Projet Génie Logiciel

Compilateur Decac

Documentation de la bibliothèque standard

Équipe GL2

Étudiants : Élina Houdé, Julien Pinto Da Fonseca,

Léa Solo Kwan, Yin Xu

Année 2019/2020

Sommaire

Introduction **3**

1. Extension TRIGO **4**

1.1. Table des méthodes - Math.decah 4

1.2. Choix de conception 6

1.3. Méthode de validation 7

1.4. Résultats et limites de conception 9

2. Références bibliographiques **12**

1.1. Extension TRIGO 12

Introduction

Ce document a pour but de spécifier les extensions de la bibliothèque standard implémentées dans le compilateur Decac. Vous pourrez retrouver dans ce document les spécifications de chaque extension et leur implémentation, ainsi que les méthodes de validation employées et leurs résultats.

1. Extension TRIGO

1.1. Table des méthodes - Math.decah

L'extension TRIGO se traduit par l'implémentation de la bibliothèque **Math.decah**, celle-ci se situant dans le dossier **src/main/resources/include/**. Toutes les méthodes de cette bibliothèque sont accessibles depuis la classe **Math**.

On retrouvera notamment les méthodes suivantes :

- **float ulp(float f)** : prend en paramètre **f** un flottant et retourne la taille de son ulp (*Unit in the Last Place*). L'ulp est la distance qui sépare deux flottants consécutifs et donc, représente l'erreur maximale qui peut être faite lors d'un arrondi. Plus formellement, la valeur de l'ulp pour un nombre flottant x est définie par :

$$ulp(x) = \begin{cases} x^+ - x & \text{si} \quad x \geq 0 \\ x - x^- & \text{si} \quad x < 0 \end{cases}$$

  où x^- et x^+ sont respectivement les flottants prédécesseur et successeur de x.

- **float sin(float rad)** : prend en paramètre **rad** un angle, en radians, et retourne la valeur du sinus de cet angle.

- **float cos(float rad)** : prend en paramètre **rad** un angle, en radians, et retourne la valeur du cosinus de cet angle.

- **float asin(float rad)** : prend en paramètre **rad** un angle, en radians, et retourne la valeur de l'arc sinus de cet angle.
  ➔ **Cette méthode n'est pour le moment pas implémentée.**

- **float atan(float rad)** : prend en paramètre **rad** un angle, en radians, et retourne la valeur de l'arc tangente de cet angle.
  ➔ **Cette méthode n'est pour le moment pas implémentée.**

- **float _getMaxValue()** : accesseur de la valeur flottante MAX_VALUE, il s'agit de la valeur maximale pouvant être stockée dans un nombre flottant (langage Déca).

- **float _getMinValue()** : accesseur de la valeur flottante MIN_VALUE, il s'agit de la valeur minimale pouvant être stockée dans un nombre flottant (langage Déca).

- **int _getExponent(float x)** : prend en paramètre **x** un flottant, et retourne la valeur (entière) de l'exposant non biaisé dans le contexte de la représentation d'un flottant.

- **float _abs(float x)** : prend en paramètre **x** un flottant, et retourne sa valeur absolue :
    - Si le paramètre n'est pas négatif, celui-ci est directement retourné.
    - Si le paramètre est négatif, la négation de celui-ci est retournée.
    - Si le paramètre est un zéro positif ou négatif, le résultat retourné est un zéro positif.

- **float _pow(float x, int e)** : prend en paramètre **x** une valeur flottante et **e** un exposant entier. Retourne la valeur du paramètre **x**, élevé à la puissance **e**, soit la valeur « x^e ».

- **float _sinX(float rad)** : prend en paramètre **rad** un angle, en radians, et retourne la valeur du sinus de cet angle. Contrairement à la méthode **float sin(float rad)**, celle-ci n'utilise aucune conversion de radians en degrés et devrait donc être plus précise.
    ➔ **Il s'agit d'une méthode expérimentale qui n'a pas été terminée.**

- **float _getPI()** : accesseur de la valeur flottante PI, représentative de la valeur du nombre pi « π ».

- **float _getMaxInt()** : accesseur de la valeur flottante MAX_INT, il s'agit de la valeur maximale pouvant être stockée dans un nombre entier (langage Déca).

- **float _toDegrees(float rad)** : prend en paramètre **rad** un angle, en radians, et retourne son équivalent en degrés. La conversion est généralement inexacte.

- **float _toRadians(float deg)** : prend en paramètre **deg** un angle, en degrés, et retourne son équivalent en radians. La conversion est généralement inexacte, il ne faut ainsi pas espérer que *cos(_toRadians(90.0))* retourne exactement *0.0*.

- **float _reduceDegrees(float deg)** : prend en paramètre **deg** un angle, en degrés, et retourne un équivalent de cet angle une fois réduit au modulo 360. Généralement on utilisera cette fonction pour éviter le dépassement arithmétique dû à un cast d'un flottant trop grand pour être contenu dans un entier. Exemples : _reduceDegrees(2147483650+100000) = 5.00000e+01 ; _reduceDegrees(5432483650.123) = 1.30123e+02 ; etc.
    ➔ **Il s'agit d'une méthode expérimentale qui n'a pas été terminée.**

- **float _sinePoly(float rad, int order)** : prend en paramètre **rad** un angle, en radians et **order** l'ordre entier maximal du polynôme. Retourne le résultat du sinus polynomial basé sur les séries de Taylor. Plus l'ordre maximum appliqué est grand et plus le résultat sera précis ; le résultat est une approximation.

- **float _sinePolyX(float rad)** : prend en paramètre **rad** un angle, en radians, et retourne le résultat du sinus polynomial basé sur les séries de Taylor. L'ordre maximum appliqué est le plus petit ordre possible permettant d'atteindre l'approximation désirée (à 1 ulp près par exemple) ; le résultat est une approximation. Contrairement à la méthode **float _sinePoly(float rad, int order)**, celle-ci est optimisée et ne fait pas de calcul inutile.
  ➔ **Il s'agit d'une méthode expérimentale qui n'a pas été terminée.**

- **float _cosPoly(float rad, int order)** : prend en paramètre **rad** un angle, en radians et **order** l'ordre entier maximal du polynôme. Retourne le résultat du cosinus polynomial basé sur les séries de Taylor. Plus l'ordre maximum appliqué est grand et plus le résultat sera précis ; le résultat est une approximation.

- **int _fact(int n)** : prend en paramètre **n** un entier, et retourne sa factorielle :
  - Si n est un entier négatif, on retourne la factorielle de sa valeur absolue.
  - Si n est null, on retourne 1.

1.2. Choix de conception

La réalisation de la méthode **ulp** a été basée sur le fonctionnement et le code de la méthode ***java.lang.Math.ulp*** (i.e. méthode ulp de la bibliothèque *Math* de référence en Java, codée par Joseph D. Darcy). On retrouvera donc pour cette méthode un code et un comportement très similaire, voir identique.

Les méthodes **sin** et **cos** sont basés sur les formules d'**approximation polynomiales**, elles-mêmes utilisant les séries de **Taylor**. Cette méthode d'approximation est très précise lorsque x se rapproche de 0 mais perd en précision lorsque x devient plus grand. Pour base de référence, on notera que pour $x = \pi/4$ radians (i.e. 45°), la formule pour le sinus aura théoriquement une marge d'erreur de **± 0.000 04**, ce marge sera de **± 0.000 004** pour le cosinus et enfin **± 0.004** pour la tangente.

Afin de pallier à ce problème de précision, pour la méthode **sin**, nous utilisons dans un premier temps la **périodicité** de la fonction sinus afin de ramener la valeur x de l'angle, en radians, passé en paramètre sur l'intervalle $[0, 2\pi[$ (i.e. $[0, 360[$ en degrés). Dans un deuxième temps, la **symétrie** de la fonction est utilisée afin de ramener la valeur de l'angle sur le premier quadrant d'un cercle trigonométrique, soit sur l'intervalle $[0, \pi/2]$ (i.e. $[0, 90[$ en degrés) ; si l'angle se situe dans le quadrant n° 3 ou n° 4, on attachera un signe **-** à la fonction sinus.

Pour exemples :

- sin(110°) = sin(70°)
- sin(200°) = **-** sin(20°) | on attache un signe **-**
- sin(330°) = **-** sin(30°) | on attache un signe **-**

Enfin, si l'angle se situe dans l'intervalle se situe désormais dans l'intervalle $]\pi/4, \pi/2]$ (i.e. $]45, 90]$ en degrés), on peut utiliser la **co-fonction**, c'est-à-dire retourner directement le résultat de la fonction cosinus($\pi/2$ - x). Dans le cas contraire, on sait que l'angle se situe dans l'intervalle $[0, \pi/4]$ (i.e. $[0, 45]$ en degrés), celui-ci est donc suffisamment petit pour utiliser la méthode d'**approximation polynomiale** basé sur les séries de **Taylor** pour le calcul du sinus. On retourne ainsi le résultat de la fonction sinus(x) avec une erreur théorique de **± 0.000 04**.

La démarche est identique pour la méthode **cos**, à la seule différence que lors de l'utilisation de la **symétrie**, on attachera un signe - à la fonction cosinus si l'angle se situe dans le quadrant n° 2 ou n° 3 (et non le quadrant n° 3 ou n° 4 comme cela est le cas pour le sinus). Également, dans le cas de l'utilisation de la **co-fonction**, on retournera directement le résultat de la fonction sinus($\pi/2$ - x). Dans l'intervalle $[0, \pi/4]$ (i.e. $[0, 45]$ en degrés), la méthode d'**approximation polynomiale** basé sur les séries de **Taylor** pour le calcul du cosinus retourne ainsi un résultat avec une erreur théorique de **± 0.000 004**.

Afin de renforcer la précision des résultats et ainsi réduire cette erreur, l'approximation polynomiale a été réalisée à l'**ordre 31**. Il s'agit d'un ordre plutôt élevé, ce qui permettra d'obtenir des résultats plus précis qu'une approximation réalisée à un ordre inférieur… La valeur de cet ordre de référence a été choisie arbitrairement de façon à ne pas effectuer de dépassement arithmétique et à ne pas effectuer de trop long temps de calculs.

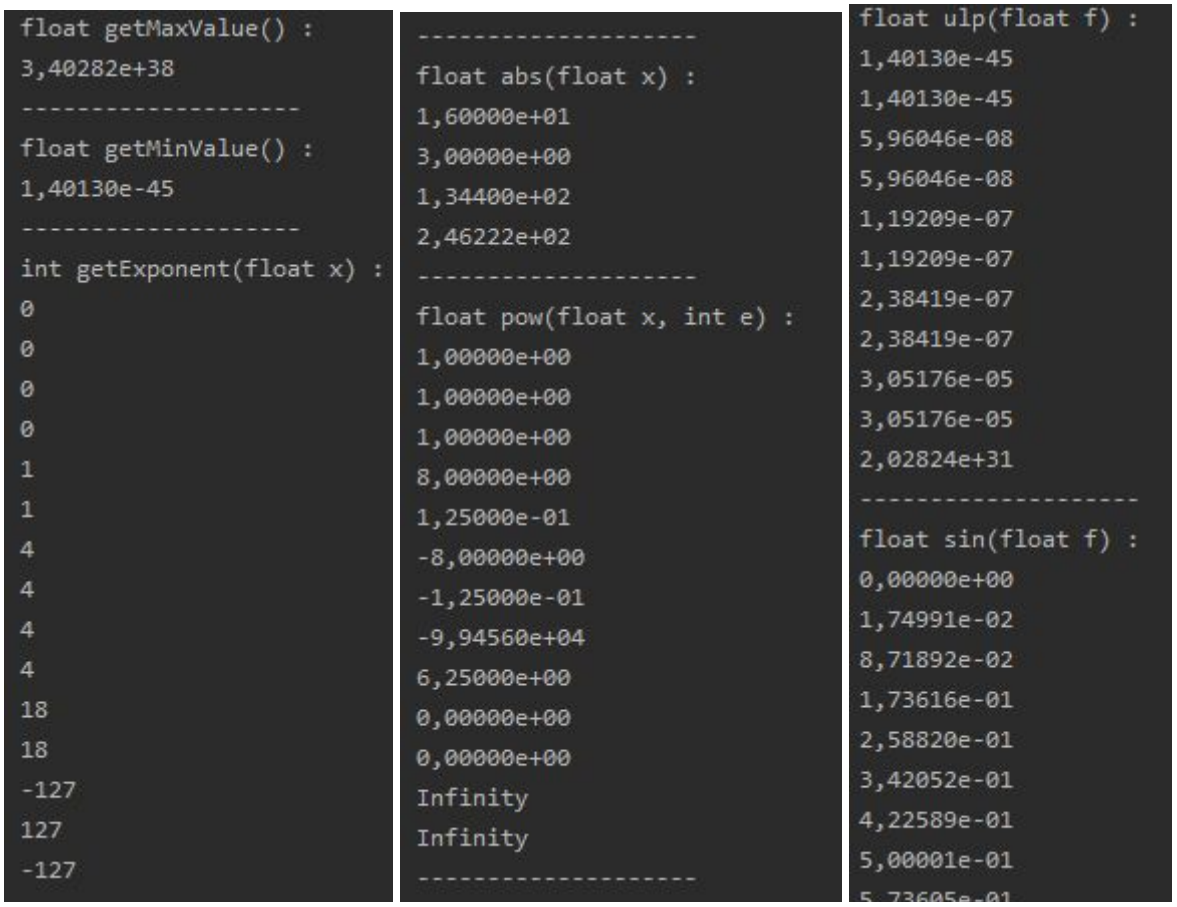
La classe Math contient d'autres méthodes, nécessaire à la réalisation des fonctions **ulp**, **sin** et **cos**. On retrouvera également parmi elles, des méthodes "**expérimentales**" qui n'ont pas été finie. Ces méthodes ont pour but d'améliorer la précision des fonctions ulp, sin et cos mais n'ont pas été finies par manque de temps ; elles pourraient être reprises par un éventuel développeur désirant maintenir ou améliorer l'extension TRIGO de la bibliothèque standard.

1.3. Méthode de validation

Afin de valider les résultats de nos méthodes réalisés en langage Deca, d'un côté, la classe **MathSimu.java** a été réalisée. Il s'agit d'une classe de test manuel se situant dans le dossier **src/test/java/include/**.

Cette classe comporte différentes méthodes, appelant, sur un lot de valeurs spécifiques, les méthodes **double ulp(double d)**, **double sin(double a)** et double **cos(double a)**, ainsi que les méthodes dites “intermédiaires” telles que **int getExponent(float f)** et **double pow(double a, double b)** de la bibliothèque *Math* de référence en Java. On sera également amené à récupérer certaines valeurs de références telles que **PI**, **Float.MAX_Value**, **Float.MIN_Value** ou encore **Integer.MAX_Value**. Ce, afin d’obtenir, pour chaque lot de valeurs spécifiques, des résultats de référence pouvant être utilisé à titre comparatif sur nos méthodes équivalentes réalisés en langage Deca.

Exemples de résultats donnés par l’exécution de la classe **MathSimu** :



D’un autre côté, un ensemble de tests systèmes ont été mis en place sur ces même méthodes en langage Déca et également sur ces mêmes lots de valeurs spécifiques. Ces tests se situent dans le dossier **src/test/deca/codegen/valid/bibliothequeStandard/**. Le niveau de précision testé est ici à la décimale près, via l’affichage des flottants et des éventuels entiers par représentation décimale de la méthode println() en Déca : par exemple *1.00000e+00* pour le flottant *1.0f* et *2.02824e+06* pour le flottant *2028241.5f*.

1.4. Résultats et limites de conception

En comparant les résultats obtenus par les méthodes Déca et les résultats de référence obtenus via MathSimu par les méthodes Java, on peut constater que pour la méthode **ulp**, la représentation décimale est identique pour les deux résultats, on en déduit donc une précision d'au moins **0.000 01** près pour notre méthode Déca.

Pour la méthode **sin**, malgré la correspondance de la majorité des valeurs testés se situant entre 0 et 2π, on remarquera des écarts sur certains résultats :

Résultat obtenu en Déca	Résultat de référence Java
-0.00000e+00	-8.74228e-08
0.00000e+00	1.74846e-07
0.00000e+00	8.74228e-08
2.31058e-01	2.26771e-01
1.15938e-01	1.15965e-01
8.73305e-01	8.73283e-01
-1.02977e-01	-1.03019e-01
-5.64268e-01	-5.64251e-01
6.13588e-03	8.59384e-03

Les trois premiers résultats correspondent à nos attentes, en effet, nos méthodes d'approximation ne possédant pas une précision à **0.000 000 1** ou **0.000 000 01** près, il est normal d'obtenir des résultats nulle pour des valeurs très petites, à e-07 et moins. On en déduit donc une précision d'au moins **0.000 01** près pour notre méthode Déca, pour les valeurs se situant sur l'intervalle $[0, 2\pi]$.

Les résultats grisés sont des résultats obtenus en testant la méthode sin sur des valeurs très élevé, ce qui est permis par notre méthode. On observe des erreurs pouvant être à **0.1**, **0.01**, **0.001** ou encore **0.000 1** près, ceci étant dû à l'utilisation de méthodes de conversion des radians en degrés et inversement, de degrés en radians.

Pour la méthode **cos**, malgré la correspondance de la majorité des valeurs testés se situant entre 0 et 2π, on remarquera des écarts sur certains résultats :

Résultat obtenu en Déca	Résultat de référence Java
-0.00000e+00	-4.37114e-08
0.00000e+00	1.19249e-08
-0.00000e+00	-4.37114e-08
0.00000e+00	1.19249e-08
-9.72940e-01	-9.73948e-01
9.93256e-01	9.93253e-01
-4.87173e-01	-4.87214e-01
-9.94684e-01	-9.94679e-01
-8.25591e-01	-8.25603e-01
9.99981e-01	9.99963e-01

Les trois premiers résultats correspondent à nos attentes, en effet, nos méthodes d'approximation ne possédant pas une précision à **0.000 000 01** près, il est normal d'obtenir des résultats nulle pour des valeurs très petites, à e-08 et moins. On en déduit donc une précision d'au moins **0.000 01** près pour notre méthode Déca, pour les valeurs se situant sur l'intervalle $[0, 2\pi]$. Au vu des résultats pour la méthode sin précédemment et ces résultats obtenus pour la méthode cos, on peut imaginer que la précision est d'au moins **0.000 001** près ; cela n'a cependant pas été testé.

Les résultats grisés sont des résultats obtenus en testant la méthode cos sur des valeurs très élevé (le même lot de valeurs que précédemment pour la méthode sin), ce qui est permis par notre méthode. On observe des erreurs pouvant être à **0.01**, **0.001**, **0.000 1** ou encore **0.000 01** près, ceci étant dû à l'utilisation de méthodes de conversion des radians en degrés et inversement, de degrés en radians. Ces résultats sont donc plus précis à une décimale près que les résultats de la méthode sin. Cela nous confirme donc que la précision pour les valeurs se situant sur l'intervalle $[0, 2\pi]$ est bien meilleure pour la méthode cos, comme nous pouvions nous y attendre.

Les conversions des radians en degrés et inversement, des degrés en radians, utilisés au sein de nos méthodes sin et cos représente ici une limitation au niveau de la conception. Ces conversions permettent de simplifier le passage des flottants aux entiers lors de cast par exemple. On pourrait imaginer une nouvelle version de nos

méthodes sans ces conversions : cela permettrait de gagner en précision, notamment sur des lots de valeurs élevés. Il existe d'ailleurs des méthodes "expérimentales" correspondant à cette amélioration, ces méthodes n'ont pas été finies par manque de temps ; elles pourraient cependant être reprises par un éventuel développeur désirant maintenir ou améliorer l'extension TRIGO de la bibliothèque standard.

D'autres techniques d'approximation peuvent également être utilisées, nous avons fait le choix d'utiliser l'approximation polynomiale basée sur les séries de Taylor à l'ordre 31, il aurait été possible d'obtenir une meilleure précision en optant pour un ordre plus élevé et/ou pour une méthode d'approximation différente.

À base de comparaison, d'après le *rapport de recherche n°5105 « Arithmétique flottante », par Vincent Lefèvre et Paul Zimmermann - Institut National de Recherche en Informatique et en Automatique (INRIA)*, page 12, pour la méthode exponentielle sur $[-1, 1]$: l'approximation de **Taylor** de degré 2 en $x = 0$, soit $1 + x + x^2/2$, donne une erreur d'environ **0.218** en $x = 1$, contre **0.082** pour l'approximation via les polynômes de **Legendre**, **0.050** pour ceux de **Chebyshev**, et **0.045** pour l'approximation **minimax**. On constate donc, en règle générale, que la méthode d'approximation polynomiale basée sur les séries de **Taylor** offre une moins bien meilleure précision qu'une approximation via les polynômes de **Legendre**, de **Chebyshev** ou encore par approximation **minimax**.

Dans la conception actuelle, on notera également que si la valeur, en radians, passée en paramètre aux méthodes sin et cos dépasse la valeur maximale d'un entier en Déca (qui correspond également à la valeur maximale d'un entier en Java), on obtiendra une erreur de dépassement arithmétique dû à l'utilisation de cast de ce paramètre flottant en entier. Le message indiqué par IMA est alors le suivant : **« Erreur : cast float -> int impossible. »**.

2. Références bibliographiques

1.1. Extension TRIGO

❑ Thèse « *Résolution de contraintes sur les flottants dédiée à la vérification de programmes* », par Mohammed Belaid
➔ https://tel.archives-ouvertes.fr/tel-00937667/document

❑ Article « *Faster Math Functions* », par Robin Green - Sony Computer Entertainment America (robin_green@playstation.sony.com)
➔ https://basesandframes.wordpress.com/2016/05/17/faster-math-functions/
➔ https://basesandframes.files.wordpress.com/2016/05/rgreenfastermath_gdc02.pdf

❑ E-book « *The Algebra Help* », par MathOnWeb (http://mathonweb.com)
➔ http://mathonweb.com/help_ebook/html/algorithms.htm
➔ http://mathonweb.com/help_ebook/html/functions_1.htm

❑ Projet « *The Improving Mathematics Education in Schools (TIMES) Project* », par l'AMSI (Australian Mathematical Sciences Institute)
➔ https://amsi.org.au/teacher_modules/The_trigonometry_functions.html
➔ https://amsi.org.au/teacher_modules/pdfs/The_trigonometry_functions.pdf

❑ Page internet « *Trigonometric Tables* », par The World of Math Online (http://math.com)
➔ http://www.math.com/tables/trig/tables.htm

❑ Cours « M@ths en LIgne - Développements limités », par Bernard Ycart - Université de grenoble (http://ljk.imag.fr)
➔ https://ljk.imag.fr/membres/Bernard.Ycart/mel/dl/dl.pdf
➔ https://ljk.imag.fr/membres/Bernard.Ycart/mel/dl/node6.html

❑ Rapport de recherche n°5105 « Arithmétique flottante », par Vincent Lefèvre et Paul Zimmermann - Institut National de Recherche en Informatique et en Automatique (INRIA)
➔ https://www.vinc17.net/research/papers/arithflottante.pdf

❑ Article « Extending the Range of the Cody and Waite Range Reduction Method », par Peter Kornerup et Jean-Michel Muller - University of Southern Denmark et CNRS-LIP-Arénaire
➔ http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.9012&rep=rep1&type=pdf