Rapport extension

Projet GL

* Groupe g3 | Equipe gl16 *
Ange Romuald Ossohou
Nadir Ait Lahmouch
Hamza Benjelloun
Oussama Fennane
Younes Zaibila

30 janvier 2020

Table des matières

1 Introduction

Ce document a pour but d'expliquer la conception de la classe Math.decah.
Il présente à la fois les algorithmes utilisés, ainsi que les choix mathématiques et informatiques nécessaires à l'implémentation, mais aussi la démarche qui a permis d'aboutir.
La validation et les limites de nos algorithmes sont décrites au fur et à mesure, expliquant les choix que nous avons étés amenés a prendre.

La classe Math.deca contient d'abord quelques méthodes de base, ainsi que des constantes mathématiques qu'on utilisera dans nos calculs, par exemple la méhtode puissance ou la constante pi mais qu'on va pas trop détailler dans ce document.

2 Ulp

La fonction ulp (Unit in the Last Place) est fréquement utilisée pour exprimer des erreurs en arithmétiques flottantes. Ainsi, l'ulp d'un flottant f répresente la distance ce flottant et le flottant le plus proche.

2.1 Notion de norme normalisée et denormalisée selon la norme IEEE 754

Il existe de type de réprensentation des flottants à savoir la réprensatation simple précision (codée sur 32 bits) et la double précision (codée sur 64 bits, elle principalement utilisée pour le type **double**. Ainsi notre architecture nous permet d'utiliser la répresentation simple précision vu que nos flottants sont codés sur 32 bits. La réprésentation normalisé d'un flottant f est donné ci-dessous :

$$f = s.2^{e-127}.(1,m)\ ,\ avec \begin{cases} s = \pm 1\ représente\ le\ signe(selon\ le\ bit\ de\ signe) \\ e\ est\ l'exposant\ (codé\ sur\ 8\ bits) \\ m\ reépresente\ la\ partie\ fractionnaire\ ou\ la\ mantissesur\ 23\ bits) \end{cases}$$

Par ailleurs, lorsque l'exposant $e = 0$, la répresentation normalisée laisse sa place à la réprensatation denormalisée qui se note comme suite pour un flottant g :

$$g = s.2^{-126}.(0,m)\ ,\ avec \begin{cases} s = \pm 1\ représente\ le\ signe(selon\ le\ bit\ de\ signe) \\ m\ reépresente\ la\ partie\ fractionnaire\ ou\ la\ mantissesur\ 23\ bits) \end{cases}$$

2.2 Implémentation de la méthode Ulp

Pour avoir une idée de l'algorithme rien de plus simple que de le tester sur un exemple simple. Ainsi, dertermions $ulp(2)$. La réprensation normalisée du flottant 2 est la suivante :

$$2 = +1.2^1.1,000..00$$

d'où le plus proche voisin (superieur) du flottant 2 est le flottant $\tilde{2}$ qui se note (en réprensation normalisée) :

$$\tilde{2} = +1.2^1.1,000..01$$

On peut donc s'amuser à calculer la distance entre le flotant 2 et le flottant $\tilde{2}$.

$$diff = 2^{-23}.2^1 = 2^{-22}$$

Par définition, $ulp(2) = diff$. cet exemple, nous a permis d'avoir une idée de l'algo. Grâce à cet exemple, on en déduit facilement ulp de toutes les puissances de 2. Par conséquent, le reste du travail consiste à trouver ulp des flottants présents entre ces differentes puissances de 2. Autre remarque, qu'on peut tirer de cet exemple, est que le calcul de l'ulp necessite uniquement de connaître l'exposant de la réprensentation normalisé (l'exposant de la réprensentatin denormalisé est déjà connue par définition). D'où, pour tout flottant f compris entre $2^p \leq f < 2^{p+1}$ avec p un entier , $ulp(f) = 2^{-23+p}$.
Finalement, l'implémentation de la methode Ulp se réduit à la recherche de la plus grande puissance p de 2 inférieure au flottant en question. Une fois cet exposant trouvé une vérification (ie on verifie que $p \geq -126$ dans ce cas on utlise la forme normalisée) est faite pour savoir s'il s'agit de la réprensentation normalisé dans ce cas notre algo retourne 2^{-23+p}, sinon s'il s'agit de la réprensentation denormalisée, l'algo retourne $2^{-126-23} = 2^{-149}$.
Concernant la gestion des valeurs particulières à savoir MAX_VALUE, MIN_VALUE ET 0, un traitement particulier leur a été accordé.
Le flottant 0 utilse par definition la répresentation dérnormalisée, ainsi comme mentionnée plus haut, $ulp(0) = 2^{-149}$. De même, la puissance p de 2 inferieure à la MIN_VALUE est égale à -126 d'où $ulp(MIN\_VALUE) = 2^{-126-23} = 2^{-149}$. Pour finir, $2^1 27 \leq MAX\_VALUE$, en appliquant encore la definition de l'ulp énoncé ci-dessus, on obtient donc que $ulp(MAX\_VALUE) = 2^{127-23} = 2^{104}$

2.3 Validation de l'algorithme

La validation de notre méthode ulp a été faite sur base de comparaison avec elle celle de java, on peut donc s'en convaincre avec le tableau sur dessous :

Intervalles	Nombres d'érreurs	Erreur Max (en ulp)	pas	Nombre de tests
[0, 1]	0	0	2^{-14}	16384
[1 000, 10 000]	0	0	2^{-7}	1 152 128
[10 000, 100 000]	0	0	2^{-7}	1 150 128

On peut donc constater sinon dire que notre méthode Ulp donne des résultats totalement identiques à celle de Java, ce qui très satisfaisant.

NB : L'erreur en un point x est pris en compte lorsque la différence absolue entre notre ulp et celui de java est supérieure à notre ulp en ce point.

2.4 Réferences bibliographiques

Les liens utlisés pour notre documentation concernant la méthode ulp sont les suivants :
— Lien wikipedia sur la norme IEEE 745
— Nombre binaire à virgule flottante
— Conversion du décimal au flottant

3 Cos et Sin

Dans cette section, nous allons vous représenter les différentes étapes suivies pour l'implementation de la fonction **cos(float f)**.
Comme toutes les autres fonctions de la classe **Math** la difficulté réside dans l'optimisation des calculs et l'utilisation des méthodes adéquates pour augmenter la précision des résultats, c'est-à-dire une précision à quelques dizaines d'ULP pour le plus grand ensemble de valeur possible.

3.1 Conception

la partie extension est beaucoup moins guidée que le reste du compilateur, c'est pour cette raison qu'il faut être très prudent dans le choix des algorithmes et des méthodes d'implementation.
Après plusieurs heures de documentation, on a décidé d'implementer d'abord l'algorithme Cordic et celui de Taylor vu qu'ils sont les plus utilisés pour l'approximation du cosinus et de comparer leurs resultats, comme on representera aussi la methode de reduction utlisée avec sa justification.

3.1.1 Algorithme de Cordic

CORDIC (sigle de COordinate Rotation DIgital Computer, « calcul numérique par rotation de coordonnées ») est un algorithme de calcul des fonctions trigonométriques et hyperboliques, notamment utilisé dans les calculatrices. Il a été décrit pour la première fois en 1959 par Jack E. Volder. Il ressemble à des techniques qui avaient été décrites par Henry Briggs en 1624.

La méthode de cordic consiste à calculer le cosinus d'un angle qui est égal ou très proche du vrai angle.
Soit $v_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

Plus formellement, à chaque itération i, on calcule un nouveau vecteur grâce à la multiplication du vecteur vi avec la matrice de rotation Ri : $v_{i+1} = R_i v_i$
La matrice de rotation Ri s'obtient selon la formule suivante :

$$R_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i \\ \sin\theta_i & \cos\theta_i \end{pmatrix}$$

à chaque itération on ajoute un angle θ_i tq $\theta_i > \theta_{i+1}$, généralement on prend $\theta_i = \arctan 2^{-i}$, et donc

l'angle θ est la limite de la somme $\sum \theta_i$.
vous trouverez l'implementation de cet algorithme dans le fichier ProjetGL/extension/src/TrigoCordic.py

3.1.2 Algorithme de Taylor

le développement en série de Taylor de $cos()$ en a, est une série entière construite à partir de $cos()$ et de ses dérivées successives en a. cette série coincide avec la fonction $cos()$ au voisinage de a.
la série de taylor au voisinage de 0 est la suivante : $\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$
comme vous pouvez le constater le calcul de cette somme necessite un très grand nombre d'opérations, c'est pour cette raison qu'on a décidé d'optimiser ces calculs en utilisant la formule de récurrence suivante :
$a_n = \frac{-a_{n-1}}{(2n-1)(2n)} x^2$

et donc la série devient $s = \sum_{n=0}^{\infty} a_n$ et de cette manière la complexité des calculs est linéaire, ce qui nous permettra de gagner plus de précision et plus de temps.

En effet avant l'utilisation de cette formule, le nombre d'opérations pour chaque iteration i était ; $\mathcal{O}(i)$ mais en sauvegardant la valeur de a_{i-1} le nombre d'operation devient $\mathcal{O}(1)$ ce qui rend le calcul linéaire. Comme indiqué précédemment, cette formule est valide au voisinage de 0, c'est pour cette raison qu'on utilisera cette méthode pour les valeurs dans l'intervalle $[-\pi, \pi]$ seulement.

vous trouverez l'implementation de cet algorithme dans le fichier ProjetGL/extension/src/TrigoTaylor.py ou dans ProjetGL/extension/src/Puissance.java

NB : le choix de l'intervalle $[-\pi, \pi]$ sera expliqué dans la partie d'analyse et de validation.

3.1.3 Algorithme de Réduction

L'étape de réduction prend une partie très importante de l'extension, parce que les méthodes cos et sin par exemple ne prennent que des valeurs réduites et donc la sortie de cet algorithme sera une entrée pour d'autres, dans le cadre de notre projet cet algorithme a pour but de remplacer la fonction modulo qui n'est pas disponible dans le langage de notre compilateur deca. le but de cette étape de réduction est de transformer un réel x à $\alpha + k\pi$ et donc la méthode de reduction doit nous renvoyer la valeur α.

dans l'implemtation on se basera sur la formule suivante :
$\alpha = x - \lfloor \frac{x}{\pi} \rfloor \pi$
avec cette formule on peut retrouver une valeur proche de celle renvoyé par le modulo avec une soustraction et une multiplication seulement, la partie entière de x peut être obtenue de cette manière :

```
int k = (int) (f/pi);
```

ensuite on obtient la valeur α avec :

```
float alpha =  x - k*(pi);
```

les résultats de cette méthode seront analysés dons les prochaines sections. Vous trouverez l'implementation dans le fichier ProjetGL/src/main/ressources/include/math.decah

3.2 Analyse

Dans cette partie on va analyser les résultats de chaque algorithme pour qu'on puisse voir les limites de chaque méthode et calculer la précision qui déterminera la qualité de notre programme. Comme dans la section de conception on commencera d'abord par l'algorithme Cordic, puis on passera à l'analyse de Taylor et de l'algorithme de réduction.

3.2.1 Algorithme de Cordic

Après l'implementation de l'algorithme de Cordic, on commence à analyser les resultas obtenus.

le premier reflexe est de calculer la différence entre **Math.cos()** et notre algorithme on peut résumer les résultats dans le tableau suivant :

Intervalles	pas	difference moy
[0, 0.1]	2^{-7}	$1.95 * 10^{-10}$
[0.1, 1]	2^{-5}	$5 * 10^{-10}$
$[1, \frac{\pi}{2}]$	2^{-5}	$6 * 10^{-9}$

Alors comme on peut remarquer à travers cette premiere analyse, la méthode de Cordic donne une différence moyenne de 10^{-10} mais l'inconvénient c'est qu'elle est définie en $[0, \frac{\pi}{2}]$ seulement.

3.2.2 Algorithme de Taylor

Dans cette partie on va analyser les résultats de l'algorithme de Taylor décrit ci-dessus.

Encore une fois on calcule d'abord la différence entre **Math.cos()** et la méthode de Taylor.

Intervalles	pas	difference moy
[0, 0.1]	2^{-7}	$2 * 10^{-10}$
[0.1, 1]	2^{-5}	$6 * 10^{-10}$
$[1, \frac{\pi}{2}]$	2^{-5}	$7 * 10^{-10}$

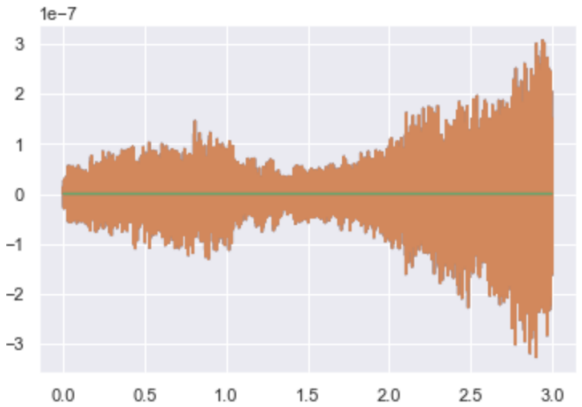
Les résultats des deux tableaux sont presque identiques, sauf que la méthode de Taylor est plus précise au voisinage de 0 la différence avec la vraie valeur est nulle, c'est pour cette raison qu'on a décidé d'utiliser cet algorithme.

3.2.3 Validation

pour valider le choix précédent, on va continuer à analyser l'algorithme de Taylor pour connaitre ses limites.

Remarque : Dans la suite de cette partie on étudiera la valeur de Cos pour un réel positif vu la parité de la série de Taylor.

le graphe ci-dessous represente la différence entre Math.cos et notre méthode :



différence avec l'algorithme de Taylor

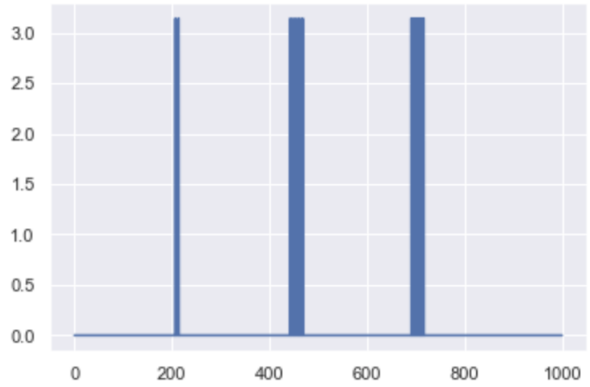
Comme on peut le constater à partir de ce graphe, la différence augmente dans l'intervalle $[2, \pi]$ mais

elle reste satisfaisante puisque $\frac{10^{-7}}{Ulp(2.5)}$ est strictement inférieure à 10 et donc on peut valider notre choix d'approximation du cosinus avec la méthode de Taylor sur l'intervalle de $[-\pi, \pi]$

3.2.4 Analyse de l'algorithme de réduction

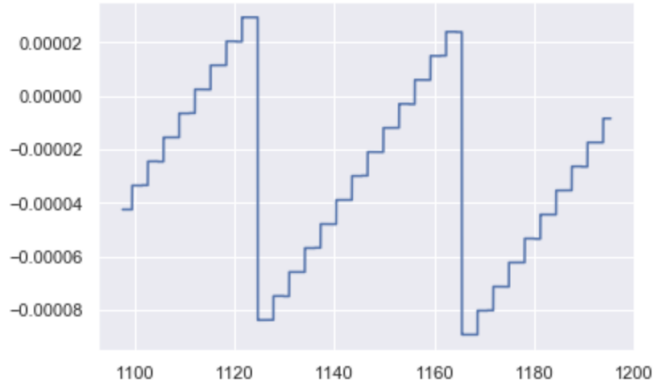
Pour valider l'algorithme de reduction, on peut calculer sa différence avec le modulo de java.

en affichant cette différence d'abord sur l'intervalle [0, 1000]



différence avec modulo de java

On remarque que la méthode de réduction utilisée renvoie des fois π au lieu de 0(à partir de 200) et donc cette réduction peut poser des problèmes de signe. c'est pour cette raison que dans la méthode cos implemetée on n'utilisera pas les résultats de la réduction pour déterminer le signe.



différence avec modulo de java

On constate aussi que pour des valeurs supérieures à 1100 la difference est de l'ordre de 10^{-4}.
Les résultas montrent que pour les valeurs inferieures à 1000 la différence est plus petite. Alors avant de valider cette méthode, on calcule d'abord le cosinus sur R.

On compare les deux méthodes cos(reduce(float f)) et Math.cos($f \mod \pi$) avec reduce(float f) notre méthode de reduction.

On calcule d'abord quelques statistiques pour les deux méthodes :

```
In [443]: dataset["pythonCosModulo"].describe()

Out[443]: count    200407.000000
          mean          0.000052
          std           0.707118
          min          -1.000000
          25%          -0.707075
          50%           0.000105
          75%           0.707192
          max           1.000000
          Name: pythonCosModulo, dtype: float64
```

```
In [250]: dataset["myCos"].describe()

Out[250]: count    200407.000000
          mean          0.000074
          std           0.707118
          min          -1.000000
          25%          -0.707063
          50%           0.000061
          75%           0.707131
          max           1.000000
          Name: myCos, dtype: float64
```

caracteristiques des deux méthodes

On peut voir que les deux méthodes ont les mêmes caracteristiques, en plus $\frac{1}{10}$ des valeurs seulement ont une erreur superieure à 10^{-5}. avec quelques de erreurs de signe aussi, c'est pour cette raison qu'on a décidé d'utiliser le pseudocode suivant pour trouver le signe independement de la reduction :

```
int m;
int p
m = (int) (f/PI_2); // PI_2 la valeur de pi/2
p = m%4; // p est la position de reduce(f) dans le cercle unit
if (p == 1 || p == 2) {
    return -s // s est la valeur absolue du cos(f)
}
```

Finalement on peut résumer les résultats obtenus dans ce tableau :

Intervalles	Nombre d'erreurs	Erreur relative moyen	Erreur moyen max	pas de la boucle	Nombre de test
[0.0, pi]	1663	-3.963071E-8	-3.0403606E-7	pow(2, -10)	3207
[pi, 100 pi]	2415	4.147462E-5	6.931883E-4	pow(2, -3)	2489
[100 pi , 100 000]	3115	0.015910711	-5.668972	pow(2, 5)	3116

3.3 Sin

le calcul du sinus est similaire à celui du cosinus c'est pour cette raison qu'on prefere faire un appel du cosinus au lieu de recalculer une série de Taylor meme si le nombre total d'opérations va augmenter de 1 en utilisant la formule $sin(x) = cos(x - \frac{\pi}{2})$

On obtient les resultats suivants :

Intervalles	Nombre d'erreurs	Erreur relative moyen	Erreur moyen max	pas de la boucle	Nombre de test
[0.1, pi]	1974	2.5990856E-7	2.3482336E-7	pow(2, -10)	3207
[pi, 100 pi]	1215	2.0544464E-4	2.1214543E-4	pow(2, -3)	1245
[100 pi , 100 000]	3116	7.686560E-4	-0.3116466	pow(2, 5)	3116

3.4 Références bibliographiques

-https ://fr.wikipedia.org/wiki/CORDIC

-https ://www.apmep.fr/IMG/pdf/cordic.pdf

-http ://www.trigofacile.com/maths/trigo/calcul/cordic/cordic.htm

-http ://zanotti.univ-tln.fr/ALGO/I31/Cordic.html

-https ://ljk.imag.fr/membres/Bernard.Ycart/mel/dl/node6.html

-https ://fr.wikipedia.org/wiki/DC3A9veloppement$_limitC3A9$

4 Arctan

La méthode Arctangente présente deux propriétés intéréssantes à savoir un développement en série entière et une dérivée relativement simple. On pouvait également utiliser l'algorithme de Cordic pour calculer l'arctangente. Cependant cet algorithme présente des inconvénients. En effet, il nécessite de stocker de nombreuses valeurs calculées utltérieurement. La précision à avoir sur ces valeurs et l'arrondi nécessaire sont difficiles à estimer. Sans collection à savoir un tableau, il serait compliqué de manipuler toutes ses données.Mais on aurait pu mettre toutes ces valeurs comme des attributs d'une classe en pouvant les acceder grâce à des getteurs. Mais par soucis de diversification et de stockage, nous avons opté d'utiliser l'approximation de Padé

4.1 Approximation de Padé

4.1.1 Motivation

Soit f une fonction de classe C^∞ au voisinage de 0, on dit qu'elle admet un développement limité à l'ordre n s'il existe un polynôme P de degré au plus n tel que $f(x) - P(x) = o(x^n)$. Le polynôme P est parfois appelé polynôme de Taylor de f à l'ordre n.Il fournit une approximation locale de f.On peut regarder expérimentalement la qualité de l'approximation de f par son polynôme de Taylor. Dans le cas des fonctions usuelles ($cos(x)$, $sin(x)$, $tan(x)$), on constate que l'approximation semble de plus en

plus fine lorsque le degré augmente, mais parfois sur un domaine limité. C'est en substance la notion de fonction développable en série entière,le polynôme de Taylor donne une bonne approximation sur]1,1]). Si l'on désire généraliser la notion de développement limité, il semble légitime de remplacer le polynôme P par une fraction rationnelle F.On définit ainsi la notion d'approximant de Padé.

4.1.2 Définition de l'approximation de Padé

Soit f une fonction de classe C^∞ sur un intervalle contenant 0. On dit que $F = P/Q \in R(X)$ est un [p/q] approximant de Padé de f si $degP \leq p$, $degQ \leq q$, tel que $Q(0) = 1$ et $f(x) - P(x)/Q(x) = o(x^{p+q})$

4.2 Implémentation de méthode atan

Au vue de la définition de l'approximation de Padé, le but de notre algo sera de trouver deux polynômes (ie les coefficients de ses deux polynômes). Comme précedement, nous allons appliquer cet algo sur exemple simple. Déterminons donc l'approximation de Padé [1/2] de la fonction $asin(x)$. La prémière étape est de déterminer le développement limité à l'ordre souhaité, pour notre exemple on prendre l'ordre n = 3. On a donc $asin(x) = x + \frac{x^3}{6} + \mathrm{O}(x^5)$, l'approximation de Padé [1/2] de la fonction $asin(x)$ s'écrit comme suit : $asin_{[1/2]}(x) = \frac{a_0 + a_1 x}{1 + b_1 x + b_2 x^2}$. Il reste donc à déterminer les coefficients a_i et b_j. Par définition, $a_0 + a_1 x = (x + \frac{x^3}{6})(1 + b_1 x + b_2 x^2) + K(x)x^4$, après avoir développé, on obtient par identification que $a_0 = 0$, $a_1 = 1$, $b_1 = 0$ et enfin $b_2 = \frac{-1}{6}$.
On obtient donc finalement que l'approximation de Padé de la fonction $asin_{[1/2]}(x) = \frac{x}{1 - \frac{1}{6}x^2} = \frac{6x}{6 - x^2}$.
Cet exemple de savoir comment fonctionnent l'approximation de Padé. Concernant notre cas à savoir la fonction arctangente, nous avons opté pour une approximation de padé [3/4] qui utilise un developpement limité à l'ordre 7 de la fonction arctangente (on aurait dû aller plus loin dans le dévloppement limité pour gagner en précision), pour gagner en efficacité, les coefficients des ces différents ont été calculé mis comme variable dans le corps de la méthode (bonne nouvelle tous les coefficients sont des entiers, on a donc pas eu des problèmes pour les approximer, de plus pour encore réduire l'imprécition, le calcul des ces polynômes a été combiné avec la méthode de Horner qui a permis de limiter le nombre d'opérations. Cependant, n'oublions pas que même si le domaine de convergence de l'approxiamtion de Padé est grand que celui du developpement en série entière, elle demeure plus efficace au voisinage de 0, étant donné que la fonction $atan(x)$ est définie sur $\mathbf{R}$ un problème se pose. Pour pallier à ce problème nous décidé d'utiliser de scinder notre intervalle $[0, +\infty[$ en trois intervalles à savaoir : $[0, \sqrt{2} - 1[$, $[\sqrt{2} - 1, \sqrt{2} + 1[$ et $[\sqrt{2} + 1, +\infty[$ (par parité, on fait de même pour l'intervalle $]-\infty, 0]$). L'idéé dérrière cette partition de l'intervalle est de pouvoir toujours se ramener en utilisant des méthodes, à l'intervalle $[0, \sqrt{2} - 1[$ là où l'approximation de Padé nous assure une bonne précision. La méthode évoquée est tout simplement l'identité remarquable de l'arctangente pour x non nul :

$$atan(x) + atan(\frac{1}{x}) = \frac{\pi}{2}$$

Ainsi $\forall x \in [\sqrt{2}-1, \sqrt{2}+1[$, on utilisant la formule ci-dessus nous nous ramenons dans l'intervalle $[0, \sqrt{2}-1[$ pour un calcul plus efficace.
Pareil $\forall x \in [\sqrt{2} + 1, +\infty[$, en utlisant la même formule un plus raffinée ie $atan(x) + atan(\frac{1}{x}) = \frac{\pi}{2} \iff atan(x) = \frac{\pi}{4} + atan(\frac{x-1}{x+1})$, encore une fois on se raméne à l'intervalle contenant 0 c'est à dire pour notre cas $[0, \sqrt{2} - 1[$.

4.3 Validation de méthode atan

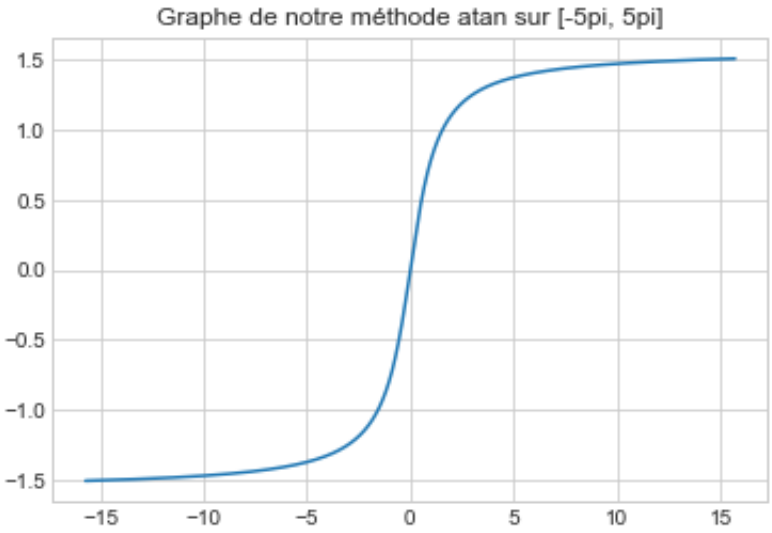
La validation de notre méthode atan a été faite en fonction des résultats de la fonction atan de java. Ce qui, nous a permis d'obtenir le tableau ci-dessous. A noter que le calcul de l'erreur en nombre d'ulp est fait selon la formule suivante :

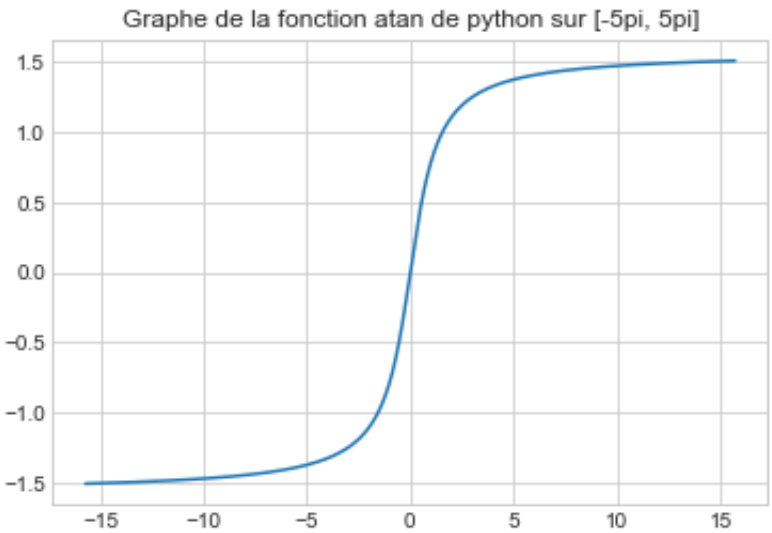
$$\frac{|atanDeca(x) - atanJava(x)|}{Ulp(atanJava(x)}$$)

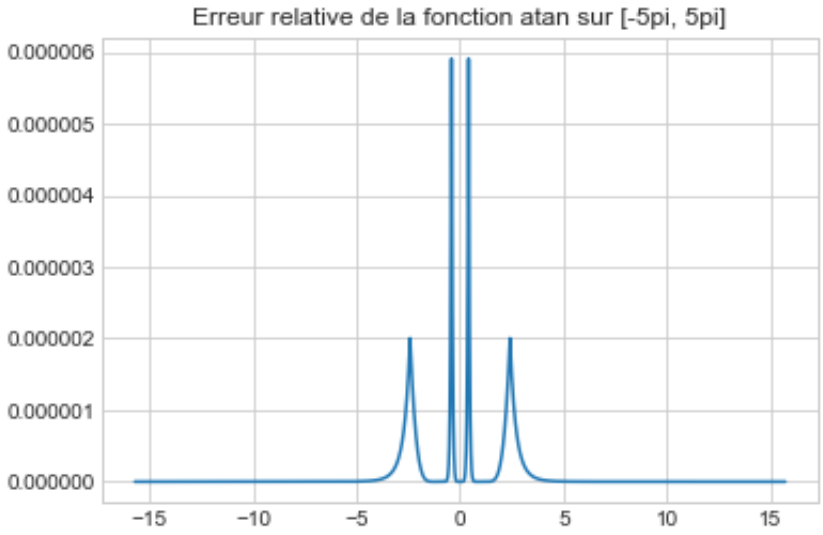
Intervalles	Nombres d'érreurs	Erreur moyenne	Erreur Max	pas	Nombre de tests
[0, 10]	86526	2.5658689295	21.0	2^{-14}	163 841
[10, 10 000]	468787	9.946524E-4	1	2^{-7}	1 278 721
[10 000,1 000 000]	715970	6.8431956386E-6	1	2^{-1}	1 980 001

On peut constater que pour de très grande valeur, notre fonction arctan est précise (avec pour erreur max égale exactement à 1 ulp) par contre pour des valeurs proches de 0 du fait de la très faible valeur de ulp en ces points, on a une erreur max de l'ordre de 21 ulp (mais on peut garantir qu'on a une précision de l'ordre de 10E-6).

On peut encore se convaincre de l'éfficacité, avec ses différents grpahes fait sur python.







En regardant le graphe d'erreur relative, on constate que notre algo garanti effectivement une précision de l'ordre de 10E-6

4.4 Réferences bibliographiques

Les liens utlisés pour notre documentation concernant la méthode atan sont les suivants :
— Lien wikipédia pour l'approxmation de Padé
— poly sur l'approximation de Padé
— Méthode pour trouver l'approximation de Padé de quelques fonctions

5 Arcsin

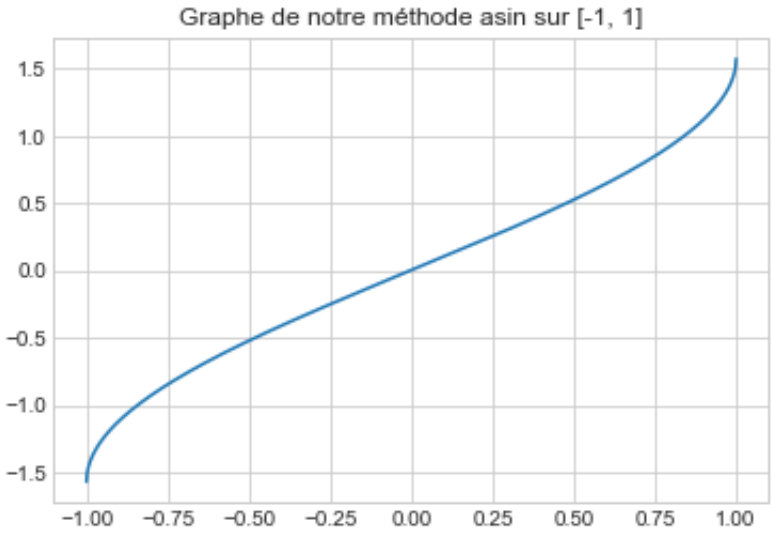
Compte tenu de la statisfaction qu'on a eu face aux résulats de la fonction $atan(x)$, nous avons d'utiliser encore une fois l'approximation de Padé comme détaillée dans la section atan

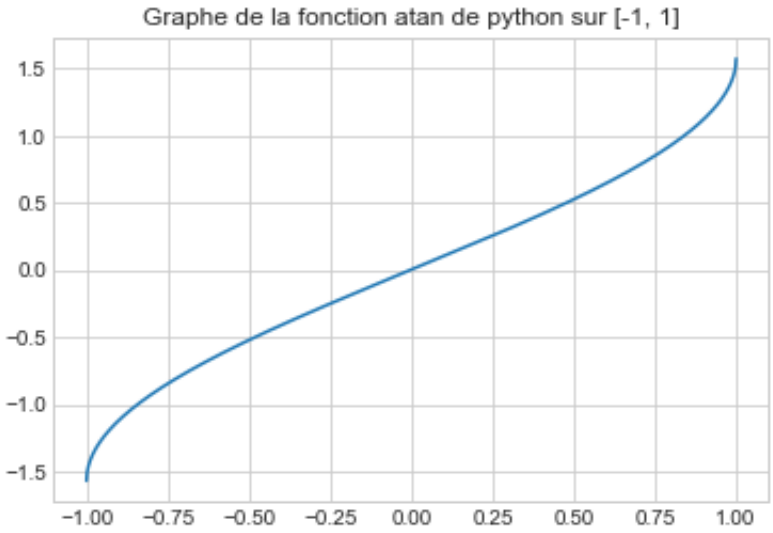
5.1 Implémentation de la méthode asin

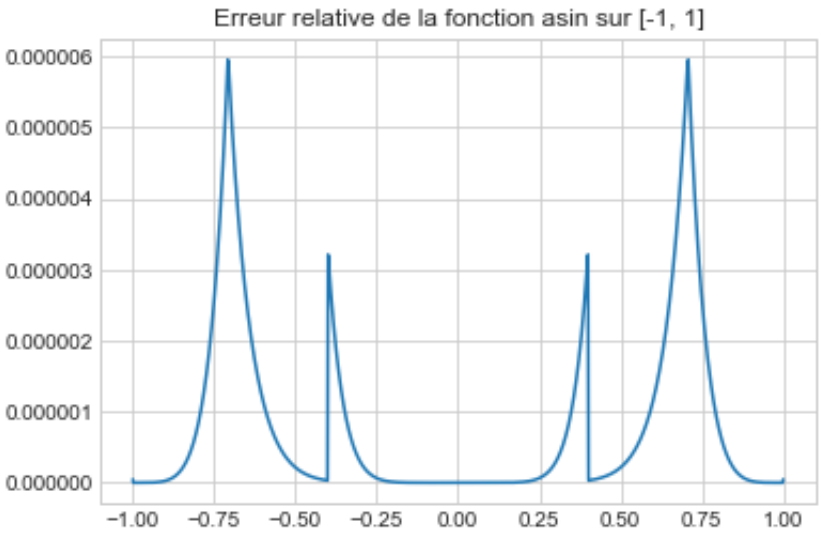
De manière analogue au travail fait dans la section *atan*, le but est de trouver un quotient de polynômes qui approche au mieux la fonction $asin(x)$.Pour la fonction $asin(x)$, nous décidé d'utliser une approximation de Padé [5/2] utilisant un développement limité à l'orde 7 de la fonction *asin*. Compte tenu de la précision qu'à la fonction *atan*, nous avons de l'utiliser dans l'implémentation de la fonction *asin*. Ainsi, nous avons partitionné notre domaine de définition $[-1, 1]$ en deux à savoir $[0, 0.4[$ et $[-0.4, 1]$ (en se focalise sur l'intervalle $[0, 1]$ par parité. Le but de cette partition est encore d'utiliser l'approximation de Padé au voisinage de zéro pour l'intervalle $[0, 0.4[$ et la rélation entre *asin* et *asin* à savoir $asin(x) = atan(\frac{x}{1+\sqrt{1-x^2}})$ est utilisé sur l'intervalle $[0.4, 1]$.

5.2 Validation de la méthode

Intervalles	Nombres d'érreurs	Erreur relative moyenne	Erreur relative Max	pas	Nombre de tests
[0.0 , 0.3]	711 400	1.1069499E-7	4.9364695E-7	2^{-22}	1 258 292
[0.3, 0.6]	74 312	6.9339506E-7	3.4045765E-6	2^{-18}	78 644
[0.6, 1.0]	354	1.7692203E-6	5.9919876E-6	2^{-10}	410







Au vue de tous ses résultats, nous pouvons garantir une précision à 6 chiffres pour la fonction *asin*, on aurait pu faire mieux si on prénait le développemenet de la fonction *asin* à un ordre plus grand que 7, intrésèquement on augmenterait la précision de notre approximation de Padé.