L'ALGORITHME CORDIC

(Christophe.Devalland@ac-rouen.fr)

Utiliser sa calculatrice pour déterminer la valeur d'un cosinus, d'un logarithme ou d'une racine carrée est un geste devenu tellement banal que plus personne ne se demande pourquoi et comment cela marche. Mais, pour reprendre une formule bien connue, a-t-on vraiment besoin de soulever le capot de sa voiture et de comprendre le fonctionnement du moteur pour s'en servir ? Evidemment non. Pourtant, un jour en classe de première, un élève de ma femme un peu curieux a été fier d'expliquer à ses camarades comment on pouvait extraire une racine carrée à la main. A cette occasion, il s'est ensuite demandé si la calculatrice utilisait la même méthode pour calculer une racine carrée. Nous nous sommes également posé cette question sans pouvoir y répondre. On pourrait même aller plus loin en se demandant comment la calculatrice peut donner la valeur d'un sinus ou d'un logarithme avec autant de précision.

Quand il s'agit d'opérations simples telles que l'addition ou la multiplication, on peut assez bien imaginer comment elle procède parce que nous savons faire ces opérations à la main. Mais pour les autres fonctions, comment fait-elle réellement ? Utilise-t-elle des développements limités, des approximations de fonctions, ou d'autres mécanismes plus complexes ? C'est en fouillant à droite et à gauche que j'ai trouvé quelques réponses qui m'ont servies de base pour cet article. Je me suis par ailleurs amusé à programmer certains algorithmes sur ma calculatrice pour vérifier que, malgré leur simplicité, ils donnent de bons résultats.

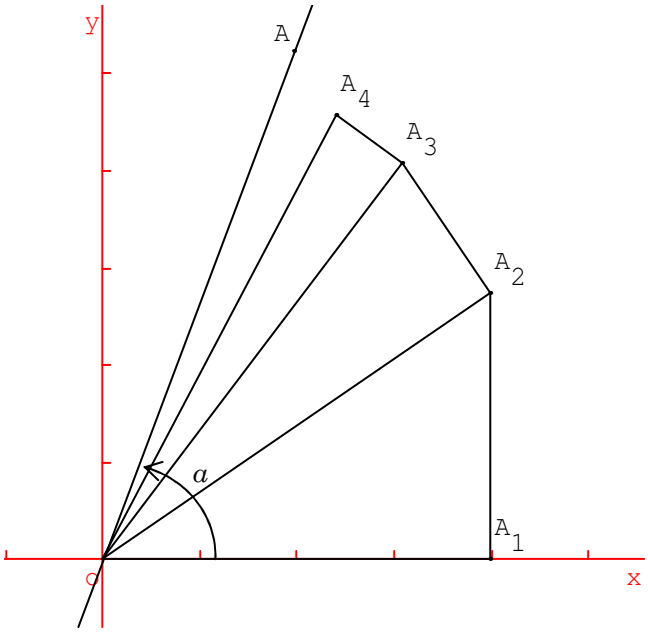
Un peu d'histoire

A la fin des années 50, les calculateurs électroniques sont en plein essor. Les domaines où ils interviennent sont de plus en plus variés et on les trouve par exemple dans les avions où ils servent d'assistants à la navigation aérienne. De ce fait, les besoins de calculs en temps réel se font de plus en plus sentir, notamment pour les calculs trigonométriques. Ainsi, en 1959, Jack E. Volder met au point un algorithme qui permet d'approximer des fonctions trigonométriques à partir d'opérations élémentaires (additions, soustractions et multiplications). Cet algorithme appelé " algorithme CORDIC " (pour Coordinate Rotation DIgital Computer) repose, comme son nom l'indique, sur le calcul des coordonnées de vecteurs auxquels on applique une rotation bien choisie. Très vite, grâce à sa mise en œuvre matérielle très simple (l'ingéniosité de l'algorithme permet en effet un câblage électronique extrêmement simple), l'algorithme CORDIC est repris dans des domaines très diverses tels que : le traitement de signaux radars, les coprocesseurs mathématiques (intel i8087 par exemple) ou les calculatrices scientifiques (toutes marques confondues). Un autre avantage non négligeable de cet algorithme est qu'il permet d'obtenir une précision déterminée à l'avance en effectuant un nombre d'itération donné. Mais pour être certain du résultat, il faut restreindre l'intervalle d'utilisation au domaine de convergence de l'algorithme, comme on le verra plus bas (il existe toutefois des techniques permettant de contourner cette limitation).

Première approche

J'ai repris ici un article de Yves Suprin paru dans le bulletin n°10 de novembre 1996 de la régionale de Haute–Normandie. Il s'agit en fait d'un devoir à la maison proposé en classe de 1ère S dont l'objectif est de permettre le calcul d'une valeur approchée du sinus d'un angle a appartenant à l'intervalle $[0 ; \pi/2[$ en s'inspirant de l'algorithme CORDIC.

L'idée est la suivante : a peut se décomposer en une somme $a_1+a_2+a_3+...+a_n$. Les angles a_i étant de plus en plus petits, et suffisamment, pour assurer la convergence ; un angle a_i pouvant être utilisé plusieurs fois dans cette décomposition. On converge alors vers l'angle a de la façon suivante :



avec $a_i = (\overrightarrow{OA_i}\ ;\ \overrightarrow{OA_{i+1}})$ et $(OA_i) \perp (A_iA_{i+1})$

En posant : $OA_i = R_i$; $\omega_i = \tan a_i$ et $a = a_1 + a_2 + \ldots + a_n$, on obtient les expressions :

$$\cos a = \frac{x_{n+1}}{R_{n+1}},\ \sin a = \frac{y_{n+1}}{R_{n+1}} \text{ et } \tan a = \frac{y_{n+1}}{x_{n+1}}. \ ①$$

On démontre ensuite les relations :

$$\text{Pour } i \in \{1\ ;\ \ldots\ ;\ n\} \begin{cases} x_{i+1} = x_i - \omega_i y_i \\ y_{i+1} = \omega_i x_i + y_i \\ R_{i+1} = R_i \sqrt{1+\omega_i^2} \end{cases} \ ②$$

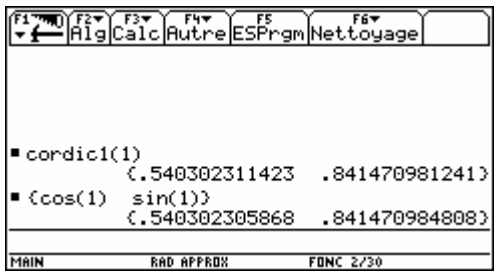
Il suffit donc de connaître x_1, y_1, R_1 et la suite des ω_i pour pouvoir calculer x_n, y_n, R_n et ainsi en déduire cos a et sin a.
Pour les calculs, on peut choisir les angles a_i dans l'intervalle [0 ; π/2[tels que $\tan a_i = 10^{-i}$. Cela permettra des calculs assez simples dans les relations ② puisqu'on aura alors $\omega_i = 10^{-i}$.
Les angles a_i se déterminent une fois pour toute avec la calculatrice pour $i \leqslant 4$ et peuvent être confondus avec 10^{-i} pour $i \geqslant 5$ (c'est d'ailleurs le but de la première partie du devoir que de démontrer que lorsque $0 \leqslant a_i \leqslant 10^{-5}$, $\tan a_i = a_i$ à 10^{-14} près. On ne sera donc pas pénalisé par cette approximation si l'on veut une valeur approchée de sin a et cos a à 10^{-8} près par exemple.).

Pour faire tourner l'algorithme, il suffit donc de donner à la calculatrice la table des arctan(10^{-i}), ces nombres étant des constantes prédéfinies dans le programme. Puis, pour un angle donné a, on soustrait autant de fois qu'il est possible les angles a_1, a_2, a_3, etc... jusqu'à ce que le reste soit inférieur à la précision désirée.

Voici une version possible de cet algorithme pour la TI 92 Plus (précision 10^{-8}) :

```
Cordic1(a)
Func
Local  angle,x,y,z,r,i
{0.785398163,0.099668652,0.009999667,0.001,0.0001,1E-005,1E-006,1E-007,1E
-008}→angle
1→x
0→y
1→r
For  i,0,8
 While  a>=angle[i+1]
  x-y/10^i→z
```

```
  y+x/10^i→y
  z→x
  r*√(1+10^(-2*i))→r
  a-angle[i+1]→a
 EndWhile
EndFor
Return  {x/r,y/r}
EndFunc
```



Il faut toutefois noter que cet algorithme n'est pas à proprement parler l'algorithme CORDIC. En effet, s'il en reprend le principe, il n'en a pas la simplicité originelle. Il faut se souvenir que dans les années 50, il aurait certainement été très pénalisant en temps d'exécution de réaliser un système électronique capable de réaliser une boucle " While " reposant sur la comparaison de deux nombres. De plus, le calcul d'une racine carrée à chaque itération est en contradiction avec les contraintes liées à l'utilisation d'opérations élémentaires. Mais puisque nous avons aujourd'hui des moyens plus performants à notre disposition, utilisons les et l'avantage de cette méthode est sa rapidité de convergence : avec 8 constantes (les $\arctan(10^{-i})$) on obtient une précision de l'ordre de 10^{-8}. Ce n'est pas le cas avec l'algorithme CORDIC car il en faut beaucoup plus, comme nous allons le voir. Ce que l'on gagne en simplicité, on le perd en rapidité de convergence.

L'algorithme CORDIC

Afin de respecter les contraintes matérielles de l'époque et pour pouvoir effectuer des calculs en temps réel, il fallait donc trouver un algorithme n'utilisant pas de test (ou seulement une comparaison avec zéro, très facile à mettre en œuvre) et dont la durée d'exécution soit toujours constante. Ce n'est pas le cas avec l'algorithme précédent puisqu'on ne sait pas à l'avance combien de passages dans la boucle While seront nécessaires pour se rapprocher de l'angle de départ.
La question que l'on peut se poser est la suivante : peut-on approcher un angle θ avec une précision donnée à l'avance en le décomposant en une somme d'angles prédéterminés ? En d'autres termes θ peut-il s'écrire :

$$\theta \approx \sum_{k=0}^{n} \delta_k \varepsilon_k$$

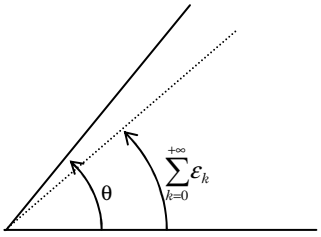
où (ε_n) est une suite d'angles distincts prédéfinis et δ_k ne pouvant prendre comme valeur que -1 ou $+1$? Plus précisément, comme l'on veut atteindre une précision donnée, peut-on écrire θ comme une combinaison d'angles ε_k tels que :

$$\left|\theta - \sum_{k=0}^{n} \delta_k \varepsilon_k\right| \leq \varepsilon_n \quad ③ \ ?$$

(Si tel était le cas, on serait certain d'effectuer $n+1$ étapes exactement pour approcher θ à ε_n près). Essayons de répondre à cette question.
D'abord, on voudrait que la série $\sum \delta_n \varepsilon_n$ converge vers θ conformément à l'inégalité ③, ce qui impose à ε_n de tendre vers 0. Plus simplement nous utiliserons la condition :

(ε_n) est une suite de réels positifs décroissant vers 0 ④

Dans la pratique, nous choisirons même une suite (ε_n) telle que la série $\sum \delta_n \varepsilon_n$ soit absolument convergente, c'est-à-dire telle que la série $\sum \varepsilon_n$ converge.
La convergence de la série $\sum \delta_n \varepsilon_n$ vers θ est-elle toujours possible quelque soit le réel θ ? En fait, non. δ_k ne pouvant prendre comme valeur que -1 ou $+1$, si $\theta > \sum \varepsilon_n$ ou $\theta < -\sum \varepsilon_n$ alors on voit bien qu'il sera impossible d'approcher l'angle θ avec la série $\sum \delta_n \varepsilon_n$:



Plus simplement, pour être certain de pouvoir approcher l'angle θ à ε_n près il faut que :

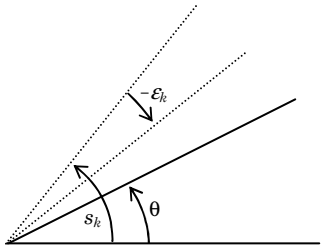
$$|\theta| \leq \sum \varepsilon_n \text{ ou } |\theta| \leq \sum_{k=0}^{n} \varepsilon_k \text{ en se limitant aux } n+1 \text{ premiers termes. ⑤}$$

Voyons maintenant comment choisir la suite (δ_n).

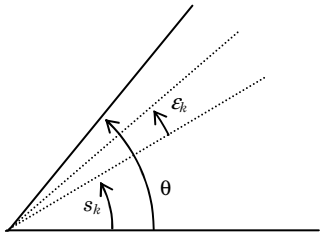
Posons $s_i = \sum_{k=0}^{i-1} \delta_k \varepsilon_k$ et construisons la suite (s_n) de la façon suivante :

- Si $\theta \geqslant 0$ alors $s_1 = \varepsilon_0$ (on se rapproche de θ en partant de ε_0 ; c'est-à-dire : $\delta_0 = +1$) sinon $s_1 = -\varepsilon_0$ (on se rapproche de θ en partant de $-\varepsilon_0$; c'est-à-dire : $\delta_0 = -1$)
- Si $\theta - \delta_0\varepsilon_0 \geqslant 0$ alors $s_2 = \varepsilon_1 + \varepsilon_1$ (on se rapproche de θ en ajoutant ε_1 ; ie $\delta_1 = +1$) sinon $s_2 = \varepsilon_1 - \varepsilon_1$ (on se rapproche de θ en retranchant ε_1 ; ie $\delta_1 = -1$)
- etc...

Plus généralement :



Si s_k est supérieure à la valeur de θ alors $s_{k+1} = s_k - \varepsilon_k$ donc $\delta_k = -1$



Si s_k est inférieure à la valeur de θ alors $s_{k+1} = s_k + \varepsilon_k$ donc $\delta_k = +1$

Finalement :

$$\delta_k = \text{signe}(\theta - s_k) \text{ ⑥}$$

Il reste maintenant à étudier si l'inégalité ③ implique certaines conditions sur la suite (ε_n).
Supposons que cette inégalité soit vraie pour l'entier n et voyons quelle condition doit vérifier ε_{n+1} pour qu'elle reste vraie pour l'entier $n+1$.

On a, au rang $n+1$: $\left|\theta - \sum_{k=0}^{n+1} \delta_k \varepsilon_k\right| = \left|\theta - \sum_{k=0}^{n} \delta_k \varepsilon_k - \delta_{n+1}\varepsilon_{n+1}\right|$

- Si $\theta - \sum_{k=0}^{n} \delta_k \varepsilon_k \geqslant 0$ alors :

  comme on suppose l'inégalité ③ vraie au rang n : $0 \leqslant \theta - \sum_{k=0}^{n} \delta_k \varepsilon_k \leqslant \varepsilon_n$.

  De plus $\delta_{n+1} = \text{signe}(\theta - s_{n+1}) = +1$ donc : $-\varepsilon_{n+1} \leqslant \theta - \sum_{k=0}^{n} \delta_k \varepsilon_k - \varepsilon_{n+1} \leqslant \varepsilon_n - \varepsilon_{n+1}$.

  D'où : $\left|\theta - \sum_{k=0}^{n+1} \delta_k \varepsilon_k\right| \leqslant \max(\varepsilon_{n+1} \ ; \ \varepsilon_n - \varepsilon_{n+1})$.

  Comme on veut avoir $\left|\theta - \sum_{k=0}^{n+1} \delta_k \varepsilon_k\right| \leqslant \varepsilon_{n+1}$, cela équivaut à $\varepsilon_n - \varepsilon_{n+1} \leqslant \varepsilon_{n+1}$.

  C'est-à-dire $\varepsilon_{n+1} \geqslant \frac{1}{2}\varepsilon_n$.

- Si $\theta - \sum_{k=0}^{n} \delta_k \varepsilon_k \leqslant 0$ je vous laisse le soin de vérifier que l'on trouve la même condition sur ε_{n+1}.

Finalement, pour que l'inégalité ③ soit vraie pour tout $n \in \mathbb{N}$, il faut que la suite (ε_n) respecte la condition nécessaire et suffisante :

$$\frac{1}{2}\varepsilon_n \leqslant \varepsilon_{n+1} \leqslant \varepsilon_n \text{ pour tout } n \in \mathbb{N} \text{ ⑦}$$

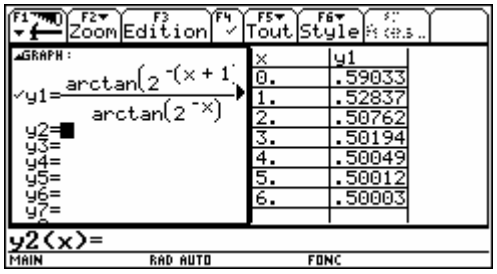
Sous cette condition, il est simple de démontrer par récurrence que ③ est vraie pour tout entier naturel, sous réserve que θ respecte la condition de départ : $|\theta - \delta_0\varepsilon_0| \leq \varepsilon_0$.

Il est maintenant possible de répondre à notre question :
En respectant les conditions ④ à ⑦, il est possible d'approcher l'angle θ à ε_n près en faisant un nombre déterminé d'additions ou de soustractions d'angles ε_k prédéfinis.

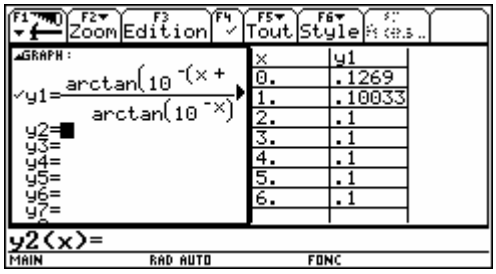
En revenant à l'exemple précédent, on peut remarquer que la suite (ε_n) définie par $\varepsilon_k = \arctan(10^{-k})$ pour $k \in \mathbb{N}$ *ne vérifie pas* la condition ⑦, ce qui oblige à répéter autant de fois que possible la soustraction de l'angle ε_k (ou a_k pour respecter les notations).
Cependant, l'intérêt d'avoir travaillé avec $\omega_i = 10^{-i}$ est d'obtenir des calculs très simples dans la relation ② puisque diviser par 10^i est effectivement très simple pour nous (humains). Mais il faut penser qu'un calculateur électronique ne compte bien souvent qu'en base 2. Il est donc beaucoup plus facile pour lui de diviser par des puissances de 2. Par exemple :

- 14 s'écrit $1\times2^3+1\times2^2+1\times2^1+0\times2^0$ ce qui donne en base 2 :
$$14_{10} = 1110_2$$
- $14\div2 = 7$ s'écrit $1\times2^2+1\times2^1+1\times2^0$ ce qui donne en base 2 :
$$7_{10} = 111_2 \text{ (un décalage à droite)}$$
- $14\times2 = 28$ s'écrit $1\times2^4+1\times2^3+1\times2^2+0\times2^1+0\times2^0$ ce qui donne en base 2 :
$$28_{10} = 11100_2 \text{ (un décalage à gauche)}$$

Tout comme nous le faisons en base 10, multiplier ou diviser par 2 en base 2 revient à décaler à droite ou à gauche les chiffres du nombre. Cette propriété est très intéressante car tout système électronique un peu élaboré est capable d'effectuer ce genre d'opération sur des cases mémoires, appelées aussi des registres.
La base 2 est d'autant plus intéressante que si l'on considère les angles $\varepsilon_k = \arctan(2^{-k})$ pour $k \in \mathbb{N}$, on remarque qu'ils vérifient la condition ⑦. (Je laisse le soin au lecteur de faire l'étude fastidieuse de la fonction $x \mapsto \arctan(\frac{1}{2^{x+1}}) - \frac{1}{2}\arctan(\frac{1}{2^x})$ pour montrer qu'elle est bien toujours positive).
Constatons expérimentalement que $\frac{\arctan(2^{-(n+1)})}{\arctan(2^{-n})} \geq \frac{1}{2}$ pour $n \in \mathbb{N}$:



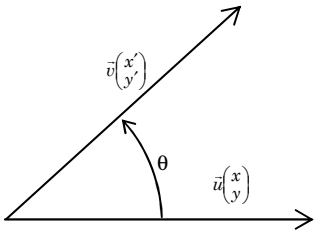
Ca marche en base 2



Ca ne marche pas en base 10

Application au calcul du sinus d'un réel

Reprenons un réel θ que l'on veut approcher par une combinaison de n nombres ε_k de plus en plus petits. Ainsi $\theta \approx \delta_0\varepsilon_0+\delta_1\varepsilon_1+\ldots+\delta_{n-1}\varepsilon_{n-1}$ où $\delta_k = \pm 1$.
Considérons maintenant la matrice de la rotation vectorielle d'angle θ pour obtenir la relation suivante entre les coordonnées des vecteurs $\vec{u}$ et $\vec{v}$.



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

ce qui revient à :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\varepsilon_0 & -\sin\varepsilon_0 \\ \sin\varepsilon_0 & \cos\varepsilon_0 \end{bmatrix} \begin{bmatrix} \cos\varepsilon_1 & -\sin\varepsilon_1 \\ \sin\varepsilon_1 & \cos\varepsilon_1 \end{bmatrix} \cdots \begin{bmatrix} \cos\varepsilon_{n-1} & -\sin\varepsilon_{n-1} \\ \sin\varepsilon_{n-1} & \cos\varepsilon_{n-1} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

L'idée maintenant est de faire apparaître $\tan\varepsilon_k$ en factorisant par $\cos\varepsilon_k$ chaque matrice :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \prod_{k=0}^{n-1} \cos\varepsilon_k \begin{bmatrix} 1 & -\tan\varepsilon_k \\ \tan\varepsilon_k & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

En prenant $\varepsilon_k = \arctan(2^{-k})$ et en définissant $\delta_k = \text{signe}(\theta - (\delta_0\varepsilon_0 + \delta_1\varepsilon_1 + \ldots + \delta_{k-1}\varepsilon_{k-1}))$ on obtient :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \prod_{k=0}^{n-1} \cos\varepsilon_k \begin{bmatrix} 1 & -\delta_k 2^{-k} \\ \delta_k 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Compte tenu du choix des suites (ε_n) et (δ_n), on est assuré d'obtenir l'égalité :

$\theta = \lim_{n\to+\infty} \sum_{k=0}^{n} \delta_k\varepsilon_k$, à condition que θ respecte la condition ⑤ c'est-à-dire que $\theta \leqslant \sum \arctan(2^{-k}) \approx$ 1,7. Mais puisque les fonctions sinus et cosinus sont 2π-périodiques, on pourra se limiter à $|\theta| \leq \frac{\pi}{2}$

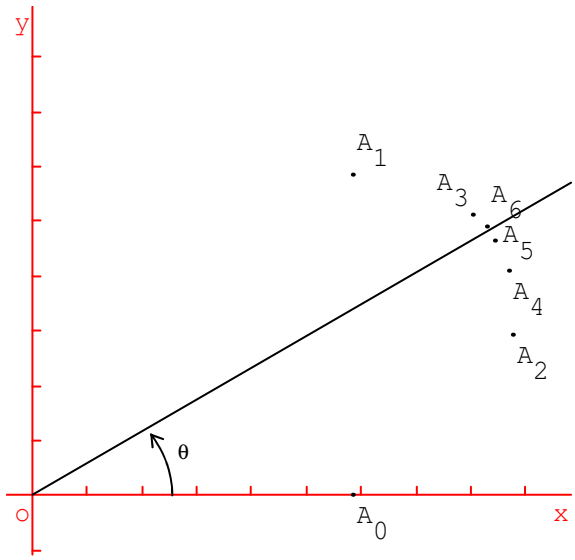
Ainsi, en appelant K le réel $\prod_{k=0}^{\infty} \cos(\arctan(2^{-k}))$ (qui est une constante ≈ 0,60725), on a la relation :

$$\begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = K \prod_{k=0}^{\infty} \begin{bmatrix} 1 & -\delta_k 2^{-k} \\ \delta_k 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \prod_{k=0}^{\infty} \begin{bmatrix} 1 & -\delta_k 2^{-k} \\ \delta_k 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} K \\ 0 \end{bmatrix}.$$

Cette étude préliminaire permet d'en déduire les relations de l'algorithme CORDIC définissant les suites (x_n) et (y_n) convergeant vers cosθ et sinθ :

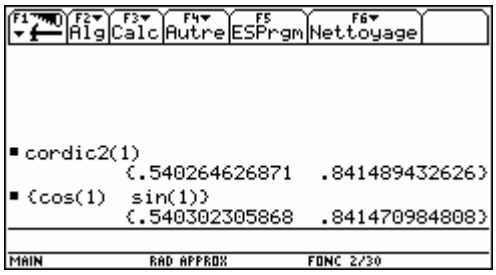
$$\begin{cases} x_0 = K \; ; y_0 = 0 \; ; z_0 = \theta \\ \delta_k = signe(z_k) \\ x_{k+1} = x_k - \delta_k y_k 2^{-k} \\ y_{k+1} = y_k + \delta_k x_k 2^{-k} \\ z_{k+1} = z_k - \delta_k \varepsilon_k \end{cases} \quad \text{où } \varepsilon_k = \arctan(2^{-k}) \; ⑧$$

En notant A(cosθ ; sinθ) et $A_i(x_i ; y_i)$, on obtient la suite de points suivant :



Voici ci-dessous un programme qui donne une valeur approchée de cosθ et sinθ en utilisant 15 constantes ε_k. Cela permet une précision de l'ordre de 2^{-14} soit $\approx 6\times10^{-5}$. C'est beaucoup moins que la précision de 10^{-8} obtenue avec 9 constantes grâce à l'algorithme précédent en base 10 mais l'exécution est beaucoup plus rapide.

```
Cordic2(z)
Func
Local angle,x,y,temp,s,i
{.7854,.46365,.24498,.12435,.06242,.03124,.01562,.00781,.00391,.00195,9.8E
-4,4.9E-4,2.4E-4,1.2E-4,6.E-5}→angle
.60725→x
0→y
For i,0,14
  when(z=0,1,signe(z))→s
  x-s*y/2^i→temp
  y+s*x/2^i→y
  temp→x
  z-s*angle[i+1]→z
EndFor
Return {x,y}
EndFunc
```



Notons que cet algorithme est " réversible ". En reprenant le système ⑧ et en modifiant les conditions suivantes :

Si $z_0 = 0$ et $\delta_k = -\text{signe}(y_k)$
alors $z_n \approx \arctan(y_0/x_0)$.

La preuve repose sur les égalités suivantes :

$$\frac{y_{k+1}}{x_{k+1}} = \frac{\frac{y_k}{x_k} + \delta_k \tan\varepsilon_k}{1 - \delta_k \frac{y_k}{x_k}\tan\varepsilon_k} = \tan(\arctan\frac{y_k}{x_k} + \delta_k\varepsilon_k)$$ (d'après une formule trigonométrique bien connue !)

d'où : $\arctan(\frac{y_{k+1}}{x_{k+1}}) = \arctan(\frac{y_k}{x_k}) + \delta_k\varepsilon_k$.

Et par une récurrence immédiate : $\arctan(\frac{y_n}{x_n}) = \arctan(\frac{y_0}{x_0}) + \sum_{i=0}^{n-1}\delta_i\varepsilon_i$

De plus, $\delta_k = -signe(y_k) = -signe(\arctan(\frac{y_k}{x_k}))$; il faudra donc s'arranger pour que x_k soit toujours positif.

On en déduit : $\delta_k = -signe(\arctan(\frac{y_0}{x_0}) + \sum_{i=0}^{k-1}\delta_i\varepsilon_i) = signe(-\arctan(\frac{y_0}{x_0}) - \sum_{i=0}^{k-1}\delta_i\varepsilon_i)$.

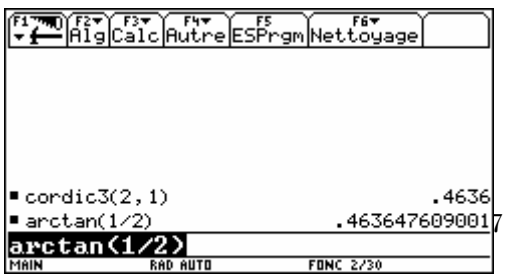
Ainsi, comme les propriétés ④, ⑥ et ⑦ sont vérifiées, on a :

$\left|-\arctan(\frac{y_0}{x_0}) - \sum_{i=0}^{n-1}\delta_i\varepsilon_i\right| \le \varepsilon_n$, c'est-à-dire $\lim_{n\to+\infty}\sum_{i=0}^{n-1}\delta_i\varepsilon_i = -\arctan(\frac{y_0}{x_0})$.

Or, par construction, $z_n = z_0 - \sum_{i=0}^{n-1}\delta_i\varepsilon_i = -\sum_{i=0}^{n-1}\delta_i\varepsilon_i$. La suite (z_n) converge donc bien vers $\arctan(\frac{y_0}{x_0})$.

Voici ci-dessous un programme qui donne une valeur approchée de arctan(*y*/*x*) à 6×10^{-5} près.

```
Cordic3(x,y)
Func
Local angle,temp,z,s,i
{.7854,.46365,.24498,.12435,.06242,.03124,
.01562,.00781,.00391,.00195,9.8E-4,4.9E
-4,2.4E-4,1.2E-4,6.E-5}→angle
0→z
For i,0,14
  when(y=0,-1,-signe(y))→s
```



```
  x-s*y/2^i→temp
  y+s*x/2^i→y
  temp→x
  z-s*angle[i+1]→z
EndFor
Return z
EndFunc
```

Extension à d'autres fonctions

En 1971, John Walther a montré qu'il était possible d'étendre l'algorithme CORDIC à d'autres fonctions. En effet, en modifiant un peu le système (8), on arrive à exprimer les fonctions hyperboliques, les fonctions logarithme et exponentielle ainsi que la multiplication et la division ! Voici la généralisation de cet algorithme :

$$\begin{cases} x_{k+1} = x_k - m\delta_k y_k 2^{-k} \\ y_{k+1} = y_k + \delta_k x_k 2^{-k} \\ z_{k+1} = z_k - \delta_k \varepsilon_k \end{cases} \quad \text{où } m = 0 \text{ ou } \pm 1 \text{ ; } \varepsilon_k \text{ sont des constantes prédéfinies ; } \delta_k = \pm 1. \ ⑨$$

En choisissant $m = 1$, on retrouve le cas des fonctions trigonométriques étudiées plus haut.

Cas des fonctions hyperboliques ($m = -1$)

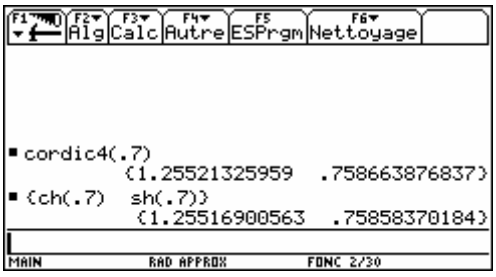
Comme pour les fonctions trigonométriques, on peut montrer qu'en prenant :

$$\varepsilon_k = \text{argth}(2^{-k}) \text{ ; } \delta_k = signe(z_k) \text{ ; } x_0 = \prod_{i=0}^{+\infty} \text{ch}(\text{argth}(2^{-i})) \text{ ; } y_0 = 0 \text{ ; } z_0 = \theta$$

alors $x_n \approx \text{ch}\theta$ et $y_n \approx \text{sh}\theta$.

On obtient par conséquent que $x_n + y_n \approx e^{\theta}$.

```
Cordic4(z)
Func
Local  angle,x,y,temp,s,i
{0.54931,0.25541,0.12566,0.06258,0.03126,0.01563,0.00781,0.00391,0.00195,0.
00097,0.00049,0.00024,0.00012,6E-005}→angle
1.20513→x
0→y
For  i,1,14
  when(z=0,1,sign(z))→s
  x+s*y/2^i→temp
  y+s*x/2^i→y
  temp→x
  z-s*angle[i]→z
EndFor
Return  {x,y}
EndFunc
```



Faisons quelques remarques importantes pour ce programme :
D'abord, il faut noter que les résultats fournis ne seront proches de ch$z$ et sh$z$ qu'à la condition où $|z| < \sum_{i=1}^{+\infty} \text{argth}(2^{-i})$ (soit environ 1,05). On peut alors se dire que cet algorithme n'a pas beaucoup d'intérêt puisque les fonctions ch et sh ne sont pas périodiques comme l'étaient les fonctions sinus et cosinus : on n'aura donc jamais accès avec ce programme à des valeurs de ch$z$ et sh$z$ avec $z > 2$ par exemple. Mais il est possible de contourner ce problème. M. Vern m'a donné quelques indices : " quelque soit le type de développement ou d'algorithme que l'on utilise, il faut en général restreindre le domaine de calcul. Pour arctan, par exemple, on utilise :

$$\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right) \text{ ou bien } \arctan(x) = \arctan(k) + \arctan\left(\frac{x-k}{1+kx}\right),$$

pour ln,

$$\ln(x) = (\ln(2)\times n)+\ln(r) \text{ avec } x = r\times 2^n.$$

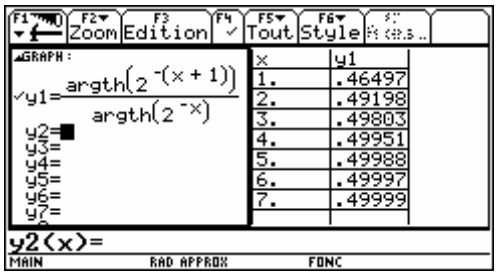
pour asin, on utilise :

$$\arctan\left(\frac{x}{\sqrt{1.0-x^2}}\right) \text{ si } x < 0.9 \text{ et } \pi/2 \pm \arctan\left(\frac{\sqrt{1.0-x^2}}{x}\right) \text{ si } x > 0.9 ,$$

pour sinh et cosh, on peut utiliser les égalités suivantes pour restreindre le domaine de calcul :

$$\cosh(x\pm y) = \cosh(x)\cosh(y)\pm\sinh(x)\sinh(y) \text{ et } \sinh(x\pm y) = \sinh(x)\cosh(y)\pm\cosh(x)\sinh(y) \text{ "}$$

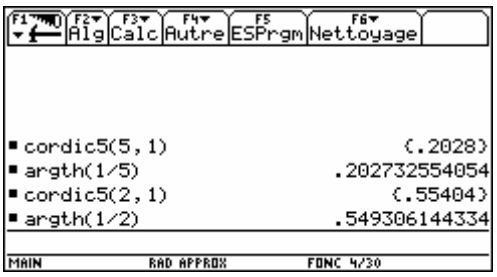
Il y a un autre problème : tel quel, la précision des résultats obtenus par ce programme n'est pas du tout garantie. En effet, la suite (ε_n) ne vérifie pas la condition ⑦, comme le suggère l'écran ci–dessous :



On remarque bien ce problème de convergence en choisissant comme conditions initiales :

$z_0 = 0$; $\delta_k = -\text{signe}(y_k)$
On obtient alors : $z_n \approx \text{argth}(y_0/x_0)$, ce qui permet d'obtenir en plus une approximation de $\ln x = \text{argth}(\frac{x-1}{x+1})$

```
Cordic5(x,y)
Func
Local angle,temp,z,s,i
{.54931,.25541,.12566,.06258,.03126,.01563,.00781,.00391,.00195,9.7E-4,4.9E
-4,2.4E-4,1.2E-4,6.E-5}→angle
0→z
For i,1,14
  when(y=0,-1,-signe(y))→s
  x+s*y/2^i→temp
  y+s*x/2^i→y
  temp→x
  z-s*angle[i]→z
EndFor
Return {z}
EndFunc
```



Suivant la valeur de départ, la précision obtenue est différente : argth(1/5) est bien approchée à 6×10^{-5} près alors que argth(1/2) ne l'est qu'à $5*10^{-3}$ près.

Précision apportée par M. Vern : on peut là encore pallier ce problème. Lors de la programmation des fonctions hyperboliques (sur un microcontrôleur par exemple), il suffit de répéter les itérations 4, 13, 40 … pour assurer la convergence avec une précision constante. C'est ce qui est fait dans la pratique.

Cas des fonctions linéaires (m=0)

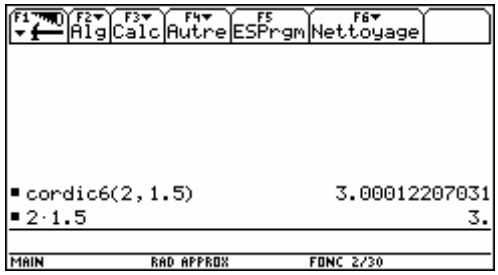
Avec cette valeur de *m,* on donne des valeurs approchées de la multiplication et de la division de deux nombres.

Reprenons le système ⑨ et choisissons :

$$\varepsilon_k = 2^{-k} \ ; \ y_0 = 0 \ ; \ \delta_k = \text{signe}(z_k).$$
On obtient alors $y_n \approx x_0 \times z_0$

On peut justifier ce résultat ainsi :
Le système ⑨ nous donne : $x_{k+1} = x_0$; $y_{k+1} = y_k + \delta_k x_0 2^{-k}$; $z_{k+1} = z_k - \delta_k 2^{-k}$
D'où : $y_n = y_0 + x_0(\sum_{i=0}^{n-1} \delta_i 2^{-i})$; $z_n = z_0 - (\sum_{i=0}^{n-1} \delta_i 2^{-i})$.
Comme $\delta_k = \text{signe}(z_0 - \sum_{i=0}^{k} \delta_i 2^{-i})$ alors $\left| z_0 - \sum_{i=0}^{n} \delta_i 2^{-i} \right| \le \varepsilon_n$.
Finalement : $y_n \approx y_0 + x_0 z_0 = x_0 z_0$.

```
Cordic6(x,z)
Func
Local angle,y,s,i
{1,.5,.25,.125,.0625,.03125,.01563,.00781,.00391,.00195,9.8E-4,4.9E-4,2.4E
-4,1.2E-4,6.E-5}→angle
0→y
For i,0,14
  when(z=0,1,signe(z))→s
  y+s*x/2^i→y
  z-s*angle[i+1]→z
EndFor
Return y
EndFunc
```

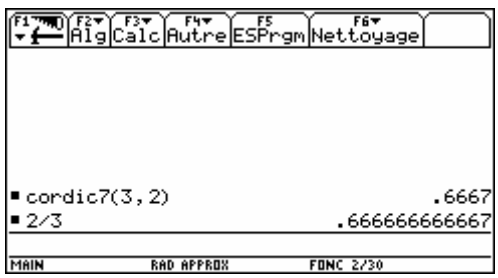


Notons que cet algorithme ne fonctionne que pour les valeurs de z vérifiant :
$|z| \le \sum_{i=0}^{+\infty} \frac{1}{2^i} = 2$. Comme pour les fonctions hyperboliques, son intérêt est assez limité à cause de cette contrainte.

Choisissons maintenant les conditions initiales suivantes :

$$\varepsilon_k = 2^{-k} \ ; \ z_0 = 0 \ ; \ \delta_k = -\text{signe}(y_k).$$
On obtient alors $z_n \approx y_0 / x_0$

```
Cordic7(x,y)
Func
Local  angle,z,s,i
{1,0.5,0.25,0.125,0.0625,0.03125,0.01563,0.00781,0.00391,0.00195,0.00098,0.
00049,0.00024,0.00012,6E-005}→angle
0→z
For  i,0,14
  when(y=0,-1,-sign(y))→s
  y+s*x/2^i→y
  z-s*angle[i+1]→z
EndFor
Return  z
EndFunc
```



Pour conclure

On voit sur ces exemples qu'un besoin très simple de calculs en temps réel datant des années 50 a fini par déboucher sur la mise au point d'une multitude d'algorithmes permettant d'approcher la plupart des fonctions usuelles. Il est d'ailleurs étonnant, lorsqu'on se penche sur ce genre de problème, de constater la diversité des théories mathématiques mises en œuvre pour les résoudre. Par exemple, pour effectuer des calculs avec un grand nombre de chiffres (dont la précision est fixée à l'avance), certains calculateurs utilisent une approximation polynomiale ou rationnelle de la fonction calculée ou bien des algorithmes de type CORDIC. Mais si la précision doit être dynamique, les calculateurs vont plutôt utiliser des développements de Taylor ou des méthodes quadratiques utilisant la moyenne "arithmético-géométrique" de Gauss–Legendre. Ce sont autant de méthodes différentes qui mériteraient chacune une étude particulière (voir à ce sujet le site très instructif http://www.ens-lyon.fr/~jmmuller/).

Finalement, on voit bien que ce sont les mathématiques qui ont permis le développement du calcul électronique et, plus encore aujourd'hui avec la généralisation du calcul formel et des calculs en grande précision, elles restent indispensables à la conception de logiciels pour les ordinateurs et les calculatrices.

Ainsi, affirmer comme on a pu l'entendre récemment de la part d'un ministre de l'Education que les mathématiques étaient en train de se dévaluer à cause des machines relève de l'incompétence ou de la malhonnêteté intellectuelle. Le "OU" n'étant pas exclusif, bien entendu !