

Petite balade à la frontière des Mathématiques et du Numérique.

<i>Préambule</i>	<i>P1</i>
<i>I Méthode de calcul par développement limité</i>	<i>P2</i>
1.1 Les diverses formulations	P2
1.2 Notion de précision de calcul	P2
1.3 Méthode de calcul	P4
1.3.1 Les indispensables de préparation	P4
1.3.2 Méthode de calcul du développement limité	P5
1.4 Définitions algorithmique du total	P6
1.5 Le code C++ résultat de ce travail	P8
1.6 Un peu de réflexion supplémentaire	P9
<i>II La méthode CORDIC</i>	<i>P10</i>
2.1 Un peu d'histoire	P10
2.2 La Méthode	P10
2.3 Choix des a_i	P12
2.4 L'algorithme à mettre en place	P12
2.5 Le codage en C/C++	P13
2.6 Résultats	P16
<i>Références :</i>	<i>P16</i>

Préambule

Récemment ma fille m'a posé la question : "mais en fait à part bricoler les limites de fonctions en maths (sic), à quoi servent les développements limités ..?"

Après réflexion, j'ai réussi à lui dire que certainement sa machine à calcul utilisait cette technique pour calculer les fonctions trigonométriques (ce qui n'est pas tout à fait sûr), mais que certainement, il fallait être malin pour éviter les calculs inutiles.

Dans le même temps, pour un projet des sections ISN, j'envisage de réaliser un robot contrôlé par une carte Arduino et un récepteur GPS. Pour ce faire, il me faut une bibliothèque trigonométrique très rapide, et donc je me suis penché sur le berceau de la bibliothèque maths Arduino (dont je n'ai pas trouvé les sources) et ai essayé de faire mieux en termes de rapidité. Peine perdue. Néanmoins, sur les quelques lignes qui suivent, j'ai essayé de mettre noir sur blanc mes recherches et essais (formateurs et infructueux sur le résultat) à propos du calcul des fonctions sin et cos.

Les algorithmes ont été codés en C++, compilés par gcc avec comme cible le microcontrôleur 8 bits cadencé à 16Mhz d'un Arduino Uno. Les capacités limitées de ce processeur tant en mémoire qu'en Mflop, obligent à se poser les vraies questions de l'optimisation de la mémoire et de l'efficacité du code source une fois compilé.

I Méthode de calcul par développement limité

1.1 Les diverses formulations

Formulation de base :

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

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

avec $0! = 1$

Plus explicitement :

$$\sin(x) = x - \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 4 \cdot 3 \cdot 2} - \frac{x^7}{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2} + \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + o(x^{2n+2})$$

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{4 \cdot 3 \cdot 2} - \frac{x^6}{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2} + \dots + (-1)^n \frac{x^{2n}}{(2n)!} + o(x^{2n+1})$$

nota : $o(x^{2n+1})$ signifie fonction négligeable devant x^{2n+1} lorsque $x \rightarrow \infty$
(notation de Landau)

Autre formulation :

$$\sin(x) = \sum_{n=0}^{\infty} k_n(x) \text{ avec } \forall n, k_{n+1}(x) = -k_n(x) \cdot \frac{x^2}{(2n+1) \cdot (2n)} \text{ et } k_0(x) = x$$

$$\cos(x) = \sum_{n=0}^{\infty} h_n(x) \text{ avec } \forall n, h_{n+1}(x) = -h_n(x) \cdot \frac{x^2}{(2n) \cdot (2n-1)} \text{ et } h_0(x) = 1$$

1.2 Notion de précision de calcul :

Nous remarquons que quel que soit x avec $x < 1$ la suite $k_n(x)$ est alternée, que $|k_n(x)|$ est décroissante, et que $\lim_{n \rightarrow \infty} [k_n(x)] = 0$. Nous pouvons démontrer que $|k_n(x)| > \left| \sum_{i=n}^{\infty} k_i(x) \right|$. Il en est de même avec $h_n(x)$. En d'autre terme, limiter le développement au terme d'ordre n permet d'assurer que l'erreur commise est de toute façon inférieure à la valeur absolue du dernier terme calculé.

Pour une précision absolue donnée p , il revient donc de trouver n minimum qui respecte cette précision.

- Pour la fonction \sin $p > \frac{x^{2n+1}}{(2n+1)!}$, pour $x \in [0,1]$ soit $p > \frac{1}{(2n+1)!}$

- Pour la fonction cos, $p > \frac{x^{2n}}{(2n)!}$, pour $x \in [0,1]$ soit $p > \frac{1}{(2n)!}$

Il suffit donc, suivant la ligne trigonométrique calculée, de se limiter en précision (donc en termes calculés) en suivant la table ci-dessous

n	$(2n)!$	$\frac{1}{(2n)!}$	$(2n+1)!$	$\frac{1}{(2n+1)!}$
0	1	1	1	1
1	2	0,5	6	0,16666667
2	24	0,04166667	120	0,00833333
3	720	0,00138889	5040	0,00019841
4	40320	2,4802E-05	362880	2,7557E-06
5	3628800	2,7557E-07	39916800	2,5052E-08
6	479001600	2,0877E-09	6227020800	1,6059E-10
7	8,7178E+10	1,1471E-11	1,3077E+12	7,6472E-13
8	2,0923E+13	4,7795E-14	3,5569E+14	2,8115E-15
9	6,4024E+15	1,5619E-16	1,2165E+17	8,2206E-18

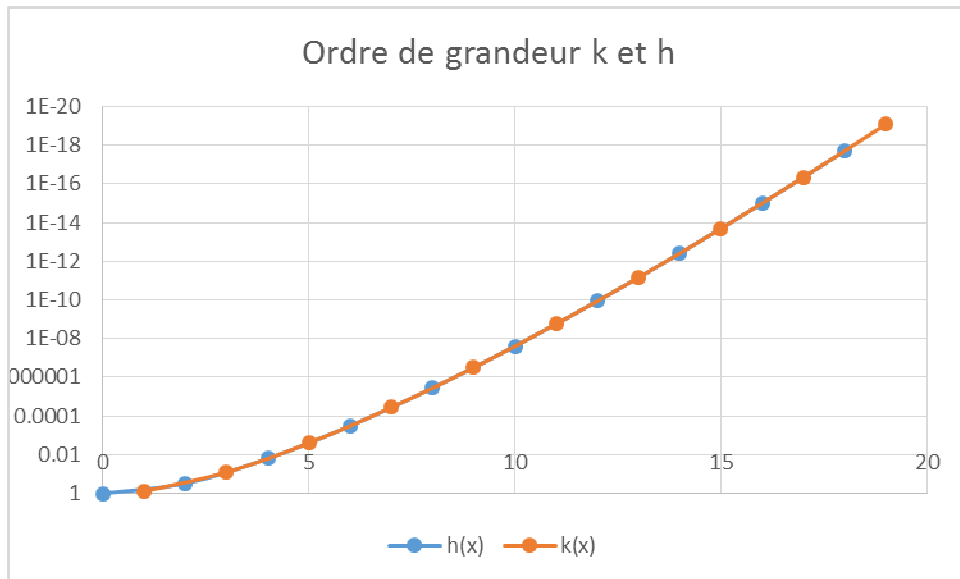
comme le calcul du sin peut conduire au calcul du cos CF méthode de calcul, il est nécessaire (de manière brute) de placer le curseur précision en se basant sur la colonne $\frac{1}{(2n)!}$. En affinant un peu la précision et comme nous le verrons peu après $x < \frac{\pi}{4}$ la

condition réelle sur la précision devient $p > \frac{\left(\frac{\pi}{4}\right)^{2n}}{(2n)!}$ Soit le tableau ci-dessous

n	1	2	3	4	5
$2n$	2	4	6	8	10
précision	0,30842	0,01585	0,00032	3,59E-06	2,46E-08

n	6	7	8	9
$2n$	12	14	16	18
précision	1,15E-10	3,9E-13	1E-15	2,02E-18

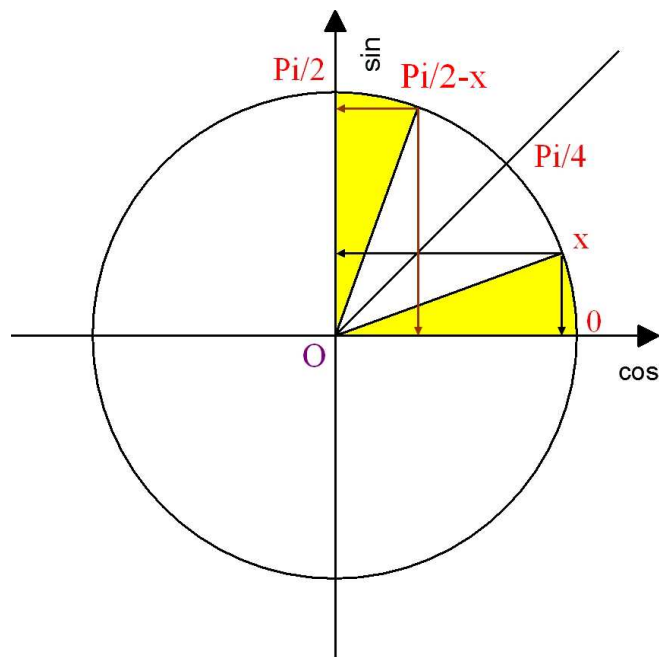
Pour une précision de l'ordre de 10^{-8} , il convient donc d'effectuer un développement limité à l'ordre 10 et donc de calculer 5 termes du développement limité.



Mais nous y reviendrons un peu plus loin en essayant d'être malin.

1.3 Méthode de calcul :

1.3.1 Les indispensables de préparation



Il s'agit au préalable de ramener au préalable l'argument de la ligne trigonométrique dans un intervalle strict $[0,1[$ pour pouvoir effectuer un développement limité "convergeant" des lignes trigonométriques.

La périodicité 2π permet de ramener l'argument dans l'intervalle $[-\pi, +\pi]$.

$while(x > \pi)$ <i>debut</i> $x = x - \pi$ <i>fin</i>	; et	$while(x < -\pi)$ <i>debut</i> $x = x + \pi$ <i>fin</i>
---	------	--

la parité des fonctions permet de se limiter à $[0, \pi]$ $\begin{cases} \sin(-x) = -\sin(x) \\ \cos(-x) = \cos(x) \end{cases}$.

Pour la fonction sin : $if(x < 0)$ <i>debut</i> $x = -x$ <i>multiplier resultat par -1</i> <i>fin</i>	pour la fonction cos :	$if(x < 0)$ <i>debut</i> $x = -x$ <i>fin</i>
--	------------------------	---

La symétrie ou antisymétrie par rapport à la droite verticale passant par O permet de se limiter à l'intervalle $[0, \pi/2]$ $\begin{cases} \sin(\pi - x) = \sin(x) \\ \cos(\pi - x) = -\cos(x) \end{cases}$.

Pour la fonction sin : $if(x > \pi/2)$ <i>debut</i> $x = \pi - x$ <i>fin</i>	pour la fonction cos :	$if(x > \pi/2)$ <i>debut</i> $x = \pi - x$ <i>multiplier resultat par -1</i> <i>fin</i>
--	------------------------	---

La singularité de la droite passant par O de pente 1 (angle de $\frac{\pi}{4}$ par rapport à l'axe Ox) permet de se ramener au cas $x \in [0, \pi/4]$. En effet dans le premier quadrant nous

avons $\begin{cases} \sin(x) = \cos(\pi/2 - x) \\ \cos(x) = \sin(\pi/2 - x) \end{cases}$.

Pour la fonction sin : $if(x > \pi/4)$ <i>debut</i> $resultat = \cos(\pi/2 - x)$ <i>fin</i>	pour le cos :	$if(x > \pi/4)$ <i>debut</i> $resultat = \sin(\pi/2 - x)$ <i>fin</i>
---	---------------	---

1.3.2 Méthode de calcul du développement limité :

Pour une fois, essayons d'être futé

Recette de maitre renard :

Nous pourrions de manière peu futée calculer les termes du développement tels qu'ils sont écrits dans la formulation de base. Pour un développement calculé avec 5 termes, nous comptabiliserons 5 additions réelles et (10+8+6+4+2) multiplications réelles pour les termes x^{2n} , 10 multiplications entières pour $(2n)!$ suivi d'une division réelle.

Soient 30 multiplications réelles, 10 multiplications entières, 5 additions entières, 5 division réelle.

En utilisant la formulation faisant intervenir les $h_{n+1}(x)$ en fonction des $h_n(x)$:

$$h_{n+1}(x) = -h_n(x) \cdot \frac{x^2}{(2n) \cdot (2n-1)}$$

et en pré calculant le terme x^2 on peut estimer les

opérations à

5 fois (1 addition, 1 multiplication, 1 division réelle; 2 multiplications entières; 1 division réelle), soit 5 multiplications réelles, 10 multiplications entières, 5 additions réelles, 5 divisions réelles.

Au bas mot, un gain de 25 multiplications réelles sur 30 (en négligeant les opérations sur les entiers, on obtient un facteur de gain de 7 !!).

Recette de la cigale (partisane du moindre effort dans le long terme) :

Pour une précision donnée, nous avons encadré l'erreur en imposant l'ordre du développement. Cette méthode quoique fonctionnelle, consomme beaucoup de temps calcul pour les valeurs de x petites (en effet le terme x^{2n} ou x^{2n+1} se « tasse très rapidement autour de 0 lorsque x est petit) et donc nous pourrions gagner du temps calcul en éliminant les termes d'ordre le plus élevé. Un petit test sur les $h(x)$ ou $k(x)$ nous permettrait de gagner sur ce point.

<pre> resultat = 0 pour n de 0 à n max debut $h_n(x) = \dots$ resultat = resultat + $h_n(x)$ fin </pre>	<p>peut se transformer en un algorithme « auto-adaptatif ». Cet algorithme quoique plus gourmand en opération sur une boucle y trouve une accélération statistique sur le calcul.</p>	<pre> $h_0(x) = \dots$ resultat = $h_0(x)$ n = 0 while ($h_n(x) > precision$) debut $h_n(x) = \dots$ resultat = resultat + $h_n(x)$ n = n + 1 fin </pre>
---	---	--

1.4 Définitions algorithmique du total

Partie "ramener l'argument entre 0 et pi/4

Pseudo code	Structure BNF (en boites)
-------------	---------------------------

<pre>while (x > π) debut x = x - π fin while (x < -π) debut x ← x + π fin signe ← 1 if (x < 0) debut x ← -x signe ← -1 fin if (x > π/2) debut x ← π - x fin if (x > π/4) debut resultat ← signe • cos(π/2 - x) fin else debut resultat ← signe * sin(x) fin</pre>	<table><tr><td colspan="2">while (x > π)</td></tr><tr><td colspan="2">x ← x - π</td></tr></table> <table><tr><td colspan="2">while (x < -π)</td></tr><tr><td colspan="2">x ← x + π</td></tr></table> <table><tr><td colspan="2">s ← 1</td></tr></table> <table><tr><td colspan="2">(x < 0)?</td></tr><tr><td>Oui</td><td>Non</td></tr><tr><td>x ← -x s ← -1</td><td></td></tr></table> <table><tr><td colspan="2">(x > π/2)?</td></tr><tr><td>Oui</td><td>Non</td></tr><tr><td>x ← π - x</td><td></td></tr></table> <table><tr><td colspan="2">(x > π/4)?</td></tr><tr><td>Oui</td><td>Non</td></tr><tr><td>r ← s • cos(π/2 - x)</td><td>res ← s • sin(x)</td></tr></table>	while (x > π)		x ← x - π		while (x < -π)		x ← x + π		s ← 1		(x < 0)?		Oui	Non	x ← -x s ← -1		(x > π/2)?		Oui	Non	x ← π - x		(x > π/4)?		Oui	Non	r ← s • cos(π/2 - x)	res ← s • sin(x)
while (x > π)																													
x ← x - π																													
while (x < -π)																													
x ← x + π																													
s ← 1																													
(x < 0)?																													
Oui	Non																												
x ← -x s ← -1																													
(x > π/2)?																													
Oui	Non																												
x ← π - x																													
(x > π/4)?																													
Oui	Non																												
r ← s • cos(π/2 - x)	res ← s • sin(x)																												

partie calcul du développement limité du sinus

Pseudo code	Structure BNF (en boites)
-------------	---------------------------

```

h ← x
res ← k
n ← 1
while (k > precision)
    debut
         $k \leftarrow -k \cdot \frac{x^2}{(2n+1) \cdot (2n)}$ 
        res ← res + k
        n ← n + 1
    fin

```

```

h ← x
res ← k
n ← 1
while (k > precision)
     $k \leftarrow -k \cdot \frac{x^2}{(2n+1) \cdot (2n)}$ 
    res ← res + k
    n ← n + 1

```

Pour le développement limité du cosinus, on obtient une formulation tout à fait analogue.

1.5 Le code C++ résultat de ce travail

Nous présenterons ici deux codes, l'un "brut" et aisément compréhensible l'autre plus compact, permettant d'optimiser les ressources processeur en essayant de garder au maximum les données dans les registres du processeur. Malgré toutes ces ruses, nous ne pouvons pas lutter contre le code de la bibliothèque intégré au compilateur (elle est certainement écrite en assembleur). Un facteur 2.9 pour le code "brut" et un facteur 1.7 pour le code "amélioré" sont à constater pour les deux codes proposés (pas mal tout de même !!)

ci dessous une partie des codes que vous trouverez dans le fichier trigdev.ino

Code optimisé	Code non optimisé
<pre> #define precision 1E-7 // precision des float #define _2M_PI 2*M_PI // Les coefficients précalculés const float _CoefSin[10] = { 0, //i=0 0.166666666666, //i=1 0.05, //i=2 0.0238095230, //i=3 0.013888888888, //i=4 0.00909090909, //i=5 0.0064102564, //i=6 0.0047619048, //i=7 0.0036764706, //i=8 0.0023809524 //i=9 }; // Calcul du sinus avec finesse </pre>	<pre> float sindevunitbrut(float x) { static float xstosin = 0; // pour sauver la dernière valeur de l'angle static float resultatsin = 0; // dernière valeur calculée if (x == xstosin) { return resultatsin; // on ne recalcule pas ! }; xstosin = x; float k = x; resultatsin = x; // premier terme byte i = 1; while ((abs(k) > precision)) { k = k * (-sq(x) / (((2 * i + 1) * 2 * i))); resultatsin = resultatsin + k; i = i + 1; } } </pre>

<pre>float sindevunit(float x) { static float xstosin = 0; static float resultatsin = 0; if (x == xstosin) { return resultatsin; }; resultatsin = x; xstosin = x; float k = x; float x2 = sq(x); byte i = 1; while ((k > precision) (k < -precision)) { // version compacte et améliorée // la * est plus rapide que la / resultatsin += (k *= -x2 * _CoefSin[i]); i++; } return (resultatsin); }</pre>	<pre>return (resultatsin); }</pre>
--	------------------------------------

avec le programme trigdev.ino sur un Arduino Uno, pour le calcul de 10000 valeurs aléatoires d'angle, nous obtenons les résultats suivants :

```
sin =>1108 ms efficacite / bibli : 1.00
cos =>1108 ms efficacite / bibli : 1.00
sindevbrut =>3237 ms efficacite / bibli : 2.92
cosdevbrut =>3148 ms efficacite / bibli : 2.84
sindev =>1930 ms efficacite / bibli : 1.74
cosdev =>1869 ms efficacite / bibli : 1.69
```

1.6 Un peu de réflexion supplémentaire :

Nous pourrions être tentés comme je l'ai été de travailler avec une table de valeur pré calculées puis d'effectuer un développement limité autour de ces valeur. Cette technique permet certainement de limiter d'un cran (en prenant 10 valeurs entre $\left[0, \frac{\pi}{4}\right]$) la

précision du développement limité, mais (il y a toujours un mais !)...

La sacro-sainte formule d'addition nous donne $\sin(x+a) = \sin(a)\cos(x) + \cos(a)\sin(x)$.

Imaginons que pour calculer un sin par la méthode du développement limité brut, nous calculons 6 termes pour obtenir la précision souhaitée.

Utilisons la deuxième méthode avec 10 valeurs de sin(a) et cos(a) pré calculées. On peut estimer que le nombre de termes à calculer pour sin(x) diminue de 1 (soyons fou de 2) par rapport à la première méthode. Il en est de même pour le cos(x). En réalité, nous devons calculer 4 termes du sin(x) plus 4 termes du cos(x) et effectuer 2 multiplications supplémentaires (équivalentes à 1 terme de développement en temps calcul). Notre

méthode dite hyper futé consomme en réalité $(4+4+1)/6=1.5$ fois plus de temps que la méthode simple (j'ai essayé statistiquement nous ne sommes pas loin du facteur 1.5). Encore une fois se méfier du mieux qui peut être l'ennemi du bien.

II La méthode CORDIC

(sigle de *CO*ordinate *RO*tation *DI*gital *CO*mputer : « calcul numérique par rotation de coordonnées »)

2.1 Un peu d'histoire

C'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.

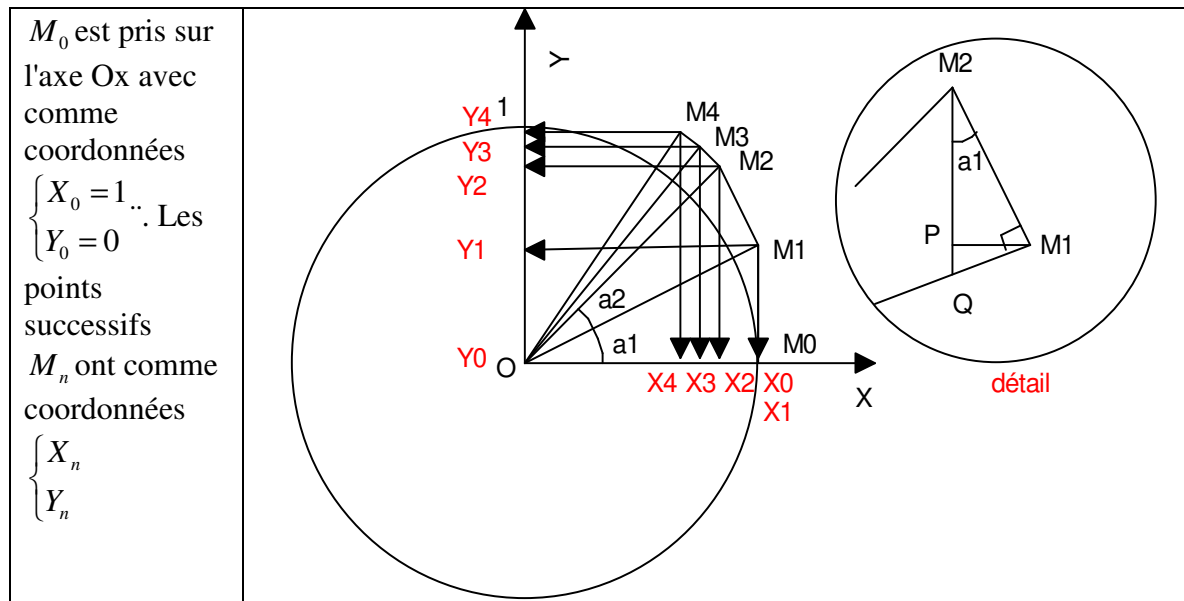
Il s'agit d'un algorithme de choix lorsque aucune implantation matérielle d'un multiplicateur n'est disponible (sur certains microcontrôleurs simples ou des FPGA). De plus, l'algorithme du CORDIC s'adapte bien au calcul à la chaîne. À l'origine, la programmation du CORDIC reposait sur un système binaire.

Cet algorithme a tout son 'charme' dans le cas d'un calculateur utilisant une représentation de nombres à virgule fixe (nombre de chiffres imposé avant le point décimal et nombre de chiffres imposé après le point décimal).

Durant les années 1970, les versions décimales du CORDIC (avec des nombres codés en BCD) commencèrent à apparaître, notamment dans les calculatrices où les critères de coût du matériel sont plus importants que la vitesse de traitement. Un autre avantage du CORDIC est sa flexibilité puisqu'il permet de calculer plusieurs fonctions avec quasiment le même code. Nous ne développerons ici que la version initiale pour le calcul des lignes trigonométriques simples.

2.2 La Méthode

On approche l'angle par des rotations successives a_i décroissantes ($angle = \sum a_i$) dont on connaît les tangentes $\tan(a_i) = \omega_i$. Le choix des a_i n'est pas anodin dans l'efficacité du calcul, mais nous y reviendrons plus tard. Pour l'instant un peu de géométrie :



$$\frac{\overline{M_2 M_1}}{\overline{O M_1}} = \tan(a_2) \text{ notée } \omega_2 \text{ d'où } \overline{M_2 M_1} = \omega_2 \overline{O M_1}$$

Dans le triangle (M_2, P, M_1) nous avons $\frac{\overline{P M_1}}{\overline{M_2 M_1}} = \sin(a_1)$ d'où $\overline{P M_1} = \sin(a_1) \overline{M_2 M_1}$

donc $\overline{P M_1} = \sin(a_1) \cdot \omega_1 \cdot \overline{O M_1}$ or $Y_1 = \sin(a_1) \cdot \overline{O M_1}$ donc $\overline{P M_1} = \omega_1 \cdot Y_1$

$$X_2 = X_1 - \overline{P M_1} \text{ et donc } \boxed{X_2 = X_1 - \omega_2 \cdot Y_1}$$

Dans le triangle (M_2, P, M_1) nous avons $\frac{\overline{P M_2}}{\overline{M_2 M_1}} = \cos(a_1)$ d'où $\overline{P M_2} = \cos(a_1) \overline{M_2 M_1}$

donc $\overline{P M_2} = \cos(a_1) \cdot \omega_1 \cdot \overline{O M_1}$ or $X_1 = \cos(a_1) \cdot \overline{O M_1}$ donc $\overline{P M_2} = \omega_1 \cdot X_1$

$$Y_2 = Y_1 + \overline{P M_2} \text{ et donc } \boxed{Y_2 = Y_1 + \omega_2 \cdot X_1}$$

Par récurrence, il vient
$$\begin{cases} X_{n+1} = X_n - \omega_{n+1} \cdot Y_n \\ Y_{n+1} = Y_n + \omega_{n+1} \cdot X_n \end{cases} \dots$$

Le calcul des lignes trigonométriques s'obtient donc très facilement en évaluant la pente de la droite (O, M_p) . nous avons $\tan\left(\sum_{i=1}^{i=p} a_i\right) = \frac{Y_p}{X_p}$, $\sin\left(\sum_{i=1}^{i=p} a_i\right) = \frac{Y_p}{\sqrt{X_p^2 + Y_p^2}}$,

$\cos\left(\sum_{i=1}^{i=p} a_i\right) = \frac{X_p}{\sqrt{X_p^2 + Y_p^2}}$. Il suffit donc de trouver un algorithme simple pour trouver la

succession des a_i tel que $angle = \sum_{i=1}^{i=p} a_i$ à la précision demandée (remarque : on ne s'interdit pas $a_j = a_{j+1}$).

2.3 Choix des a_i

La méthode à été mise au point pour des calculateurs "rustiques" ne pouvant simplement effectuer que des opérations binaires simples sur des entiers et ne possédant pas la notion de réels (au mieux celle de décimaux à virgule fixe) : addition soustraction et décalage.

Un décalage d'un rang vers la gauche sur un entier revient à le multiplier par 2. Un décalage vers la droite revient à diviser par 2

Les formules de récurrence évoquées en 2.2 possèdent une multiplication que nous transformerons en un décalage si les ω_i peuvent s'écrire sous la forme $\omega_i = 2^j$ avec j un entier positif ou négatif.

les a_i sont donc choisis tels que $\tan(a_i) = \omega_i = 2^{-i}$ soit le tableau suivant :

i	$\tan(a_i)$	a_i	i	$\tan(a_i)$	a_i
0	1	0,78539816	15	3,0518E-05	3,0518E-05
1	0,5	0,46364761	16	1,5259E-05	1,5259E-05
2	0,25	0,24497866	17	7,6294E-06	7,6294E-06
3	0,125	0,12435499	18	3,8147E-06	3,8147E-06
4	0,0625	0,06241881	19	1,9073E-06	1,9073E-06
5	0,03125	0,03123983	20	9,5367E-07	9,5367E-07
6	0,015625	0,01562373	21	4,7684E-07	4,7684E-07
7	0,0078125	0,00781234	22	2,3842E-07	2,3842E-07
8	0,00390625	0,00390623	23	1,1921E-07	1,1921E-07
9	0,00195313	0,00195312	24	5,9605E-08	5,9605E-08
10	0,00097656	0,00097656	25	2,9802E-08	2,9802E-08
11	0,00048828	0,00048828	26	1,4901E-08	1,4901E-08
12	0,00024414	0,00024414	27	7,4506E-09	7,4506E-09
13	0,00012207	0,00012207	28	3,7250E-09	3,7250E-09
14	6,1035E-05	6,1035E-05	29	1,8620E-09	1,8620E-09
			30	9,3130 E-10	9,3130E-10
			31	4,6560E-10	4,6560E-10

On remarquera que :

- La résolution de l'algorithme dépend du terme le plus petit que l'on est susceptible d'ajouter à la somme des a_i pour obtenir l'angle souhaité.
- La division par 2 conduit à une multitude de termes à pré calculer et stocker en mémoire (27 pour une résolution de 10^{-8}).
- A partir du rang 10 pour une résolution de 10^{-8} , il est possible de confondre la tangente et l'angle, ce qui peut simplifier les pré calculs.
- Le calcul produit la tangente,
- En utilisant les formules de l'angle moitié, on peut s'affranchir de l'opération racine

$$\text{carrée coûteuse en temps calcul : } \begin{cases} \text{avec } t = \tan\left(\frac{\alpha}{2}\right) \\ \sin(\alpha) = \frac{2 \cdot t}{1+t^2} \text{ et } \cos(\alpha) = \frac{1-t^2}{1+t^2} \end{cases} \cdot \text{ Cette méthode,}$$

nous pénalise sur la précision (p sur $\frac{\alpha}{2} \Rightarrow 2p$ sur α).

2.4 L'algorithme à mettre en place

Pseudo langage	Représentation BNF
<pre> reste ← angle i ← 0 X ← 1 Y ← 0 while (reste > precision) debut while (reste > a_i) debut stoX ← X - ω_i · Y Y ← Y + ω_i · X X ← stoX reste = reste - a_i fin fin i ← i + 1 fin R ← √(X² + Y²) Cos ← X/R Sin ← Y/R </pre>	<pre> reste ← angle i ← 0 X ← 1 Y ← 0 while (reste > precision) while (reste > a_i) stoX ← X - ω_i · Y Y ← Y + ω_i · X X ← stoX reste = reste - a_i i ← i + 1 R ← √(X² + Y²) Cos ← X/R Sin ← Y/R </pre>

2.5 Le codage en C/C++

Parmi les divers essais réalisés, nous développerons dans cette partie que trois codages différents. *Cordic10* dont le seul but est de bien comprendre comment on procède, *tancordbin* et *tancordbinl* qui se rapprochent de la méthode utilisée dans des calculateurs câblés. Pour *tancorcbn* nous utiliserons des entiers standards, nous nous limiterons à une précision de 10^{-4} (sur un entier codé sur deux octets allant de +32768 à -32767, nous pouvons représenter un nombre à virgule fixe compris entre +3.2768 et -3.2767 à la précision de 10^{-4} près. Or les fonctions sin et cos sont dans l'intervalle [-1.0000, 1.0000] qui rentre bien dans [-3.2768, 3.2767]). Pour *transcordbinl*, nous utiliserons des entiers longs qui nous permettent une précision de 10^{-8} .

La fonction transcordbin est en fait un "depliage" de la fonction cordic10 avec un choix judicieux des ω_i et a_i associés. En prenant $\omega_i = \frac{1}{2^i}$, et en codant les nombres en virgule fixe (représentés sur un entier), nous pouvons transformer la coûteuse multiplication $\omega_i \cdot Y$ par une opération de décalage à droite de i crans du contenu de la variable Y (idem pour $\omega_i \cdot X$), d'où l'algorithme presque linéaire

```

reste ← angle
i ← 0
X ← 1000000000
Y ← 0
si (reste ≥ a0)
    debut
        stoX ← X − Y; Y ← Y + X; X ← stoX; reste = reste − ai
    fin
si (reste ≥ a1)
    debut
        stoX ← X − Y shr 1; Y ← Y + X shr 1; X ← stoX; reste = reste − a1
    fin
....
si (reste ≥ a30)
    debut
        stoX ← X − Y shr 30; Y ← Y + X shr 30; X ← stoX; reste = reste − a30
    fin
Tan ← Y/X

```

Nous présenterons ici le code de tancordbin, pour le reste, se reporter au fichier trigcord.ino.

```

// *****
// Fonction cordic en base 2
// Cette fonction calquée sur un calculateur cablé
// n'utilise aucune boucle
// et travaille sur des entiers
// Ici, nous travaillons avec une précision de 1E-4
// *****

#define _b0 7854 // atan(1/2^0)*1E4
#define _b1 4636 // atan(1/2^1)*1E4
#define _b2 2450 // atan(1/2^2)*1E4
#define _b3 1244 // atan(1/2^3)*1E4
#define _b4 624 // atan(1/2^4)*1E4
#define _b5 312 // atan(1/2^5)*1E4
#define _b6 156 // atan(1/2^6)*1E4
#define _b7 78 // atan(1/2^7)*1E4
#define _b8 39 // atan(1/2^8)*1E4
#define _b9 20 // atan(1/2^9)*1E4
#define _b10 10 // atan(1/2^10)*1E4
#define _b11 5 // atan(1/2^11)*1E4
#define _b12 2 // atan(1/2^12)*1E4

```

```

#define _b13 1 // atan(1/2^13)*1E4
#define _b14 1 // atan(1/2^14)*1E4

//*****
// Calcul de la tangente
// pour un angle de 0 à pi/4
// *****

float tancordbin(float ar)
{ static float _tancordbin = 0;
  static float asto = 0;

  if (asto == ar) {
    return _tancordbin; // déjà calculé !!
  }
  asto = ar; // reinitialisation de la dernière valeur calculée
  int x = 10000;
  int y = 0;
  int sto;
  int a = int(ar * 1E4); // conversion en entier

  // pour chaque test, si réussi
  // sto=x-y*(2^-n)
  // y=y+x*2^-n
  // x=sto
  // a=a-_bn

  if (a >= _b0) { a -= _b0; sto = x - (y >> 0); y += x >> 0; x = sto;}
  if (a >= _b1) { a -= _b1; sto = x - (y >> 1); y += x >> 1; x = sto;}
  if (a >= _b2) { a -= _b2; sto = x - (y >> 2); y += x >> 2; x = sto;}
  if (a >= _b3) { a -= _b3; sto = x - (y >> 3); y += x >> 3; x = sto;}
  if (a >= _b4) { a -= _b4; sto = x - (y >> 4); y += x >> 4; x = sto;}
  if (a >= _b5) { a -= _b5; sto = x - (y >> 5); y += x >> 5; x = sto;}
  if (a >= _b6) { a -= _b6; sto = x - (y >> 6); y += x >> 6; x = sto;}
  if (a >= _b7) { a -= _b7; sto = x - (y >> 7); y += x >> 7; x = sto;}
  if (a >= _b8) { a -= _b8; sto = x - (y >> 8); y += x >> 8; x = sto;}
  if (a >= _b9) { a -= _b9; sto = x - (y >> 9); y += x >> 9; x = sto;}
  if (a >= _b10) { a -= _b10; sto = x - (y >> 10); y += x >> 10; x = sto;}
  if (a >= _b11) { a -= _b11; sto = x - (y >> 11); y += x >> 11; x = sto;}
  if (a >= _b12) { a -= _b12; sto = x - (y >> 12); y += x >> 12; x = sto;}
  if (a >= _b13) { a -= _b13; sto = x - (y >> 13); y += x >> 13; x = sto;}
  if (a >= _b14) { a -= _b14; sto = x - (y >> 14); y += x >> 14; x = sto;}

  float xr = float(x);
  float yr = float(y);
  _tancordbin = yr/xr;
  return _tancordbin;
}

```

2.6 Résultats

Bien qu'annoncé comme ultrarapide, mes essais multiples avec l'algorithme cordic10 se sont soldés par de cuisants échecs au niveau rapidité. En effet un décompte des opérations élémentaires à effectuer nous amène à un nombre voisin de 10 fois celui utilisé dans la méthode utilisant les développements limités. Les deux Cordicbin, eux se comportent beaucoup mieux (mais n'égale pas ceux de la bibliothèque).

Ces dernières méthodes sont par contre adaptées à un calculateur câblé (c'est celle qui avait été implémentée dans les processeurs arithmétiques x87 avec une structure câblée) .
efficacité des algorithmes

Le tableau ci-après donne les résultats obtenus sur un arduino Uno par le programme trigcord.ino :

Fonction	Temps en ms pour 10^4 calculs	Efficacité par rapport aux fonctions de la bibliothèque	précision
sin	3152	1.0	10^{-8}
cos	3151	1.0	10^{-8}
sincord10	110523	17.46	10^{-8}
coscord10	110666	17.55	10^{-8}
sincordbin	4444	1.43	10^{-4}
coscordbin	4153	1.31	10^{-4}
sincordbinl	13463	3.29	10^{-8}
coscordbinl	10250	2.27	10^{-8}

Nous présenterons ici le code de tancordbin, pour le reste, se reporter au fichier trigcord.ino.

Références :

https://fr.wikipedia.org/wiki/D%C3%A9veloppement_limit%C3%A9
<https://fr.wikipedia.org/wiki/CORDIC>
<http://cdeval.free.fr/IMG/pdf/cordic.pdf>
<http://www.trigofacile.com/maths/trigo/calcul/cordic/cordic.htm>
<http://documents.irevues.inist.fr/bitstream/handle/2042/1682/03.PDF+TEXTE.pdf?sequence=1>
<https://www.math.u-psud.fr/~perrin/CAPES/analyse/Suites/Cordic.pdf>

Annexes :

Les deux fichiers trigcord.ino et trigdev.ino respectivement dans les répertoires trigcord et trigdev constitue les sources C/C++ qui ont permis l'élaboration des tests sur une plateforme arduino Uno.