Decac

Groupe Génie Logiciel n°29

Compilateur pour le langage Deca

Extension TRIGO

Janvier 2024

Table des matières

1 Généralités **2**
1.1 Méthodes et attributs de la classe Math proposée 2
1.2 Utilisation 2

2 Implémentation des méthodes **3**
2.1 Fonction trigonométriques : idée générale, exemple avec *cos* 3
2.2 *sin* 5
2.3 *asin* 6
2.4 *atan* 7
2.5 Unit of Least Precision: ULP 8
2.6 Comparaison des méthodes en Java 9

3 Limites de la classe Math **10**
3.1 Périmètre et fonctionnalités 10
3.2 Gestion des cas limites 10
3.3 Précisions des calculs 10
3.3.1 Fonctionnalité des algorithmes 10
3.3.2 Puissance de calcul 11

4 Bibliographie **12**

1 Généralités

1.1 Méthodes et attributs de la classe Math proposée

Parmi les diverses extensions proposées, nous avons choisi l'extension TRIGO. Elle consiste à implémenter diverses fonctions trigonométriques, ainsi que la fonction ULP dans un fichier `Math.decah`. Ce fichier est destiné à enrichir la bibliothèque standard Deca. Les méthodes de la classe `Math` de Deca que nous proposons sont les suivantes :

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

Ces fonctions étaient imposées par le cahier des charges de l'extension, mais nous avons pris la liberté d'ajouter une classe `Float` avec des méthodes et des attributs qui facilitaient l'implantation des méthodes susmentionnées :

- `float _modulo(float f1, float f2)` renvoie $f_1 modulo f_2$
- `int _sign(float f)` renvoie $+1$ si le flottant est positif, -1 sinon
- `float _abs(float f)`: fonction valeur absolue
- `float _pow(float f, int exp)` renvoie f^{exp}
- `int _fact( int n)` renvoie $n!$

1.2 Utilisation

Pour utiliser les classes Float et Math, il faut tout d'abord inclure le fichier `Math.decah` pour qu'il soit compilé en même temps que le fichier principal. Il faut ensuite instancier la classe Math pour pouvoir appeler les différents attributs et méthodes de la classe Math. Par exemple :

```
1 #include "Math.decah"
2
3 {
4     Math m = new Math();
5     println(m.cos(3.4785523));
6 }
```

2 Implémentation des méthodes

2.1 Fonction trigonométriques : idée générale, exemple avec *cos*

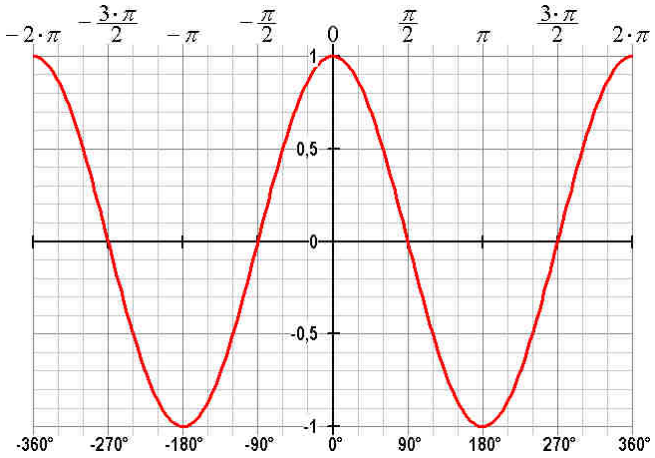

Figure 1: Courbe de la fonction cosinus

Pour évaluer la valeur d'une fonction trigonométrique en un point x, *cosinus* par exemple, on utilise son Développement en Série Entière :

$$\forall x \in \mathbf{R} : \cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

Cette expression est évidemment purement mathématique, on ne peut matériellement pas calculer une somme infinie. Il faut donc approcher cette expression par une somme finie :

$$\cos x \approx \sum_{n=0}^{p} \frac{(-1)^n}{(2n)!} x^{2n}$$

On prend arbitrairement $p = 255$. On peut alors naïvement traduire cette expression en une méthode Java suivante :

```java
public float cosNaif(float x){
    float somme = 0;
    int n = 0;
    while (n!=255){
        somme=somme+UMath.pow(-1,n)*UMath.pow(x,2*n)/UMath.fact(2*n)
        n=n+1;
    }
    return somme;
}
```

On implémente les méthodes `cosNaif`, `pow` (puissance) et `fact` (factorielle) dans une classe `UMath.java (src/main/java/fr/ensimag/trigo/UMath.java` et on la teste avec `TestUMath.java (src/main/java/fr/ensimag/trigo/TestUMath.java)`. On remarque un temps de calcul très élevé comparé à la méthode cos de `java.math` et que le calcul peut être considéré comme erroné :

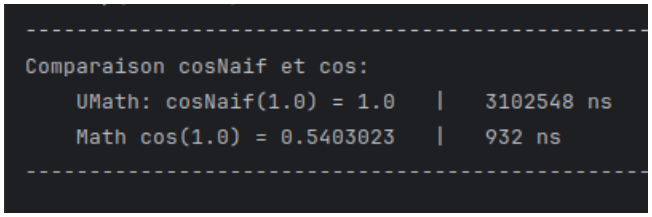

Figure 2: Notre méthode naïve est plus lente que la méthode standard d'un facteur 3329 et très imprécise

Ces résultats ne sont pas suprenants. Chaque terme de la somme appelle en effet 2 fois la fonction `pow` et une fois la fonction `fact`. Ces 2 fonctions sont récursives, elles sont donc appelées un total de respectivement $3n$ et $2n$ fois par terme de somme, soit $5n+1$ appels récursifs. Pour évaluer cette somme en un flottant ($n_{max} = 255$), on réalise donc de l'ordre de 160000 appels récursifs !

Pour améliorer les performances de calcul et limiter les imprécisions, on utilise la méthode de Horner, qui consiste à factoriser la somme finie.

À l'ordre 2 par exemple :

$$\sum_{n=0}^{2} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} = 1 - x^2(\frac{1}{2!} - \frac{x^2}{4!})$$

À l'ordre 3 :

$$\sum_{n=0}^{3} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} = 1 - x^2(\frac{1}{2!} - x^2(\frac{1}{4!} - \frac{x^2}{6!}))$$

On peut donc écrire :

$$\forall p \in \mathbf{N}^*, \sum_{n=0}^{p} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - x^2 \times F_{cos,1}$$

avec la relation de récurrence pour les facteurs de Horner :

$$\forall i \in [1,p], F_{cos,i} = \frac{1}{(2i)!} - x^2 \times F_{cos,i+1}$$

$$F_{cos,p} = \frac{1}{(2p)!}$$

Pour encore factoriser les calculs, on exprime le terme factoriel à l'ordre n en fonction de celui à l'ordre $n-1$, et on stocke dans une variable la valeur de x^2. Par ailleurs, la fonction *cosinus* est 2π périodique, *ie* $\cos(x+2\pi) = cos(x)$. On peut donc travailler uniquement sur l'intervalle $[-2\pi; 2\pi]$. Pour ce faire, on prend l'argument de la fonction modulo 2π. Cela permet de travailler avec des "petits" nombres et minimiser les erreurs dues à la manipulation de "grands" nombres.

On obtient l'algorithme suivant :

```
    float cosHorner(float f2, int n, float invFact) {

        invFact = invFact/(2*n*(2*n-1));

        if (n = 255) {
            return f2 * invFact;
        }

        return invFact - f2 * this.cosHorner(f2, n + 1, invFact);
    }

    float cos(float f) {

        float f2 = f*f;

        return 1-f2*this.cosHorner(f2, 1, 1);
    }
```

Grâce à la méthode de Horner et à ces différentes factorisations, on réduit le temps de calcul par un facteur 119. L'ordre de grandeur du temps de calcul se rapproche de celui de la méthode `cos` de la classe `Math` de Java. La précision du résultat est également grandement améliorée.

```
------------------------------------------------------------
Comparaison cosNaif et cos:
    UMath: cosNaif(1.0) = 1.0    |   929248 ns
    UMath: cosHorner(1.0) = 0.5403023    |   7815 ns
    Math cos(1.0) = 0.5403023    |   2355 ns
------------------------------------------------------------
```

Figure 3: Comparaison des résultats et de la rapidité de CosNaif, cosHorner et cos

2.2 *sin*

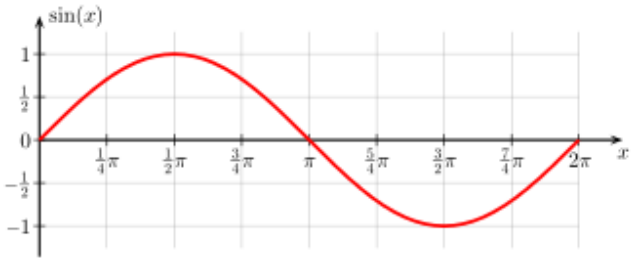

Figure 4: Courbe de la fonction sinus

Le DSE de la fonction *sinus* est :

$$\forall x \in \mathbf{R} : \sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \approx \sum_{n=0}^{p} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

À l'ordre 2 :

$$\sum_{n=0}^{2} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} = x(1 - x^2(\frac{1}{3!} - \frac{x^2}{5!}))$$

À l'ordre 3 :

$$\sum_{n=0}^{3} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} = x(1 - x^2(\frac{1}{3!} - x^2(\frac{1}{5!} - \frac{x^2}{7!})))$$

On généralise donc $\forall p \in \mathbf{N}^*$:

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

Avec la relation de récurrence pour les facteurs de Horner :

$$\forall n \in [1, p-1] F_{sin,n} = \frac{1}{(2n+1)!} - x^2 \times F_{sin,n+1}$$

$$F_{sin,p} = \frac{1}{(2p+1)!}$$

De même que pour la fonction *cos*, on exprime le terme factoriel de manière récursive, on stocke la valeur de x^2 dans une variable et on travaille avec $x[2\pi]$.

2.3 *asin*

Le DSE de *arcsinus* est:

$$\forall x \in [-1;1] : \arcsin x = \sum_{n=0}^{\infty} \frac{\binom{2n}{n}}{2^{2n} \times (2n+1)} \times x^{2n+1} = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n \times (2n+1) \times (n!)^2} \times x^{2n+1}$$

$$\approx \sum_{n=0}^{p} \frac{(2n)!}{4^n \times (2n+1) \times (n!)^2} \times x^{2n+1}$$

Et $\forall p \in \mathbf{N}^*$:

$$\sum_{n=0}^{p} a_n \times x^{2n+1} = x \times F_{asin,1}$$

Avec la relation de récurrence pour les facteurs de Horner :

$$\forall n \in [1, p-1] F_{asin,n} = a_n - x^2 \times F_{asin,n+1}$$

$$F_{asin,p} = a_p$$

et :

$$\forall n \in [1, p-1] a_n = \frac{(2n)!}{4^n \times (2n+1) \times (n!)^2}$$

$$a_0 = 1$$

On remarque qu'une partie de l'expression de a_n est récursive :

$$\forall n \in [1, p-1] a_n = \frac{rec(n)}{2n+1}$$

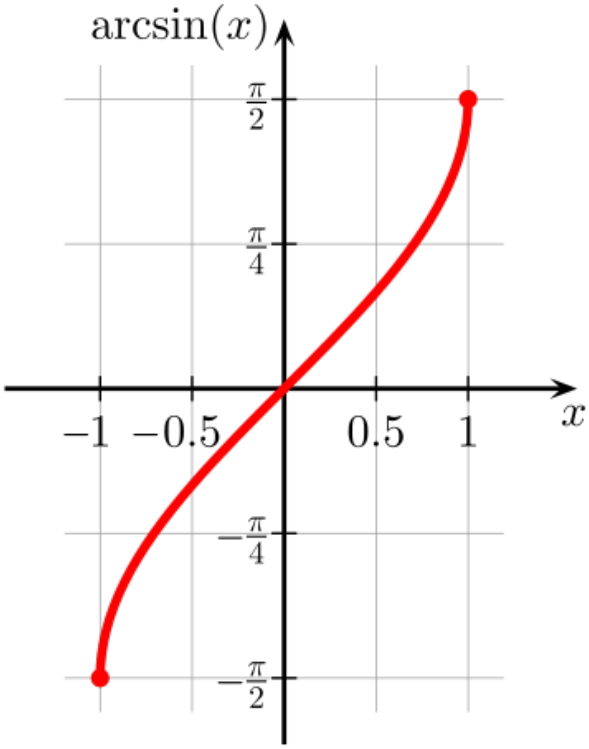

Figure 5: Courbe de la fonction arcsinus

$$\forall n \in [1, p-1] rec(n) = \frac{(2n)!}{4^n \times (n!)^2} = rec(n-1) \times (1 - \frac{1}{2n})$$
$$rec(0) = 1$$

De même que pour les fonctions précédentes, on stocke la valeur de x^2 dans une variable de manière à limiter les répétitions de calculs qui engendrent des imprécisions.

2.4 *atan*

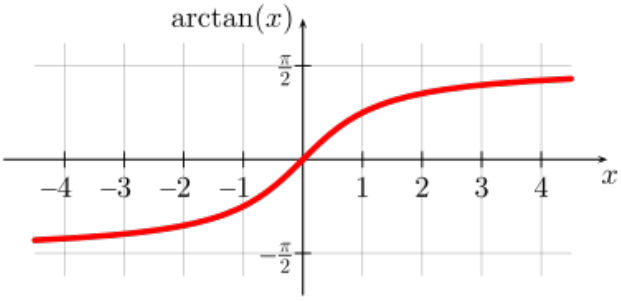

Figure 6: Courbe de la fonction arctangente

Le DSE de la fonction *arctan* est :

$$\forall x \in [-1;1] : \arctan x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \times x^{2n+1} \approx \sum_{n=0}^{p} \frac{(-1)^n}{2n+1} \times x^{2n+1}$$

Et $\forall p \in \mathbf{N}^*$:

$$\sum_{n=0}^{p} \frac{(-1)^n}{2n+1} \times x^{2n+1} = x \times F_{atan,0}$$

Avec la relation de récurrence pour les facteurs de Horner :

$$\forall n \in [0, p-1] F_{atan,n} = \frac{1}{2n+1} - x^2 \times F_{atan,n+1}$$

$$F_{atan,p} = \frac{1}{2p+1}$$

De même que pour les fonctions précédentes, on stocke la valeur de x^2 dans une variable de manière à limiter les répétitions de calculs qui engendrent des imprécisions.

Ce DSE n'est valable que sur un intervalle fini de flottants $[-1;1]$. En dehors de cet intervalle, notre méthode est invalide. Il faut donc se ramener au cas $x \in [-1;1]$. Pour ce faire, on utilise la formule suivante :

$$\forall x \in \mathbf{R}^* arctan(x) = arctan(\frac{1}{x}) + sign(x) \times \frac{\pi}{2}$$

Ainsi, si $|x| > 1$, $|\frac{1}{x}| < 1$ donc on calcule $arctan(x)$ grâce à la formule précédente, sinon on calcule directement $arctan(x)$.

2.5 Unit of Least Precision: ULP

L'ULP (*Unit of Least Precision* ou *Unit in Last Position*), littéralement unité de plus petite précision/en dernière position, se définit comme l'écart entre 2 flottants successifs. En effet, dans la norme IEEE 754, un flottant est représenté sur 32 bits comme suit :

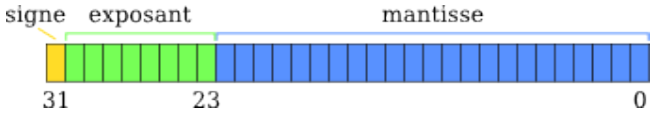

Figure 7: Représentation binaire d'un flottant 32 bits suivant la norme IEEE 754

Avec 1 bit de signe s, 8 bits pour représenter l'exposant e et 23 bits pour la mantisse m. La valeur du flottant est alors :

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

Cette représentation binaire n'est pas linéaire, c'est-à-dire que les bits de poids faibles ne sont pas nécessairement à droite, et ceux de poids forts à gauche. Par exemple, pour $e = 0$ et $m = 1$, on a $f = s \times 2^{-127} \times (1 + 2^{-23})$. Le dernier bit d'exposant (de rang 23) a donc moins de poids que le dernier bit de la mantisse (rang 0). La distribution des réels représentables par des flottants n'est en conséquence pas uniforme. L'ULP permet d'évaluer cette distribution.

On peut exprimer l'ULP d'un flottant, noté $ulp(f)$ par la formule suivante :

$$\text{Si } 2^e < f \leq 2^{e+1} \text{ alors } ulp(f) = 2^{e-24},\ e \in [-127, 128]$$

Pour calculer l'ULP d'un flottant, on procède de manière itérative, avec une boucle `while` pour encadrer f par des puissances de 2 successives 2^e et 2^{e+1}. On renvoie alors 2^{e-24}.

2.6 Comparaison des méthodes en Java

La classe `TestUMath` permet de comparer les fonctions `cos, sin, atan, ulp et asin` de notre classe `UMath` où sont retranscrites les méthodes ci-dessus en Java à celles de la classe `Math` Java sur 10 valeurs aléatoires, et de calculer l'erreur moyenne.

On obtient des erreurs moyennes de l'ordre de:

- 10^{-1} pour *cos*
- 10^{-1} pour *sin*
- 10^{-5} pour *atan*
- 0 pour *ulp*
- 10^{-9} pour *asin*

Les résultats peuvent être considérés comme acceptable dans une certaine mesure. Les algorithmes sont corrects.

3 Limites de la classe Math

3.1 Périmètre et fonctionnalités

La classe Math que nous proposons est très incomplète et loin d'être adaptée à une résolution de problèmes mathématiques complexes. Nous n'avons en effet implémenté que très peu de fonctions. L'utilisateur se retrouvera rapidement limité dans sa capacité à résoudre certains problèmes, comme l'utilisation de la fonction mathématique *exp*. La classe Math peut donc être complétée par une classe définie par l'utilisateur.

Toutefois, les algorithmes que nous proposons sont corrects et permettent une bonne approximation des valeurs théoriques.

3.2 Gestion des cas limites

D'autre part, la machine IMA utilisée pour le développement de notre projet fonctionne avec des flottants 32 bits sous la norme IEEE-754, alors que la plupart des systèmes actuels fonctionnent sur 64 bits. De plus, nous n'avons pas accès à la représentation binaire des flottants. Les flottants représentants des constantes telles que l'infini (positif et négatif) et le type NaN (Not a Number) manquent à la classe Math, faute de pouvoir les représenter correctement en écriture décimale et même hexadécimale.

Concrètement, en Java, certaines constantes comme POSITIVE_INFINITY ou NaN sont définis avec les méthodes `floatToIntBits / IntBitsToFloat` de la classe `Float` de Java. On ne peut pas implémenter de méthode similaire dans la classe Math de Deca, car nous ne disposons pas d'outils pour récupérer la représentation binaire d'un flottant en Assembleur. Il est donc impossible de représenter correctement ces constantes. Par exemple, POSITIVE_INFINITY est défini comme le flottant de représentation binaire IEEE-754 identique au résultat de la méthode `Float.floatToIntBits(0x7f800000)`, c'est-à-dire 01111111100000000000000000000000 ou en décimal 2^{128}. Malheureusement, l'exécution de la commande `pow(2.0,128)` provoque un overflow et l'arrêt complet du programme. On ne peut donc ni travailler avec ces constantes ni proprement gérer les cas limites des fonctions.

3.3 Précisions des calculs

Deux facteurs indépendants entrent en jeu dans la précision des résultats obtenus avec nos classes Float et Math :

- la fonctionnalité des algorithmes
- la puissance de calcul d'IMA.

3.3.1 Fonctionnalité des algorithmes

Avant d'être implémentés dans les classes Float et Math de Deca, tous les algorithmes ont été testés en Java (avec `UMath.java` et `TestUmath.java`). Ce fut réalisé dans le but de vérifier la fonctionnalité des algorithmes et de les tester séparément de la machine IMA, ce qui élimine les sources d'erreurs

et d'imprécisions. L'exécution de la méthode `Main` de la classe `TestUMath` compare les résultats des méthodes de la classe `UMath` et de ses attributs à ceux de la classe `Math` de Java. On remarque que toutes les constantes ($\pm\infty, NaN$, etc...) sont pour la majorité correctement évalués, ce qui prouve que la machine virtuelle IMA est bien source d'erreurs.

3.3.2 Puissance de calcul

La puissance de calcul est ici limitée par 2 choses :

- la représentation des flottants en 32 bits, que nous avons déjà abordé
- les arrondis réalisés lors des calculs sur les flottants

La précision des calculs sur les flottants, notamment la multiplication suivie d'une addition, laissent en effet grandement à désirer. Nous n'avons pas utilisé l'instruction assembleur FMA (Fusion Multiplication Addition) à notre disposition. Néanmoins, les résultats des opérations concordent en général à quelques chiffres après la virgule avec ceux obtenus avec les classes Float et Math de Java. Par exemple, en remarquant que $arctan1 = \frac{\pi}{4}$, on peut aisément approcher la valeur de π, c'est-à-dire trouver le flottant qui se rapproche le plus du réel π. On obtient `4*atan(1)=3.99217`, ce qui est évidemment une très mauvaise approximation de π. De ce fait, on définit π par la valeur littérale de `Math.PI` de la classe Math de Java (`3.1415927`).

La classe `CommpareDecaJava` permet de comparer les fonctions `cos, sin, atan, ulp et asin` de notre classe `Math` Deca à celles de la classe `Math` Java sur 10 valeurs, et de calculer l'erreur moyenne. Il faut pour cela compiler le fichier `CompareDecaJava.deca` à l'aide de la commande: `mvn compile && src/main/bin/decac src/test/deca/trigo/CompareDecaJava.deca && ima src/test/deca/trigo/CompareDecaJava.ass > src/test/deca/trigo/ResCompareDecaJava.txt`

On obtient des erreurs moyennes suivantes :

- 0 pour *sin*
- 0.3064401 pour *cos*
- 0.38108847 pour *atan*
- 0.22904143 pour *ulp*
- 0.36698166 pour *asin*

Comparé aux résultats précédents (comparaison des méthodes en Java), on remarque une nette dégradation de la précision pour la majorité des fonctions. Les défauts de la machine IMA et de son utilisation comptent donc beaucoup, les résultats ne sont plsu du tout acceptables.

4 Bibliographie

- Norme IEEE 754
- Unit in the last place