CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

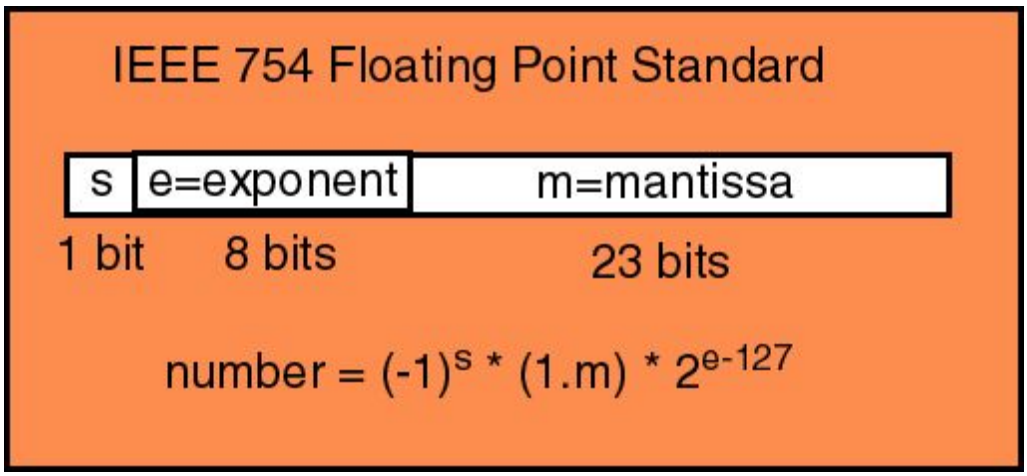
Documentation de l'extension MATH

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Objectifs

La classe MATH cherche à calculer quelques fonctions trigonométriques de flottants. La difficulté réside dans la précision des calculs. L'objectif est de se rapprocher des résultats jusqu'à atteindre la précision maximale que permet la représentation des flottants IEEE 754 simple précision.

Quesque la représentation IEEE 754 simple précision ?



Cette représentation des nombres à virgules flottantes utilisent 32 bits. Le premier bit renseigne le signe du flottant. Les 8 bits suivants représente l'exposant de l'écriture binaire scientifique du flottant. Enfin les 23 derniers représente les chiffres après la virgule de cette même représentation. Le plus grand nombre représentable est donc de l'ordre de 100 milliard de milliard de milliard de milliard. En revanche les 23 bits de mantisses sont très limitant dans ce cas là. En effet la différence entre deux nombres consécutifs de cet ordre est de l'ordre de 100 mille milliard de milliard de milliard, puisque le plus petit bit qu'on peut modifier ajoute 2^{127-23}. C'est ainsi que la précision que nous essayons d'atteindre en x est de ulp(x).

Quesqu'un ulp

L'ulp, " Unit of Least Precision ", est la précision maximale qui peut être atteinte pour un nombre donné. Il correspond à la différence entre deux floats consécutifs aux alentours de ce nombre. Les limites de précision sont dûs aux 23 seuls bits destinés à la mantisse qui impose naturellement la définition suivante :

$ulp(x) = 2^{e(x)} \times 2^{-23}$ où $e(x)$ est l'exposant de la représentation scientifique binaire

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Exemple:

$ulp(\pi) = 2^{-22} \approx 2,4 \times 10^{-7}$

en effet, $\pi = 1.5707963705062866 \times 2^{1}$ et $2^{1}2^{-23} = 2^{-22}$

Ainsi la meilleure représentation de π en IEEE 754 simple précision est $3,1415927$

Cas particulier

$ulp(0) = 2^{-149}$, car le plus petit nombre représentable est celui d'exposant décalé 0 et de mantisse 000 0000 0000 0000 0000 0001.

Conséquence importante

Une conséquence importante de la définition de l'ulp est que la précision à atteindre est d'autant plus petite que le nombre lui même est petit. Une attention particulière a donc été portée aux zéros des différentes fonctions à développer.

L'importance de raisonner en binaire

Il était important de raisonner en binaire durant tout le développement de cette extension car la représentation elle même est en binaire et cette approche permet souvent de comprendre des détails qui seraient restés incompris.
C'est ainsi que $1,1 \oplus 1,1 \neq 2,2$
où $\oplus$ est l'addition faite dans l'ensemble des flottants IEEE 754. En effet la représentation IEEE 754 la plus proche de 1,1 est :

$0\ \ 01111111\ \ 00011001100110011001101$ soit donc $\quad 1.100000023841858 \times 2^{0}$

Le groupe GL 35 peut donc désormais se vanter d'être composé de bons élèves de l'Ensimag ayant compris cette mauvaise égalité.
Mieux encore, nous savons en revanche que :

$1,25 \oplus 1,25 = 2,5$ car c'est deux nombres sont une parfaites somme de puissance de deux.

Les opérations utilisant les puissances de deux ont donc été avantagées dans tous nos programmes.

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Développement en série et entière et algorithme cordic

Conception de la classe

Aucune classe particulière n'a été implémentée pour cette partie. Nous avons en effet essayé dans un premier temps d'implémenter des algorithmes simples, évidents et approchant les fonctions trigonométriques suivantes :

- sinus
- cosinus
- arcsin
- arctan

Algorithmes utilisés

Développement en série entière

Les quatres fonctions trigonométriques à implémenter sont développables en série entière. Le premier choix d'algorithme s'est donc naturellement porté pour le développement en série entière.

Cosinus

Le cosinus est développable en série entière sur R, soit donc sur la totalité de son domaine de définition et :

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

Sinus

Le sinus est développable en série entière sur R, soit donc sur la totalité de son domaine de définition et :

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

Arc sinus

Arc sinus est développable en série entière sur $]-1,+1[$, soit donc sur la quasi-totalité de son domaine de définition qui est $[-1,+1]$ *et* :

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

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

Deux cas particuliers sont donc levé pour -1 et +1 pour cet algorithme.

Arc tangente

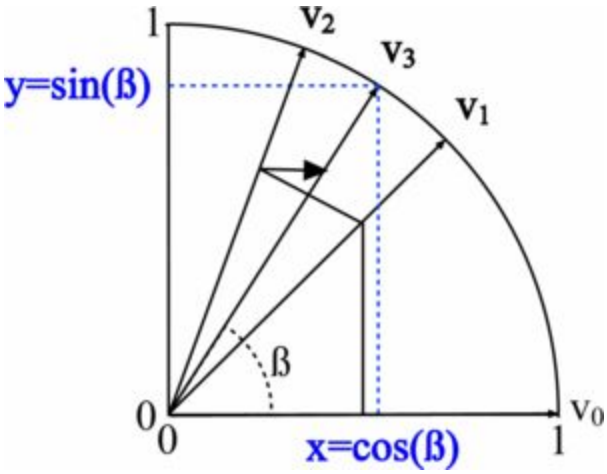
Arc tangente est développable en série entière sur $]-1,+1[$, et :

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

Arc tangente étant définie sur R, l'utilisation de la formule suivante permettait le calcul pour l'ensemble de son domaine de définition :

$$\forall x \in R \qquad arctan(x) = \frac{\pi}{2} - arctan(\frac{1}{x})$$

Algorithme cordic



$Soit \ \theta \in R$

L'algorithme cordic s'appuie sur la représentation géométrique de $cos(\theta)$ et de $sin(\theta)$ qui représente successivement l'abscisse et l'ordonné de l'angle θ dans le cercle trigonométrique.

Plus précisément, en initialisant un vecteur à $x_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, il s'agit de réitérer des rotations sur ce vecteur jusqu'à s'approcher de l'angle θ.

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Ainsi :

$$v_{i+1} = R_i v_i$$

où R_i, la matrice de rotation est définie de manière à comparer le vecteur actuel au vecteur défini par θ :

Le choix réalisé a été : $R_i = cos(arctan(2^{-i}))\begin{pmatrix} 1 & -signe(vi-\theta)\times 2^{-i} \\ signe(vi-\theta)\times 2^{-i} & 1 \end{pmatrix}$

Cet algorithme fonctionne pour des angles inférieurs à $\frac{\pi}{2}$, des formules trigonométriques ont donc été utilisés pour compléter la fonction.

Cody and Waite:

Si les nombres dans l'intervalle $[-\pi, \pi]$ suffisent à définir les fonctions sinus et cosinus grâce à la 2π périodicité, la translation doit tout de même être réalisé, ce qui en fait une nouvelle source d'erreur.

Pour réaliser une translation précise. nous utilisons l'algorithme de cody and waite. Cet algorithme est basé sur la décomposition de n'importe quelle constante utilisée dans nos calculs en une somme de constantes pour diminuer les erreurs dues aux opérations de calcul successives. Ici, il s'agit donc de composer 2π. Ensuite, pour toute opération , on effectue cette opération sur toutes les constantes qui représentent sa décomposition.

On a choisi d'utiliser cet algorithme pour implémenter une fonction nommée *adapt* qui permet d'adapter un angle quelconque et le translater pour avoir enfin son équivalent trigonométrique dans $[-\pi, \pi]$. Cette fonction permet de faire les opérations d'ajout ou soustraction de 2π plusieurs fois avec plus de précision à l'aide de Cody and Waite.
D'abord on a commencé à l'implémenter avec seulement deux constantes, une qui était très proche de la valeur de 2π et une valeur qui la complète pour avoir exactement 2π comme somme.

On a testé cette fonction en faisant une comparaison avec celle implémentée sans Cody and Waite et on a remarqué que la précision a légèrement augmenté donc on a décidé ensuite d'utiliser plusieurs constantes pour avoir un maximum de précision.

Et puisque l'utilisation des nombres qui sont des puissances de 2 permettent de minimiser les erreurs de calculs , on a choisi de faire la décomposition de 2π en la somme de ses puissances de 2.

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Enfin, cette méthode a été testée et donne de bons résultats pour le moment mais elle reste améliorable avec la méthode de Grand Float (voir section 3).

Puisque cet algorithme a donné de bons résultats , on l'a utilisé aussi pour faire des opérations tel que π -x et $(\pi/2)$-x en décomposant cette fois π et $\frac{\pi}{2}$

Autres algorithmes et formules:

Fonction arctangente:

Concernant cette fonction , le développement en série entière n'est pas précis au voisinage de -1 et 1 donc on a cherché un autre algorithme qui permet de régler ce problème.

On a bien trouvé un autre mais qui utilise la fonction racine carrée donc on s'est proposé de l'implémenter.

Voici l'algorithme utilisé pour calculer $\sqrt{}$ x : il est basé sur une suite réelle a_n de premier terme $a_0 = \frac{x}{2}$ et la relation de récurrence a_{n+1} = (½) * ($\frac{x+a_n}{a_n}$) , cette suite converge vers $\sqrt{x}$.

Une fois la fonction sqrt(x) implémentée , on l'utilise dans l'algorithme de calcul de atan(x) qui suit:

$a = \frac{1}{\sqrt{1+x^2}}$, $b = 1$

pour i de 1 à 12 :

$a = \frac{a+b}{2}$

$b = \sqrt{a * b}$

On retourne à la fin $\frac{x}{a*\sqrt{1+x^2}}$

Cet algorithme règle bien les problèmes de précision pour $|x| \sim 1$ pour la fonction $atan(x)$.

Fonction arcsinus:

A part le modèle en séries entières, on a choisi d'utiliser une relation entre $asin(x)$ et $atan(x)$ trouvée sur internet:

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

Fonction tangente:

On a implémenté cette fonction à partir de la formule suivante:

$tan(x) = \frac{sin(x)}{cos(x)}$

Fonction arccosinus:

On a implémenté cette fonction en utilisant cette formule:

$acos(x) = \frac{\pi}{2} - asin(x)$

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Choix mathématiques

On essaye toujours de se placer dans l'intervalle ou les algorithmes disponibles sont les plus précis possibles et ce , à l'aide des formules trigonométriques qui permettent de calculer d'une autre façon un angle situé dans un intervalle où c'est difficile de le calculer avec précision.

Par exemple , pour le développement en série entière la précision est maximale au voisinage de 0 tandis qu'elle diminue en s'éloignant de cette zone.
Donc pour les fonctions trigonométriques on a choisi de procéder comme suit:

D'abord, vu la périodicité de certaines fonctions trigonométriques (sinus , cosinus...) on a implémenté une fonction qui permet de translater un angle et le positionner dans l'intervalle $[-\pi,\pi]$ puisque les fonctions trigonométriques utilisées sont 2π périodiques.

Cette fonction permet de retourner l'équivalent dans $[-\pi,\pi]$ de n'importe quel angle en faisant à chaque fois des ajouts ou soustractions de 2π pour avoir juste à considérer cet intervalle pour nos calculs algorithmiques.

Ensuite,pour la fonction sinus , on se ramène toujours à l'intervalle $[0,\frac{\pi}{2}]$, pour les angles entre $\frac{\pi}{2}$ et π (on nomme x l'angle considéré) il suffit de calculer $sin(\pi - x)$ et pour les angles entre $-\pi$ et 0 on calcule $-sin(-x)$.
Pour la fonction cosinus , on se ramène aussi à l'intervalle $[0,\frac{\pi}{2}]$, pour les angles entre 0 et $\frac{\pi}{2}$ on calcule $sin(\frac{\pi}{2} - x)$ et pour les angles situés dans l'intervalle $[-\pi,0]$ il suffit de calculer $cos(-x)$ avec x l'angle considéré.

Pour la fonction arctangente, on se ramène toujours à l'intervalle [0,1] , donc pour les angles supérieurs à 1, il suffit de calculer $atan(\frac{1}{x})$ et déduire le résultat à partir de la formule $atan(x) = \frac{\pi}{2} - atan(\frac{1}{x})$.
Pour les angles négatifs on calcule $-atan(-x)$ pour revenir aux cas précédents.
Concernant la fonction arcsinus , elle est déjà définie seulement sur l'intervalle [-1,1] ce qui donne déjà une bonne précision sans utilisation de formules trigonométriques.

Choix informatiques

Pour les choix informatiques , on s'est basé sur les tests pour décider quels algorithmes laisser et quels algorithmes choisir pour avoir le maximum de précision.

On choisit parfois deux ou trois algorithmes pour une seule fonction et on utilise chaque algorithme dans un certain intervalle ou il est plus précis que les autres.

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Pour la fonction sinus , on déduit d'après les tests que la version en séries entières est plus précise pour les angles entre -2 et 2 mais en s'approchant de π la version en CORDIC devient plus précise.

Pour la fonction cosinus , la version en CORDIC paraît plus précise mais en passant en Deca , celle de séries entières devient plus précise.

Pour la fonction arctangente, on utilise les séries entières sauf au voisinage de -1 et 1 vu que ça diverge, donc pour cette zone on a opté pour le deuxième algorithme qui règle ce problème.

Concernant la fonction arcsinus , la formule qui utilise arctangente est la plus précise en Java et nous donne de très bons résultats, mais contrairement , en Deca , la version en séries entières devient plus précise.

Test et résultats

Classe FonctionsTests:

C'est une classe implémentée qui permet de réaliser les différents tests des fonctions de la classe Math. Cette classe contient deux attributs , un flottant test qui définit l'erreur maximale tolérée et un booléen qui permet de choisir si cette erreur sera définie à la main ou bien calculée à partir de la fonction ulp.
Cette fonction est munie d'un constructeur qui permet de fixer les deux valeurs des attributs.

On implémente une méthode *Erreur* qui permet de nous donner une idée sur les erreurs de calcul pour les fonctions trigonométriques. Cette fonction prend en paramètre trois flottants, calcule la différence entre les deux premiers et calcule ensuite le rapport entre cette différence et le troisième. Cette méthode nous permet de calculer l'erreur relative pour nos fonctions trigonométriques , par exemple pour la fonction sinus , on calcule la différence entre notre sinus et le sinus de Java ensuite on la divise par l'ulp.

A l'aide de cette fonction on peut approximer l'erreur en nombre de ulp. On implémente après, plusieurs fonctions qui permettent de tester chacune une fonction trigonométrique (*testSinus,testCosinus,testTan,testAtan*,...).

Donc pour chaque fonction trigonométrique on a une méthode qui permet de la tester et on dispose du même modèle pour toutes les méthodes de test (ces méthodes prennent un flottant en paramètre) .

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Voici le modèle utilisé décrit dans le cas de la fonction sinus:
Au début on stocke la valeur du sinus calculée à partir des algorithmes utilisés dans notre classe Math, ensuite on calcule et on stocke la valeur de l'ulp de ce sinus avec la fonction ulp implémentée.

On effectue un test sur l'attribut booléen de notre classe *FonctionsTest* pour voir si on va utiliser la valeur de l'ulp ou bien une autre valeur choisie manuellement pour le test de la précision.

Ensuite on utilise la méthode Erreur qui est dans cette classe pour comparer la différence entre notre sinus et le sinus implémenté par Java avec l'ulp.
Si on arrive à une précision de l'ordre de l'ulp , qui est la précision voulue , on affiche un message "SIN PASS" sinon on affiche "SIN NO".

Ce modèle est le même pour les autres fonctions trigonométriques , il suffit de remplacer le sinus par cosinus , tangente , arctangente…

Il faut en revanche noter que le sinus de la classe Math de Java n'est pas forcément la référence absolue.

Classe TestMathe:

Cette classe contient des tests élémentaires des fonctions basiques comme la valeur absolue , max , min , puissance, factorielle et ulp…

Elle permet de faire des tests et valider ces petites fonctions avant de se lancer dans le reste du code vu que ces fonction seront certainement utilisées pour coder les fonctions trigonométriques.

Bien que ces fonctions sont plus ou moins faciles à implémenter par rapport aux autres , il faut bien vérifier leur bon fonctionnement pour détecter les erreurs à ce niveau là avant de se lancer dans l'implémentation des fonctions complexes et avancés qui utilisent certainement nos fonctions basiques.

Cette étape est très importante et nous fait gagner du temps contrairement à ce qui peut paraître pour d'autres personnes.

Classe TestMathe2:

Cette classe est une classe qui contient le main de test et qui permet de générer une suite de tests des fonctions trigonométriques implémentées.
On commence par afficher un message, ensuite stocker la valeur de pi dans une variable pour l'utiliser plus facilement.

On définit après une instance *FonctionsTest* avec ses deux attributs au choix en fonction des tests voulus.

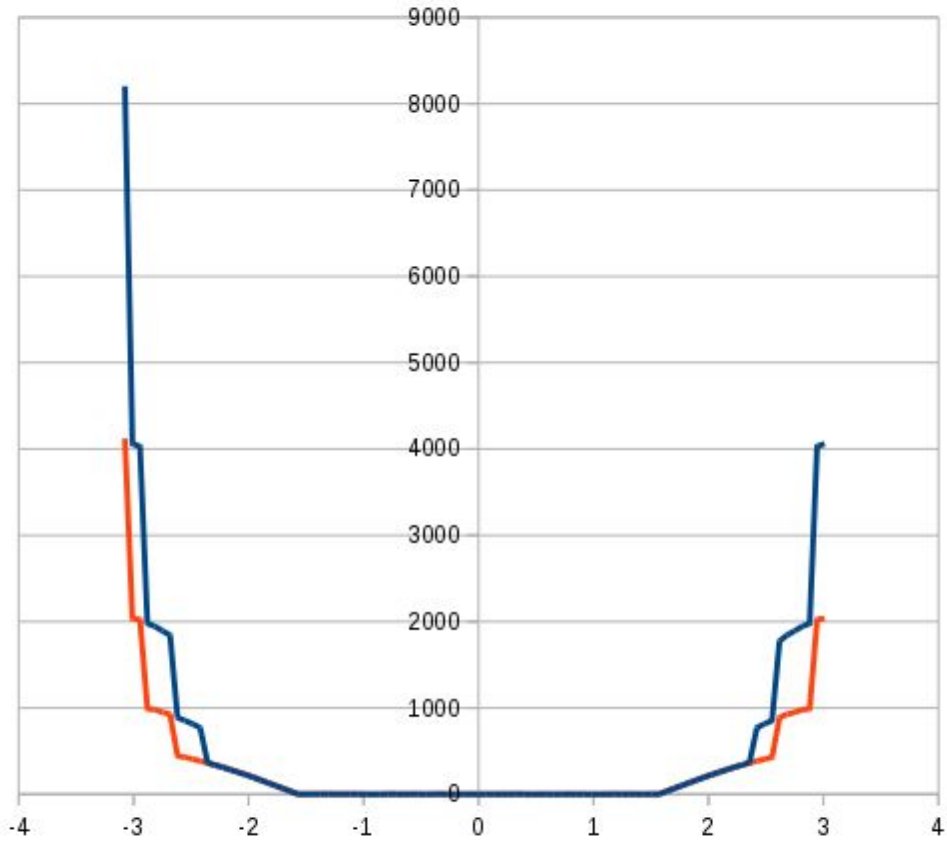
CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

On effectue après une boucle pour chaque fonction trigonométrique.Cette boucle permet de tester cette fonction pour plusieurs valeurs de flottants entre $-\pi$ et π, en affichant à chaque fois la valeur du flottant, son image par la fonction trigonométrique et fait l'appel à la fonction de test spécifique pour cette fonction qui permet d'afficher l'erreur et affiche "PASS" ou "NO" selon la précision de calcul.

On améliore à chaque fois ce modèle de fonction en ajoutant le système de calcul de l'erreur relative par rapport à l'ulp : $\left|\frac{valeur\ calculée - valeur\ réelle}{ulp}\right|$ pour avoir une idée sur nos erreurs de calcul et essayer toujours de les minimiser soit en changeant de méthode ou en changeant d'algorithme et cela grâce aux tests effectués en parallèle avec l'implémentation des fonctions trigonométriques.

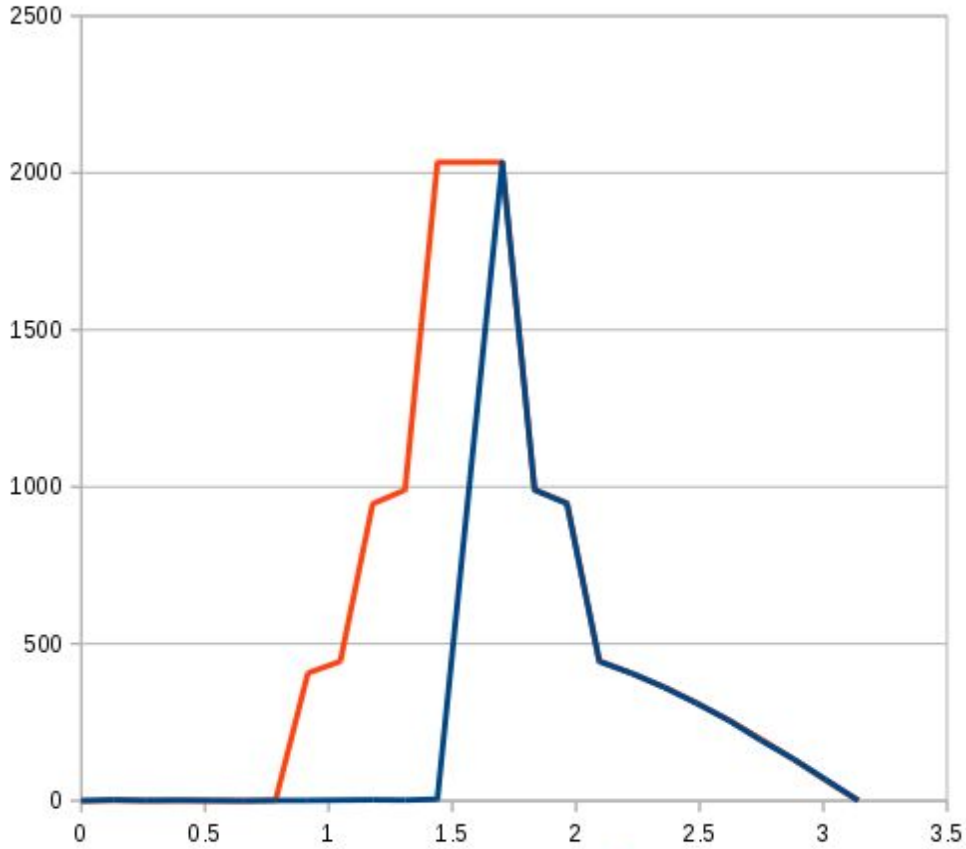
Voici les résultats obtenus pour ces tests (en abscisse la valeur de l'angle et en ordonnée l'erreur relative):

Fonction sinus: en rouge version CORDIC et en bleu version séries entières:



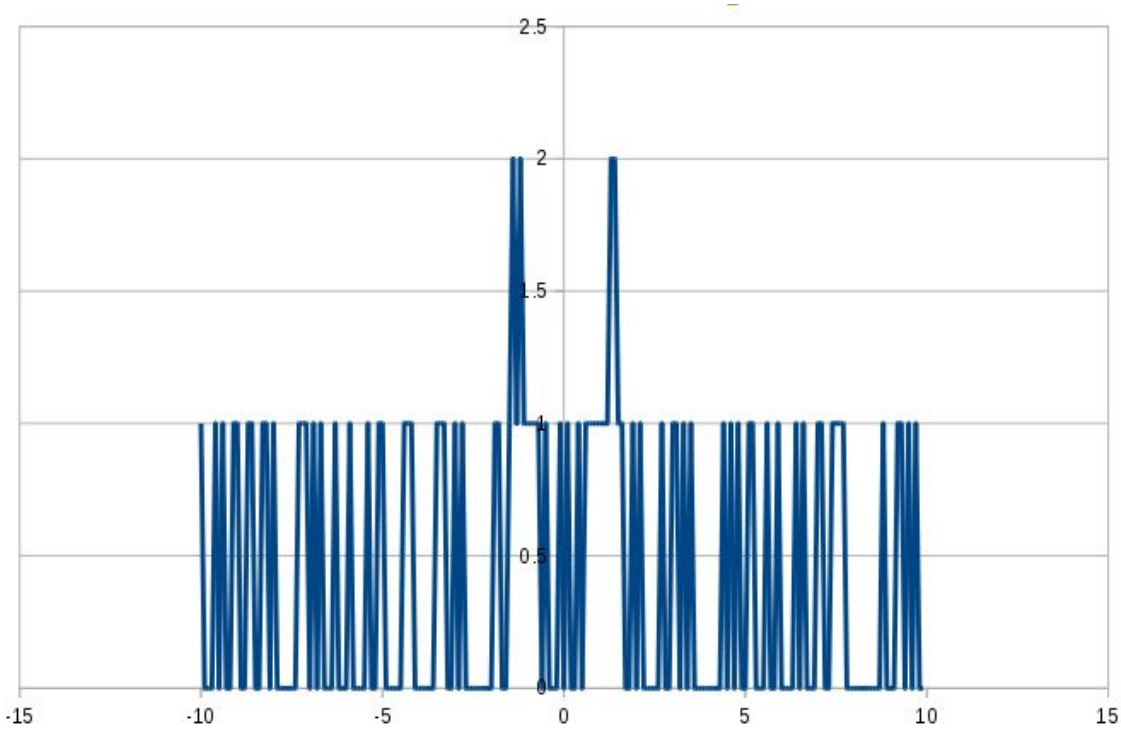
CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Fonction cosinus: en bleu version CORDIC et en rouge version séries entières:

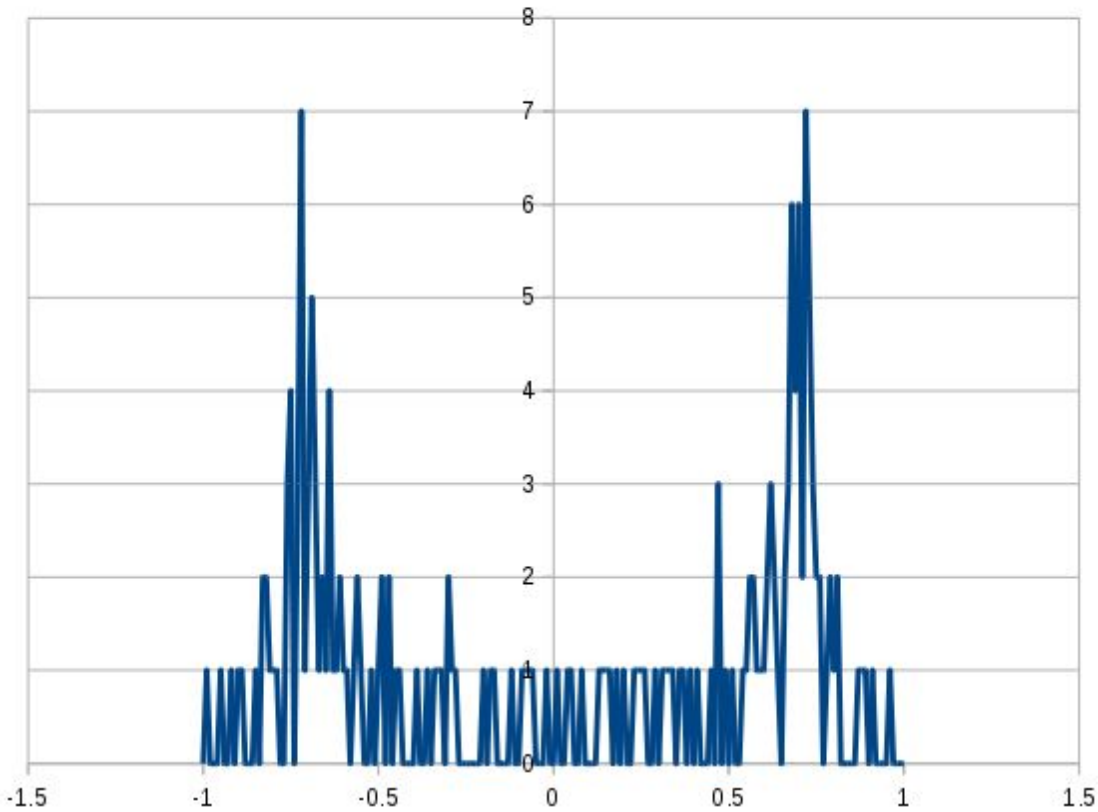


Fonction arctangente: utilisation des deux algorithmes implémentés:

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami



Fonction arcsinus: utilisation de la formule avec arctangente qui est la meilleure en précision:



CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Analyse théorique de la précision

Développement en série entière

Pour les séries alternées on sait que le reste sera majoré par le terme u_{n+1} si on a calculé la somme $\sum_{k=0}^{n} u_k$ donc on peut majorer l'erreur qu'on aura pour nos calculs concernant les fonctions sin, cos et atan.

Sinus: l'erreur est majorée par $u_7 = \frac{x^{15}}{15!}$ (on obtient 10^{-17} pour x=0.5 et 10^{-13} pour x=1)

Cosinus: l'erreur est majorée par $u_7 = \frac{x^{14}}{14!}$ (on obtient 10^{-16} pour x=0.5 et 10^{-11} pour x=1)

Arc tangente: l'erreur est majorée par $u_{1001} = \frac{x^{2003}}{2003}$ (on obtient 10^{-4} pour x=1 et 10^{-95} pour x=0.9)

Amélioration de la précision par des Grand float

Conception de la classe

Générale

Un grand float dans notre extension math est représenté par deux floats. Le premier décrit la représentation float IEEE 754 simple précision du nombre qu'on approche, le second représente l'erreur faite à cause de cette représentation. Ainsi la classe consiste à définir une nouvelle addition et une nouvelle multiplication pour cette structure qui permettent de regrouper les erreurs faites et de les ajouter régulièrement aux bits de poids forts. Les grands floats ne sont donc pas des doubles IEEE 754. Puisque les doubles sont normalement représentés par 1 bit de signe, 11 bit pour l'exposant et 52 pour la mantisse.
Les attributs de cette classe sont donc naturellement :

- f représentant le premier flottant
- erreur représentant le deuxième flottant

cette structure nous permet par exemple d'avoir une représentation plus précise du nombre $\pi = 3.1415926535897$ en stockant :

- dans le premier float f = 3.141592
- dans l'erreur =0.0000006535897

Ce qui est possible grâce à la virgule flottante de la représentation IEEE 754 qui pour des nombres très petits, a une plus grande précision.
Les principales méthodes de cette classes sont donc :

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

- additionSimple. Cette fonction additionne deux flottants et récupère les erreurs faites et les garde dans notre structure.
- multiplicationSimple. Cette fonction multiplie deux flottants et récupère les erreurs faites et les garde dans notre structure
- additionGrandFloat. Cette fonction redéfinie l'addition pour notre structure
- multiplicationGrandFloat. Cette fonction redéfinie la multiplication pour notre structure

Le détail de ces fonctions est décrit sur la section : Algorithme utilisées. D'autres méthodes sont spécifiées dans la spécification de l'extension Math.

Ca marche ?

Une fois l'addition et la multiplication. Nous réappliquons les algorithmes détaillés dans la section 1 aux Grands Floast. La précision augmente, c'est une certitude. Voici un exemple concret :
$sin(3.141592) = 6.535898 \times 10^{-7}$ soit donc une précision à atteindre de :
$ulp = 2^{-44}$

	Sans Grand float	Avec Grand float
Java	$-4.4460827 \times 10^{-4}$	6.535898×10^{-7}
Deca	-4.44608×10^{-4}	6.535900×10^{-7}

Ainsi le passage au Grand float améliore l'écart avec la solution en java de :
$\Delta = 7 \times 10^{9}\ ulp$ à $\Delta = 0\ ulp$
et en déca de :
$\Delta = 7 \times 10^{9}\ ulp$ à $\Delta = 4$

En revanche, autour des zéros des différentes fonctions, la sensibilité est très élevée, car l'ulp est très faible. Cet exemple peut être donc paraître trompeur : Pour d'autres valeurs aux alentours, Plus d'erreurs sont faites dans la représentation binaire du float même et le résultat est moins précis. Le détail complet des erreurs est analysé dans les parties Tests et résultats et Analyse théorique de la précision.

Algorithmes utilisés

Addition simple

L'algorithme GrandFloat addisionSimple(Float f, float g) additionne des flottant simple et récupère l'erreur dans notre structure. Il s'appuie sur le théorème de Dekker suivant :

Théorème 1 :

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Soit a et b deux flottants simple précision tel que $|a| > |b|$
alors

$$a + b = a \oplus b \oplus (b \ominus (x \ominus a))$$ où

$x = a \oplus b$ et $\oplus$ et $\ominus$ sont les additions et soustractions des flottants.

Ainsi l'algorithme stock dans notre structure les flottants :
f : $a \oplus b$
erreur : $(b \ominus (x \ominus a))$
Il a aussi été adapté pour éviter de comparer a et b et faire une erreur en conséquence.

Multiplication simple

L'algorithme GrandFlaot multiplicationSImple(float f , float g) multiplie des flottants simple et récupère l'erreur dans notre structure. Il s'appuie sur un théorème analogue : le théorème de Vetkamp.

Il assure alors pareillement que les valeurs f et erreur stockés dans notre structure assure pour deux flottants a et b:

$$a \times b = f + erreur$$

Addition Grand Float

Comme son nom l'indique GrandFLoat additionGrandFLoat(GrandFloat ff, GrandFloat gg) additionne deux Grand flottants. La fonction commence par additionner séparément les deux flottants f et les deux flottants erreur grâce à la fonction addition simple. Il s'agit ensuite de récupérer toutes les erreurs possible :

- la somme des erreurs
- l'erreur de la somme des erreurs

Multiplication Grand Float

Pareillement, GrandFloat multiplicationGrandFloat(GrandFloat ff, GrandFLoat gg) multiplie deux Grand Flottants :

$$(f^1 + erreur^1) \times (f^2 + erreur^2) = f^1 f^2 + erreur^1 f2 + f^1 erreur^2 + erreur^1 erreur^2$$

Tous les multiples des erreurs sont d'abord ajouté à l'erreur puis grâce à additionSimple, les erreurs les plus significatives sont intégrés au float.

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Choix mathématiques

Les choix mathématiques restent inchangés par rapport à la section 1. En effet les choix qui ont été fait pour un contexte générale. Ici la redéfinition de l'addition et de la multiplication ne change pas les décisions prises mais améliore seulement leur résultat.

Choix informatiques

Seul l'algorithme utilisant le développement en série entière a été développé en Grand Float car s'est avéré satisfaisant.

Deux fonctions particulière ont été implémentée pour s'adapter aux choix mathématiques :

- pimoinsf : calcul pour un flottant f, de $\pi - f$ avec une précision Grand Float. Cette fonction tente de minimiser les erreurs sensibles autour du zéro de la fonction sinus. en effet, une erreur peu significative sur un nombre f autour de π^+ ($ulp = 2^{-22}$) peut se transformer en une erreur importante pour $sin(\pi^+) \approx 0$ ($ulp\ autour\ de\ 2^{-50}$)
- pisurmoinsf :effectue le même le travail autour du zéro de la fonction cosinus : $\frac{\pi}{2}$

Test et résultat

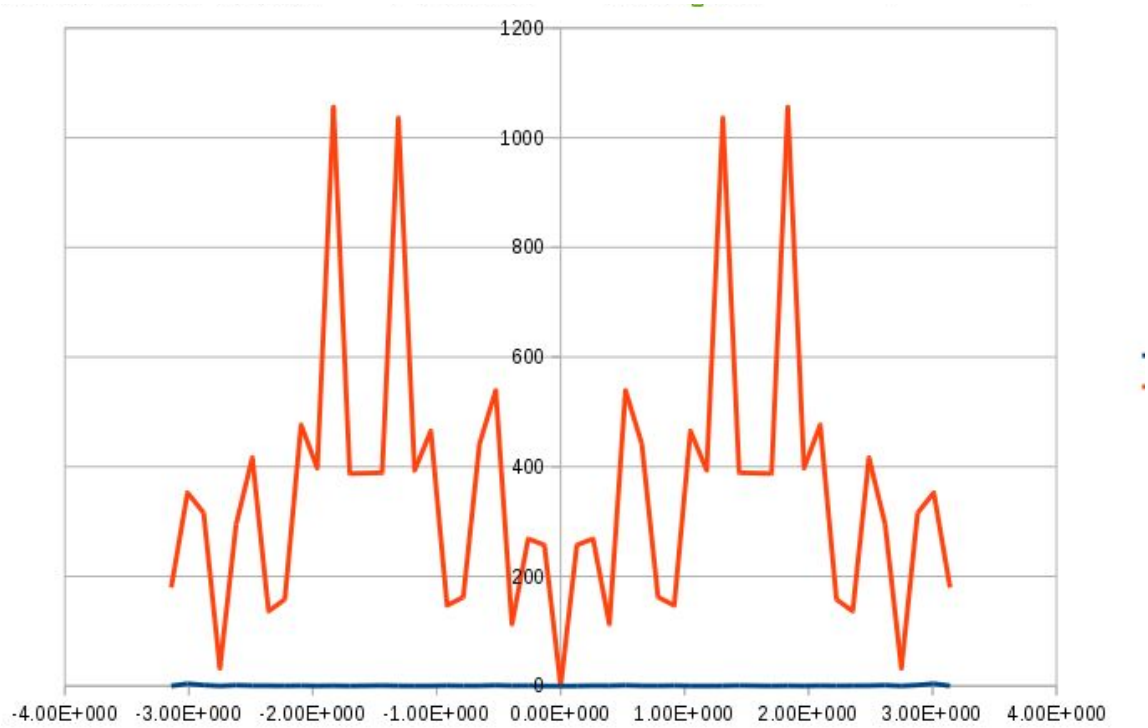
Concernant les fonctions trigonométriques implémentées avec le système de Grand Floats on a effectué des tests en Java et en Deca vu que la précision obtenue n'est pas la même dans les deux cas.

On a tracé les courbes qui permettent de représenter *l'erreur relative* (par rapport à l'ulp) de ces fonctions trigonométriques (sin,cos) dans l'intervalle *[-π , π]* .

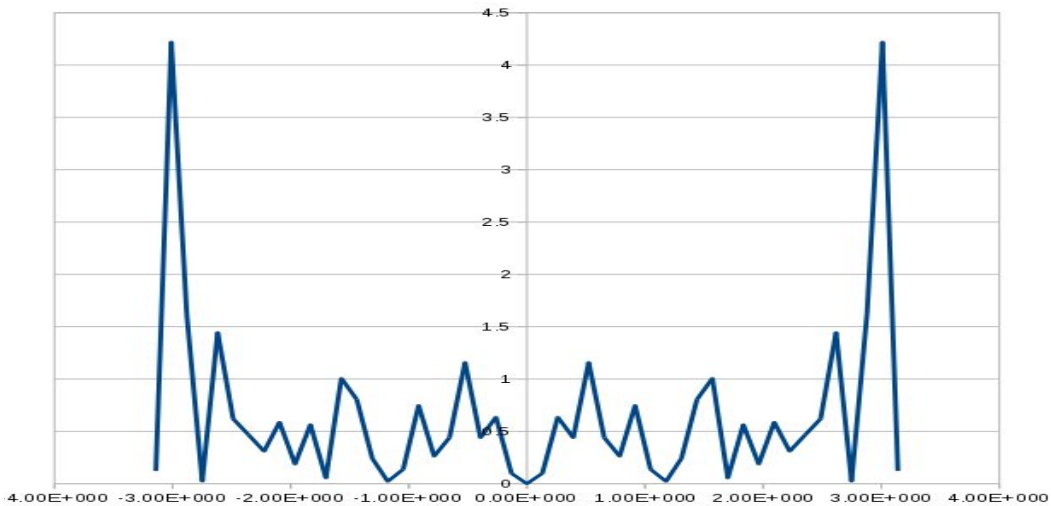
$Erreur\ relative = \left|\frac{valeur\ calculée - valeur\ réelle}{ulp}\right|$

Voici les résultats obtenus pour la fonction sinus avec Java en bleu et Deca en rouge:

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

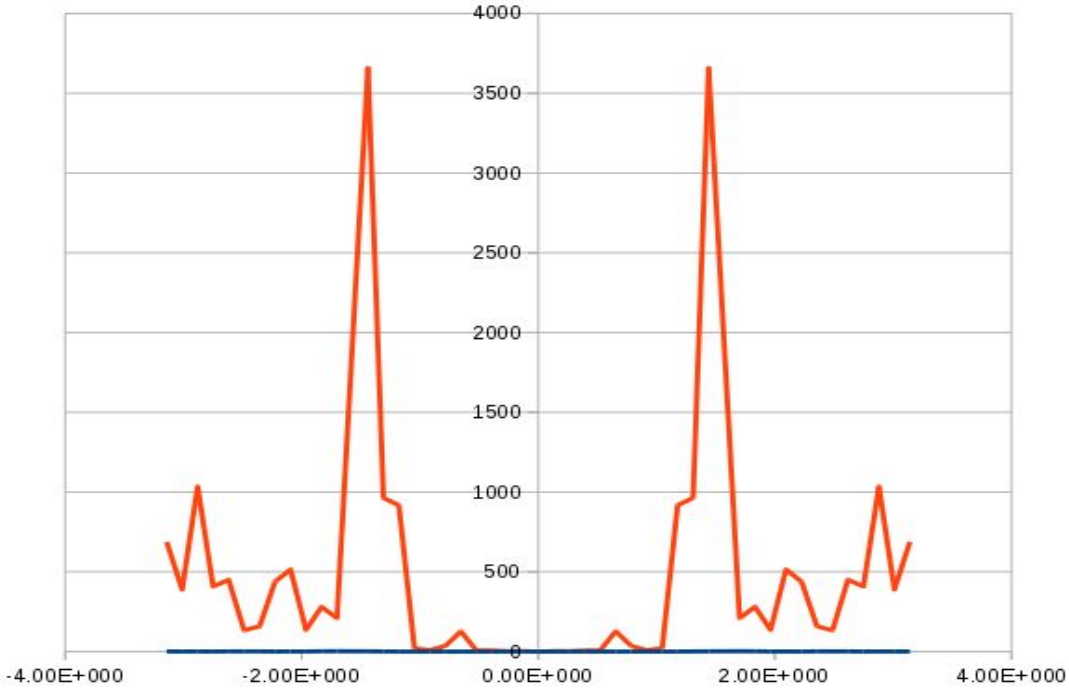


Zoom sur le voisinage de 0 pour la courbe bleue pour notre sinus en Java:

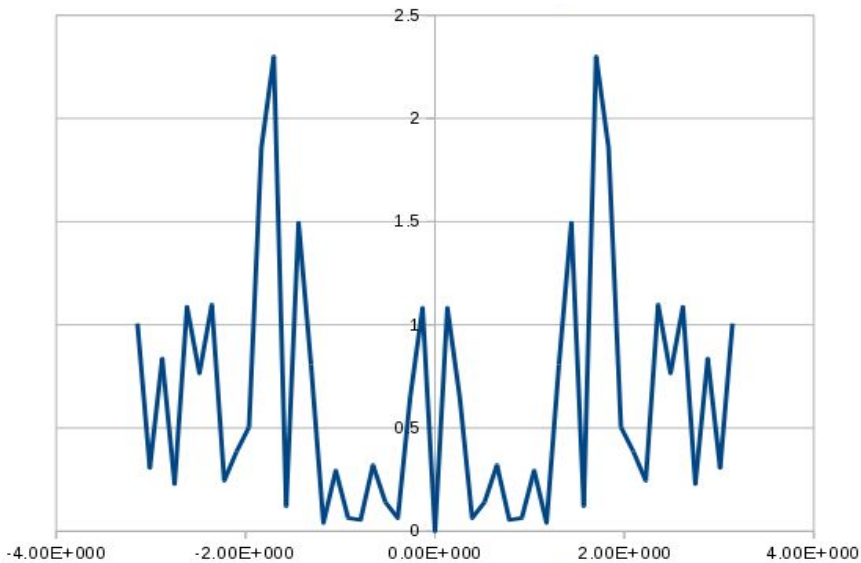


CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

Voici les résultats de nos tests sur la fonction cosinus de la même manière:

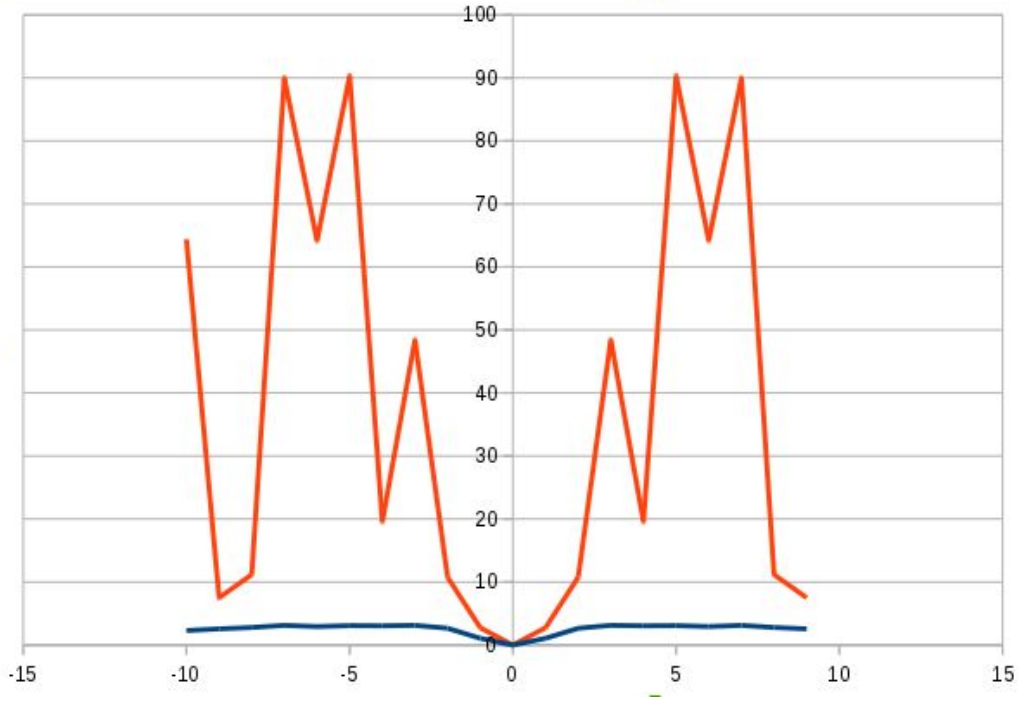


Zoom sur le voisinage de 0 pour la courbe bleue pour notre sinus en Java:

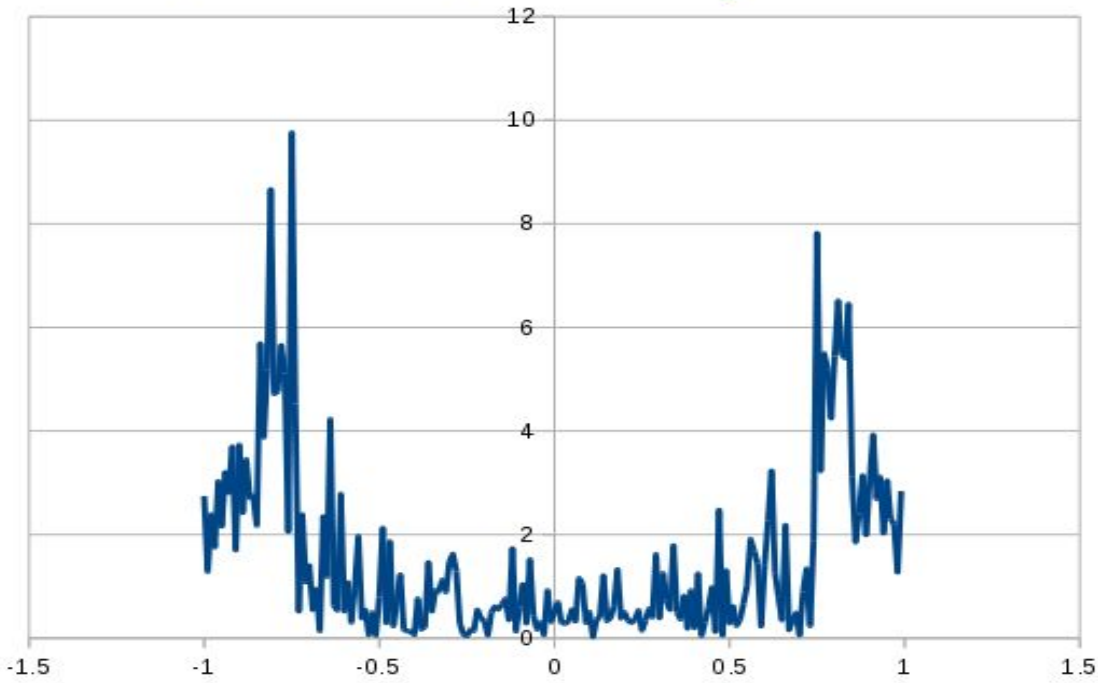


CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

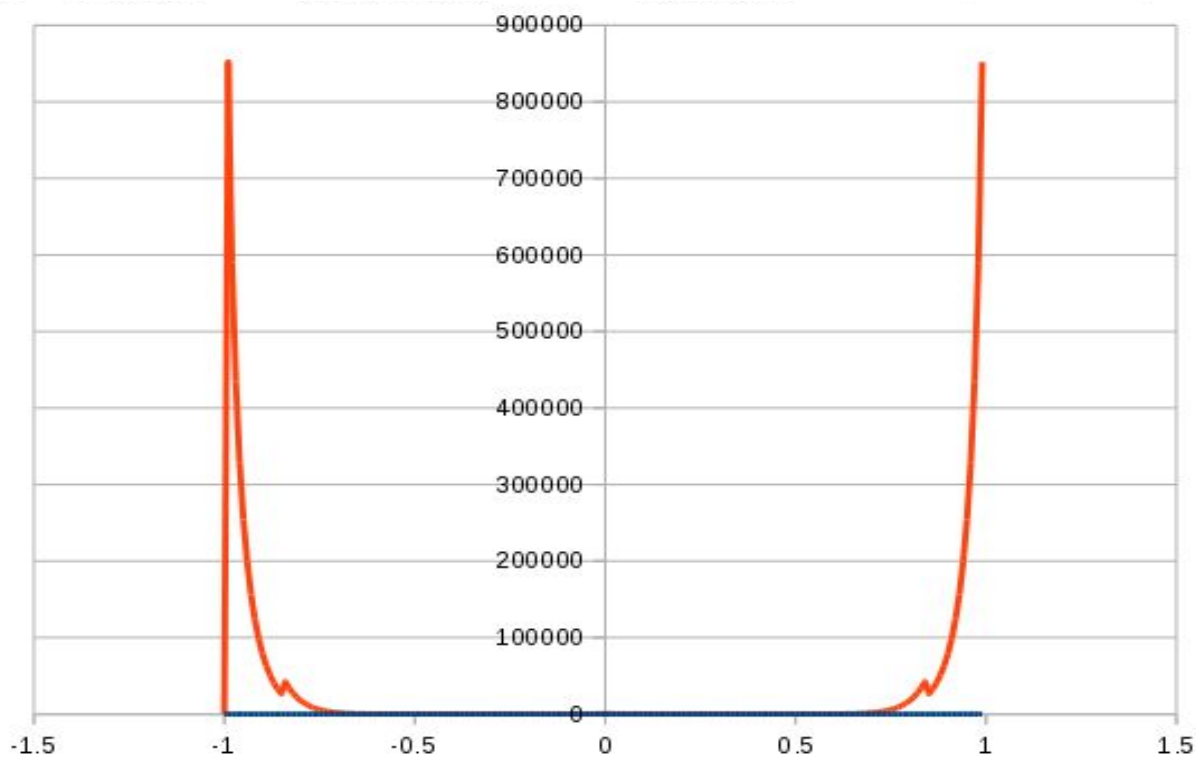
Pour la fonction arctangente , on peut dire que les résultats obtenus en Deca étaient bon par rapport aux autres fonctions.
Voici la courbe qui illustre les tests effectués entre -10 et 10 pour l'erreur relative:



Pour la fonction arcsinus , on a eu une très bonne précision avec les deux algorithmes (Séries entières et formule avec atan) en Java mais en passant en Deca on perd beaucoup de précision. Voici nos résultats en Java, ensuite la différence avec Deca.



CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami



Analyse théorique de la précision

L'algorithme sélectionné étant le développement en série entière. La limite de précision espérée reste la même que celle de la section 1.

Références bibliographiques:

Développements en séries entières:
http://www.panamaths.net/Documents/Formulaires/FORMU_DSEUSUELS.pdf
http://www.trigofacile.com/maths/trigo/calcul/cordic/cordic.htm

IEE754:
https://fr.wikipedia.org/wiki/IEEE_754

Formules trigonométriques:
https://www.maths-france.fr/MathSup/Cours/FormulaireTrigo.pdf
https://www.maths-france.fr/Terminale/TerminaleS/FichesCours/FormulesTrigonometrie.pdf

Algorithme arctangente:

CLASTRES Thomas
BERTRAND Lucas
BUTHOD-GARCON Tony
BEL HADJ SALAH Nizar
BENJELLOUN Thami

http://www.gladir.com/CODER/TPASCAL7/arctan.htm

Grands Floats:
http://www.dptinfo.ens-cachan.fr/Conferences/muller.pdf