Projet GL 2021:
Documentation de l'extension TRIGO

Équipe n°31 : Antoine Briançon, Axel Glorvigen, Thibault Launay, Maxime Martin et Pierre Pocreau

Sommaire:

I. Introduction

L'extension TRIGO doit permettre l'utilisation de fonctions trigonométriques dans les programmes du langage Deca. Le compilateur Decac ne doit rien faire de particulier pour reconnaître ces fonctions mathématiques. Cela nécessite l'implémentation d'une classe Math dans la bibliothèque standard pour pouvoir être ajouté à n'importe quel programme avec un include. Les méthodes demandées sont sinus, cosinus, arcsinus, arctangente et la fonction ULP (Unit in the Last Place) qui retourne l'intervalle entre deux flottants représentables. Les méthodes trigonométriques doivent prendre en entrée un angle en radians. La principale problématique de cette extension réside dans le niveau de précision apporté par les fonctions trigonométriques.

Afin que la classe Math soit dans la bibliothèque standard, elle est placée dans le fichier : s*rc/main/resources/include/Math.decah*. Les méthodes évoquées précédemment seront au sein de cette classe Math. Ainsi, un simple include permet d'importer la classe et d'utiliser les fonctions trigonométriques. Par exemple :

```
#include "Math.decah"
{
        Math m = new Math();
        println("cos(0.0) = ", m.cos(0.0));
}
```

Afin de vérifier que la classe Math fonctionne bien, il a fallu mettre en place un ensemble de tests. Ces tests ont été construits de manière à être valables pour n'importe quelle classe Math. Inversement, la classe Math a été implémentée pour être valable pour n'importe quel compilateur Decac. La majeure partie du travail a résidé dans l'élaboration d'algorithmes pour les fonctions trigonométriques qui évitent ou du moins limitent l'approximation du résultat flottant par rapport à la valeur réelle. Il y a plusieurs sources d'erreurs. En effet, à chaque calcul d'un flottant nous pouvons obtenir une approximation de la valeur réelle et donc une perte d'information. A ces erreurs, il faut également ajouter l'incertitude du au calcul fini de séries mathématiques infinies. De plus, pour avoir des flottants très proches de la valeur réelle, il a fallu prendre en compte la densité de répartition des flottants qui est très variable. Elle est par exemple très forte autour de 0.

II. Implémentation de la classe Math

L'ensemble des méthodes deca nécessaires à la mise en place de l'extension TRIGO a été implémenté dans la classe Math. On retrouve dans celle-ci les cinq méthodes demandées pour l'extension, à savoir ulp, sin, cos, asin et atan. De plus, ces fonctions font appel à des méthodes intermédiaires pendant leur exécution pour réaliser certains de leurs calculs. Cependant, pour éviter des conflits d'identification de méthode avec des fonctions d'un utilisateur utilisant la bibliothèque Math, le nom des méthodes intermédiaires est précédé du caractère " _". Tout comme les fonctions trigonométriques, les méthodes intermédiaires sont présentes dans la classe Math et sont _abs, _modulo2PI, _sqrt, _power2 et _getExponent. La classe Math comporte donc un total de dix méthodes dont seulement cinq accessibles par l'utilisateur de la bibliothèque. A cela s'ajoutent trois variables globales accessibles par toutes les méthodes de la classe, à savoir PI, PI_2 et PI_4 qui sont respectivement des approximations en flottant de π, π/2 et π/4.

Afin d'optimiser les méthodes deca de la classe Math, il a été décidé lors de sa conception de factoriser le plus possible le code. C'est pour cette raison qu'on retrouve autant de méthodes intermédiaires. Pour certaines, elles ne sont appelées que par une seule méthode rendant la factorisation du code moins pertinente. Cependant, cette implémentation permet ainsi théoriquement d'enrichir la classe Math plus rapidement en laissant la possibilité aux futures méthodes d'utiliser les méthodes intermédiaires.

A. Valeur absolue

La méthode implémentée la plus basique est *float _abs (float a)* qui prend en entrée un flottant *a* quelconque et retourne sa valeur absolue sous la forme d'un flottant également. Si a est strictement inférieur à 0 alors la méthode retourne *-a*, sinon elle retourne *a*.

C. Modulo 2π

La méthode float *_modulo2PI (float a)* prend en entrée un flottant *a* et retourne son modulo 2π sous forme de flottant avec un décalage de -π. Cette méthode ne retourne pas un flottant entre 0 et 2π comme on pourrait s'y attendre à partir de son nom, mais en réalité le flottant retourné est compris entre -π et π. L'intérêt de retourner un flottant dans cet intervalle est de minimiser l'erreur dû à l'arrondi de la valeur trouvée au flottant le plus proche. En effet, les flottants ne sont pas répartis de manière uniforme dans l'ensemble des nombres qu'ils représentent et la densité est très forte autour de 0, plus qu'autour de π, d'où l'intérêt de décaler

le modulo 2π de -π. Ainsi, si *a* est strictement inférieur à -π, on lui ajoute 2π jusqu'à ce qu'il entre dans l'intervalle [-π, π] et on le retourne. A l'inverse, si *a* est strictement supérieur à π, on lui soustrait 2π jusqu'à ce qu'il entre dans l'intervalle [-π, π] et on le retourne. Sinon, *a* est déjà dans l'intervalle, donc il est retourné directement.

D. Racine carrée

La méthode *float _sqrt (float a)* prend en entrée un flottant *a* et retourne la racine carrée de celui-ci sous forme de flottant. La racine carrée obtenue est le plus souvent la valeur exacte arrondie la plus proche mais son calcul peut également donner une valeur approximative notamment avec de grands nombres. Cette méthode est tout de même très efficace puisqu'elle utilise la méthode de Héron pour approcher la valeur exacte grâce à une suite qui converge rapidement. La méthode de Héron est initialement définie pour approximer la racine carrée d'un entier mais elle est totalement adaptable avec l'utilisation de nombres flottants. Si *a* est inférieur ou égal à 0 alors la méthode retourne 0. Sinon elle boucle sur la suite $u_{n+1} = (u_n + a / u_n) / 2$ avec $U_0 = a$ en calculant successivement tous les termes jusqu'à ce que $u_{n+1} = u_n$, la méthode retourne ensuite *Un* qui correspond à l'approximation de la racine carrée de *a*. La boucle a une limite d'une trentaine d'itérations pour éviter exécution trop longue. Dans ce cas, u_n est quand même retourné et tend tout de même vers la racine carrée de *a* même si u_{n+1} est différent de u_n. Cependant, le seuil d'arrêt de la boucle est fixé de sorte à ce que ce dernier cas de figure n'arrive pas. Même avec les plus grands flottants possibles, la suite converge avant l'arrêt de la boucle.

E. 2 puissance n

La méthode *float _power2 (int pow)* prend entrée un entier *pow* et retourne le résultat de 2 puissance *pow* sous la forme d'un flottant. Si *pow* est strictement supérieur à 0, la méthode boucle *pow*-fois en multipliant à chaque boucle la valeur à retourner par 2 et donc jusqu'à atteindre 2 puissance *pow*. A l'inverse, si *pow* est strictement inférieur à 0, la méthode boucle *pow*-fois en divisant à chaque boucle la valeur à retourner par 2 et donc jusqu'à atteindre 2 puissance *pow*. Si *pow* est égal à 0, la méthode retourne 1.0 directement.

F. Exposant non-biaisé d'un flottant

La méthode *float getExponent (float f)* prend en entrée un flottant *f* et retourne uniquement sa partie exposant sous forme de flottant. La partie exposant retournée est non-biaisée ce qui demande une modification de la partie exposant récupérée sur un flottant. En effet, l'exposant peut être positif ou négatif mais la représentation des nombres signés par un complément à 2, comme fait habituellement, rend la

comparaison entre les nombres flottants beaucoup plus difficile. Ainsi, dans la norme IEEE 754, l'exposant est volontairement biaisé pour être stocké sous forme d'un nombre non signé. Ce biais est de $2^{e-1} - 1$, avec e = 8 pour notre représentation des flottants 32 bits, et donc $2^7 - 1$.

G. Unit in the Last Place

La méthode *float ulp (float f)* prend en entrée un flottant *f* et retourne l'Unit in the Last Place de *f* sous la forme d'un flottant. L'ULP d'un nombre permet de mesurer la précision absolue d'un nombre flottant, elle correspond à la distance qui le sépare du flottant qui lui succède directement. Les flottants n'étant pas répartis de manière uniforme dans l'ensemble des nombres représentables par des flottants 32 bits, la valeur de l'ULP varie directement en fonction du flottant pour lequel on souhaite calculer son Unit in the Last Place. Plus la répartition des flottants est dense localement, plus la distance qui les sépare successivement est courte et donc moins l'ULP est élevée. L'ULP n'est donc pas fonction des parties signe et mantisse d'un nombre flottant mais uniquement de sa partie exposant codée sur 8 bits et de la précision relative de la représentation des flottants, à savoir 2 puissance -23. La méthode *ulp* retourne directement la multiplication entre la précision relative et 2 à la puissance de la partie de l'exposant non-biaisé du flottant pour lequel on calcul l'ULP. Pour cette fonction, on utilise les méthodes intermédiaires *_power2* et *_getExponent*.

H. Sinus

La méthode *float sin(float a)* prend en entrée un flottant *a* correspondant à un angle en radians et retourne une approximation du sinus de cet angle, également sous la forme d'un flottant. Dès le début de la méthode, l'angle *a* est ramené à une valeur comprise entre -π et π grâce à l'appel de la méthode *_modulo2PI* qui utilise un modulo 2π décalé de -π. Ensuite, le signe de l'argument *a* est conservé par une variable valant 1 ou -1 en testant si *a* est inférieur ou non à 0. Puis, *a* est remplacé par sa valeur absolue grâce à la méthode *_abs*. L'objectif de ces transformations est d'effectuer par la suite des calculs uniquement avec un angle compris entre 0 et π/4 et d'augmenter ainsi la précision du résultat. En effet, l'algorithme de cette méthode trouve une valeur approximative de sinus de a grâce à un développement en série de Taylor de la fonction sinus avec les sept premiers termes et la valeur absolue de *a*. Cependant, cette approximation est très précise pour un angle d'une valeur absolue comprise entre 0 et π/4 mais beaucoup moins pour un angle plus grand.

Ainsi, la méthode calcul directement le développement en série de sinus lorsque la valeur absolue de l'angle est comprise entre 0 et π/4, puis, une fois le

résultat obtenu, il est multiplié par la variable ayant enregistrée le signe de *a* et retourné.

Si l'angle est compris entre π/4 et π/2 alors c'est la fonction cosinus (*cos*) qui est appelée afin de se retrouver avec un angle compris entre 0 et π/4. Pour cela, on se base sur la formule sin(x) = cos(π/2 - x). La méthode appelle donc *this.cos(this.PI_2 - a)* avec *this.PI_2* la variable globale correspondant au flottant représentant π/2. Une fois le résultat obtenu, il est multiplié par la variable ayant enregistrée le signe de *a* et retourné.

Sinon, l'angle est donc supérieur ou égal à π/2 et inférieur ou égal à π et la fonction appelle la méthode cosinus en se basant sur la formule sin(x) = cos(x - π/2) lorsque x ≥ 0, c'est-à-dire en deca *this.cos(a - this.PI_2).* La fonction cosinus se retrouve donc avec un angle compris entre 0 et π/4 permettant un calcul plus précis. Une fois le résultat obtenu, il est multiplié par la variable ayant enregistrée le signe de *a* et retourné.

I. Cosinus

La méthode *float cos(float a)* prend en entrée un flottant *a* correspondant à un angle en radians et retourne une approximation du cosinus de cet angle, également sous la forme d'un flottant. La méthode cosinus a un fonctionnement similaire à celle de sinus (*sin*). Dès le début de la méthode, l'angle *a* est ramené à une valeur comprise entre -π et π grâce à l'appel de la méthode *_modulo2PI* qui utilise un modulo 2π décalé de -π. Ensuite, le signe du futur résultat est calculé à partir de l'argument *a*. Si a est inférieur à -π/2 ou supérieur à π/2 alors une variable conserve la valeur -1 sinon elle conserve 1. Puis, *a* est remplacé par sa valeur absolue grâce à la méthode *_abs*. Comme pour la méthode sinus, l'objectif de ces transformations est d'effectuer par la suite des calculs uniquement avec un angle compris entre 0 et π/4 et d'augmenter ainsi la précision du résultat. En effet, l'algorithme de cette méthode trouve une valeur approximative de cosinus de a grâce à un développement en série de Taylor de la fonction cosinus avec les huit premiers termes et la valeur absolue de *a*. Cependant, cette approximation est très précise pour un angle d'une valeur absolue comprise entre 0 et π/4 mais beaucoup moins pour un angle plus grand.

Ainsi, la méthode calcul directement le développement en série de cosinus lorsque la valeur absolue de l'angle est comprise entre 0 et π/4, puis, une fois le résultat obtenu, il est multiplié par la variable ayant enregistrée le signe du résultat et retourné.

Si l'angle est compris entre π/4 et π/2 alors c'est la fonction sinus (*sin*) qui est appelée afin de se retrouver avec un angle compris entre 0 et π/4. Pour cela, on se

base sur la formule cos(x) = sin(π/2 - x). La méthode appelle donc *this.sin(this.PI_2 - a)* avec *this.PI_2* la variable globale correspondant au flottant représentant π/2. Une fois le résultat obtenu, il est multiplié par la variable ayant enregistrée le signe du résultat et retourné.

Sinon, l'angle est donc supérieur ou égal à π/2 et inférieur ou égal à π et la fonction appelle la méthode sinus en se basant sur la formule cos(x) = sin(x - π/2) lorsque π/2 ≤ x ≤ π, c'est-à-dire en deca *this.sin(a - this.PI_2)*. La fonction sinus se retrouve donc avec un angle compris entre 0 et π/4 permettant un calcul plus précis. Une fois le résultat obtenu, il est multiplié par la variable ayant enregistrée le signe du résultat et retourné.

J. Appels mutuels de sinus et cosinus

Pour résumer, les deux méthodes sinus et cosinus manipulent l'argument de pris en entrée pour le convertir en un angle compris entre 0 et π. Si l'angle est inférieur à π/4 alors une approximation de la valeur demandée est directement calculée par un développement en série pour un nombre finis de termes. Sinon, la méthode appelle l'autre pour lui passer un argument inférieur à π/4 grâce à des formules trigonométriques. Cela permet de ne calculer des développement limités que pour des arguments inférieur à π/4 et donc d'obtenir un résultat le plus précis possible. En outre, pour réduire les temps de calculs et les approximations intermédiaires sur des flottants, les coefficients des développements limités des deux méthodes ont été calculés au préalable et enregistrés dans des flottants directement initialisés au début de chacune des méthodes.

K. Arcsinus

La méthode *float asin (float a)* prend en entrée un flottant *a* et retourne une approximation de l'arcsinus de *a*, également sous la forme d'un flottant. L'entrée doit être comprise entre -1 et 1 sinon la méthode retourne directement 0. Si ce prérequis est respecté, le signe *a* est conservé par une variable valant 1 ou -1 en testant si *a* est inférieur ou non à 0. Ensuite, a est remplacé par sa valeur absolue avec la méthode *_abs.* On se retrouve donc avec une variable comprise entre 0 et 1. Si la valeur absolue de *a* est strictement inférieur à *0.5* alors la méthode calcule le développement limité de la fonction mathématique arcsinus avec ses quatre premiers termes et la valeur absolue de *a*. Une fois le résultat obtenu, il est multiplié par la variable ayant enregistrée le signe de *a* et retourné. Sinon, la valeur de a est comprise entre 0.5 et 1 et la méthode calcul une approximation avec une série ainsi que π/2 et une racine carrée, *this.PI_2 - this._sqrt(1 - a) * (A1 + a * A2 + square * A3 + cube * A4)* en deca avec *this.PI_2* la variable globale correspondant au flottant représentant π/2 et A1, A2, A3, et A4 les coefficients sous forme de flottants de la

suite. Une fois le résultat obtenu, il est multiplié par la variable ayant enregistrée le signe de *a* et retourné. Pour les deux calculs d'approximation, les coefficients des séries ont été calculés au préalable et enregistrés dans des flottants directement initialisés au début de la méthode.

L. Arctangente

La méthode *float atan (float a)* prend en entrée un flottant *a* et retourne une approximation de l'arctangente de *a*, également sous la forme d'un flottant. La méthode est relativement simple et utilise deux calculs d'approximation en fonction de la valeur de *a*. Si *a* est compris entre -1 et 1, l'approximation est obtenue en calculant $\pi/4a + 0.273a\ (1-|a|)$. Sinon l'approximation est obtenue en calculant 2arctan(a / (1 + sqrt(1 + a ²))) avec sqrt le calcul de la racine carrée. Dans ce deuxième cas, la méthode utilise un algorithme récursif. Une fois l'un des calculs effectué, le résultat est directement retourné.

III. Validation des algorithmes

Pour valider les différents algorithmes on utilise trois sortes de méthodes:

A.Script

Pour lancer le script de test de l'extension, il faut se placer dans le répertoire src/test/script et lancer la commande ./test_trigo.sh. Cette dernière va d'abord lancer quelques calculs basiques prédéfinis dans des programmes .deca (dans le répertoire src/test/deca/trigo). Le script continue son exécution en lançant l'un après l'autre 4 autres programmes .deca destinés à la comparaison avec des valeurs attendues. Chaque programme fait plusieurs appels à une des fonctions principales (sinus, cosinus, asin et atan) pour un ensemble de valeurs fixes.

Dans cette deuxième partie du script, on fait une comparaison entre les résultats obtenus et les résultats donnés par les fonctions contenues dans la librairie Math de Java, préalablement écrits dans des fichiers .txt localisés au même endroit que les programmes .deca. Un simple appel à la commande diff permet de faire cette comparaison et d'afficher les différences directement sur le terminal.

Ce script est surtout utile lors du développement et de l'optimisation des fonctions trigonométriques. Il permet d'avoir rapidement un aperçu de la précision des 4 fonctions.

Actuellement il est très utile pour évaluer les fonctions sinus et cosinus. Malheureusement, notre compilateur provoquant encore quelques erreurs dans le cadre de l'extension, nous n'avons pas de résultats sur les méthodes asin et atan.

Ci-dessous un extrait de la sortie du script à son exécution:

```
cos:
1,6c1,6
< 1.0
< 0.5403023058681398
< -0.8390715290764524
< 0.8623188722876839
< 0.562379076290703
< -0.9521553682590148
---
> 1.00000e+00
> 5.40302e-01
> -8.30048e-01
> 8.55752e-01
> 5.63142e-01
> -9.59370e-01
```

Dans la première moitié, on affiche les valeurs écrites préalablement dans le fichier .txt relatif à la méthode cos. Ce sont les résultats sortis par la librairie Math de Java.

La deuxième moitié montre les résultats de notre fonction cosinus, il suffit alors de comparer les valeurs deux à deux. Ici on peut voir qu'on a une précision plutôt satisfaisante pour la plupart des valeurs.

B.Classes Java

Localisation: src/test/java/fr/ensimag/trigo

Les tests passant par une classe Java nous permettent d'exploiter la librairie Math de Java dynamiquement. On peut donc directement comparer nos fonctions avec celles de la librairie qu'on veut approcher. Pour cela, on copie directement nos fonctions deca dans la classe Java qui les teste.

TestUlp:

Cette classe permet de visualiser les différences entre les méthodes getExponent, power2 et ulp de notre extension avec celles de la librairie Math de Java (puisque la méthode ulp dépend du bon fonctionnement des deux autres). Pour ce faire, on calcule le résultat de notre méthode et celui de la méthode de la librairie et on les affiche pour que l'utilisateur puisse se rendre compte des plages de valeurs pour lesquelles la méthode donne un résultat satisfaisant et celles pour lesquelles le résultat est trop éloigné (pour getExponent et ulp on teste environ 100000 valeurs différentes).

Après avoir exécuté la classe, on peut voir que pour la grande majorité des valeurs en entrée, getExponent, power2 et ulp renvoient des valeurs très satisfaisantes. Ci-dessous un extrait de la sortie de la classe après exécution lors du test de la méthode ulp pour une plage de valeurs prise au hasard.

```
9592.748: 9.765625E-4 | 9.765625E-4
9592.848: 9.765625E-4 | 9.765625E-4
9592.947: 9.765625E-4 | 9.765625E-4
9593.047: 9.765625E-4 | 9.765625E-4
9593.146: 9.765625E-4 | 9.765625E-4
9593.246: 9.765625E-4 | 9.765625E-4
9593.346: 9.765625E-4 | 9.765625E-4
9593.445: 9.765625E-4 | 9.765625E-4
9593.545: 9.765625E-4 | 9.765625E-4
9593.645: 9.765625E-4 | 9.765625E-4
9593.744: 9.765625E-4 | 9.765625E-4
9593.844: 9.765625E-4 | 9.765625E-4
9593.943: 9.765625E-4 | 9.765625E-4
9594.043: 9.765625E-4 | 9.765625E-4
9594.143: 9.765625E-4 | 9.765625E-4
9594.242: 9.765625E-4 | 9.765625E-4
9594.342: 9.765625E-4 | 9.765625E-4
9594.441: 9.765625E-4 | 9.765625E-4
```

Valeur en entrée: sortie de notre méthode | sortie de la méthode Math.java

On peut donc attester de la validité de notre fonction ulp qui sera utilisée pour les futurs tests.

TestTrigo:

Cette classe, à l'exécution, affiche pour un nombre de valeurs fixe le résultat renvoyé par chaque fonction (sinus, cosinus, asin et atan) de notre extension ainsi que celui renvoyé par les fonctions de la librairie Math. C'est un premier test basique qui permet de voir si les résultats de nos fonctions, même si elles ne sont pas parfaitement précises, sont cohérents pour des valeurs clé.

Ci-dessous un extrait de la sortie de la classe après exécution:

```
----- TEST SINUS -----
sin(0) = 0.0 (0.0)
sin(0.5) = 0.47942555 (0.479425538604203)
sin(1) = 0.8327205 (0.8414709848078965)
sin(2) = 0.9064868 (0.9092974268256817)
sin(π/4) = 0.70710677 (0.7071067966408575)
sin(π/2) = 1.0 (0.999999999999999)
sin(π) = 0.0 (-8.742278000372475E-8)
sin(-π/2) = -1.0 (-0.999999999999999)
sin(-π) = -0.0 (8.742278000372475E-8)
sin(-30.5π) = -1.0 (-0.9999999999889881)
```

Avec entre parenthèses la sortie de la librairie Math. Dans l'ensemble les fonctions retournent des valeurs très proches de celles attendues.

TestAuto:

C'est la classe qui regroupe les tests les plus complets puisqu'elle teste un grand nombre de valeurs pour chaque fonction. De plus, elle fournit des informations supplémentaires sur les différences entre les résultats de nos fonctions et ceux attendus en les estimant en nombre d'ULP.

Une démonstration vaut mieux qu'une longue description, ci-dessous la sortie après exécution de cette classe:

```
Tests sinus:
    Moyenne d'erreur = 0.009464272
    Diff. max = 0.049216986 atteinte pour f = 9997.361 soit 825724.0 ulp
    Diff. min = 0.0 atteinte pour f = 0.0 soit NaN ulp

Tests cosinus
    Moyenne d'erreur = 0.009464634
    Diff. max = 0.04909748 atteinte pour f = 9976.941 soit 823719.0 ulp
    Diff. min = 0.0 atteinte pour f = 0.0 soit 0.0 ulp

Tests asin:
    Moyenne d'erreur = NaN
    Diff. max = 4.6133995E-5 atteinte pour f = 0.5 soit 774.0 ulp
    Diff. min = 0.0 atteinte pour f = 0.0 soit NaN ulp

Tests atan:
    Moyenne d'erreur = 2.1863862E-5
    Diff. max = 0.0075204372 atteinte pour f = 2.2 soit 63086.0 ulp
    Diff. min = 0.0 atteinte pour f = 0.0 soit NaN ulp
```

On peut ainsi savoir quelles sont les valeurs pour lesquelles les résultats sont les plus éloignés de ceux attendus. Ce test permet d'avoir un bon aperçu des éventuels problèmes de précision sur une grande plage de valeur.

C.Traçage de courbes

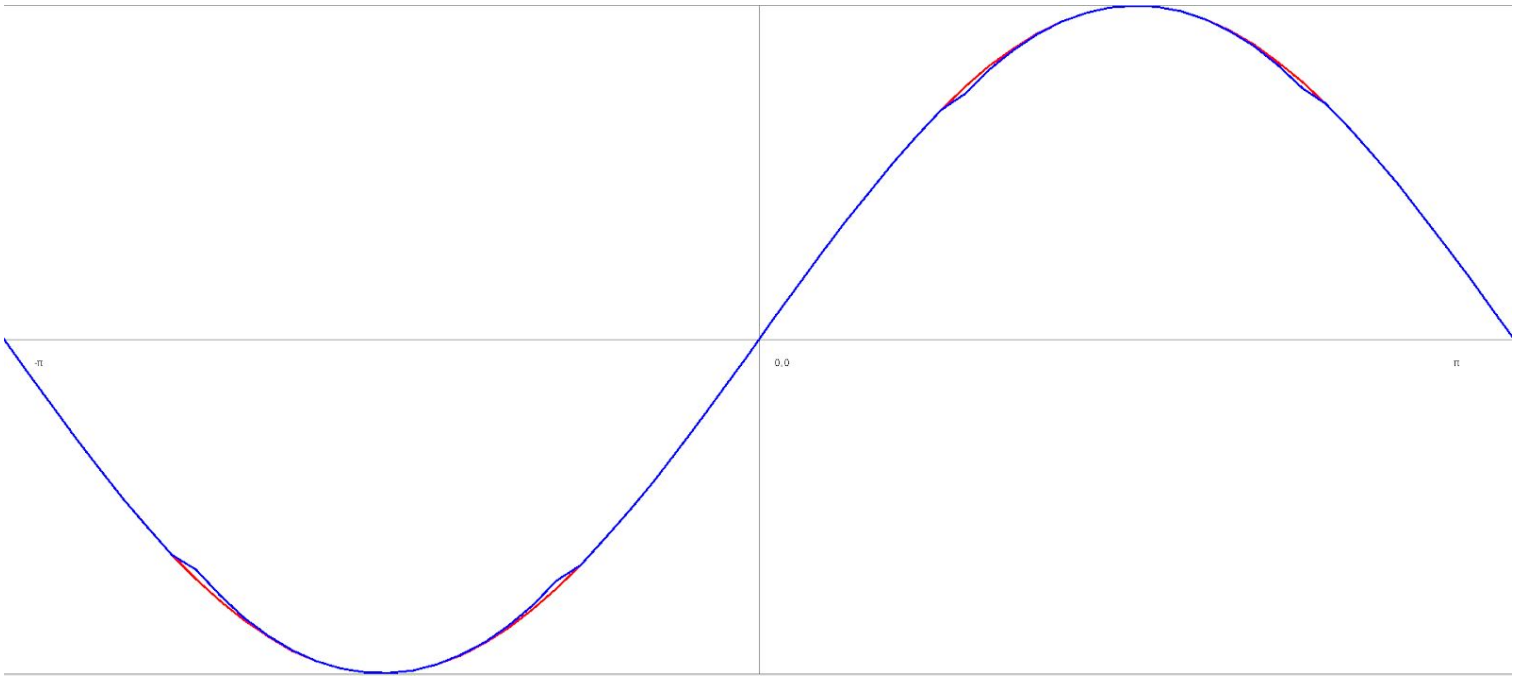
Localisation: src/test/java/fr/ensimag/trigo

Une dernière méthode de validation consiste à tracer les courbes de chacune des fonctions trigonométriques. Pour chaque fonction, on trace sur le même graphe la courbe de nos propres fonctions deca et celles de la librairie Math de Java.

Comme pour les tests dans les classes Java, on doit passer par une classe à part entière pour pouvoir appeler les méthodes de la librairie Java. On copie donc également nos fonctions pour pouvoir les exploiter.

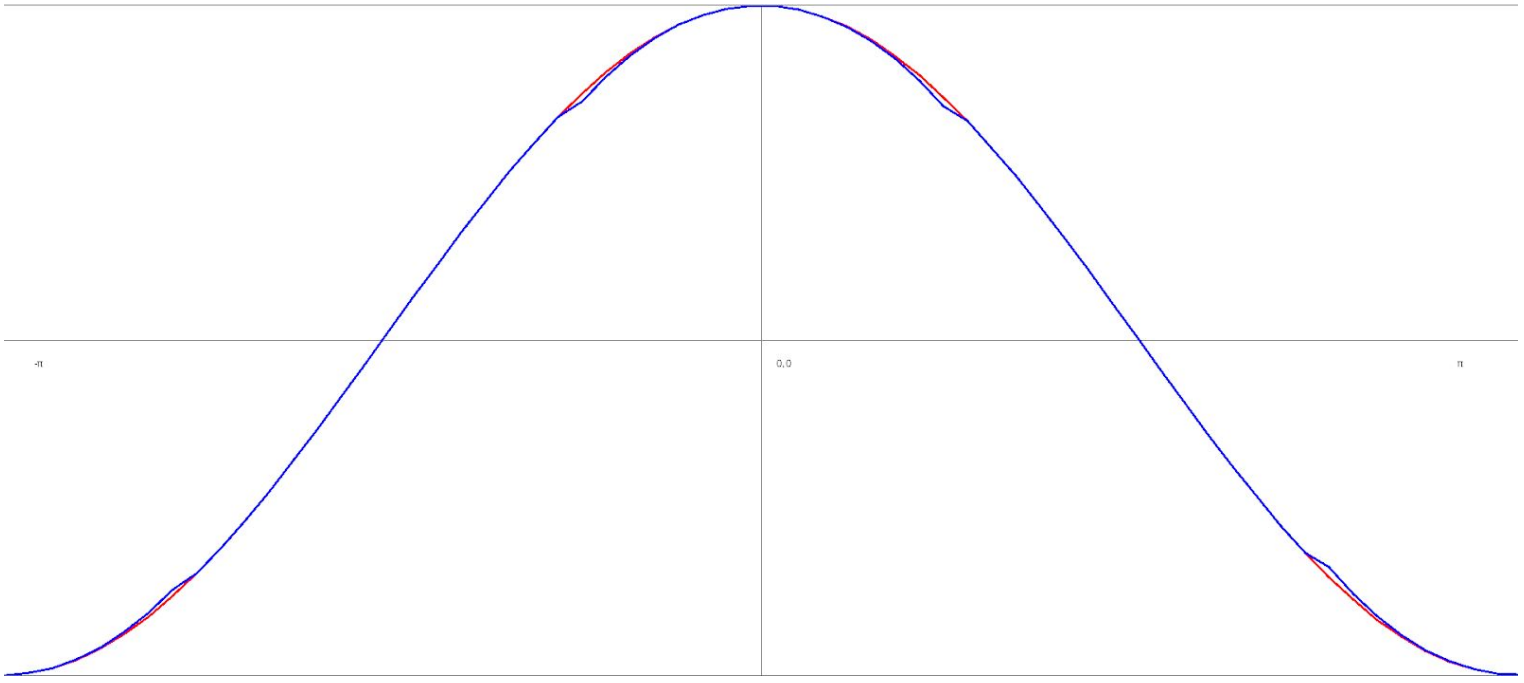
Pour tracer la courbe de la fonction que l'on souhaite, il faut modifier les fichiers CourbeCos.java (pour sinus et cosinus) et CourbeArc.java (pour asin et atan). Pour lancer le tracé il faut lancer CurveTracer.java (pour sinus et cosinus) et CurveTracer2.java sinon.

Ci-dessous les courbes obtenues, avec en bleu les fonctions de notre extension et en rouge les fonctions de la librairie Java:



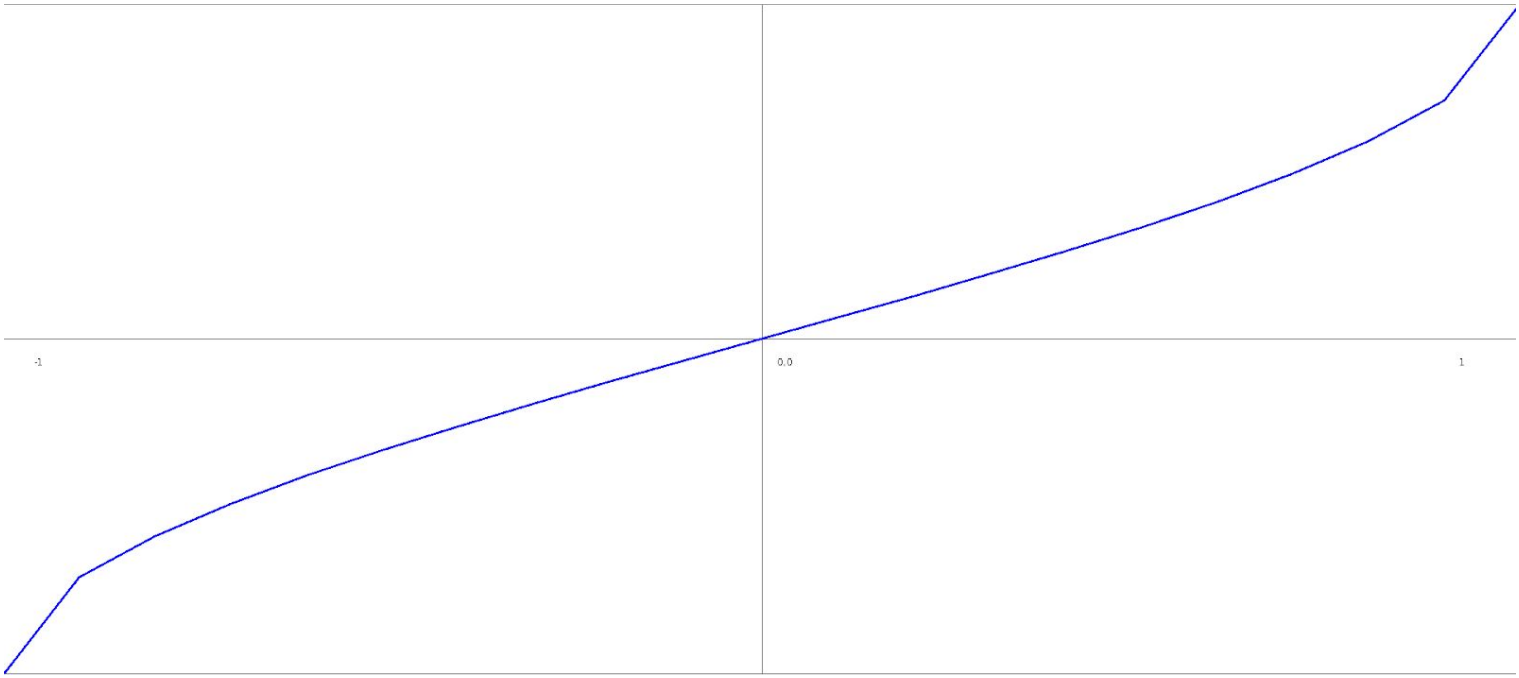
Tracé de la fonction sinus entre -π et π

On peut voir que notre fonction sinus est assez proche de celle attendue, malgré un petit écart autour des valeurs π/3, -π/3, 2π/3 et -2π/3.



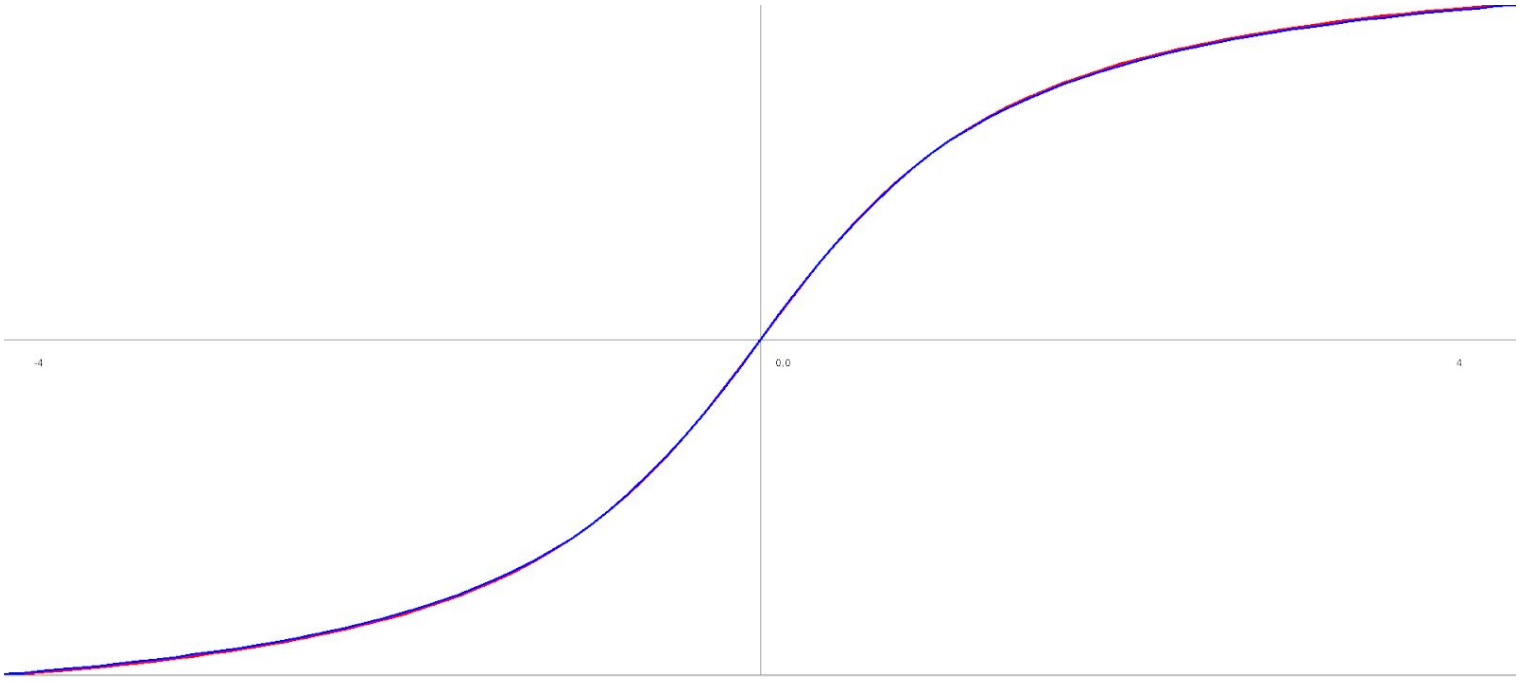
Tracé de la fonction cosinus entre -π et π

On peut voir que notre fonction sinus est assez proche de celle attendue, malgré un petit écart autour des valeurs π/3, -π/3, 2π/3 et -2π/3 (comme pour la fonction sinus mais sachant que l'une dépend de l'autre ce n'est pas étonnant).



Tracé de la fonction asin entre -1 et 1

On peut voir que le tracé de notre fonction asin est exactement celui de la fonction asin contenue dans la librairie Java. Évidemment, un tel tracé n'a pas une précision suffisante pour qu'on puisse conclure que notre fonction asin est parfaite, mais c'est un bon aperçu.



Tracé de la fonction atan entre -4 et 4

On peut voir que plus on s’éloigne de 0, plus on perd en précision par rapport à la fonction de la librairie Java. La différence n’est pas très grande mais il est intéressant de le noter.

IV. Limites et améliorations

La classe Math implémentée pour ce projet répond aux exigences et fonctionne correctement. Les fonctions trigonométriques retournent des valeurs approximatives très proches des valeurs exactes pour une grande majorité des angles passés en entrée. Cependant, cette implémentation n'est pas parfaite et présente certaines limites. La classe Math pourrait être améliorée à plusieurs niveaux.

A. Code assembleur

L'ensemble des méthodes de la classe Math a été implémenté en deca. Certains calculs ou méthodes auraient pu être codés directement en assembleur pour gagner du temps lors de la compilation du programme mais également pour optimiser les instructions exécutés par la suite en utilisant *FMA* par exemple. En effet, *FMA* fait en une seule instruction ce que fait une instruction une instruction *MUL* suivie d'une instruction *ADD* lors d'un calcul de la forme *a + x * b*. Cette amélioration pourrait donc réduire les temps d'exécutions ainsi que l'impact énergétique de la classe Math. De plus, cette amélioration apporterait davantage de précision pour les méthodes trigonométriques puisqu'il y aurait moins d'approximation dû aux flottants intermédiaires lors des calculs.

B. Méthode modulo

La méthode modulo (*_modulo2PI*) fonctionne très bien et donne des résultats très précis à l'exception des très grands nombres. Premièrement, la manière dont est construite la méthode, fait que l'exécution peut être assez longue pour les très grands nombres. En effet, soustraire 2π à un nombre de l'ordre des dizaines de millions implique tout de même des millions de soustractions, rendant l'algorithme peut optimisé. Il aurait été pertinent de réfléchir à un algorithme plus efficace pour les grands nombres. Deuxièmement, plus le flottant est grand positivement ou négativement, plus l'ULP est important et donc moins la précision est bonne. Au bout d'un certain ordre de grandeur, l'ULP est supérieur à 2π rendant les calculs de moins en moins pertinent car il y a une grosse perte de précision au point de perdre toute pertinence dans le résultat. C'est là une limite de la classe Math, puisque toutes les méthodes trigonométriques utilisent la méthode modulo, ainsi pour de très grands nombres les résultats ne sont pas exploitables. Une des solutions pour résoudre ce problème serait de faire des calculs sur des flottants avec plus de chiffres significatifs en codant le flottant reçu en entrée de la méthode sur plusieurs flottants.

C. Gestion des préconditions

La gestion des préconditions des méthodes de la classe Math a été faite de manière minimaliste. Par exemple, pour la méthode de la racine carrée (*_sqrt*), si la valeur en entrée est inférieure à 0 alors la méthode s'arrête et retourne 0. Dans l'implémentation actuelle de la classe Math, les flottants envoyés à la méthodes sont tous forcément supérieurs ou égaux à 0 donc le cas de figure de la racine d'un nombre négatif n'est pas possible. Cependant, lors de l'envoi d'un nombre négatif à la méthode racine carrée, il serait préférable de retourner une exception pour indiquer à l'utilisateur que le nombre est invalide et doit être supérieur ou égale à 0 plutôt que de retourner simplement 0. Cela serait particulièrement pertinent si la classe Math a vocation à être améliorée par d'autres développeurs par la suite.

De même que pour la fonction racine carrée, la méthode arcsinus (*asin*) a des prérequis obligatoires sur le flottant récupéré en entrée. En effet, la fonction mathématique arcsin(x) est définie uniquement pour $x \in [-1, 1]$. Actuellement, la méthode *asin* retourne directement 0 si le flottant passé en entrée est à l'extérieur de l'intervalle [-1, 1]. Il serait beaucoup pertinent de retourner une exception indiquant à l'utilisateur que le nombre envoyé à la méthode *asin* est en dehors du domaine de définition et qu'il doit être compris entre -1 et 1. Cette exception n'a pas été implémentée pour laisser davantage de temps à la conception des algorithmes mais il aurait été préférable de la mettre en place.

V. Références bibliographiques

La conception des algorithmes d'approximation des fonctions trigonométriques a nécessité une période non négligeable de recherche d'information et d'analyse. Internet a été mis à contribution pour ces recherches. Il a notamment permis un approfondissement de la compréhension de la représentation des flottants 32 bits et la découverte de plusieurs formules d'approximation des fonctions trigonométriques plus ou moins précises. Ci-dessous, les références bibliographiques des ressources utilisées :

Représentation des flottants :
Arithmétique flottante - Inria
Java's new math : Floating-point numbers - IBM
Codage des nombres - desaintar.free.fr
Les calculs sur les flottants - Grenoble INP Ensimag

Calcul de l'Unit in the Last Place :
Unit in the last place - Wikipedia
What is Unit in the Last Place? - Programming Words
src/share/classes/java/lang/Math.java - Java.net
IEEE 754 - Wikipédia

Approximation de la racine carrée par une suite :
La méthode de Héron - amcac.net

Approximation d'arcsinus :
Looking for an arcsin algorithm - Stack Exchange

Approximation d'arctangente sur l'intervalle [-1, 1] :
Efficient Approximation Arctg Function - Université de Montréal