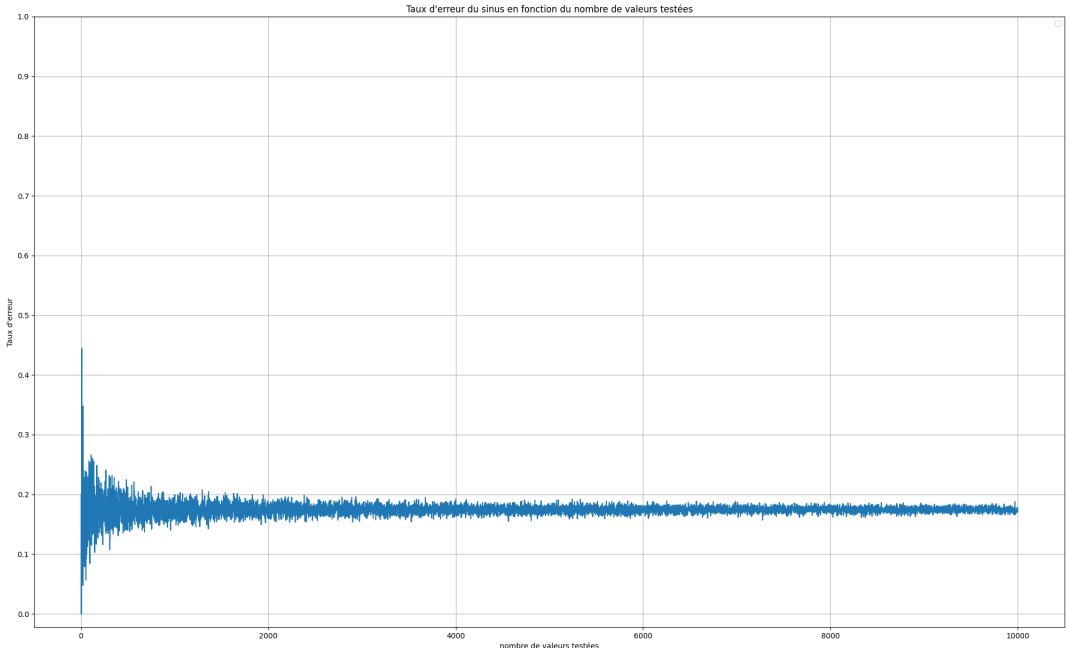
Et de plus avec aussi un raisonnement similaire à celui employé pour démontrer l'encadrement de l'ULP dans le cas de la représentation sur 32 bits, on prouve que :

$$2^{e-1023} \leq |x_0| < 2^{e-1022}$$

Donc on implémente les méthodes précédentes (ULP, cosinus, sinus, ...) en travaillant sur des *doubles* et plus des *floats*, mais au final **On retrouve les mêmes 40 pourcent d'erreurs**.Ainsi l'origine du problème **n'était pas la représentation en float plutôt que en double**.

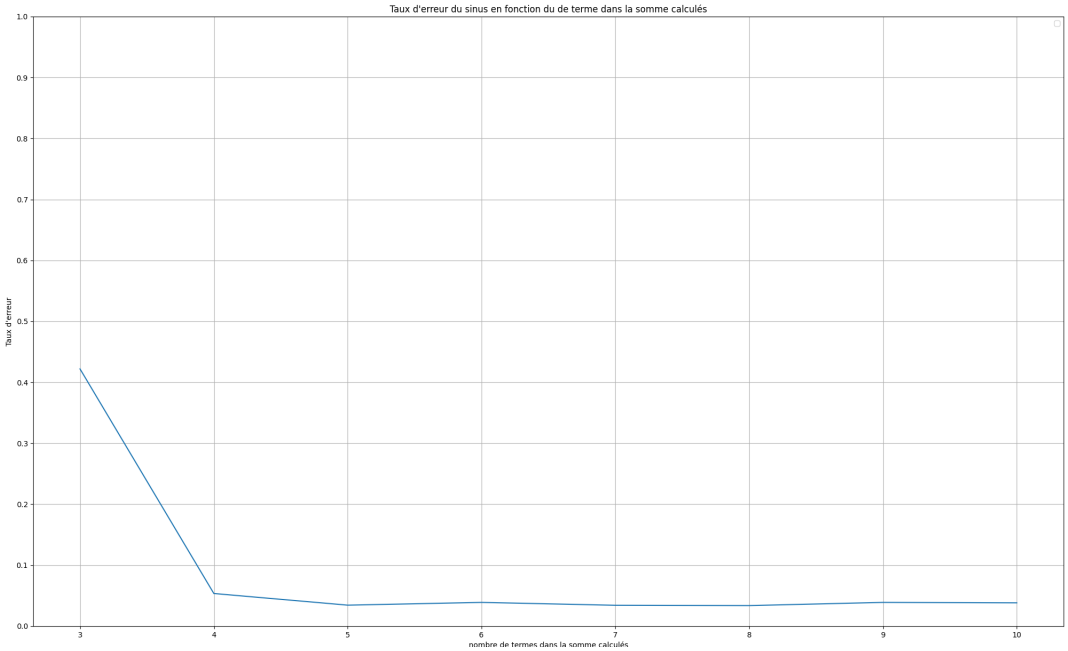
4.6 Solution du premier problème

Finalement, nous nous somme rendu compte que nous sommions les termes dans le mauvais ordre. Nous sommions les termes dans la somme de la série entière du sinus, du plus grand au plus petit. En commençant par sommer les termes du plus petit au plus grand, le résultat est le suivant :



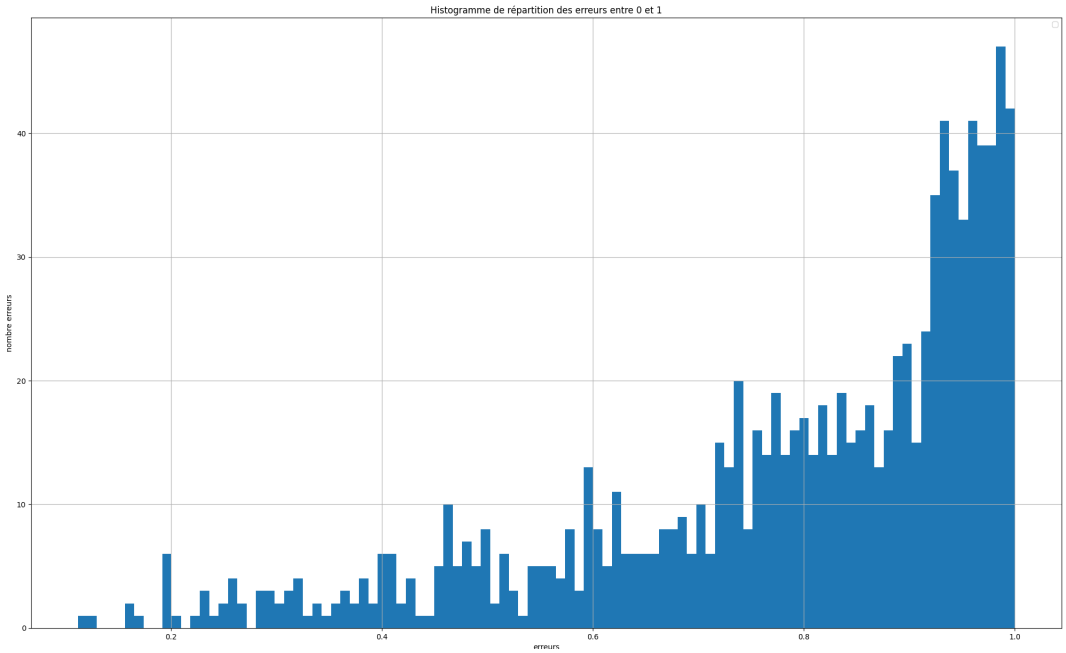
On se rend compte que l'on a réussi à **réduire à 17 pourcent le taux d'erreur en changeant l'ordre des termes**.

On regarde si on peut réduire encore plus le taux d'erreurs en modifiant la condition d'arrêt de l'algorithme. Pour cela on impose le nombre de termes calculés dans la série entière du sinus. On obtient le graphique suivant :



On se rend compte que si on **force l'algorithme à calculer jusqu'au 5ème terme dans la somme** du développement en série entière du sinus, **le taux d'erreur passe à 3,5 pourcent.**

Ainsi on met à jour nôtre algorithme de calcul du sinus en le forçant à calculer jusqu'à 5 termes maintenant. Puis nous cherchons à voir la répartition des valeurs créant des erreurs avec nôtre nouvelle version de notre algorithme. On obtient l'histogramme suivant :

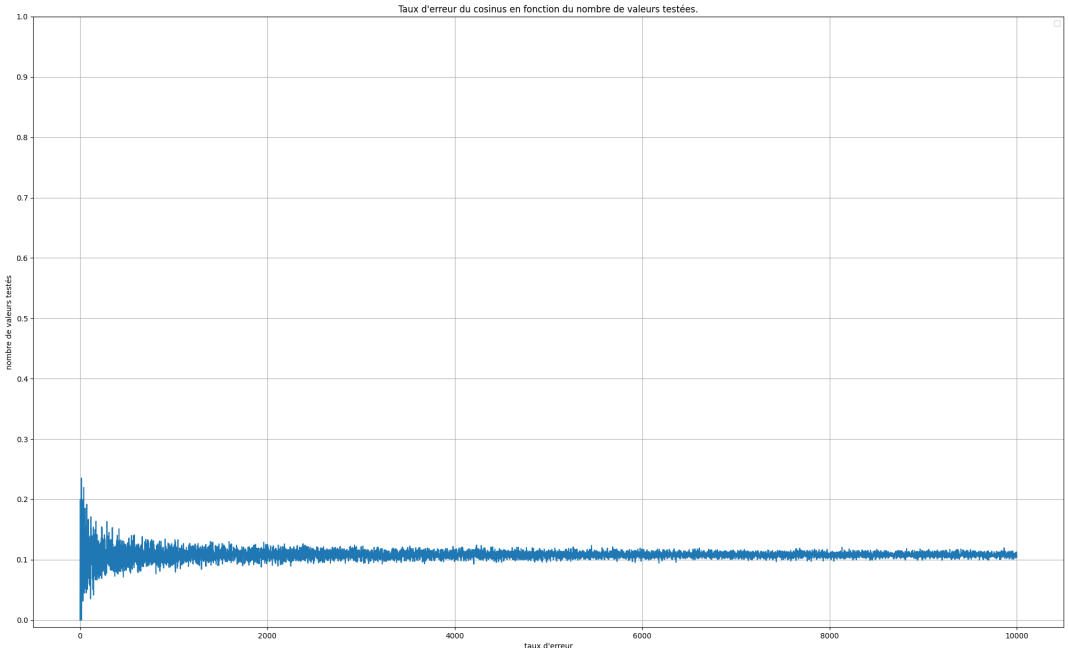


On reconnaît que plus on se rapproche de 1 plus il y a d'erreurs.

Maintenant que nous obtenons un résultat de précision (environ 3.5 pourcent d'erreurs) acceptable pour le sinus, concentrons nous sur la manière dont nous recodons le cosinus.

Nous avons choisi de recopier la nouvelle méthode utilisée pour le sinus (on calcul les 5 premiers termes de la somme du développement en série entière du cosinus, et on somme les termes du plus petit au plus grand ensuite).

Avec cette méthode on obtient le résultat suivant :

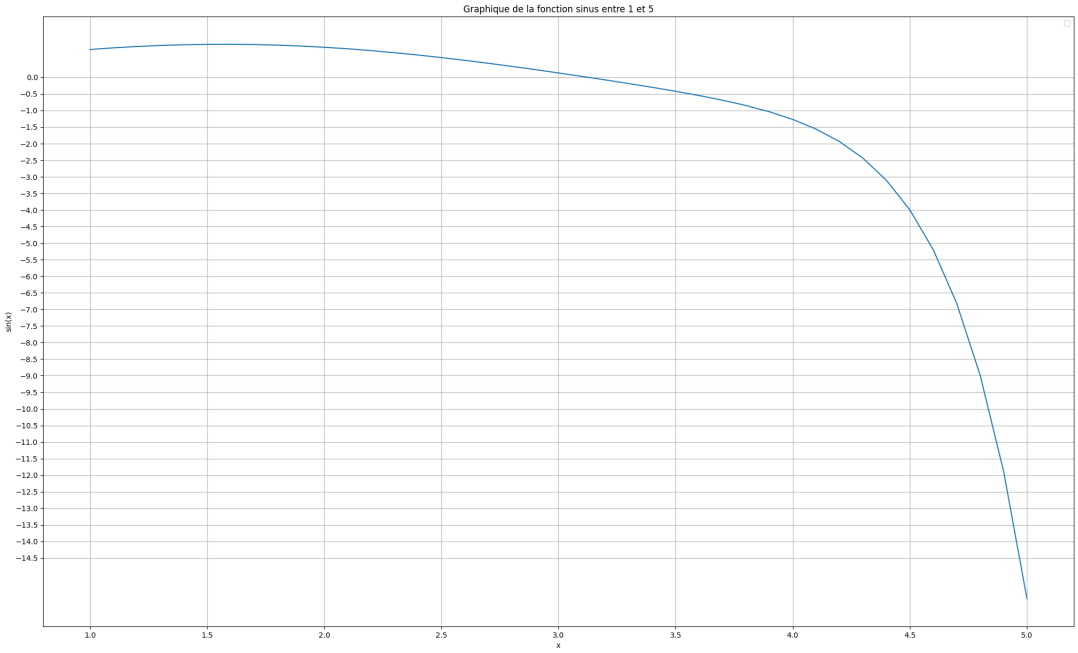


On obtient donc environ 11 pourcent d'erreurs avec cette méthode. Elle est donc bien moins efficace que pour le sinus (qui a 3.5 pourcent d'erreurs). Mais ce résultat reste acceptable.

4.7 Second problème

Un second problème est apparu lorsque nous cherchions à **tester nos fonctions trigonométriques pour d'autres valeurs** que celles comprisent entre $[-1, 1]$.

En effet, si on regarde le taux d'erreur de notre fonction sinus pour des valeurs supérieurs stricts à 1, on obtient le graphique suivant :



On peut observer que pour des valeurs supéieurs à 3, notre fonction n'est même plus comprise entre -1 et 1. Et cela parce que comme notre x augmente, et bien la vitesse de convergence de notre algorithme diminue.

Par exemple si on regarde pour x = 4. Nous avons que le 5ème terme de notre somme vaut :

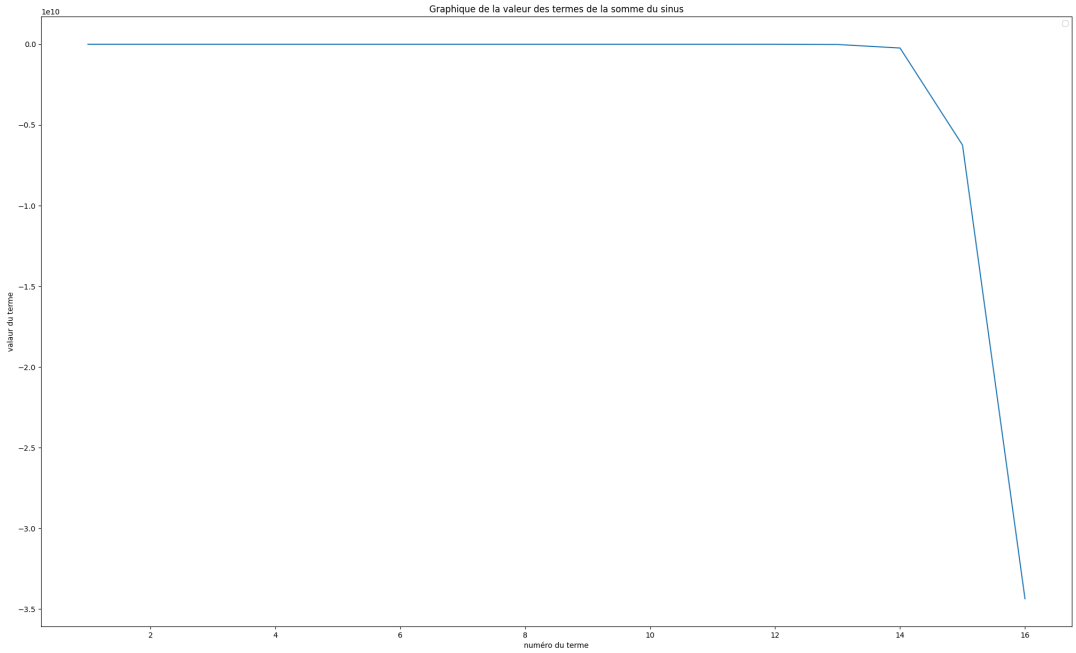
$$terme_5 = -\frac{4^{11}}{11!}$$

donc

$$terme_5 = 0.10507616$$

Ainsi le 5ème terme dans la somme n'est pas inférieur à l'ulp de 4 (*qui vaut* 2^{-21}). On suppose donc que il suffit de calculer plus de termes dans la somme, et nous atteindront au bout d'un moment un terme inférieur à l'ulp. Mais bien que ce raisonnement soit théoriquement correctement, en pratique il nous emmène vers notre **second problème : le dépassement d'entiers**.

En effet si nous continuons de calculer les prochains termes dans la somme de la série entière du sinus évaluée en 4 on obtient le résultat suivant :



On a bien que la valeur des termes est négative, et tend vers $-\infty$. C'est parce que la valeur du dénominateur dans le terme générale de la série entière du sinus **à dépasser la valeur représentable pour un entier sur 32 bits**. En effet les entiers sur 32 bits sont représentés en Java en complément à 2, et peuvent donc couvrir l'ensemble des entiers dans :

$$[-2^{31}, 2^{31}]$$

Or lorsque l'on fait le calcul de

$$factorielle(2n+1) = (2n+1)!$$

avec $n > 6$, alors la valeur dépasse 2^{31} et donc il y a **dépassement d'entier** ce qui vient fausser la représentation de l'entier qui est compris comme négatif.

4.8 Solution second problème

Pour résoudre le problème **du dépassement d'entiers** au dénominateur, nous n'avons pas eu d'autres choix que d'utiliser les formules trigonométriques suivantes afin de réduire la valeur du flottant dont on calcul le sinus **forcément entre** $[-1, 1]$:

- $\forall k \in Z, sin(x + k2\pi) = sin(x)$. Cette formule nous permet de réduire la valeur de x à l'intervalle $[0, 2\pi]$.
- $sin(x - \pi) = -sin(x)$. Cette formule nous permet de réduire la valeur de x à l'intervalle $[0, pi]$.

- $sin(\pi - x) = sin(x)$. Cette formule nous permet de réduire la valeur de x à l'intervalle $[0, \frac{\pi}{2}]$.
- $sin(\frac{\pi}{2} - x) = cos(x)$. Cette formule nous permet de réduire la valeur de x à l'intervalle $[0, 1]$.

Ainsi en appliquant dans les cas appropriés **une certaine combinaison de ces formules** on peut toujours calculé la valeur du sinus d'un flottant à partir d'un flottant se trouvant entre 0 et 1.

4.9 Implémentation finale du sinus / cosinus

Au final le code en *decah* de nos fonctions **sinus et cosinus** sont les suivant :

Listing 3: Implémentation en decah du sinus

```
1     float sin(float f){
2         //Fonction sinus d finie sur R.
3         float newF = 1.0; //On commence par initialiser
              toutes les valeurs.
4         //Puis on applique les formules trigo
              correspondantes selon le domaine dans lequel on
              se trouve.
5         //On verifie si on est entre 0 et 1 :
6         if ((f <= 1.0) && (f >= 0.0)){
7             return _falseSinus(f);
8             }
9         //On verifie si on est entre [1, Pi/2] :
10        if ((f > 1.0) && (f <= (_pi()/2))){
11            newF = (_pi()/2) - f;
12            return _falseCosinus(newF);
13            }
14        //On verifie si on est entre [Pi/2, Pi] :
15        if((f > (_pi()/2)) && (f <= _pi())){
16            newF = _pi() - f;
17            return sin(newF);
18        }
19        //On verifie si c'est entre [Pi; 2 Pi] :
20        if((f > _pi()) && (f <= (2 * _pi()))){
21            newF = f - _pi();
22            return -sin(newF);
23            }
24        //Maintenant si la valeur est pas entre 0 et 2 Pi :
25        else{
26            if(this.getX() < 0){
27                newf = f;
28                while(newF < 0){
29                    newF =(2 * _pi()) + newF;
30                }
31            }
```

```
32              else{
33                      newF = f % (2 * _pi());
34              }
35              return sin(newF);
36          }
37      }
```

Avec comme fonction *falseSinus(float f)* la fonction suivante :

Listing 4: Implémentation en decah de $_falseSinus()$

```
1           float _falseSinus(float f){
2       //Cette fonction sinus fonctionne seulement entre -1 et
            1 avec un taux d'erreur a 3.5%.
3           float somme = 0.0;//On initialise la somme a 0.
4           int compteur = 5;//On initialise le compteur a 5, et
                on decroit ensuite de 1 par tour de boucle pour
               calculer les 5 premiers termes de la somme.
5           int exposant = 11;
6           float terme;
7           while(compteur >= 0){ //Tant que on a pas calcul
               les 5 premiers termes.
8               exposant = (2 * compteur) + 1;
9               terme = _power(-1, compteur) *  (_power(f,
                    exposant)/this._factorial(exposant));
10              somme = somme + terme;//On met a jour la somme.
11              compteur = compteur - 1;//On met a jour le
                    compteur.
12              }
13          return somme; //On renvoi la somme.
14      }
```

Le code en *decah* du cosinus est de la même forme.

4.10 Performance des algorithmes.

Au final les performances de nos algorithme sont les suivantes :

Pour la fonction sinus :

Complexité temporelle :
Le temps de calcul de la fonction falseSinsus est constant car dans tous les cas nous calculons les 5 premiers termes dans la serie entière du sinus. Cependant celui du sin est variable, car il dépend de l'intervalle dans lequel se trouve le flottant initialement.

Précision des calculs :
La précision des calculs dépend fortement de l'intervalle dans lequel se trouve le flottant dont l'on cherche à calculer le sinus. Ainsi on a le tableau suivant:

Intervalle	[- π,- $\frac{\pi}{2}$]	[- $\frac{\pi}{2}$, **-1**]	**[-1, 1]**	**[1,** $\frac{\pi}{2}$]	[$\frac{\pi}{2}$, π]
Pourcentage d'erreur	67	50	3.5	21	70
Pourcentage des erreurs égales à 1 ULP	35	80	100	100	49
Moyenne d'erreur	1×10^{-7}	2×10^{-8}	1 ULP	1ULP	7.3×10^{-8}

Les performances du sinus sont donc acceptable.

Pour la fonction cosinus :

Complexité temporelle :
Le temps de calcul de la fonction falseCosinus est constant car dans tous les cas nous calculons les 5 premiers termes dans la série entière du cosinus. Cependant celui du cos est variable, car il dépend de l'intervalle dans lequel se trouve le flottant initialement.

Précision des calculs :

La précision des calculs dépend fortement de l'intervalle dans lequel se trouve le flottant dont l'on cherche à calculer le sinus. Ainsi on a le tableau suivant:

Intervalle	[- π,- $\frac{\pi}{2}$]	[- $\frac{\pi}{2}$, **-1**]	**[-1, 1]**	**[1,** $\frac{\pi}{2}$]	[$\frac{\pi}{2}$, π]
Pourcentage d'erreur	75	99	11	97	75
Pourcentage des erreurs égales à 1 ULP	0	0	100	0	100
Moyenne d'erreur	1×10^{-7}	1.2×10^{-7}	1 ULP	4.2×10^{-8}	1ULP

Les performances du cosinus sont moin bonnes que celles du sinus mais restent acceptables.

5 Fonction Asin

5.1 Méthode utilisée

Par soucie de temps, nous avons implémenté la fonction Asin de la même manière que nous l'avons fait pour les foncitons sinus et cosinus, c'est-à-dire, à l'aide de son **développement en série entière**.

On rappelle que le développement en série entière de l'arcsinus est le suivant :

$$\forall x \in]-1,1[, arcsin(x) = \sum_{n=0}^{+\infty} \frac{(2n!) \times x^{2n+1}}{(n!)^2 \times (2n+1)}$$

Cependant, **cette somme étant infinie**, il nous faut trouver une condition d'arrêt dans le calcul de la somme tout en ayant la précision maximale. Cependant, comme **le terme au dénominateur diverge très rapidement** nous ne pouvons pas prendre le même critère d'arrêt que pour le sinus/cosinus, qui était : tant que le prochain terme est plus grand que l'ULP de la somme partielle.

En effet il y a **dépassement d'entier** à partir du 9ème terme dans la somme de la série entière de l'arcsin. Cependant dans la plupart des cas, le 8ème terme n'est pas inférieur à la valeur de l'ULP de la somme partielle.

Cela dépend de la vitesse de convergence de la suite des termes de la série vers 0. **Pour les valeurs proches de 0 la convergence sera plus forte**, et donc l'incertitude sur le résultat plus faible.

Pour maximiser la précision, qui ne peut dans tous les cas pas être parfaite, surtout pour les valeurs proche de 1 et -1, nous avons choisi **d'imposer le calcul de la somme jusqu'à son 9ème terme**.

5.2 Implémentation

Ainsi le code en *decah* de notre fonction **asin** est le suivant :

Listing 5: Implémentation de l'asin en decah

```
1     float asin(float f) {
2         float somme = 0.0;//On initialise la somme    0.
3         float terme = 1.0;//On initialise    1 le premier
              terme.
4         int compteur = 8; //On initialise le compteur    8.
5         int exposant = 17; //On initialise l'exopsant (mais
              sa valeur    peut d'exposant).
6
7         if( (f < -1.0) || (f > 1.0)) {//On v rifie que la
              valeur du flottant est dans le domaine de
              definition de la fonction.
8                 println("Error: The asin function receives
                      only values between -1 and 1");
9                 return -1.0;
10        }
11              while(compteur >= 0){ //Tant que on a pas
                    calcul  les 8 premier termes.
12            exposant = 2 * compteur + 1;
13            terme = (_power(f,exposant) * _factorial(2  *
                  compteur))/(exposant * _power(_factorial(
                  compteur), 2) * _power(4 , compteur)) ;//On
                  calcule le terme.
14            somme = somme + terme;
15            compteur = compteur - 1;//On decremente le
                  compteur.
16        }
17        return somme;
18    }
```

5.3 Performance de l'algorithme

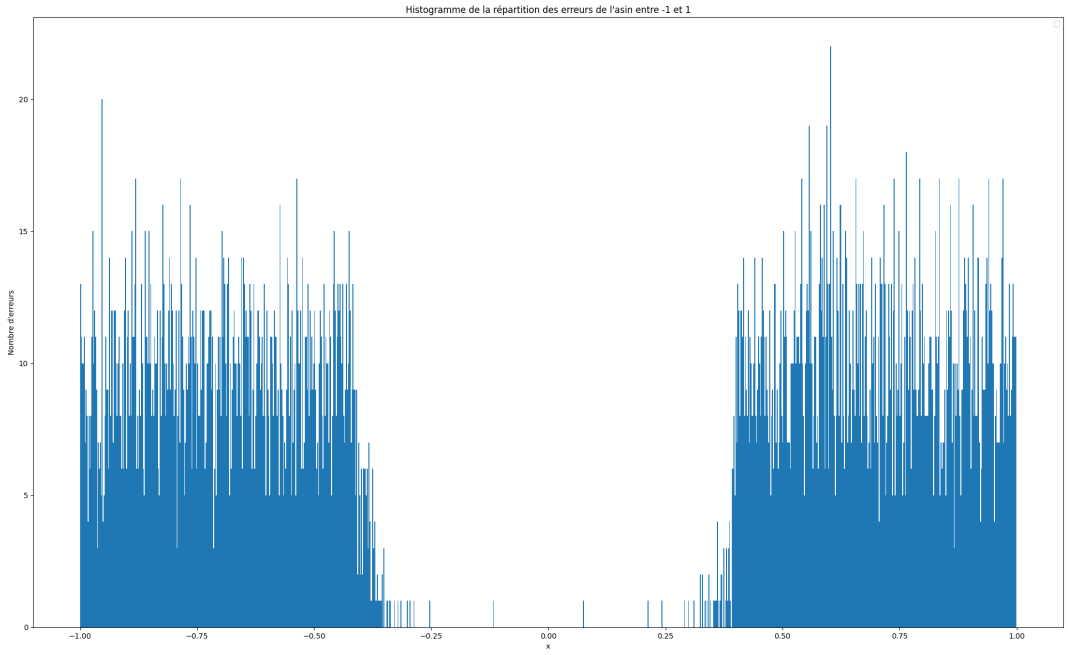
Nous souhaitons maintenant analyser la performance de notre fonction *asin*

Complexité temporelle :

Le temps de calcul de la fonction asin est constant car dans tous les cas nous calculons les 8 premier termes.

Précision des calculs :
Pour tester nôtre algorithme, nous comparons nos résultat avec la fonction *asin* de la bibliothèque *Math* de Java.

On commence par regarder la répartition des erreurs entre -1 et 1 de notre fonction *asin* :



On observe que comme prévu théoriquement, les valeurs proches de 0 convergent rapidement et sont avec peu d'erreur, tandis que lorsque l'on tend vers -1 et 1, la précision diminue. (le palier autour de -0.3 et 0.3 est due au fait que après cette valeur le 8 ème terme n'est plus inférieur à l'ULP de la somme partielle ce qui crée de l'erreur).

Ainsi on observe bien que la précision de notre fonction dépend de l'intervalle auquel appartient le flottant dont l'on chercher à calculer l'arcsinus. On a donc

le tableau suivant :

Intervalle	**]- 1,- 0.3**	**]-0.3, 0.3[**	**[0.3, 1[**
Pourcentage d'erreur	99	0.3	11
Pourcentage des erreurs égales à 1 ULP	0	100	0
Moyenne d'erreur	8×10^{-3}	1 ULP	8×10^{-3}

Ainsi la précision de notre fonction *asin* est la moin bonne de toutes nos fonctions avec de grosses erreurs proche de 1 et -1. Cependant elle reste excellente entre les valeurs -0.3 et 0.3 et ce niveau d'erreur peut être réduit en codant les entiers sur 64 bits plutôt que 32 pour éviter le **dépassement d'entiers**.

6 Fonction Atan

6.1 Méthode utilisée

Par soucie de temps, nous avons implémenté la fonction Atan aussi de la même manière que pour les autres fonctions , c'est-à-dire, à l'aide de son **développement en série entière**.

On rappelle que le développement en série entière de l'arctan est le suivant :

$$\forall x \in]-1,1[, arcsin(x) = \sum_{n=0}^{+\infty} \frac{(-1)^n \times x^{2n+1}}{(2n+1)}$$

Cependant, **cette somme étant infinie**, il nous faut trouver une condition d'arrêt dans le calcul de la somme tout en ayant la précision maximale.
Mais cette fois le terme au dénominateur diverge lentement par rapport aux autres fonctions. Ainsi il n'y a **pas de risque de dépassement d'entier**. Mais par soucie de complexité temporelle nous devons tout de même choisir une condition d'arrêt choisit à 100 termes calculés (car le meilleur ratio temps/précision).

6.2 Implémentation

Ainsi le code en *decah* de notre fonction **atan** est le suivant :

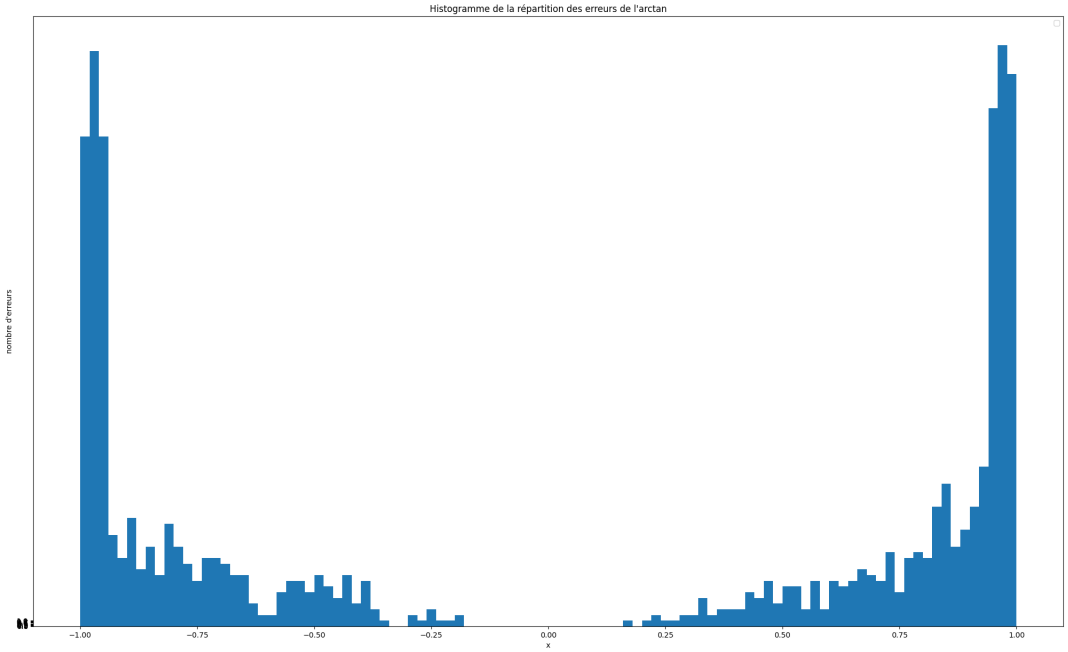
Listing 6: Implémentation de l'asin en decah

```
1    float atan(float f) {
2        //Fonction qui calcule la valeur de l'arctan d'un
             flottant entre -1 et 1.
3        float somme = 0.0;//On initialise la somme    0.
4        float terme = 1.0;//On initialise    1 le premier
             terme.
```

```
5           int compteur = 100;
6           int exposant = 201;
7
8           if( (f < -1.0) || (f > 1.0)) {
9                   println("Error: The atan function receives
                        only values between -1 and 1");
10                  return -1.0;
11          }
12
13          while(compteur >= 0){ //Tant que on a pas calcul
                les 100 premier termes.
14              exposant = 2 * compteur + 1;
15              terme = (_power(f,exposant) * _power(-1 ,
                    compteur))/(exposant);
16              somme = somme + terme;
17              compteur = compteur - 1;
18          }
19          return somme;
20      }
```

6.3 Performance de l'algorithme

On commence par regarder la répartition des erreurs entre -1 et 1 de l'arctan.



On a bien que autour des valeurs -1 et 1 pour lesquels la convergence est plus lente que la le nombre d'erreurs augmente.

7 Conclusion

Ainsi bien que nos fonctions ne soient pas exact aux valeurs théoriques des fonctions trigonométriques, leur approche reste bonne, avec un taux d'erreur toujours quantifié. Cette extension est fonctionnelle et sait trouver un bon compromis entre précision et vitesse de calcul. Pour l'utiliser, écrivez "include Math" dans vôtre fichier .decah.

8 Bibliographie

- BibMath, lien : https://www.bibmath.net/dico/index.php?action=affichequoi=./s/seriealt.html pour les théorème utilisés.
- Wikipédia, lien : https://fr.wikipedia.org/wiki pour la représentation IEEE754.
- TrigoFacile, lien : http://www.trigofacile.com/maths/trigo/calcul/cordic/cordic.htm pour la méthode COORDIC.