CORDIC

COördinate Rotation DIgital Computer

Guilhem Gamard (il/lui)

Université publique et les 64 nains

Journée doctorant·es Mocqua 2026

Contexte

Qu'est-ce qui se passe

CORDIC est un algorithme pour calculer $\cos(\theta)$ et $\sin(\theta)$, donné θ.

Contexte

Qu'est-ce qui se passe

CORDIC est un algorithme pour calculer $\cos(\theta)$ et $\sin(\theta)$, donné θ.

Algo conçu en **1956** par Jack E. Volder

- idée de base dans un livre de 1946
- techniques similaires connues depuis le $17^{\text{ème}}$ siècle

Bref, ce n'est **pas de la recherche actuelle.**

Contexte

Qu'est-ce qui se passe

CORDIC est un algorithme pour calculer $\cos(\theta)$ et $\sin(\theta)$, donné θ.

Algo conçu en **1956** par Jack E. Volder

- idée de base dans un livre de 1946
- techniques similaires connues depuis le 17$^{\text{ème}}$ siècle

Bref, ce n'est **pas de la recherche actuelle.**

Il y a des gens qui améliorent CORDIC encore aujourd'hui (pour $\exp$, $\log$, etc.).
Ce n'est **pas mon sujet** donc je connais mal.

(Image d'autrice ou auteur inconnu.)

Cahier des charges

On veut une **exécution rapide** et une **faible conso mémoire** en pratique (pas en $O(\cdot)$).

À éviter :

- Grosses tables
- Multiplications et divisions

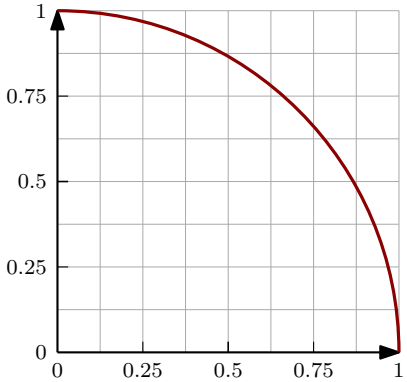
$\Longrightarrow$ Pas de polynômes, ni de Newton, ni de tables de valeurs

À privilégier :

- Additions et soustractions
- Tests ($\leq, <, \geq, >, \ldots$)
- Décalages binaires

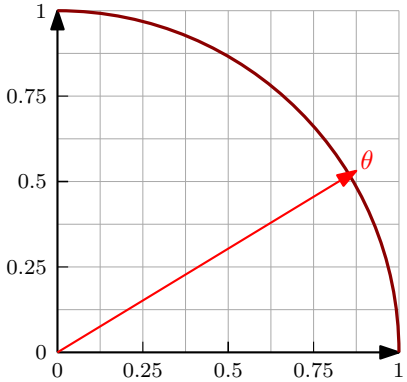
Idée générale

Pour $\theta \in [0; \pi/2]$, on va faire une **recherche dichotomique**.



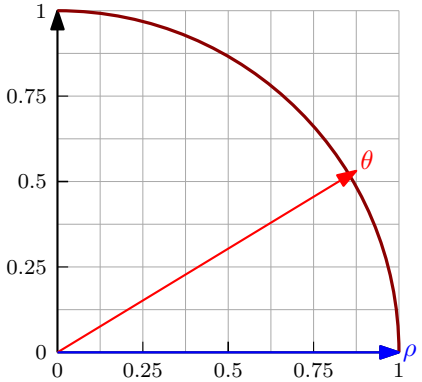
Idée générale

Pour $\theta \in [0; \pi/2]$, on va faire une **recherche dichotomique**.



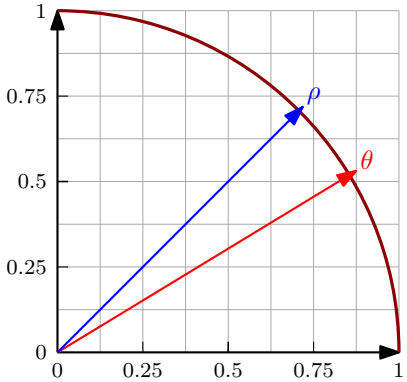
Idée générale

Pour $\theta \in [0; \pi/2]$, on va faire une **recherche dichotomique**.



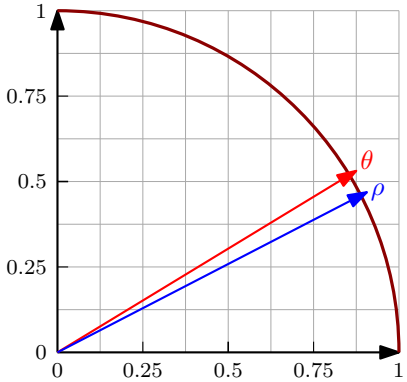
Idée générale

Pour $\theta \in [0; \pi/2]$, on va faire une **recherche dichotomique**.



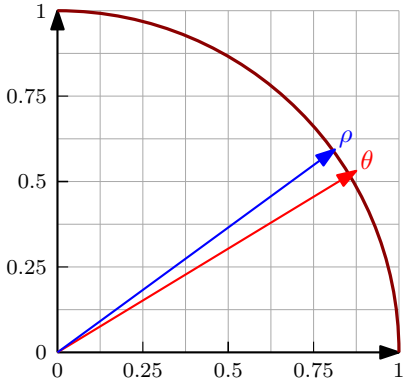
Idée générale

Pour $\theta \in [0; \pi/2]$, on va faire une **recherche dichotomique**.



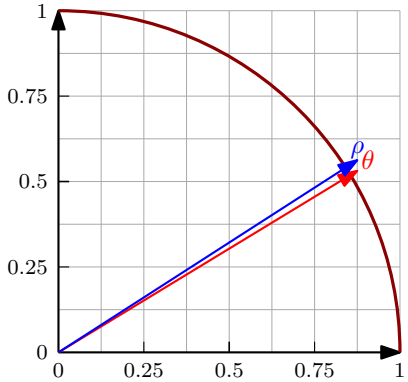
Idée générale

Pour $\theta \in [0; \pi/2]$, on va faire une **recherche dichotomique**.



Idée générale

Pour $\theta \in [0; \pi/2]$, on va faire une **recherche dichotomique**.



Algo général

Initialiser $\rho \leftarrow 0$ et $v \leftarrow (1, 0)$

Pour α dans $\{\frac{\pi}{4},\ \frac{\pi}{8},\ \frac{\pi}{16},\ \frac{\pi}{32},\ \dots\}$ **:**

- **Si** $\theta > \rho$ **:**

  Tourner v de α radians à gauche

  $\rho \leftarrow \rho + \alpha$

- **Sinon :**

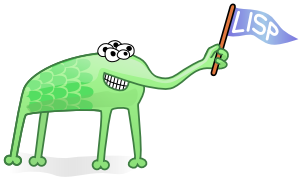
  Tourner v de α radians à droite

  $\rho \leftarrow \rho - \alpha$

Renvoyer v

Trigger warning

Attention : les slides qui suivent contiennent du code en Common Lisp.



(Image de Conrad Barski, domaine public.)

Rotation d'un vecteur

Pour l'instant, nous allons faire la rotation à l'ancienne :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

```
(defun rotate-$ (v α)
  (let ((x (first v))
        (y (second v)))
    (list (+ (* x (cos α)) (* y (- (sin α))))
          (+ (* x (sin α)) (* y (cos α)))))))
```

Implémentation naïve

```
(defparameter splits
  (loop :for k :from 2 to 32 :collect (/ π (expt 2 k))))

(defun cordic-$ (θ)
  (let ((v (list 1 0))
        (ρ 0))
    (dolist (α splits v)
      (if (< ρ θ)
          ;; then
          (setf v (rotate-$ v α)
                ρ (+ ρ α))
          ;; else
          (setf v (rotate-$ v (- α))
                ρ (- ρ α))))))
```

Mais c'est de la triche, on utilise cos et sin dans `rotate-$` !

Mais c'est de la triche, on utilise cos et sin dans `rotate-$` !

Oui, mais toujours sur les mêmes valeurs (les « splits ») :

- $\frac{\pi}{4}$, $\frac{\pi}{8}$, $\frac{\pi}{16}$, ... pour la rotation à gauche
- $-\frac{\pi}{4}$, $-\frac{\pi}{8}$, $-\frac{\pi}{16}$, ... pour la rotation à droite

$\Longrightarrow$ On va **précalculer**.

Rotation d'un vecteur

Nouvelle formule :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos(\alpha) \cdot \begin{bmatrix} 1 & -\tan(\alpha) \\ \tan(\alpha) & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotation d'un vecteur

Nouvelle formule :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos(\alpha) \cdot \begin{bmatrix} 1 & -\tan(\alpha) \\ \tan(\alpha) & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

On a $\tan(-\alpha) = -\tan(\alpha)$, donc il suffit de stocker dans une table :

$$\tan\left(\frac{\pi}{4}\right),\ \tan\left(\frac{\pi}{8}\right),\ \tan\left(\frac{\pi}{16}\right),\ \ldots$$

Rotation d'un vecteur

Nouvelle formule :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos(\alpha) \cdot \begin{bmatrix} 1 & -\tan(\alpha) \\ \tan(\alpha) & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

On a $\tan(-\alpha) = -\tan(\alpha)$, donc il suffit de stocker dans une table :

$$\tan\left(\frac{\pi}{4}\right),\ \tan\left(\frac{\pi}{8}\right),\ \tan\left(\frac{\pi}{16}\right),\ \ldots$$

On a aussi $\cos(-\alpha) = \cos(\alpha)$, donc il suffit de stocker la valeur :

$$\cos\left(\frac{\pi}{4}\right) \times \cos\left(\frac{\pi}{8}\right) \times \cos\left(\frac{\pi}{16}\right) \times \ldots$$

et de faire cette multiplication à la fin.

Implémentation avec tables

```
(defparameter tan-splits
  (mapcar #'tan splits))

(defparameter cos-correction
  (reduce #'* (mapcar #'cos splits)))

(defun rotate/cos (v tan-α)
  (let ((x (first v))
        (y (second v)))
    (list (- x           (* y tan-α))
          (+ (* x tan-α) y         ))))

(defun cos-correct (v)
  (mapcar (λ (z) (* z cos-correction)) v))
```

Implémentation avec tables

```
(defun cordic-$$ (θ)
  (let ((v (list 1 0))
        (ρ 0))
    (loop :for α :in splits
          :for tan-α :in tan-splits
          :do (if (< ρ θ)
                  ;; then
                  (setf v (rotate/cos v tan-α)
                        ρ (+ ρ α))
                  ;; else
                  (setf v (rotate/cos v (- tan-α))
                        ρ (- ρ α)))
          :finally (return (cos-correct v)))))
```

Taille des tables

CORDIC donne 1 bit de précision supplémentaire par itération.

Pour des nombres sur k bits, il faut stocker $k+1$ valeurs de k bits chacune.
(Il y a k splits et la cos-correction.)

- Pour $k = 32$ cela représente 132 octets (gratuit).
- Pour $k = 64$ cela représente 520 octets.

Taille des tables

CORDIC donne 1 bit de précision supplémentaire par itération.

Pour des nombres sur k bits, il faut stocker $k+1$ valeurs de k bits chacune.
(Il y a k splits et la cos-correction.)

- Pour $k = 32$ cela représente 132 octets (gratuit).
- Pour $k = 64$ cela représente 520 octets.

Le compilateur va **dérouler les boucles** et **propager les constantes**.
Les tables seront donc directement dans le texte du programme et pas dans un tableau à part.

(Oui, Common Lisp est compilé et oui, les compilos peuvent optimiser ça.)

Mais tu avais dit que les multiplications et les divisions, ça coûte trop cher !

Mais tu avais dit que les multiplications et les divisions, ça coûte trop cher !

Oui, sauf si ce sont des multplications **par des puissances de 2** :
celles-là sont gratuites (décalages binaires).

$\Longrightarrow$ On va **changer les splits** pour faire apparaître des puissances de 2 !

Divisions par 2

Prenons comme splits :

$$\arctan(1),\ \arctan\left(\frac{1}{2}\right),\ \arctan\left(\frac{1}{4}\right),\ \arctan\left(\frac{1}{8}\right),\ ...$$

Alors appliquer $\tan$ aux splits va donner :

$$1,\ \frac{1}{2},\ \frac{1}{4},\ \frac{1}{8},\ ...$$

Et toutes nos multiplications vont devenir des décalages binaires.

Implémentation «puissances de 2»

```
(defparameter splits-2
  (loop :for k :from 0 :to 32 :collect (atan 1 (expt 2 k))))

(defparameter cos-correction-2
  (reduce #'* (mapcar #'cos splits-2)))
```

Implémentation «puissances de 2»

```
(defun rotate/cos-2 (v k left)
  (let* ((x (first v))
         (y (second v))
         (x*tan-α (/ x (expt 2 k)))
         (y*tan-α (/ y (expt 2 k))))
    (if left
        ;; then
        (list (- x       y*tan-α)
              (+ x*tan-α y))
        ;; else
        (list (+ x y*tan-α)
              (- y x*tan-α)))))

(defun cos-correct-2 (v)
  (mapcar (λ (z) (* z cos-correction-2)) v))
```

Implémentation «puissances de 2»

```
(defun cordic-2 (θ)
  (let ((v (list 1 0))
        (ρ 0))
    (loop :for α :in splits-2
          :for k :from 0
          :do (if (< ρ θ)
                  ;; then
                  (setf v (rotate/cos-2 v k T)
                        ρ (+ ρ α))
                  ;; else
                  (setf v (rotate/cos-2 v k nil)
                        ρ (- ρ α)))
          :finally (return (cos-correct-2 v)))))
```

Mais il reste des divisions.
Comment être sûrs que le compilo les optimise ?

Mais il reste des divisions.
Comment être sûrs que le compilo les optimise ?

CORDIC a été conçu pour tourner sur des **nombres entiers**.
C'est de l'**« arithmétique à virgule fixe »**.

$\Longrightarrow$ Prenons des entiers **64 bits** vus comme des multiples de 2^{61}.

(De façon équivalente, notre unité est maintenant le $1/2^{61}$ème de radian.)

Implémentation «arithmétique à virgule fixe»

```
(defun float-to-fixed (f)
  (round (* f (expt 2 61))))

(defun fixed-to-float (x)
  (* x (expt 2 -61)))

(defparameter splits-f
  (list 1811004864519280640 1069098597953152896
        564882337777596224  286743094836456896
        143927976672616096   72034151524184360
         ;; ...
))

(defparameter cos-correction-f
  1400229935014939136)
```

Implémentation «arithmétique à virgule fixe»

```
(defun rotate/cos-f (v k left)
  (let* ((x (first v))
         (y (second v))
         (x*tan-α (ash x (- k)))
         (y*tan-α (ash y (- k))))
    (if left
        ;; then
        (list (- x       y*tan-α)
              (+ x*tan-α y      ))
        ;; else
        (list (- y*tan-α x)
              (- x*tan-α y)))))

(defun cos-correct-f (v)
  (mapcar (λ (z) (* cos-correction-f z)) v))
```

Implémentation «arithmétique à virgule fixe»

```
(defun cordic-f (θ)
  (let ((v (list 1 0))
        (ρ 0))
    (loop :for α :in splits-f
          :for k :from 0
          :do (if (< ρ θ)
                  ;; then
                  (setf v (rotate/cos-f v k T)
                        ρ (+ ρ α))
                  ;; else
                  (setf v (rotate/cos-f v k nil)
                        ρ (- ρ α)))
          :finally (return (cos-correct-f v)))))
```

Résultat des courses

Pour chacune des k itérations :

- **6** tests ($\rho < \theta$)
- **9** additions ou soustractions
- **8** décalages binaires
- **25** mouvements

Une seule fois (à la fin) :

- **2** multiplications (pour la cos-correction)

Pour rappel, la conso mémoire :

- $k \cdot (k+1)/8$ bits (pour $k = 64$ cela fait 520 octets)
- entièrement dans le texte exécutable

```
(defun cordic (θ)
  (declare (optimize (speed 3) (safety 0) (compilation-speed 0)))
  (let ((x 1) (y 0) (ρ 0))
    (declare (type fixnum θ x y ρ))
    (loop :for α fixnum :in (list 1069098597953152896 ...)
          :for k fixnum :from 0
          :for x*tan-α fixnum = (ash x (- k))
          :for y*tan-α fixnum = (ash y (- k))
          :do (if (< ρ θ)
                  (setf x (- x y*tan-α)
                        y (+ x*tan-α y)
                        ρ (+ ρ α))
                  (setf x (+ x y*tan-α)
                        y (- y x*tan-α)
                        ρ (- ρ α)))
          :finally (return (values (* x 1400229935014939136)
                                   (* y 1400229935014939136))))))
```

Merci ! Questions ?