Grenoble INP – ENSIMAG

École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Rapport sur l'extension TRIGO du Projet Génie Logiciel

Implémentation des fonctions trigonométriques Ulp, Sin, Cos, Arcsin, Arctan

Groupe 01- GL 01

05 Janvier - 28 Janvier (23 jours)

Sommaire

1 Choix d'un algorithme

1.1 Approximation par les séries entières

Une première idée naïve était d'utiliser les développements en série entière des fonctions:

$$cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - ... + (-1)^n \frac{x^{2n}}{(2n)!} + ...$$

$$sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - ... + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + ...$$

Ces formules sont valables pour tout réel x. Ce sont des séries alternées, l'erreur commise est donc majorée en valeur absolue par le premier terme négligé.

En revanche, pour des valeurs comme x=100 ou x=200, l'évaluation est extrêmement laborieuse: plusieurs dizaines de secondes sont nécessaires, pour un résultat parfois faux. Cette première solution induit donc des pertes impressionnantes de chiffres significatifs. Pour notre extension, nous souhaitons un calcul rapide, précis et avec des fonctionnalités peu "gourmandes". C'est pourquoi nous n'avons pas retenu cette piste pour notre étude.

1.2 En passant par des intégrales

Les dérivées des fonctions arccos, arcsin ou arctan sont assez simples à intégrer. On pourrait se servir des formules suivantes:

$$arccos(x) = \int_1^x \frac{-1}{\sqrt{1-t^2}} dt$$

$$arcsin(x) = \int_1^x \frac{1}{\sqrt{1-t^2}} dt$$

$$arctan(x) = \int_0^x \frac{1}{1+t^2} dt$$

Les 2 premières formules sont valables pour x appartenant à]-1 ; 1[et la troisième est valable pour x appartenant à $\mathbf{R}$.

Vient néanmoins la question du moyen de calcul de ces intégrales, pour lequel on peut se tourner vers des principes connus comme la méthode des rectangles, des trapèzes ou encore de Simpson.

Malheureusement, après l'essai de ces méthodes sur nos machines (en Python), il est apparu que les résultats manquent souvent de précision; il faut prendre énormément de points pour obtenir une précision correcte, et il est évident que ceci n'est ni acceptable au vu du temps de calcul, ni acceptable au regard de l'énergie et de la mémoire consommée, trois points embarrassants vis-à-vis de la patience de l'utilisateur et du développement durable.

Pour des limites de temps et de moyens plus raisonnables, nous avons tourné notre regard vers les algorithmes utilisés par les premiers ordinateurs et les calculatrices.

1.3 Algorithme CORDIC

Dans un second temps, nous nous sommes penchés sur l'algorithme CORDIC (COrdinate Rotation DIgital Computer). Découvert en 1959 par Jack Volder, il a connu un succès retentissant. Il a été mis en oeuvre d'abord sur les ordinateurs, puis sur les premières calculatrices scientifiques, et il a été étendu à de nombreuses autres fonctions (fonction logarithme, hyperboliques, racine carrée, etc.).

Son avantage est énorme, notamment parce qu'il est en adéquation avec la structure de la machine. Il demande simplement que l'on ait mémorisé un certain nombre de valeurs de la fonction considérée, plus ou moins selon la précision souhaitée.

2 Principe de l'algorithme CORDIC

L'idée de l'algorithme CORDIC est de calculer $\tan(\theta)$ en faisant subir à un vecteur de coordonnées $\begin{pmatrix} X \\ Y \end{pmatrix}$ des rotations d'angles de plus en plus petits, tendant vers 0, et dont la somme est égale à θ. Le quotient $\frac{Y}{X}$ est alors clairement une approximation de $\tan(\theta)$ par définition de la tangente.

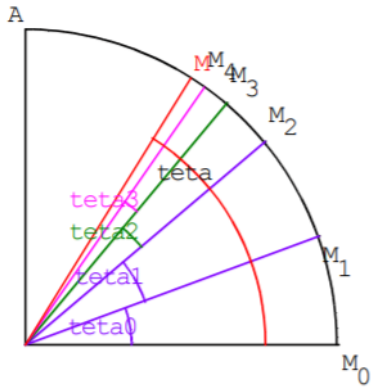

Figure 1: Différentes étapes pour s'approcher d'un angle θ

On se place dans un repère orthonormé direct (O, i, j). On considère un angle θ de l'intervalle $]0\ ;\ \frac{\pi}{2}[$ et on cherche à calculer $\tan(\theta)$. On procède par rotations successives: on part d'un point M_0 (X_0=1, Y_0=0) et par des

rotations successives d'angles à préciser, on cherche à s'approcher d'un point M de coordonnées (X,Y) ayant pour angle θ. On aura alors $\tan(\theta) = \frac{Y}{X}$.

On considère ensuite une suite décroissante de θ_k, c'est-à-dire telle que $\theta_0 \geq \theta_1 \geq ... \geq \theta_{n-1} \geq \theta_n$ avec $\theta = \sum_{j=0}^{n} \theta_j$, où n est un entier naturel.

Soit le point M0 de coordonnées $\begin{pmatrix} X_0 \\ Y_0 \end{pmatrix} = \begin{pmatrix} X_0 \\ 0 \end{pmatrix}$. Alors le point M1, image de M0 par la rotation de centre 0 et d'angle θ_0, a pour coordonnées $\begin{pmatrix} X_1 \\ Y_1 \end{pmatrix}$ vérifiant

$$\begin{cases} X_1 = X_0 \times cos(\theta_0) \\ Y_1 = X_0 \times sin(\theta_0) \end{cases}$$

On applique les rotations successives avec les différents angles θ_j. De plus, on peut montrer par récurrence la relation suivante:

$$\begin{cases} X_k = X_0 \times cos(\theta_0 + \theta_1 + ... + \theta_{k-1}) \\ Y_1 = X_0 \times sin(\theta_0 + \theta_1 + ... + \theta_{k-1}) \end{cases}$$

On obtient alors la relation suivante:

$$\frac{Y_{n+1}}{X_{n+1}} = \frac{X_0 \times sin(\theta_0 + \theta_1 + ... + \theta_n)}{X_0 \times cos(\theta_0 + \theta_1 + ... + \theta_n)} = tan(\theta)$$

Ainsi, $\tan(\theta)$ ne dépend pas de la coordonnée X0 initiale. On pourra donc prendre X0=1 dans un premier temps.
Il faut désormais choisir les θ_j judicieusement. En effet, un choix particulier des angles θ_j permet de simplifier grandement les calculs. Nous avons le choix entre la base 10 et la base 2. Si nous travaillons en base 10 par exemple, nous aurons $\theta_j = arctan(10^{-j})$ et si nous travaillons en base 2 nous aurons $\theta_j = arctan(2^{-j})$. Nous verrons en partie 6)1) les éléments qui nous ont permis de choisir de travailler en base 10 ou en base 2.

Pour les approximations de *sin* et *cos*, nous avons choisi la base 10, c'est pourquoi nous avons pris les θ_j de manière à ce que $\tan(\theta_j) = 10^{-j}$. Dès lors, le passage de M_k à M_{k+1} s'obtient uniquement à l'aide de calculs élémentaires. On obtient les relations suivantes:

$$\begin{cases} X_{k+1} = X_k - 10^{-k} \times Y_k \\ Y_{k+1} = Y_k + 10^{-k} \times X_k \end{cases}$$

Ces calculs sont simples à mettre en oeuvre informatiquement étant donné qu'ils ne nécessitent que des additions, des soustractions et de simples décalages de virgules. Il faut cependant avoir préalablement en mémoire les premières valeurs possibles des θ_k. A partir de k > 6, on pourra prendre $\theta_k = 10^{-k}$ avec l'équivalent en 0 : $\tan(\theta) \sim \theta$.

Table de valeurs à choisir pour les angles θ_k.

k	10^{-k}	$\theta_k = \text{Arctan}(10^{-k})$
0	1	0,78539816339744830962
1	0,1	0,09966865249116202737 8
2	0,01	0,00999966668666523820 63
3	0,001	0,00099999966666686666652
4	0,0001	0,00009999999966666666668667
5	0,00001	0,00000999999999966666666667
6	0,000001	0,00000099999999999966666667

Figure 2: Table des premières valeurs des angles θ_k

Le tableau ci-dessus donne les premières valeurs à choisir pour θ_k, à 10^{-20} près, exprimées en radians:

Il suffit donc de rentrer ces valeurs à l'avance dans la mémoire afin de les utiliser pour les calculs de cos, sin, tan.

Une idée d'algorithme est présentée à la fin de ce rapport, il permet d'obtenir $\tan(\theta)$. On pourra ensuite obtenir $\cos(\theta)$ ou $\sin(\theta)$ grâce aux formules suivantes:

$$cos(\theta) = \frac{1}{\sqrt{1 + tan^2(\theta)}}$$

$$sin(\theta) = \frac{tan(\theta)}{\sqrt{1 + tan^2(\theta)}}$$

On remarque aussi que cet algorithme n'est valable que pour des angles θ tels que $0 < \theta < \frac{\pi}{2}$. Pour les autres angles, on se ramène à cet intervalle à l'aide de relations issues de la symétrie des courbes trigonométriques comme $\cos(-\theta) = \cos(\theta)$ ou $\sin(\pi - \theta) = \sin(\theta)$.

Après en avoir testé l'implémentation en Python sur un vaste échantillon de valeurs, nous avons décrété que les résultats obtenus étaient suffisamment cohérents et fiables en comparaison aux valeurs données par les fonctions inhérentes à la bibliothèque math de Python, et c'est pourquoi nous avons retenu cette solution pour notre étude.

3 CORDIC inverse pour les fonctions réciproques

L'algorithme CORDIC étudié précédemment permet d'accéder à la tangente, ensuite nous pouvons aisément en déduire les valeurs de cos et sin. Toutefois, il est impossible de l'employer pour arccos, arcsin ou arctan. Nous avons donc fait appel à un autre algorithme : Le CORDIC inverse.

Ce nouvel algorithme est similaire au CORDIC vu précédemment. En fait, nous allons calculer arctan(x) en partant d'un vecteur de coordonnées $\begin{pmatrix} x \\ 1 \end{pmatrix}$ et en effectuant des rotations d'angles de plus en plus petits, la somme de tous ces angles donnera une valeur approchée de θ tel que tan(θ) = x, c'est-à-dire tel que arctan(x)=θ.

La coordonnée Y=1 (au départ) va tendre petit à petit vers 0 et une fois qu'on sera "suffisamment proche" de 0, l'algorithme retournera la valeur de l'angle θ vérifiant θ=arctan(x).

On se place dans le même repère orthonormé direct (O, i, j). On considère un vecteur de coordonnées $\begin{pmatrix} x \\ 1 \end{pmatrix}$. Attention cependant à ce que son argument soit toujours compris dans l'intervalle $]\frac{-\pi}{2}\ ;\ \frac{\pi}{2}[$.
On procède par rotations successives: on part d'un point M_0 (X_0=x, Y_0=1) et par des rotations d'angles à préciser, on cherche à s'approcher d'un point M de coordonnées (Z,0) (la valeur de Z est sans importance, seul le fait que Y=0 importe). On aura alors $\theta = \sum_{j=0}^{n} \theta_j$, avec les θ_j qui correspondent à tous les angles utilisés lors des rotations successives.

A l'instar du CORDIC classique, il faut choisir les θ_j judicieusement: ceux-ci doivent satisfaire

$$tan(\theta_j) = 2^{-j}$$

Désormais, on obtient les relations suivantes pour passer de M_k à M_{k+1} :

$$\begin{cases} X_{k+1} = X_k - 2^{-k} \times Y_k \\ Y_{k+1} = Y_k + 2^{-k} \times X_k \end{cases}$$

Ces calculs ne nécessitent que des additions, des soustractions et de simples décalages de virgule. Il faut cependant avoir préalablement en mémoire les premières valeurs possibles des θ_j, stockées comme pour le CORDIC classique dans une liste.

Le tableau de la Figure 3 donne les étapes successives de l'algorithme CORDIC inverse. Le point de départ est (2, 7) ce qui revient à calculer arctan($\frac{2}{7}$). Le résultat est en degré mais notre algorithme fonctionnera avec des radians car c'est ce qui est spécifié dans le cahier des charges.

i	L	real arg. a_i	imag. arg.b_i	b_i>0? --> sign	k_i	atan(k_i) in °	"+/- 90 + Σ(atan(k_i))
1		2	7	-1			-90
2	0	7	-2	1	1	45	-45
3	1	9	5	-1	-0,5	-26,5650512	-71,56505118
4	2	11,5	0,5	-1	-0,25	-14,0362435	-85,60129465
5	3	11,625	-2,375	1	0,125	7,12501635	-78,4762783
6	4	11,921875	-0,921875	1	0,0625	3,57633437	-74,89994392
7	5	11,97949219	-0,17675781	1	0,03125	1,78991061	-73,11003331
8	6	11,98501587	0,19760132	-1	-0,015625	-0,89517371	-74,00520702
9	7	11,98810339	0,01033545	-1	-0,0078125	-0,44761417	-74,45282119
10	8	11,98818414	-0,08332161	1	0,00390625	0,2238105	-74,22901069
11	9	11,98850961	-0,03649277	1	0,00195313	0,11190568	-74,11710502
12	10	11,98858089	-0,01307771	1	0,00097656	0,05595289	-74,06115212
13	11	11,98859366	-0,00137011	1	0,00048828	0,02797645	-74,03317567
14	12	11,98859433	0,00448369	-1	-0,00024414	-0,01398823	-74,0471639
15	13	11,98859542	0,00155679	-1	-0,00012207	-0,00699411	-74,05415801
		verific.	tan(φ) = b_i/a_i	atan(φ)			
			3,5	74,0546041			

Figure 3: Tableau des étapes du CORDIC inverse

Ce tableau se lit de gauche à droite et de haut en bas. En effet, on souhaite accéder à $arctan(3.5)$ or on peut décomposer 3.5 en quotient $\frac{7}{2}$. Dans le plan complexe, le numérateur 7 va être associé à la partie imaginaire et le dénominateur 2 va être associé à la partie réelle. Ensuite, on regarde si la partie imaginaire $b_i > 0$, si oui signe=-1 et sinon signe=1. Dans notre cas, signe=-1 sur la première ligne donc on ajoute -90 degrés sur la dernière ligne qui correspond à la somme de tous les angles pour arriver à $arctan(3.5)$. On modifie ensuite les coordonnées a_i et b_i en appliquant une rotation de -90 degrés au complexe 2+7i ce qui nous donne 7-2i, c'est-à-dire les coordonnées de la ligne suivante (la ligne 2).

On réitère ce procédé sur la ligne 2, $-2 < 0$ donc signe = 1. On remarque que sur cette ligne ki=1 ($signe \times \frac{1}{2^0}$) et que $arctan(1) = 45$ donc l'angle à ajouter à la grande somme pour cette étape sera 45 degrés. La grande somme de la dernière colonne vaut donc $-90 + (signe \times 45) = -90 + 45 = -45$ degrés. on modifie ensuite les coordonnées a_i et b_i en appliquant une rotation de +45 degrés au complexe 7-2i ce qui nous donne 9+5i, c'est-à-dire les coordonnées de la ligne 3.

On réitère ce procédé pour la ligne 3 avec le complexe 9+5i, $5 > 0$ donc signe = -1. Pour cette ligne ki=-0.5 ($signe \times \frac{1}{2^1}$) et $arctan(-0.5) = -26.5850512$ donc l'angle à ajouter à la grande somme pour cette étape sera -26.5850512 degrés. La grande somme de la dernière colonne vaut donc $-90 + 45 - 26.5850512 = -71.58505118$.

En réitérant ce procédé sur un grand nombre de ligne, nous allons approcher la valeur de $arctan(\frac{7}{2})$ qui correspond à environ 74.05415801 degrés sur la dernière colonne de la dernière ligne (il faut prendre l'opposé de la grande somme). Une version détaillée de notre algorithme est fournie dans les annexes.

Cette deuxième version de l'algorithme CORDIC inverse permet d'obtenir $arctan(x)$. On pourra ensuite obtenir $arccos(x)$ ou $arcsin(x)$ grâce aux formules suivantes:

$$arcsin(x) = arctan(\frac{x}{\sqrt{1-x^2}})$$

Le cas de la fonction $arccos$ est plus délicat:
si $-1 \leq x < 0$ alors :

$$arccos(x) = \pi + arctan(\frac{\sqrt{1-x^2}}{x})$$

si $0 < x \leq 1$ alors :

$$arccos(x) = arctan(\frac{\sqrt{1-x^2}}{x})$$

Une implémentation en Python nous a convaincu de la cohérence et de la fiabilité de cet algorithme, et c'est pourquoi nous avons retenu cette solution pour notre étude.

4 Calcul de l'ULP

La définition de l'ulp que nous avons décidé d'utiliser est celle du pdf du sujet, c'est-à-dire que notre implémentation doit correspondre à celle de java (Math.ulp()).

Notre approche a alors été simple, nous avons tenté d'imiter l'implémentation de java. Pour cela nous avons trouvé le code source de l'implémentation java.

En java, la technique utilisé est d'accéder à l'exposant avec des shifts, comme on peut le voir sur l'image ci dessous.

```
/**
 * Returns a floating-point power of two in the normal range.
 */
static float powerOfTwo(int n) {
    assert(n >= Float.MIN_EXPONENT && n <= Float.MAX_EXPONENT);
    return Float.intBitsToFloat(((n + FloatConsts.EXP_BIAS) <<
                                 (FloatConsts.SIGNIFICAND_WIDTH-1))
                                & FloatConsts.EXP_BIT_MASK);
}
```

Figure 4: shift en java

Malheureusement, il est impossible, à notre connaissance, d'accéder directement aux "champs" de la représentation IEEE754 en deca ou même avec une instruction de la machine abstraite. Cette représentation est nécessaire pour accéder à l'exposant, qui est nécessaire pour trouver l'ulp.

Il a donc été nécessaire de développer un algorithme pour retrouver l'exposant, ce qui coûte du temps de calcul. L'idée est simplement de:

- diviser le nombre par des puissances de deux de plus en plus grandes s'il est positif

- diviser le nombre par des puissances de deux de plus en plus petites s'il est négatif

Vous pouvez voir l'algorithme ci dessous.

```
int _trouveExposant(float a, Math m) {
    float fractionPart;
    int exponent;
    if(a >= 1.0) {
        exponent =0;
        fractionPart =a/m._puissanceDe2(exponent);
        while(fractionPart>=2.0){
            exponent = exponent +1 ;
            fractionPart =a/m._puissanceDe2(exponent);
        }
    }else {
        exponent =-1;
        fractionPart =a/m._puissanceDe2(exponent);
        while(fractionPart<1.0){
            exponent = exponent -1 ;
            fractionPart =a/m._puissanceDe2(exponent);
        }

    }
    return exponent;
}
```

Figure 5: code pour trouver l'exposant en deca

Cependant, cela nécessite d'implémenter une fonction qui calcule les puissances de 2. Hors, pour des questions d'optimisation, cela a été implémenté avec des shifts (instruction SHL) Cependant, les shifts ne fonctionnent qu'avec des entiers. Les entiers sur 32 bits signés peuvent aller jusqu'à 2^{31} c'est à dire $\approx 2 \times 10^9$ Pour cette raison, l'ULP ne fonctionne pas sur des nombres supérieurs à 2×10^9 (crash à l'exécution) Nous admettons avec honnêteté que ce bug nous est passé sous le nez car nous n'avons pas testé la fonction ulp sur des valeurs suffisamment grandes. Par "chance" nous avons repéré ce bug avant la soutenance, mais pas avant les rendus.

5 Précision de l'extension

Pour évaluer la précision de l'extension nous allons comparer avec une implémentation de référence, telle que celle de python par exemple. Pour simplifier l'analyse on va se concentrer sur les puissances de 10 jusqu'à 10^9.

Nous avons développé un programme deca dans lequel nous avons rentré manuellement les valeurs correctes (venant de Python), et nous comparons ces valeurs correctes avec la sortie de nos fonctions. Sur les captures d'écran suivantes vous voyez à gauche l'entrée de la fonction, au milieu l'erreur par rapport à la valeur attendue et à droite l'ulp. Nous n'avons pas inclus l'ulp pour arcSin et arcTan car les valeurs testées sont toutes proches de 1.

```
[cathelib@ensipc276 extension]$ ima cos_range.a
1.00000e+09: -1.35767e+00 : ulp :  6.40000e+01
5.00000e+08: 4.68608e-01 : ulp :  3.20000e+01
1.00000e+08: 1.03080e+00 : ulp :  8.00000e+00
5.00000e+07: 3.48890e-01 : ulp :  4.00000e+00
1.00000e+07: 1.50426e-01 : ulp :  1.00000e+00
5.00000e+06: -1.33355e-01 : ulp :  5.00000e-01
1.00000e+06: -1.01008e-02 : ulp :  6.25000e-02
5.00000e+05: 2.56944e-03 : ulp :  3.12500e-02
1.00000e+05: 1.03354e-04 : ulp :  7.81250e-03
5.00000e+04: -1.39127e-03 : ulp :  3.90625e-03
1.00000e+04: -8.49366e-05 : ulp :  9.76563e-04
5.00000e+03: -1.37135e-04 : ulp :  4.88281e-04
1.00000e+03: 2.31862e-05 : ulp :  6.10352e-05
5.00000e+02: -6.49691e-06 : ulp :  3.05176e-05
1.00000e+02: -1.25170e-06 : ulp :  7.62939e-06
5.00000e+01: -3.57628e-07 : ulp :  3.81470e-06
1.00000e+01: -2.38419e-07 : ulp :  9.53674e-07
precision max = -1.35767e+00
```

Figure 6: précision pour cos

```
1.00000e+09: -1.40014e+00 : ulp :  6.40000e+01
5.00000e+08: 1.15642e+00 : ulp :  3.20000e+01
1.00000e+08: -1.67632e+00 : ulp :  8.00000e+00
5.00000e+07: -1.23343e+00 : ulp :  4.00000e+00
1.00000e+07: 2.33047e-01 : ulp :  1.00000e+00
5.00000e+06: 3.93006e-02 : ulp :  5.00000e-01
1.00000e+06: -2.59286e-02 : ulp :  6.25000e-02
5.00000e+05: 1.36743e-02 : ulp :  3.12500e-02
1.00000e+05: 2.78083e-03 : ulp :  7.81250e-03
5.00000e+04: 2.58684e-05 : ulp :  3.90625e-03
1.00000e+04: 2.64704e-04 : ulp :  9.76563e-04
5.00000e+03: -2.15173e-05 : ulp :  4.88281e-04
1.00000e+03: -1.57356e-05 : ulp :  6.10352e-05
5.00000e+02: 1.22488e-05 : ulp :  3.05176e-05
1.00000e+02: -2.14577e-06 : ulp :  7.62939e-06
5.00000e+01: -1.46031e-06 : ulp :  3.81470e-06
precision max = -1.67632e+00
```

Figure 7: précision pour sin

```
1.00000e+03: 0.00000e+00 ok
1.00000e+02: 1.19209e-07
1.00000e+01: 1.19209e-07
7.00000e+00: 0.00000e+00 ok
4.00000e+00: 1.19209e-07
3.00000e+00: 1.19209e-07
2.00000e+00: -1.19209e-07
1.00000e+00: 5.96046e-08
8.00000e-01: 5.96046e-08
6.00000e-01: 5.96046e-08
5.00000e-01: 5.96046e-08
4.00000e-01: 2.98023e-08
3.00000e-01: 2.98023e-08
2.00000e-01: 2.98023e-08
1.50000e-01: 1.49012e-08
1.00000e-01: 2.98023e-08
5.00000e-02: 1.86265e-08
0.00000e+00: 0.00000e+00 ok
-1.00000e+03: 0.00000e+00 ok
-1.00000e+02: -1.19209e-07
-1.00000e+01: -1.19209e-07
-7.00000e+00: 0.00000e+00 ok
-4.00000e+00: -1.19209e-07
-3.00000e+00: -1.19209e-07
-2.00000e+00: 1.19209e-07
-1.00000e+00: -5.96046e-08
-8.00000e-01: -5.96046e-08
-6.00000e-01: -5.96046e-08
-5.00000e-01: -5.96046e-08
-4.00000e-01: -2.98023e-08
-3.00000e-01: -2.98023e-08
-2.00000e-01: -2.98023e-08
-1.50000e-01: -1.49012e-08
-1.00000e-01: -2.98023e-08
-5.00000e-02: -1.86265e-08
precision max = 1.19209e-07
```

Figure 8: précision pour ArcTan

```
[cathelib@ensipc276 gl01]$ ima
0.00000e+00: 0.00000e+00 ok
1.00000e-11: 4.84812e-08
1.00000e-08: 4.84812e-08
1.00000e-03: 4.81959e-08
1.00000e-01: 8.94070e-08
1.50000e-01: 2.98023e-08
2.00000e-01: 4.47035e-08
2.50000e-01: 1.19209e-07
3.00000e-01: -2.98023e-08
3.50000e-01: 2.98023e-08
4.00000e-01: 1.78814e-07
4.50000e-01: 1.78814e-07
5.00000e-01: 1.19209e-07
5.50000e-01: 5.96046e-08
6.00000e-01: 5.96046e-08
6.50000e-01: -5.96046e-08
7.00000e-01: -1.78814e-07
7.50000e-01: 5.36442e-07
8.00000e-01: 7.15256e-07
8.51000e-01: -1.19209e-07
8.50000e-01: 1.43051e-06
8.52000e-01: 1.19209e-07
9.00000e-01: -2.38419e-07
9.50000e-01: -1.19209e-07
9.90000e-01: 1.78814e-06
1.00000e+00: -1.19209e-07
-1.00000e-11: -4.84812e-08
-1.00000e-01: -8.94070e-08
-2.00000e-01: -4.47035e-08
-3.00000e-01: 2.98023e-08
-4.00000e-01: -1.78814e-07
-5.00000e-01: -1.19209e-07
-6.00000e-01: -5.96046e-08
-7.00000e-01: 1.78814e-07
-8.00000e-01: -7.15256e-07
-9.00000e-01: 2.38419e-07
-9.90000e-01: -1.78814e-06
-1.00000e+00: 1.19209e-07
precision max = 1.78814e-06
[cathelib@ensipc276 gl01]$
```

Figure 9: précision pour Arcsin

Comme on peut le voir sur l'image ci-dessus, pour cos et sin qui sont les fonctions les plus dures à implémenter précisément du fait de leur grand domaine de définition ($\mathbb{R}$), l'erreur est toujours dans l'ordre de grandeur de l'ulp. Pour ArcTan et ArcSin, la précision est très bonne et l'erreur est de l'ordre d'un ulp également.

Une conclusion possible est que notre extension a une bonne précision et que nous n'aurions pas pu faire beaucoup mieux à cause du manque de précision intrinsèque au format IEEE754 simple précision. Toutefois, la précision actuelle n'est pas garantie d'être 1 ulp au maximum (ce qui est théoriquement possible mais très compliqué).

6 Explication détaillée de chaque algorithme

6.1 Sinus et Cosinus

6.1.1 Algorithme et choix de conception

Les fonctions Sinus et Cosinus sont basées sur l'algorithme CORDIC classique étudié dans la partie 2. L'objectif de cet algorithme était d'avoir la meilleure approximation possible de la fonction Sinus ou Cosinus. Cependant, nous nous sommes heurté à 3 problèmes majeurs lors de la conception de cet algorithme.

Tout d'abord, on remarque que cet algorithme n'est valable que pour des angles θ tels que $0 < \theta < \frac{\pi}{2}$. Il fallait donc trouver un moyen de traiter tous les angles du cercle trigonométrique. Nous avons donc décomposer le cercle en 4 zones comme sur la Figure 10 ci-dessous.

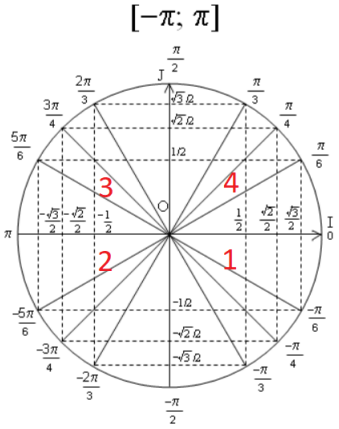

Figure 10: Division du cercle trigo en 4 parties

Notre première problématique consistait donc à traiter tous les points du cercle trigonométrique, en d'autres termes, il fallait trouver un moyen de ramener un angle des zones 1, 2, 3 à la zone 4 pour lui appliquer l'algorithme. Cela devenait encore plus compliqué lorsque l'angle en question n'apparaissait même pas sur le cercle trigonométrique (par exemple pour $sin(10)$ ou $sin(100)$). Il fallait aussi prévoir un algorithme de *range reduction* pour lui appliquer un modulo 2π et le ramener dans l'intervalle $]-\pi\ ;\ \pi[$.

Notre algorithme de *range reduction* fonctionne de la manière suivante: Admettons qu'on souhaite calculer $sin(x)$ avec x=100, il va falloir ramener ce nombre dans le cercle trigonométrique en lui appliquant un ou plusieurs modulos 2π.
On sait que $x = 2k\pi + r$ avec r appartenant à $[0\ ;\ 2\pi[$. On divise dans un premier temps x par 2π (dans notre cas on obtient 15,91).
On prend ensuite la partie entière de 15,91 = 15. On va ainsi obtenir le reste r en soustrayant $15 \times 2\pi$.
Il nous restera r appartenant à $[0\ ;\ 2\pi[$, c'est-à-dire sur le cercle trigonométrique. On pourra donc appliquer l'algorithme CORDIC à ce reste r pour en déduire $sin(x)$.

Dans le cadre de notre conception, nous avons inclus une méthode *range reduction* dans le module Math.decah, le code est disponible sur la Figure 11 ci-dessous:

```
float _rangeReduction(float angle ){
    float pi;
    float quotient;
    float integerQuotient;
    float deuxPi;
    _Pi piClass;
    piClass=new _Pi();
    pi = piClass.getPi();
    deuxPi = (2 * pi);
    quotient = angle / deuxPi;
    integerQuotient = (int)(quotient);
    angle = this._fma(angle,-integerQuotient,deuxPi);
    if(angle < -pi){
        while(angle < -pi){
            angle = this._fma(angle,2,pi);
        }
    }
    if(angle > pi){
        while(angle > pi) {
            angle = this._fma(angle,-2,pi);
        }
    }

    return angle;
}
```

Figure 11: Algorithme de "range reduction" en Deca

Une fois que l'angle se trouvait dans l'intervalle $]-\pi\ ;\ \pi[$ il restait encore à déterminer s'il appartenait à la zone 1, 2, 3 ou 4 de la Figure 10. Pour cela, nous avons mis en place une disjonction de cas. Par exemple, pour l'algorithme du Sinus nous avons utilisé quelques propriétés de la fonction *sin* comme l'imparité. Les 4 cas en question pour la fonction Sinus sont décrits dans les commentaires de la Figure 12 ci-dessous:

```
689
690         /* angle est désormais dans [-pi, pi] */
691         if(-(pi / 2) < angle && angle < 0.0){   /* zone 1 en bas a droite */
692           /* sin(-x) = - sin(x)
693            on fait le cordic(-angle) car -angle est dans [0, pi/2] et
694            on prend l'opposé du résultat (* -1) */
695             angle = -angle;
696             oppose = -1;
697         }
698         else if(-pi < angle && angle < -(pi / 2)){ /* zone 2 en bas a gauche */
699           /* sin(x+pi) = -sin(x)
700           on rajoute pi à l'angle donc il se retrouve dans [0, pi/2] et
701           on prend l'opposé du résultat (* -1) */
702             angle = angle + pi;
703             oppose = -1;
704         }
705         else if(angle < pi && (pi / 2) < angle){  /* zone 3 en haut à gauche */
706           /* sin(-x + pi) = sin(x)
707           on prend l'opposé de l'angle et on lui ajoute pi donc il se
708           retrouve dans [0, pi/2] et on prend le résultat (* 1) */
709             angle = pi - angle;
710             oppose = 1;
711         }
712         else{   /* zone 4 en haut à droite */
713             oppose = 1;
714         }
```

Figure 12: Disjonction des 4 cas pour Sinus

Nous avons procédé de la même manière pour Cosinus, nous avons fait une disjonction de cas en fonction des 4 "zones" du cercle trigonométriques et grâce aux propriétés de la fonction Cosinus, notamment sa parité, nous sommes parvenus à ramener l'angle θ de façon à ce que $0 < \theta < \frac{\pi}{2}$. Ainsi on pourra lui appliquer l'algorithme CORDIC. Les 4 cas en question pour la fonction Cosinus sont décrits dans les commentaires de la Figure 13 ci-dessous:

```
761
762         /* angle est désormais dans [-pi, pi] */
763         if(-(pi / 2) < angle && angle < 0.0){   /* zone 1 en bas a droite */
764           /* cos(-x) = cos(x)
765           on fait le cordic(-angle) car -angle est dans [0, pi/2] et
766           on prend le même résultat (* 1) */
767             angle = -angle;
768             oppose = 1;
769         }
770         else if(-pi < angle && angle < -(pi / 2)){ /* zone 2 en bas a gauche */
771           /* cos(x+pi) = -cos(x)
772           on rajoute pi à l'angle donc il se retrouve dans [0, pi/2] et
773           on prend l'opposé du résultat (* -1) */
774             angle = angle + pi;
775             oppose = -1;
776         }
777         else if(angle < pi && (pi / 2) < angle){  /* zone 3 en haut à gauche */
778           /* cos(-x + pi) = cos(x)
779           on prend l'opposé de l'angle et on lui ajoute pi donc il se
780           retrouve dans [0, pi/2] et on prend le résultat (* -1) */
781             angle = pi - angle;
782             oppose = -1;
783         }
784         else{    /* zone 4 en haut à droite */
785             oppose = 1;
786         }
```

Figure 13: Disjonction des 4 cas pour Cosinus

Le second problème rencontré concernait les constantes de l'algorithme CORDIC. Nous n'avions aucun moyen de les stocker et d'y accéder étant donné que nous n'avions pas de liste ou de LinkedList comme en Java. Nous nous sommes alors dirigés vers une fameuse structure que nous avons découverte l'année dernière à l'ENSIMAG: les listes chaînées.
Nous avons défini la structure d'une liste chaînée dans le fichier Math.decah avec plusieurs cellules. Chacune d'elle possède un contenu, une cellule suivante vers laquelle elle pointe (null si c'est la dernière cellule de la liste) et une valeur k permettant de calculer les coordonnées X et Y du point M durant les calculs. La structure de notre liste chaînée est décrite sur la Figure 14 avec la syntaxe de Deca.

```
class _Liste{
    float contenu;
    _Liste suivant;
    float valeur_k;

    void init(float contenu, _Liste suivant, float valeur_k){
        this.contenu = contenu;
        this.suivant = suivant;
        this.valeur_k = valeur_k;
    }

}

```

Figure 14: Structure de la liste chaînée utilisée

Une fois cette structure implémentée, nous l'avons utilisée pour stocker les constantes nécessaires lors des calculs de l'algorithme CORDIC. Dans un premier temps, on déclare de nouvelles listes sur la Figure 15. Ensuite, on les initialise avec le mot "new" comme sur la Figure 16.

```
class _ListeRemplie {
    _Liste a0;
    _Liste a1;
    _Liste a2;
    _Liste a3;
    _Liste a4;
    _Liste a5;
    _Liste a6;
    _Liste a7;
    _Liste a8;
    _Liste a9;
    _Liste a10;
    _Liste a11;
    _Liste a12;
    _Liste a13;
    _Liste a14;
    _Liste a15;
```

Figure 15: Déclaration des listes

```
    void initListe(){
        a0 = new _Liste();
        a1 = new _Liste();
        a2 = new _Liste();
        a3 = new _Liste();
        a4 = new _Liste();
        a5 = new _Liste();
        a6 = new _Liste();
        a7 = new _Liste();
        a8 = new _Liste();
        a9 = new _Liste();
        a10 = new _Liste();
        a11 = new _Liste();
        a12 = new _Liste();
        a13 = new _Liste();
        a14 = new _Liste();
        a15 = new _Liste();
```

Figure 16: Initialisation avec "new"

Les Figures 17 et 18 décrivent comment nous avons rentré ces valeurs dans le fichier Math.decah. Les valeurs en question étaient calculées à l'avance par Python et nous avons mis en place un programme capable de remplir les listes

automatiquement afin d'économiser du temps et de l'énergie.

```
338     void sinCosRemplissage(){
339         a44.init(1.0e-44, null, 1.0e-44);
340         a43.init(1.0e-43, a44, 1.0e-43);
341         a42.init(1.0e-42, a43, 1.0e-42);
342         a41.init(1.0e-41, a42, 1.0e-41);
343         a40.init(1.0e-40, a41, 1.0e-40);
344         a39.init(1.0e-39, a40, 1.0e-39);
345         a38.init(1.0e-38, a39, 1.0e-38);
346         a37.init(1.0e-37, a38, 1.0e-37);
347         a36.init(1.0e-36, a37, 1.0e-36);
348         a35.init(1.0e-35, a36, 1.0e-35);
349         a34.init(1.0e-34, a35, 1.0e-34);
350         a33.init(1.0e-33, a34, 1.0e-33);
```

Figure 17: Remplissage des listes

```
371         a12.init(1.0e-12, a13, 1.0e-12);
372         a11.init(1.0e-11, a12, 1.0e-11);
373         a10.init(1.0e-10, a11, 1.0e-10);
374         a9.init(1.0e-09, a10, 1.0e-09);
375         a8.init(1.0e-08, a9, 1.0e-08);
376         a7.init(9.9999999999966e-08, a8, 1.0e-07);
377         a6.init(9.999999999966666e-07, a7, 1.0e-06);
378         a5.init(9.99999999666668e-06, a6, 1.0e-05);
379         a4.init(9.9999996666666e-05, a5, 0.0001);
380         a3.init(0.0009999966666668668, a4, 0.001);
381         a2.init(0.009999666686665238, a3, 0.01);
382         a1.init(0.09966865249116204, a2, 0.1);
383         a0.init(0.785398163974483, a1, 1.0);
384     }
```

Figure 18: Utilisation des constantes

La troisième difficulté rencontrée concernait le passage de la fonction *tan* aux fonctions *sin* ou *cos*. En effet, l'algorithme CORDIC permet d'approcher la tangente d'un angle, et nous comptions nous servir des formules de la partie 2 pour obtenir des approximations de $cos(\theta)$ et $sin(\theta)$. Pour rappel, ces formules sont:

$$cos(\theta) = \frac{1}{\sqrt{1 + tan^2(\theta)}}$$

$$sin(\theta) = \frac{tan(\theta)}{\sqrt{1 + tan^2(\theta)}}$$

Le gros problème de ces formules est qu'elles utilisent une racine carré, or nous ne disposons pas d'un tel opérateur en Deca. C'est pourquoi, nos recherches bibliographiques et nos calculs nous ont menés aux formules suivantes:

$$cos(\theta) = \frac{1 - tan(\frac{\theta}{2})^2}{1 + tan(\frac{\theta}{2})^2}$$

$$sin(\theta) = \frac{2 \times tan(\frac{\theta}{2})}{1 + tan(\frac{\theta}{2})^2}$$

Ces formules sont très intéressantes car il suffit de calculer $tan(\frac{\theta}{2})$ pour avoir accès à $cos(\theta)$ et $sin(\theta)$. C'est pourquoi nous avons mis cette stratégie en place pour calculer les fonction *sin* et *cos*.

6.1.2 Précision et validation

Nous avons vu précédemment que l'algorithme a besoin de constantes pré-enregistrées pour fonctionner. Nous avons 2 possiblités pour choisir ces valeurs. Nous pouvions travailler en base 10 ou en base 2. En fait les constantes ki peuvent valoir $ki = arctan(\frac{1}{10^i})$ ou $ki = arctan(\frac{1}{2^i})$. Tout d'abord, on peut se douter que si on choisit 10, la méthode CORDIC va converger plus rapidement

que si on prend des puissances de 2 et on atteindra plus vite l'approximation de $tan(\theta)$.

Cependant, nous avons voulu vérifier ça avec des tests de rapidité et de précision. En nous aidant de Python, nous avons renvoyé une liste contenant: la véritable valeur de $sin(\frac{\pi}{3})$, la valeur approchée en utilisant la base 10 puis la valeur approchée en utilisant la base 2. Ces valeurs sont obtenues dans la Figure 19 ci-dessous:

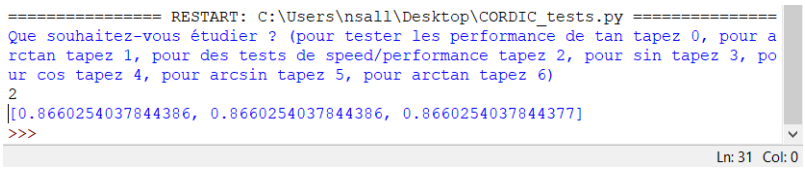

Figure 19: Sortie obtenue pour $sin(\frac{\pi}{3})$

On remarque que les 2 valeurs de droite sont très proches de la première (la valeur exacte de $sin(\frac{\pi}{3})$). Cependant, on remarque que la valeur obtenue en travaillant en base 10 est légèrement plus proche de la réalité. Pour l'instant, nous retenons la méthode en base 10 car elle semble être plus précise que la méthode en base 2. Pour nous en assurer, nous avons réalisé le même test pour $cos(\frac{\pi}{6})$. Les valeurs obtenues sont décrites dans la Figure 20 ci-dessous:

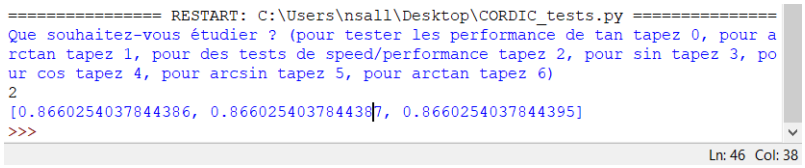

Figure 20: Sortie obtenue pour $cos(\frac{\pi}{6})$

Pour cet exemple, les 2 valeurs de droite sont très proches de la valeur exacte à gauche. Cependant, c'est une fois de plus la valeur de l'algorithme en base 10 qui donne le résultat le plus proche de la réalité. C'est pourquoi, nous avons choisi de travailler en base 10 pour calculer les fonctions *cos* et *sin*. De plus, l'avantage de travailler en base 10 est que le résultat va converger plus rapidement vers la véritable valeur de $tan(\theta)$. Nous avons aussi mené des tests en Python pour mesurer la durée d'exécution de nos algorithmes mais les temps obtenus étaient très courts (on obtenait 0.0, c'était quasiment instantané). C'est pourquoi, pour obtenir la meilleure précision, nous avons estimé qu'il valait mieux travailler en base 10 pour *cos* et *sin*. Cependant, nous allons voir dans la partie suivante que les algorithmes pour *arcsin* et *arctan* fonctionnent en base 2.

6.2 Arcsin

La fonction Arcsin est la première fonction réciproque abordée dans le sujet. Nous avons étudié dans la partie 3 un algorithme permettant d'approcher les valeurs des fonctions réciproques: Le CORDIC inverse. Tout comme les fonctions trigonométriques classiques *cos*, *sin* l'algorithme permettant d'approcher *arcsin* nécessite plusieurs constantes pré-enregistrées en mémoire. Contrairement aux fonctions classiques *cos*, *sin* où nous avions fait le choix de travailler en base 10, nos recherches bibliographiques nous ont conduits vers des algorithmes qui fonctionnent uniquement en base 2. C'est pourquoi, nous avons essayé de travailler en base 2 pour changer, les résultats se sont avérés concluants donc nous avons choisi de travailler en base 2 pour les fonctions réciproques.

Les angles à rentrer manuellement dans l'algorithme afin qu'il fonctionne correctement sont les angles qui vont satisfaire la relation suivante:

$$tan(\theta_j) = 2^{-2j}$$

Nous allons donc obtenir la relation suivante pour passer des coordonnées du point M_k au point suivant M_{k+1} tout au long de l'algorithme:

$$\begin{cases} X_{k+1} = X_k - 2^{-2k} \times Y_k \\ Y_{k+1} = Y_k + 2^{-2k} \times X_k \end{cases}$$

Dans la Figure 21 ci-dessous, on peut observer le remplissage des premières constantes utilisées par l'algorithme approchant la fonction Arcsin. Une fois de plus, nous utilisons les listes chaînées pour stocker toutes ces constantes et y accéder lors des calculs.

```
444         a15.init(3.0517578115526096e-05, a16, 9.313225746154785e-10);
445         a14.init(6.103515617420877e-05, a15, 3.725290298461914e-09);
446         a13.init(0.00012207031189367021, a14, 1.4901161193847656e-08);
447         a12.init(0.00024414062014936177, a13, 5.960464477539063e-08);
448         a11.init(0.0004882812111948983, a12, 2.384185791015625e-07);
449         a10.init(0.0009765621895593195, a11, 9.536743164062 5e-07);
450         a9.init(0.0019531225164788188, a10, 3.814697265625e-06);
451         a8.init(0.0039062301319669718, a9, 1.52587890625e-05);
452         a7.init(0.007812341060101111, a8, 6.103515625e-05);
453         a6.init(0.015623728620476831, a7, 0.000244140625);
454         a5.init(0.031239833430268277, a6, 0.0009765625);
455         a4.init(0.06241880999595735, a5, 0.00390625);
456         a3.init(0.12435499454676144, a4, 0.015625);
457         a2.init(0.24497866312686414, a3, 0.0625);
458         a1.init(0.4636476090008061, a2, 0.25);
459         a0.init(0.7853981633974483, a1, 1);
460     }
```

Figure 21: Les constantes utilisées lors du calcul d'Arcsin

Durant la conception de cet algorithme, nous avons fait attention au domaine de définition de la fonction Arcsin. De plus, nous avons simplifié notre algorithme grâce au caractère impaire de la fonction Arcsin ($arcsin(-x) = -arcsin(x)$). En effet, lorsque $-1 < \theta < 0$ il suffit d'appliquer l'algorithme du CORDIC inverse à $-\theta$ car on a $0 < -\theta < 1$. Ensuite, il nous suffit de multiplier

le résultat par -1. Nous avons aussi pris le temps de retourner une erreur si le flottant saisi en argument n'est pas compris entre -1 et 1. De même, on retourne 0.0 lorsque le flottant saisi en argument vaut 0.0. Tous ces cas particuliers que nous avons introduits dans le code sont décrits dans la Figure 22 ci-dessous:

```
827         curr=_liste.getA0();
828         if((nombre > 0 && nombre > 1) || (nombre < 0 && nombre < -1)){
829             println("Erreur de domaine (le nombre doit appartenir à [-1, 1])");
830             this._Error();
831         }
832         if(nombre == 0.0){
833             signe2=1;
834             return 0.0;
835         }
836         if(nombre > 0.0){
837             signe2 = 1;
838             w = nombre;
839         }
840         if(nombre < 0.0){
841             signe2 = -1;
842             w = - nombre;
843         }
844         i = 0;
845         while(i<74){
```

Figure 22: Les cas particuliers de l'algorithme CORDIC inverse

L'algorithme complet permettant d'approcher $arcsin(x)$ est décrit en annexe.

6.3 Arctan

La fonction Arctan est similaire à la fonction Arcsin abordée dans la partie précédente. L'algorithme permettant d'approcher la fonction Arctan est légèrement différent de celui d'Arcsin. L'algorithme étudié dans la partie 3 permet d'approcher les valeurs des fonctions réciproques, notamment celle d'Arctan. En effet, pour approcher la valeur d'$arctan(x)$ il faut faire exactement les opérations inverses de l'algorithme CORDIC classique vu en partie 2. En effet, ce dernier permet de s'approcher de la valeur de $tan(x)$ donc en partant de la fin et en effectuant les opérations inverses nous parviendrons à approcher la valeur d'$arctan(x)$.

Tout comme les fonctions trigonométriques étudiées précédemment (cos, sin, $arcsin$), l'algorithme permettant d'approcher $arctan$ nécessite, lui aussi, plusieurs constantes pré-enregistrées en mémoire. De même que pour la fonction Arcsin, nos recherches bibliographiques nous ont conduits vers des algorithmes CORDIC inverse fonctionnant uniquement en base 2. C'est pourquoi, nous avons réitéré l'algorithme en base 2 comme pour Arcsin, les résultats se sont avérés concluants donc nous avons aussi choisi de travailler en base 2 pour Arctan.

Les angles à rentrer manuellement dans l'algorithme afin qu'il fonctionne correctement sont les angles qui vont satisfaire la relation suivante:

$$tan(\theta_j) = 2^{-j}$$

Nous allons donc obtenir la relation suivante pour passer des coordonnées du point M_k au point suivant M_{k+1} tout au long de l'algorithme:

$$\begin{cases} X_{k+1} = X_k - 2^{-k} \times Y_k \\ Y_{k+1} = Y_k + 2^{-k} \times X_k \end{cases}$$

Comme pour la fonction Arcsin, nous avons regroupé les premières constantes utilisées par l'algorithme approchant la fonction Arctan dans la Figure 23 ci-dessous (dans notre algorithme il y a 145 constantes mais nous avons sélectionné les premières). Comme pour les fonctions précédentes, nous utilisons les listes chaînées pour stocker toutes ces valeurs.

```
        a18.init(3.814697265606496e-06, a19, 3.814697265625e-06);
        a17.init(7.62939453110197e-06, a18, 7.62939453125e-06);
        a16.init(1.5258789061315762e-05, a17, 1.52587890625e-05);
        a15.init(3.0517578115526096e-05, a16, 3.0517578125e-05);
        a14.init(6.103515617420877e-05, a15, 6.103515625e-05);
        a13.init(0.0001220703118936702l, a14, 0.0001220703125);
        a12.init(0.0002441406201493617l, a13, 0.000244140625);
        a11.init(0.0004882812111948983, a12, 0.00048828125);
        a10.init(0.0009765621895593195, a11, 0.0009765625);
        a9.init(0.0019531225164788188, a10, 0.001953125);
        a8.init(0.0039062301319669718, a9, 0.00390625);
        a7.init(0.007812341060101111, a8, 0.0078125);
        a6.init(0.015623728620476831, a7, 0.015625);
        a5.init(0.031239833430268277, a6, 0.03125);
        a4.init(0.06241880999595735, a5, 0.0625);
        a3.init(0.12435499454676144, a4, 0.125);
        a2.init(0.24497866312686414, a3, 0.25);
        a1.init(0.4636476090008061, a2, 0.5);
        a0.init(0.7853981633974483, a1, 1);
    }
```

Figure 23: Les constantes utilisées lors du calcul d'Arctan

Durant la conception de notre algorithme, nous avons eu une attention particulière pour la précision qui est le critère le plus important de l'extension. Étant donné que la fonction Arctan est définie sur $\mathbb{R}$ il fallait s'assurer que toutes les valeurs soient les plus précises possibles. C'est pourquoi, nous avons testé notre algorithme avec énormément de valeurs pour s'assurer qu'il n'y avait pas d'erreurs avant 10^{-6} ou 10^{-7}. De plus, nous avons simplifié notre algorithme grâce au caractère impaire de la fonction Arctan ($arctan(-x) = -arctan(x)$). En effet, lorsque $-1 < \theta < 0$ il suffit d'appliquer l'algorithme du CORDIC inverse à $-\theta$ car on a $0 < -\theta < 1$. Ensuite, il nous suffit de multiplier le résultat par -1. De même, on retourne 0.0 lorsque le flottant saisi en argument vaut 0.0. Tous ces cas particuliers sont pris en compte dans notre algorithme final. Une version en pseudo-code de l'algorithme CORDIC inverse permettant d'approcher $arctan(x)$ est décrit en annexe.

7 Résultats obtenus et Validation

La précision est le critère le plus important de notre extension, étant donné que nous avons choisi TRIGO. Les algorithmes CORDIC et ANTI CORDIC présents sur internet sont ceux utilisés par les premiers ordinateurs et les calculatrices. Ces algorithmes réalisent des approximations du Cosinus et du Sinus de l'angle passé en argument. Notre problème principal était de mettre en œuvre des solutions permettant d'avoir la meilleure précision possible pour les fonctions cos, sin, asin, atan et ulp. Le temps d'exécution est aussi un facteur important de l'extension TRIGO mais il demeure moins important que la précision. Les algorithmes pour cos et sin sont assez similaires et renvoient des résultats avec la même précision. En effet, si on se place sur le cercle trigonométrique (ex: cos(1) ou sin(1)) les 2 algorithmes vont afficher les valeurs attendues avec une précision allant de 10^{-7} à 10^{-8}. Si on sort du cercle trigonométrique, une nouvelle problématique apparaît : on doit se ramener au cercle trigonométrique. Si on souhaite se ramener à $[-\pi, \pi]$ il faut ajouter ou retrancher pi plusieurs fois ce qui peut entraîner des imprécisions ou des erreurs. Nous avons stocké une valeur approchée de π dans notre module Math (contenant toutes les décimales possibles, $\simeq 44$ pour un flottant) mais en utilisant une valeur approchée comme celle-ci, nous allons générer des approximations et des incertitudes en ajoutant $\pm\pi$. Les calculs sont arrondis par déca tout comme la valeur de π et c'est ce qui entraîne des imprécisions si on répète ces calculs un grand nombre de fois (par exemple pour cos(1000000)).

Pour se ramener à $[-\pi, \pi]$ nous avons fait appel à l'algorithme de "range reduction" présenté précédemment: cet algorithme permet de calculer cos(100000) ou sin(10000) efficacement. Cependant, il trouve ses limites quand l'angle passé en argument est trop élevé (par exemple pour $cos(10^{10})$ l'écart entre la véritable valeur et celle de notre algorithme et de 1.5 ce qui n'est pas du tout acceptable d'un point de vue précision). Cependant, notre version donne de bons résultats pour des angles inférieurs à 10 000, au delà de cette valeur, la précision va diminuer progressivement. Nous avons tracé la courbe de la Figure 24 ci-dessous en analysant les écarts entre la véritable valeur de $cos(\theta)$ et l'approximation renvoyée par notre algorithme (pour θ allant de 10 à 10^{10})

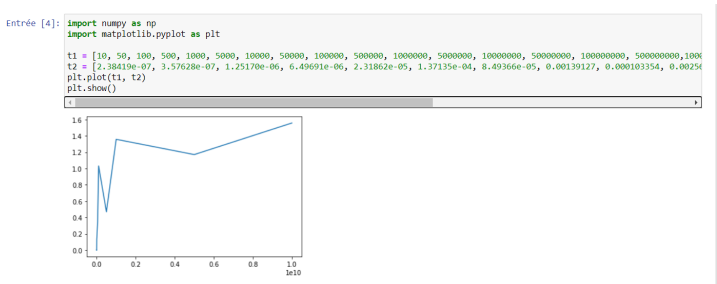

Figure 24: Ecart entre la véritable valeur de $cos(\theta)$ et celle de notre algorithme en fonction de θ

En pratique, on fera rarement appel à cos(1000000) mais il faut tout de même noter que la valeur renvoyée par notre algorithme ne sera pas précise. En effet, pour des angles "raisonnables"(qui ne sont pas de l'ordre du milliard), nos algorithmes renvoient des valeurs assez précises pour cos et sin. Pour vérifier ces valeurs, nous avons pris les valeurs de cos et de sin sur une calculatrice (Casio Graph 35+) et sur Python et nous les avons comparées avec la sortie de notre algorithme.

Quant à Arcsin et Arctan, l'algorithme est un peu atypique, il fonctionne un peu comme un CORDIC inverse mais il renvoie de bons résultats avec une précision similaire à celle de cos/sin (de 10^{-6} à 10^{-7}). Nous avons distingué les cas où le nombre en entrée est positif, négatif ou nul (la fonction renvoie 0.0). Grâce à l'imparité de ces fonctions, le cas négatif est le même que le cas positif sauf qu'on multiplie la sortie par -1.

8 Liens utiles et Bibliographie

- Méthode de calcul et algorithme : http://www.trigofacile.com
  voir l'onglet maths/trigo/calcul/cordic/cordic.htm
- Histoire et précision du CORDIC : http://assprouen.free.fr
  voir l'onglet fichiers/tables-trigonometriques/cordic-2.pdf
- Méthode du CORDIC inverse : https://www.convict.lu/Jeunes/Math/arctan.htm
- Passage de arctan à arccos/arcsin :
  http://www.panamaths.net/Documents/Exercices/SolutionsPDF/26/TRIGOC00005.pdf
- code source de la classe Math de java contenant Math.ulp()
  https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/lang/Math.java

9 Annexes

Algorithm 1 Algorithme CORDIC pour Cos et Sin

Entrées: θ l'angle donné par l'utilisateur que l'on ramène à $0 < \theta < \frac{\pi}{2}$, le tableau des θ_k, k=0, X=1 et Y=0 les coordonnées du point de départ M0, $\varepsilon = 10^{-15}$ la précision voulue sur l'angle θ
Sortie: $\frac{Y}{X}$ qui est la valeur de tan(θ)
def CORDIC(θ, table-des-θ_k, k, X, Y, ε):
while $\theta \geq \varepsilon$ **do**
  if $\theta < \theta_k$ **then**
    k = k + 1
  end if
  res = X
  X = X - $10^{-k} \times Y$
  Y = Y + $10^{-k} \times res$
  $\theta = \theta - \theta_k$
end while
return $\frac{Y}{X}$

Algorithm 2 Algorithme CORDIC inverse pour Arcsin

Entrées: *nombre* le nombre donné par l'utilisateur, le tableau des constantes θ_k et k = 0.
if 1 < abs(nombre): **then**
  print "erreur de domaine"
end if
x = 1
y = 0
z = 0
w = nombre
for i in range(0, 45): **do**
  if x < 0: **then**
    signx = -1
  else
    signx = 1
  end if
  if y ≤ w: **then**
    sgn = signx
  else
    sgn = -signx
  end if
  if sgn > 0: **then**
    for r in range(1, 3): **do**
      ox = x
      x = x - (y×θ_k)
      y = y + (ox×θ_k)
      z = z + θ_k
    end for
  else
    for r in range(1, 3): **do**
      ox = x
      x = x+ (y×θ_k)
      y = y + (ox×θ_k)
      z = z - θ_k
    end for
  end if
  w = w + (w ×(2^{-2i}))
  k += 1
end for
return z

Algorithm 3 Algorithme CORDIC inverse pour Arctan

Entrées: *nombre* le nombre donné par l'utilisateur, le tableau des constantes θ_k, L=-1, X=1 et Y=nombre les coordonnées du point de départ M0, $\varepsilon = 10^{-15}$ la précision voulue sur la valeur de sortie.
grosse-somme = $-\frac{\pi}{2}$
res = X
X = Y
Y = -res
while abs(Y) $\geq$ epsilon: **do**
  L += 1
  if Y $>$ 0: **then**
    signe = -1
  else
    signe = 1
  end if
  angle-ajoute = signe $\times\theta_L$
  grosse-somme += angle-ajoute
  res = X
  X = X - (signe $\times(2^{-L})\times$ Y)
  Y = Y + (signe $\times(2^{-L})\times$ res)
end while
return - grosse-somme