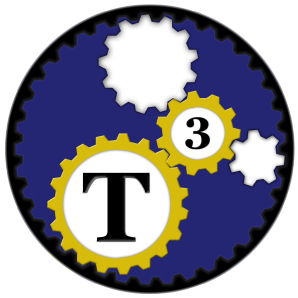
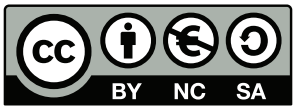


Dossier TechTheTroll

Les trajectoires courbes dans la bonne humeur * : de l'asservissement à la planification

Pierre-François Gimenez
pierre-francois.gimenez@irit.fr

Version 1.2, mai 2016
TechTheTroll.wordpress.com



https://github.com/TechTheTroll/t3-doc

*. conditions may apply

Table des matières

Trajectoire courbe à Eurobot. Allégorie.

Introduction

Où y a pas encore de troll

Ce dossier a pour objectif de présenter la solution utilisée par l'équipe TechTheTroll d'Eurobot concernant les trajectoires courbes. Ce dossier a une approche ascendante : on part du bas niveau pour arriver au haut niveau.

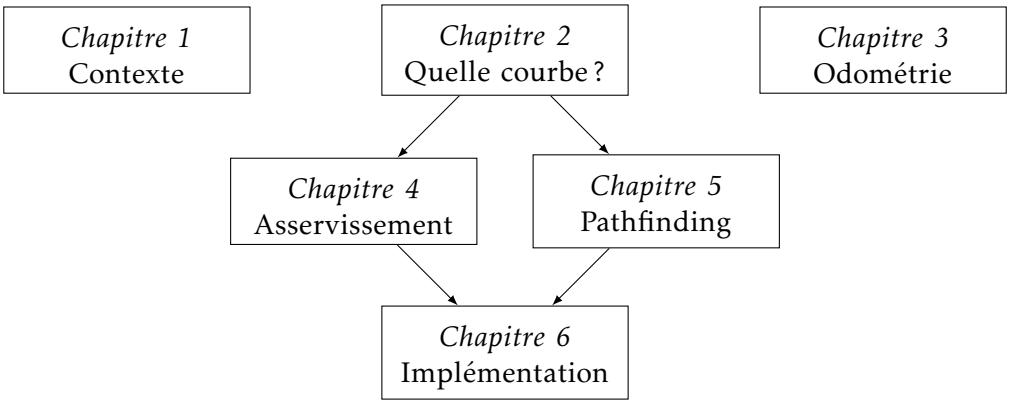
Dans ce rapport, nous considérerons uniquement un robot :

— non-holonome, qui se comporte cinématiquement comme une voiture ;
— muni de codeurs ;
— muni de capteurs de proximité ;
— qui n'a pas besoin de coopérer.

Avoir deux robots qui doivent coopérer peut changer la façon qu'a le haut niveau de prévoir les trajectoires. Nous ne traiterons pas de ce sujet.

Dans le premier chapitre sera rappelé le contexte de ce dossier : qu'est-ce qu'Eurobot, qui est l'équipe TechTheTroll. Le second chapitre entrera dans le sujet et s'interrogera sur la forme que devront prendre les trajectoires courbes. Ensuite, le troisième chapitre concernera l'odométrie qui voit ses exigences grimper du fait de l'utilisation de trajectoires courbes. Le quatrième chapitre concernera l'asservissement à une courbe, tandis que le cinquième chapitre s'attardera sur la manière de générer ses courbes avec une recherche de chemin. Enfin, le sixième chapitre proposera des algorithmes d'implémentation.

Pour ceux d'entre vous qui souhaitent seulement lire les parties qui les intéressent, sachez que les chapitres ont des dépendances et que certains doivent être lus avant d'autres. Le diagramme de dépendance est indiqué ci-dessous.



Bonne lecture !

Chapitre 1

Contexte

Où on fait du blabla avant de rentrer dans le technique

1.1 Eurobot

Eurobot est une compétition européenne de robotique qui a lieu tous les ans depuis 1994 à la Ferté-Bernard (Sarthe), puis à la Roche-Sur-Yon (Vendée) en 2016. Elle est organisée par l'association Planète Sciences [1]. Il ne s'agit pas de combats de robots ("Ohhhh...") mais de matchs au cours desquels les robots doivent prendre, déposer, empiler des objets afin de marquer des points, sans gêner l'autre robot.

Au commencement fut la Coupe E=M6 [2], créée par et diffusée dans l'émission E=M6. Avec un thème et un règlement changeant chaque année (Sumo, le football, les châteaux forts...), de nombreuses équipes venaient participer.

Dès 1998, devant le succès de la compétition en France, la Coupe s'est exportée dans d'autres pays européens avec la création d'Eurobot. Toutes ces compétitions suivent le même thème et le même règlement, ce qui permet des rencontres entre des équipes de différents pays. En 2008, 27 pays participaient à Eurobot.

En 2005, l'émission E=M6 se retire de l'organisation et de la diffusion de la compétition ; la Coupe E=M6 est renommée Coupe de France de robotique. Environ 200 équipes participent chaque année.

Le règlement d'Eurobot change chaque année. Néanmoins, voici certaines constantes que l'on retrouve invariablement d'une édition à l'autre :

- les robots sont autonomes (aucun contrôle à distance) ;
- un match oppose deux équipes et dure 90 secondes ;
- chaque équipe peut avoir un ou deux robots ;
- la table de jeu mesure 2×3 m et est globalement symétrique ou antisymétrique ;
- des emplacements sont prévus autour de la table afin de poser des balises, des petits circuits électroniques conçus par les équipes et permettant de repérer les robots sur la table.

Eurobot est une compétition conviviale. Cela peut déjà se constater au règlement :

- il y a des actions faciles à faire, de manière à ce qu'un maximum d'équipes puisse participer ;
- il y a des actions techniques, afin d'avoir des match impressionnants et de l'innovation technique ;
- il est interdit de gêner le robot adverse ou de manquer de fair-play (anti-jeu...) ;
- il y a très peu de contraintes techniques, hormis ce qui provient de mesures de sécurité. Sont par exemple interdits : air sous haute pression, haute tension, utilisation de liquides, de poudres, d'animaux [3]... Tout ceci permet d'obtenir parfois des robots qui ne font pas beaucoup de points mais sont très créatifs.
- les matchs sont ouverts au public et animés.

1. http://www.planete-sciences.org/
2. 'pas foulé sur le nom
3. morts ou vivants

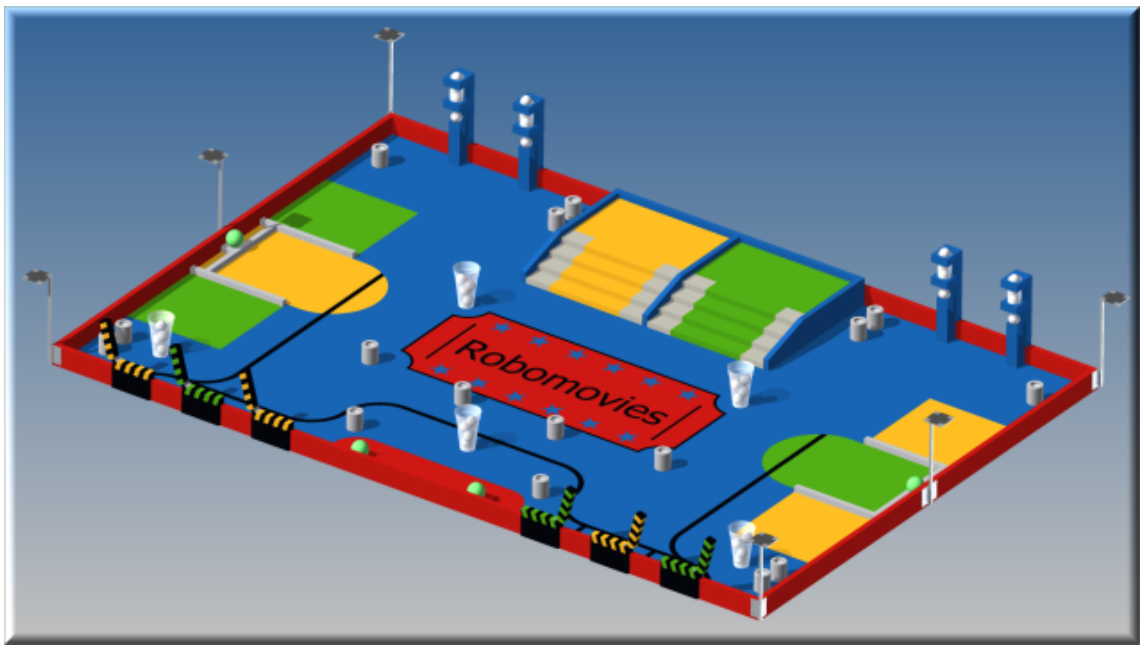


Figure 1.1 – La table d'Eurobot 2015

Ceci fait qu'au final, chaque année environ 200 équipes participent à la Coupe : IUT, clubs amateurs, écoles d'ingénieurs, universités… Technologiquement, cela va du robot construit avec des Lego au robot de plusieurs dizaines de milliers d'euros.

1.2 TechTheTroll

TechTheTroll est une équipe de deux membres toulousains qui viennent du club INTech de Télécom SudGrenoble. Notre club s'est illustré plusieurs fois lors de la Coupe de France de robotique ; nous avons notamment dans notre local la coupe 1995 et nous sommes arrivés 2^{es} en 2009 avec l'équipe TechTheWave et 9^{es} en 2012 avec l'équipe TechTheCoconut en 2015 avec l'équipe INTechnicolor et en 2016 avec l'équipe TechTheTowel.

Notre objectif principal est la transmission de connaissances ; d'ailleurs, ce sont les première année qui construisent le robot, les plus anciens s'occupant des formations, de la gestion de projet et d'assurer des transitions vers de nouvelles technologies (PIC vers AVR, AVR vers ARM, développement d'une balise, changement de langage de programmation…).

Tandis que l'équipe INTech est composée de première année, il y a souvent des anciens qui montent leurs propres projets, souvent avec l'objectif affiché de se marrer et non de gagner. Citons l'hexapode qui dansait l'année dernière à la Coupe, ou encore GobGob Technology, qui a eu le prix coup de cœur en 2013. TechTheTroll fait partie de ces spin-off du club INTech et reprend la nomenclature INTech'ienne en TechThe*.

L'objectif de TechTheTroll, c'est de faire le robot de nos rêves, avec toutes les technologies *overkill* qui nous plaisent. Et intégrer les trajectoires courbes était un point essentiel.

Au final, notre robot ne fut pas prêt à temps. Néanmoins, nous pensons pouvoir transmettre ce que nous avons appris sur le sujet.

Chapitre 2

Trajectoire courbe : quoi et pourquoi ?

Où y a quelques équations pour vous faire peur

2.1 Intérêt des trajectoires courbes

Une trajectoire courbe est une trajectoire composée d'une ligne courbe [1].

Traditionnellement, les robots d'Eurobot font deux types de mouvement : la translation (en avant ou en arrière) et la rotation à l'arrêt. De ce fait, lorsqu'un robot doit se déplacer entre deux points, il suit souvent une ligne brisée.

Ces mouvements ont l'inconvénient d'être lents, notamment à cause des multiples phases d'accélération et de décélération.

Il est important de remarquer que le rapport entre le gain de temps apporté par l'utilisation des trajectoires courbes et coût en temps de développement est pour l'immense majorité des équipes *très défavorable*. Beaucoup d'autres critères peuvent être améliorés avant d'en arriver aux trajectoires courbes.

Si je me permets d'insister là-dessus, c'est que je connais l'enthousiasme sans faille des 1A [2] qui ne se rendent pas vraiment compte de la difficulté de ce qu'ils proposent.

Je vais vous laisser dix secondes pour vous demander si, oui ou non, les trajectoires courbes sont un objectif profitable dans votre cas.

1 hippopotame, 2 hippopotames, 3 hippopotames, 4 hippopotames, 5 hippopotames, 6 hippopotames, 7 hippopotames, 8 hippopotames, 9 hippopotames, 10 hippopotames.

Vous êtes encore là ? Super, on peut commencer.

2.2 Critères concernant le choix des courbes

Il y a trois critères importants pour choisir la façon de faire ces trajectoires courbes :

— Leur expressivité, c'est-à-dire à quel point les courbes du modèle pourront se rapprocher de n'importe quelle courbe ;
— Les contraintes du robot en tant que système mécanique possédant une inertie, une précision limitée, ... (ce qui, en terme de courbe, impose la continuité de certaines grandeurs) ;
— La difficulté mathématique. Certaines courbes sont plus difficiles à exprimer et à calculer que d'autres (faut pas oublier qu'on est sur système embarqué...).

1. jusque là, tout le monde suit j'espère
2. 1e année. Les n00bs quoi

2.3 Choix des courbes

Nous traiterons uniquement de trois types de courbes.

2.3.1 Les arcs de cercles

Dans ce modèle, le robot suit une succession d'arcs de cercles et de lignes droites (qui peuvent être interprétées comme des formes dégénérées d'arc de cercles de rayon infini). Ces arcs de cercles sont enchaînés de manière à obtenir une courbe continue et dérivable (l'orientation du robot à la sortie d'un arc est la même qu'à l'entrée de l'arc suivant).

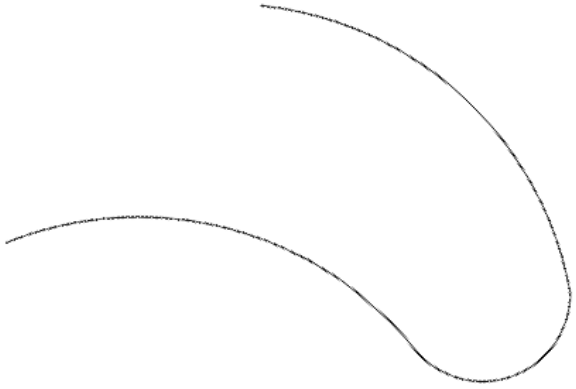


FIGURE 2.1 – Exemple d'une succession d'arcs de cercle. L'œil voit très bien ces discontinuités.

Mathématiquement, il s'agit d'un modèle simple, quasiment tous les calculs qu'on souhaite faire sont réalisables facilement. Un autre avantage : on a la preuve de l'optimalité d'une telle trajectoire en terme de minimisation de distance [RS90].

Par contre, il y a un problème : une succession d'arcs est seulement $\mathcal{C}^1$, ce qui signifie que le robot va subir des discontinuités [3] d'accélération, résultant en un mauvais suivi de la trajectoire. En effet, si on note ΔR la différence entre les rayons de deux arcs de cercle consécutifs, un robot à vitesse v et de masse m va subir au changement d'arc une force centrifuge F_c :

$$F_c = \frac{mv^2}{\Delta R}$$

Plus le robot sera rapide, plus cette force sera grande, ce qui risquera dans le meilleur des cas de faire glisser le robot orthogonalement à son orientation (induisant une erreur sur les codeurs qui ne peuvent pas mesurer ce mouvement latéral), et dans le pire des cas de soulever voire de faire chuter le robot si son centre de gravité est trop haut.

Des mécanismes d'anticipation de trajectoires existent afin de limiter les problèmes de suivi de trajectoire, mais ces discontinuités nous ont poussé à disqualifier ces courbes.

2.3.2 Les courbes de Bézier

Ah, les courbes de Bézier ! Elles sont zolies, et en plus elles sont $\mathcal{C}^\infty$, ce qui signifie que mécaniquement il n'y aura aucune discontinuité.

Le problème est cette fois plus d'ordre mathématique : leurs expressions ne sont pas simples et il n'est pas forcément facile de planifier un chemin en cherchant des points de contrôle qui peuvent être dans des obstacles.

3. que ceux qui ont pensé *"scontinuités"* sortent

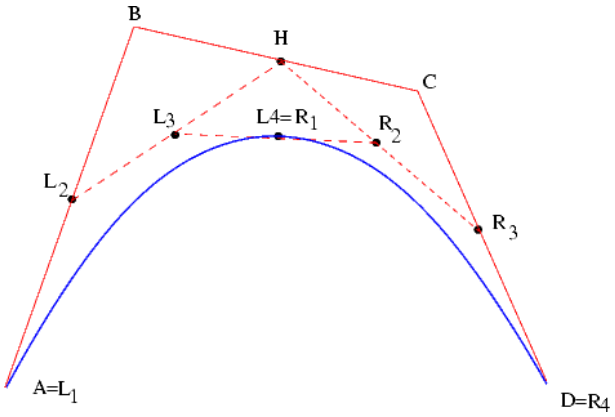


FIGURE 2.2 – Exemple d'une courbe de Bézier et de sa construction. Les points de contrôle sont A, B, C et D. La courbe de Bézier en elle-même est en bleu.

Cette fois, c'est plus pour des raisons de difficultés mathématiques que nous avons mis de côté ces courbes.

2.3.3 Les clothoïdes

Il est fort probable que vous n'ayez jamais entendu parler des clothoïdes [4], et pourtant elles ont des propriétés qui les rendent omniprésentes : saviez-vous que les autoroutes et les lignes de chemin de fer ont des arcs de clothoïdes ? [Baa82] [5] Ainsi que les montagnes russes ? Et puis, les clothoïdes, c'est joli [6] [BLP10].

Supposons que vous soyez au volant d'une voiture [7]. Vous commencez à rouler en ligne droite jusqu'à obtenir une vitesse constante, puis vous vous mettez à tourner progressivement votre volant. Vous tournez le volant à *vitesse constante* tandis que la (norme de la) vitesse de la voiture est aussi constante. Bravo, vous décrivez une spirale qui se resserre de plus en plus appelée clothoïde.

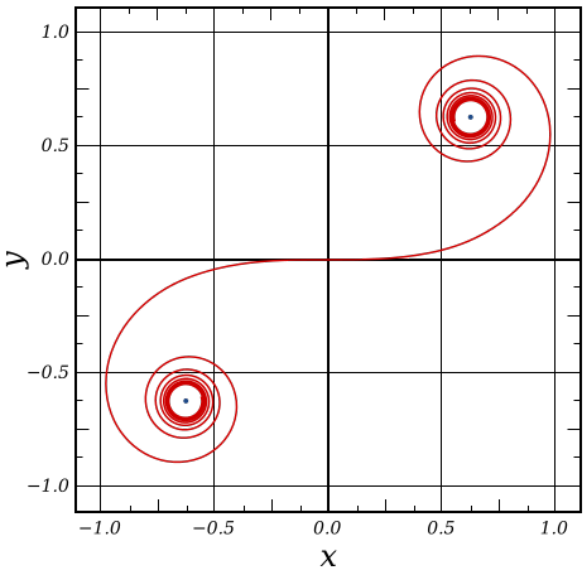


FIGURE 2.3 – La clothoïde, c'est elle *#swag*

Contrairement aux arcs de cercle où il faudrait à chaque changement d'arc tourner instantanément le volant à un autre angle, ici la modification de courbure est linéaire. Mécaniquement, ça se

4. aussi appelées spirale d'Euler ou spirale de Cornu
5. En fait, les autoroutes et les chemins de fer utilisent les clothoïdes uniquement comme interpolation. La plupart du temps, ces itinéraire sont des lignes droites ou des arcs de cercle reliés entre eux par des clothoïdes.
6. on les utilise d'ailleurs dans le design, les polices d'écriture, ...
7. ça marche aussi très bien avec un tractopelle

passe bien mieux qu'avec des arcs de cercles [SS90].

Remarque : dans la suite, $'$ désigne la dérivation par rapport à l'abscisse curviligne s.

Pour rappel, si $\theta(s)$ est l'orientation du robot en s, la courbure en s vaut :

$$\kappa(s) = \theta'(s)$$

En gros, plus la valeur absolue de la courbure est grande, plus on tourne vite. De plus, si on approche la trajectoire du robot en s par un cercle (appelé cercle osculateur), alors le rayon $R(s)$ de ce cercle est appelé rayon de courbure [8] et vaut [9] :

$$R(s) = \frac{1}{\kappa(s)}$$

Son équation paramétrique est :

$$p(t) = \int_0^t e^{iu^2}\, du \qquad (*)$$

Ainsi, une clothoïde est donc une courbe pour laquelle on a $\kappa(s)' = C$.

On peut facilement tracer une clothoïde unitaire ($C = 1$) grâce à son développement en série :

$$\begin{cases} x &= s - \frac{s^5}{1\cdot2\cdot5} + \frac{s^9}{1\cdot2\cdot3\cdot4\cdot9} - \ldots = \sum_{n=0}^{+\infty} \frac{(-1)^n \cdot s^{4n+1}}{(2n)!\cdot(4n+1)} \\ y &= \frac{s^3}{1\cdot3} - \frac{s^7}{1\cdot2\cdot3\cdot7} + \frac{s^{11}}{1\cdot2\cdot3\cdot4\cdot5\cdot11} - \ldots = \sum_{n=0}^{+\infty} \frac{(-1)^n \cdot s^{4n+3}}{(2n+1)!\cdot(4n+3)} \end{cases} \qquad (**)$$

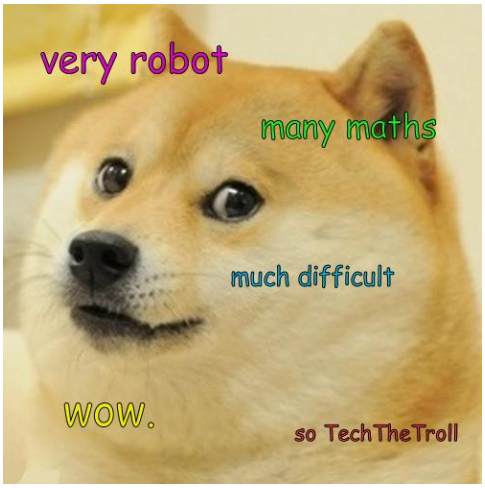
© Euler. RIP bro

Ces calculs de série peuvent être effectués une fois pour toutes, les clothoïdes étant toutes similaires entre elle (sauf dans le cas particulier de la ligne droite et des cercles, c'est-à-dire quand $\kappa' = 0$)

Par contre, il y a certains calculs qui devront être approchés, notamment la distance d'un point à la courbe [10].

De plus, si on fait intervenir la vitesse angulaire $\omega(s)$ et la vitesse instantanée $v(s)$ du robot, on a la relation suivante :

$$\omega(s) = v(s) \times \theta'(s) = v(s) \times \kappa(s)$$



Mais on n'est pas obligé de suivre une clothoïde à vitesse constante. Il suffit d'avoir :

$$\kappa(s)' = \left(\frac{\omega(s)}{v(s)}\right)' = C$$

soit [11]

$$\frac{\omega'(s) \times v(s) - v'(s) \times \omega(s)}{v(s)^2} = C$$

Au final, les clothoïdes sont bien pratiques, mécaniquement. Nous avons donc choisi une succession d'arcs de clothoïdes comme trajectoire courbe pour ces raisons. Mathématiquement, c'est globalement facile (on verra ça en détail dans un chapitre ultérieur).

Finalement, la conclusion à ce chapitre revient à l'article [BF12] : *"It is well known that the best or most pleasing curve is the clothoid, despite its transcendental form"*.

8. plus précisément, c'est le rayon algébrique, qui peut être négatif
9. notez, cette définition n'est pas utilisée par la suite. C'est juste que savoir que la courbure est l'inverse du rayon du courbure, dans la vie, c'est important
10. on en reparlera après
11. même si en pratique, rassurez-vous, on n'utilisera pas cette équation

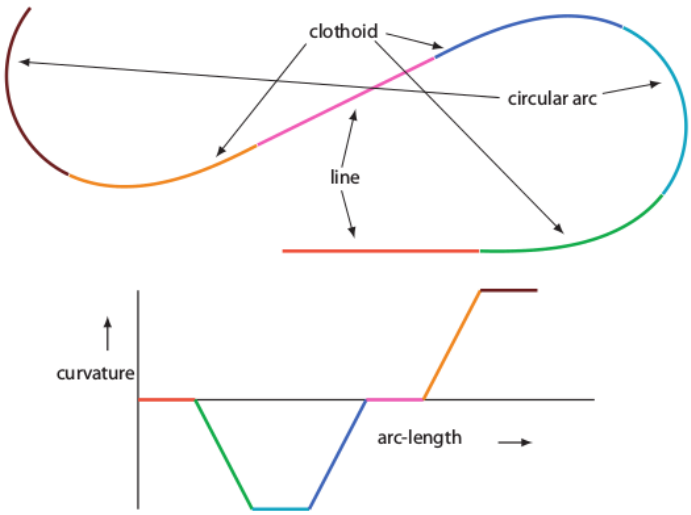


Figure 2.4 – Exemple d'une succession de clothoïdes. L'arc de cercle et la ligne droite en sont des cas particuliers. En bas, la courbure : elle est continue.

Figure 2.5 – La clothoïde ménage les moteurs comme les cœurs. Ici, deux clothoïdes sont raccordées.

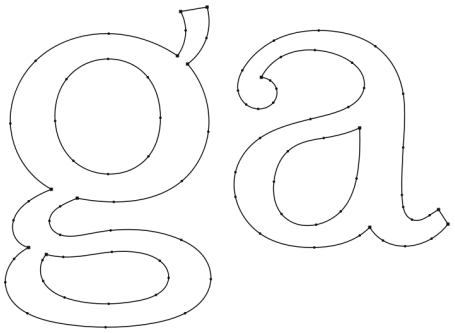


Figure 2.6 – Utilisation en design.

Chapitre 3

L'odométrie

Où je laisse RCVA faire tout le boulot

3.1 Codeurs

On appelle *codeur* un capteur rotatif qui permet de connaître les déplacements du robot. Il y en a deux, un pour chaque roue de propulsion. Ils sont soit intégrés aux moteurs, soit montés sur des roues folles. Ce deuxième cas, un peu plus lourd à mettre en œuvre mécaniquement, à l'avantage de détecter les patinages (quand les roues tournent mais glissent, ce qui fait que le robot ne bouge pas).

A partir des informations provenant de ces codeurs, on peut estimer la position du robot.

Il y a plusieurs moyens d'approcher la position du robot à partir du mouvement des codeurs en utilisant des approximations. On peut voir deux principales approximations.

3.1.1 Approximation linéaire

On suppose que le robot suit des lignes brisées. C'est un calcul simple à mettre en place qui suffit le plus souvent… sauf en cas de trajectoire courbe.

3.1.2 Approximation circulaire

Une approximation plus complexe (dont le cas linéaire est un cas particulier) mais qui est bien plus adaptée aux trajectoires courbes. En plus, c'est une approximation qui fait naturellement intervenir le calcul de la courbure que suit le robot ; et sans trop vouloir *spoiler*, ça nous sera utile.

Le lecteur intéressé est invité à lire le document de l'équipe RCVA sur le sujet [RCV10][1]. Il est complet et je n'ai rien à y ajouter.

1. http://www.rcva.fr/images/stories/site/cours/Odometrie2010/ODOMETRIE_2010.pdf

Chapitre 4

L'asservissement à une trajectoire courbe

Où on se rend compte qu'on n'est plus au Kansas

4.1 Suivi de chemin de C. Samson

4.1.1 L'équation

Le régulateur PID est un système d'asservissement qui s'appuie sur trois grandeurs : l'erreur, la dérivée de l'erreur et son intégrale. Il est très utilisé dans l'industrie et la robotique n'y fait pas exception.

Le problème avec le PID, c'est qu'il faut être asservi à une certaine valeur. Or, nous, on veut s'asservir à une *trajectoire*, qui est constituée d'une infinité de positions[réf. nécessaire].

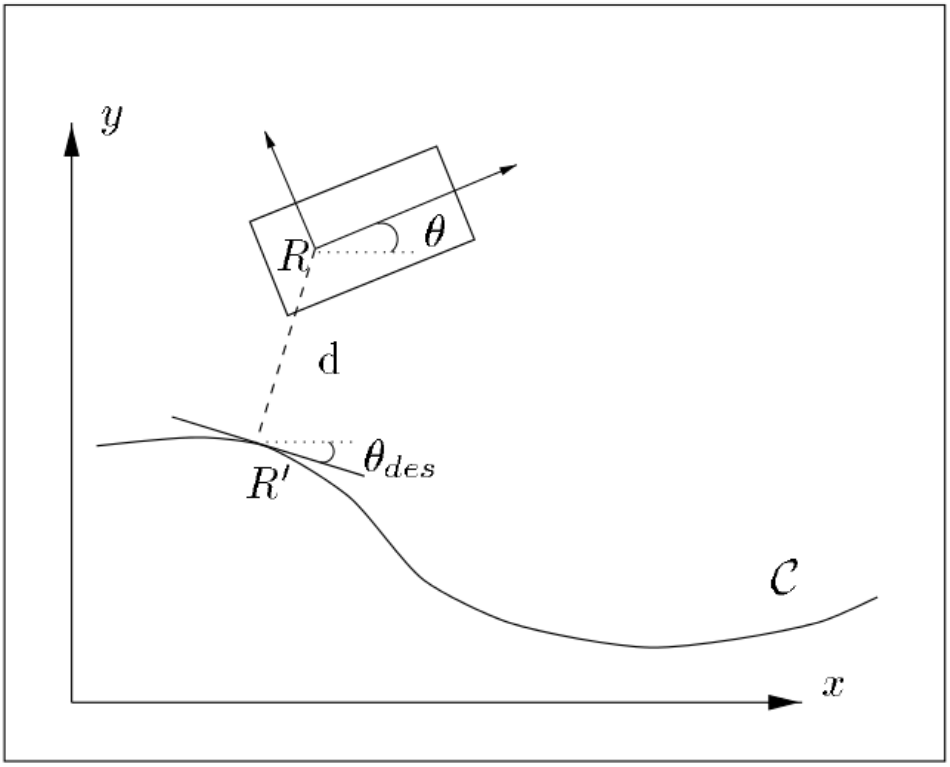


FIGURE 4.1 – Suivi de chemin (remarque : $\mathcal{C}$ sur le dessin n'est pas une succession de clothoïdes)

Introduisons quelques notations :
— R la position du robot
— R' le point de la courbe le plus proche de R
— θ l'orientation du robot

— θ_{des} cette orientation en R'
— $\kappa(R')$ la courbure de $\mathcal{C}$ en R'
— $d = d(R, R')$ l'erreur de position, algébrique (positive si le robot est à gauche de la courbe, négative sinon [1])
— $\theta_e = \theta - \theta_{des}$ l'erreur d'orientation, algébrique aussi
— κ_c la consigne de courbure à donner au robot

Alors on a [2] :

$$\kappa_c = \kappa(R') - k_1 d - k_2 \theta_e \qquad (\bigstar)$$

où $k_1 = \xi^2$ et $k_2 = 2\zeta\xi$ avec ξ la pulsation propre du système et ζ son amortissement.

La formule complète est disponible sur [MS93], [Sam95] ; la formule ci-dessus est une version approchée au premier ordre issue de [Gau99].

Ainsi, on dispose à tout instant de la consigne en courbure à fournir au robot pour qu'il suive la trajectoire. Il suffit alors de fixer une vitesse v (celle qu'on veut) pour en déduire la consigne en vitesse angulaire $\omega_c = v \times \kappa_c$.

A noter que d'autres régulateurs existent, comme [KC99]. Celui de Samson est adapté à la clothoïde car on peut très facilement connaître la courbure $\kappa(R')$ et l'orientation $\theta_{des}(R')$ en un point de la trajectoire. Seul le paramètre d est un peu compliqué à calculer.

4.1.2 Mais tout n'est pas si simple

Et concrètement, comment fait-on ? Il y a pas mal de paramètres qui entrent en jeu. Tout d'abord, pour déterminer la consigne du robot, il faut deux paramètres : κ_c (la courbure consigne) et v_c (la vitesse consigne). Grâce au régulateur de Samson, on a κ_c. J'ai dit plus haut que v_c est libre. Mais dans les faits, la vitesse est limitée par la courbure : si la courbure est grande (on tourne beaucoup), il ne faudra pas aller trop vite sinon la force centrifuge sera trop grande et le robot pourra soit lever le côté intérieur soit déraper vers l'extérieur du virage (selon la mécanique du robot).

Ainsi, la vitesse dépend de la courbure. Il y a alors deux possibilités : soit on se met à une vitesse qui ne pose pas ce genre de problème (en supposant que la courbure ne sera jamais trop grande, quitte à la caper), soit on ajuste la vitesse. Dans la suite de ce chapitre, on va supposer le second cas (le premier ne posant pas problème).

Pour la vitesse à appliquer au robot, il y a deux possibilités :
— soit la vitesse actuelle est toujours compatible avec κ_c, auquel cas on ne change rien ;
— soit il faut ralentir.

S'il faut ralentir, on se retrouve dans un scénario un peu compliqué : et si la consigne en courbure augmente si vite que la consigne en vitesse ne peut pas suivre, du fait de la décélération limitée [3] ?

4.2 Algorithme d'asservissement

On suppose disposer d'un asservissement en vitesse à chaque roue (je vous laisse adapter si ce n'est pas le cas) qui utilise les consignes v_{gc} (consigne en vitesse de la roue gauche) et v_{dc} (consigne en vitesse de la roue droite). D'ailleurs, ces asservissements sont liés, car les contraintes mécaniques des moteurs font qu'il y a trois bornes sur acc_g (accélération de la roue gauche) et acc_d (accélération de la roue droite) : $|acc_g| < C_1$, $|acc_d| < C_1$, $|acc_g + acc_d| < C_2$ et $|acc_g - acc_d| < C_3$. Contrairement aux déplacements en ligne droite et aux rotations sans translation où $|v_{gc}| \approx |v_{dc}|$), $|v_{gc}|$ et $|v_{dc}|$ peuvent être différentes dans le cas des trajectoires courbes. Je ferme la parenthèse.

On suppose enfin aller en marche avant. Toutes les vitesses sont donc supposées positives. L'algorithme s'adapte bien au cas de la marche arrière.

Définissons alors quelques nouvelles notations :
— v_r la vitesse réelle (mesurée) du robot ;

1. par rapport à la direction dans laquelle le robot avance. La formule change légèrement si le robot suit une trajectoire courbe en marche arrière
2. s'il y a une seule formule à retenir, c'est celle-là
3. en supposant, bien sûr, qu'on puisse arriver dans un tel cas limite.

— v_{max} la vitesse maximale du robot(il pourra aller moins vite s'il y est forcé) ;
— $v_{lim}(\kappa)$ la vitesse limite (maximale) en fonction de la courbure ;
— $\kappa_{lim}(v)$ la courbure limite (maximale) en fonction de la vitesse [4] ;
— κ_{max} la courbure maximale que le robot peut suivre ;
— κ_s la consigne en courbure provenant de (★) ;
— κ_c la consigne effective qu'on va suivre ;
— v_c la consigne en vitesse linéaire ;
— ω_c la consigne en vitesse angulaire ;
— C la moitié de la distance entre les deux roues de propulsion.

La relation entre courbure et vitesse est :

$$v_{lim}(\kappa) = \min\left(\sqrt{\frac{gd}{\kappa z}}, \sqrt{\frac{k_a g}{\kappa}}\right)$$

où d est la demi-distance entre les faces extérieures des roues de propulsion, z l'altitude du centre de gravité du robot, k_a le coefficient d'adhérence des roues de propulsion sur la table. Il y a deux facteurs : le premier correspond au point de décollement et le second au point de patinage. Selon le robot, ce sera l'un ou l'autre qui sera limitant.

Notre algorithme d'asservissement est le suivant [5] :

Algorithm 1: Asservissement à la courbe $\mathcal{C}$

Input: v_{max} la vitesse maximale, $\mathcal{C}$ la courbe à suivre

Loop
- $R' \leftarrow Proj_{\mathcal{C}}(R)$ // on projette R sur $\mathcal{C}$
- $\kappa_s \leftarrow EquSamson(\kappa(R'), d(R,R'), \theta_{robot} - \theta(R'))$ // calcul de κ_s avec (★)
- $\kappa_c \leftarrow \min(\kappa_{lim}(v_r), \kappa_s, \kappa_{max})$ // on s'approche au mieux de la consigne idéale
- $v_c \leftarrow \min(v_{lim}(\kappa_s), v_{max})$ // on limite la consigne en vitesse si besoin est
- $\omega_c \leftarrow \kappa_c \times v_r$ // la consigne angulaire
- $v_{gc} \leftarrow v_c - C \times \omega_c$ // consigne en vitesse moteur gauche
- $v_{dc} \leftarrow v_c + C \times \omega_c$ // onsigne en vitesse moteur droit
- $AsserVitesse(v_{gc}, v_{dc})$ // PID classique qui commande les moteurs

Grâce aux propriétés de la clothoïde, l'évolution de toutes ces grandeurs est continue. De plus, on sait que la dérivée de $\kappa(R')$ est bornée, et donc que celle de κ_c le sera probablement aussi.

L'algorithme ci-dessus permet d'être asservi à une trajectoire mais ne permet pas de commencer ou de finir la trajectoire. En effet, il y a bien un moment où il faudra s'arrêter... [6] Une manière d'asservir un robot est d'utiliser un trapèze de vitesse. Le mouvement est alors décomposé en trois parties : accélération constante, vitesse constante et enfin décélération constante.

On va supposer disposer d'un régulateur PID sur la distance entre le robot et sa destination qui nous fournira une consigne v_k pour la vitesse linéaire. Il suffira alors d'utiliser l'algorithme ci-dessus en calculant à chaque itération $v_{max} = \min(v_k, v_t)$, où v_t est la vitesse maximale que le robot ne pourra jamais dépasser, celle du sommet du trapèze.

4. si $v = v_{lim}(\kappa)$, alors $\kappa = \kappa_{lim}(v)$ (et inversement). Pour le dire plus mathématiquement, v_{lim} est la bijection réciproque de κ_{lim}. De plus, ces deux fonctions sont décroissantes (plus on tourne, moins on doit aller vite ; plus on va vite, moins on peut tourner)

5. Protip : tous les meilleurs algos contiennent une boucle infinie

6. je parle d'arrêts *conventionnels*. Se prendre un bord de table n'est pas une solution viable pour s'arrêter.

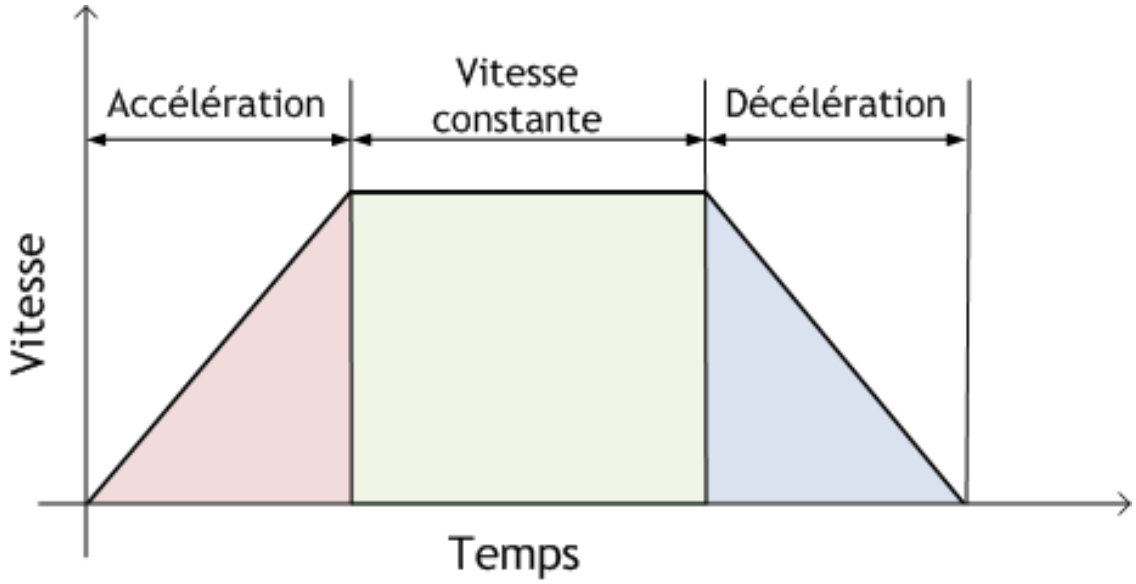


Figure 4.2 – La vitesse du robot au cours d'en déplacement en utilisant le profil trapézoïdal.

Chapitre 5

La planification de trajectoire courbe

Où on fait copuler deux pathfindings

5.1 Besoins

Super, on a un robot qui suit des trajectoires courbes ![1] Mais c'est encore mieux si le robot sait *quelle* courbe il doit suivre.

En effet, un grand avantage des trajectoires courbes est de pouvoir évoluer rapidement dans un environnement rempli d'obstacles (sinon, on prendrait une ligne droite), mais cela signifie donc que ces courbes dépendent d'obstacles dynamiques et doivent être générées dynamiquement.

Il y a une foultitude de façon de faire de la recherche de chemin. Tous les algorithmes ou presque sont des développements de l'algorithme A*.

Il faut donc définir nos besoins en terme de recherche de chemin. Il y en a deux :

— il faut récupérer une trajectoire sous la forme d'une suite de clothoïdes ;

— il faut une recherche rapide au cas où un obstacle surgit pendant qu'on roule.

Sans vouloir trop vous *spoiler*, il y a deux algorithmes de recherche de chemin utilisés ensemble ; l'un couvrira le premier besoin, l'autre le second.

Il ne sera pas ici question d'implémentation des algorithmes [2].

5.2 Cours de rattrapage rapide

Je vais trè-ès rapidement rappeler les algorithmes de base de recherche de chemin.

5.2.1 L'algorithme de Dijkstra

Cet algorithme est heureusement plus simple que le nom de son inventeur ; c'est l'ancêtre de la majorité des algorithmes de recherche de chemin modernes. Ainsi, je vous le présente non pas pour ses performances, mais parce que le comprendre est nécessaire pour saisir les prochains algorithmes.

Le principe de cet algorithme est de partir du point de départ et d'affecter à chaque autre nœud sa distance minimale au nœud de départ. Pour cela, on prend le nœud N qu'on a pas encore visité et qui minimise distance au point de départ, et on affecte à ses voisins V_i la distance de N à laquelle on ajoute la valuation de l'arc entre N et V_i. Si un nœud est revisité, alors on met à jour sa valeur si elle diminue sa distance.

Un exemple clair, étape par étape, se trouve sur kiwi-pédia. Je vous invite à le lire : c'est un algorithme fondamental, qui n'est pas bien compliqué une fois qu'il est compris.

À la fin de l'algorithme, on dispose du plus court chemin depuis le nœud de départ vers n'importe quel point. C'est-à-dire qu'il n'a pas seulement trouvé le plus court chemin du point de départ au point d'arrivée, mais beaucoup d'autres ! D'une manière générale, Dijkstra part dans toutes les

1. si ce n'est pas le cas, veuillez consulter notre SAV au `?v=Q16KpquGsIc`
2. ces questions sont traitées dans les articles de `http://techthetroll.wordpress.com/`

directions, et ne prend pas en compte le nœud d'arrivée dans ses calculs (sauf pour vérifier s'il a fini).

Une version plus rapide et plus maligne existe : l'algorithme A*.

5.2.2 L'algorithme A*

L'algorithme A* (qu'on prononce "A star" ou "A étoile") est une extension de l'algorithme de Dijkstra. Sa différence est qu'il va chercher à se diriger vers le nœud d'arrivée, à l'aide d'une heuristique. Une heuristique est une fonction qui, pour deux nœuds, renvoie une valeur. Cette fonction peut être quelconque mais doit satisfaire une contrainte : elle ne doit jamais surestimer la distance réelle.

Un exemple simple dans le cadre de la recherche de chemin est la distance euclidienne (la distance à vol d'oiseau) ; c'est une certaine distance qui est forcément inférieure ou égale à la distance réelle à parcourir. On voit bien ici que l'heuristique permet de se diriger vers l'arrivée : prendre la direction de l'arrivée à vol d'oiseau est un bon moyen d'arriver rapidement. Un autre exemple simple d'heuristique est la fonction nulle : en effet, la distance entre deux nœuds est forcément positive. Dans ce cas, l'algorithme A* se réduit à l'algorithme de Dijkstra [3].

5.3 A* pour trajectoire courbe

5.3.1 Adaptation aux clothoïdes

Remarque : on va faire simple, et minimiser la distance parcourue. Étant donné que la vitesse maximale dépend de la courbure, plus courte distance ne signifie pas nécessairement plus courte durée de trajet. Ce cas est donc laissé au lecteur [4].

Il y a plusieurs moyens de faire du pathfinding avec des clothoïdes. En voici quelques uns.

la méthode "interpolation à l'arrache" (continuité G^1) : grâce à des formules que des gens très intelligents ont trouvées, on peut trouver la bonne clothoïde pour aller d'un point A à un point B [BF12]. Problème : une clothoïde n'a pas assez de degré de liberté. Concrètement, ça veut dire qu'il y aura continuité de position et d'orientation mais pas de courbure à l'entrée et à la sortie de la clothoïde (continuité G^1). Ouille ! (à se demander pourquoi utiliser des clothoïdes si la trajectoire obtenue n'utilise pas ses propriétés...)

la méthode "interpolation propre" (continuité G^2) : si une clothoïde n'a pas assez de degré de liberté, peut-être peut-on enchaîner deux clothoïdes pour aller de A à B ? Effectivement, ça marche : on peut trouver deux clothoïdes qui vérifient en tout point la continuité de la courbure (continuité G^2) [DKK98]. Par contre les calculs sont bien plus lourds. Heureusement, il y a des cas assez simples, où la paire de clothoïde est symétrique. Cette méthode et la précédente ont un inconvénient : peut-être la courbe qui permettrait de joindre deux points est impossible à suivre par le robot ; en effet, celui-ci ne peut pas rouler à une courbure trop grande.

la méthode "à la Bézier" (avec points de contrôle) : on peut également générer une trajectoire clothoïdique [5] avec des points de contrôle, comme ce qui est fait avec les courbes de Bézier. Pas très pratique quand il y a des obstacles, car même si les points de contrôle peuvent se voir mutuellement, la trajectoire finale peut se prendre des obstacles [WM05].

la méthode "on voit où ça mène..." : la plus simple. Quand on exécute le pathfinding, on a besoin de ses voisins. Généralement, ces voisins sont des points répartis sur une grille pour lesquels on trouve un chemin élémentaire pour y aller. Dans cette approche, on fait le contraire : depuis un point de la table, on peut définir des voisins par rapport au trajet qu'on va réaliser, en faisant une sorte d'arbre des possibles [Kno06]. Par exemple, le voisin 1 va être le point où on arrive avec une clothoïde de paramètre 1 ; le voisin 2, avec une clothoïde de paramètre 2, etc.

3. quantité d'heuristiques existent : les rapides, les efficaces, celles qui paraissent "humaines" (pratique pour les IA de jeux vidéo), ...

4. et toc.

5. non, ce mot n'existe pas, peu importe la réforme de l'orthographe utilisée

Les voisins ne sont plus disposés en grille, et au final on ne peut pas demander à l'algo "va là". On est obligé de voir jusqu'où il va de lui-même aller. L'inconvénient est que même s'il y a une solution, on est pas sûr de la trouver.

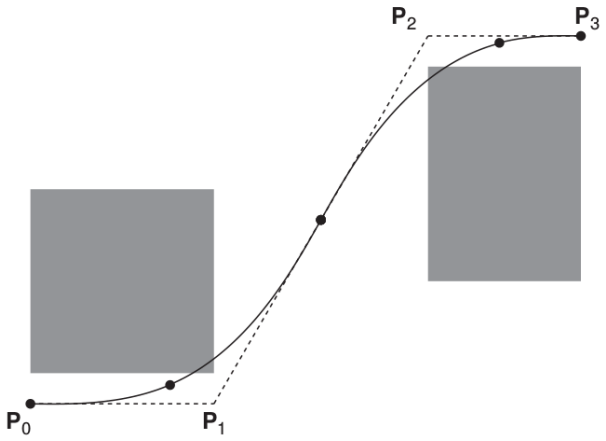


Figure 5.1 – Le problème des points de contrôle

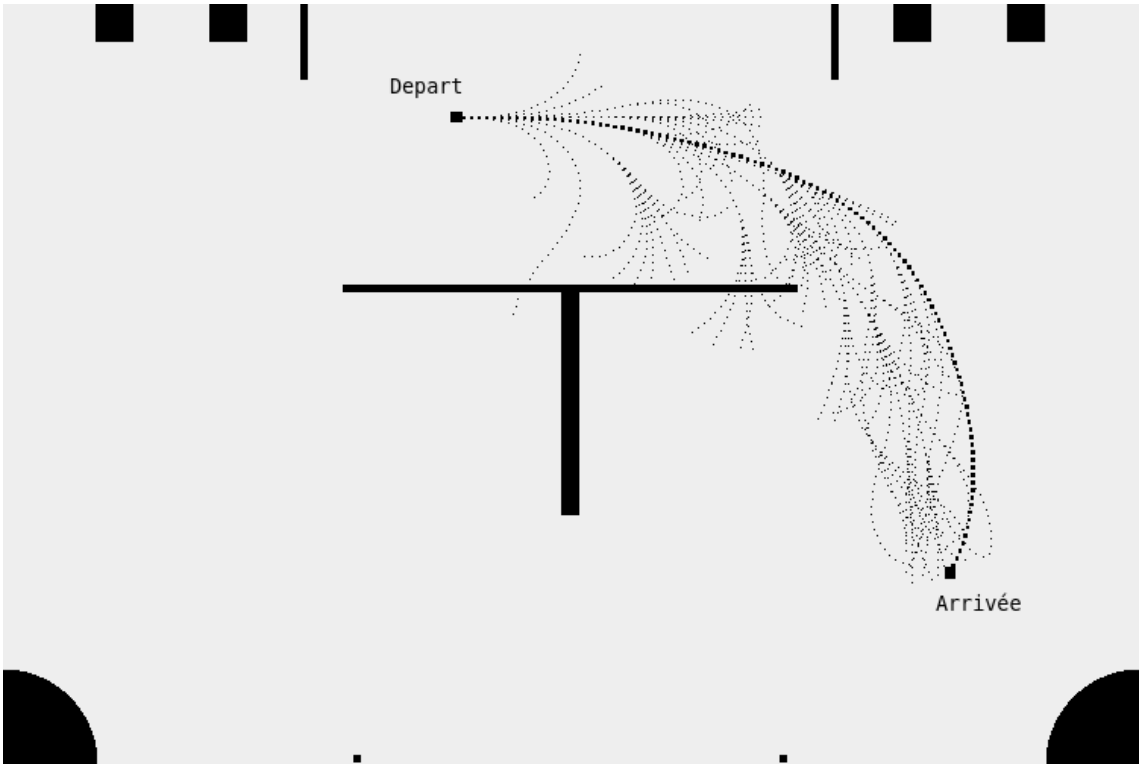


Figure 5.2 – La dernière méthode et son exploration de l'arbre des trajectoires possibles. Les trajectoires fines ont été explorées et la trajectoire épaisse a été retenue au final.

Nous avons choisi la dernière méthode. Bien qu'elle ne soit pas forcément la plus pratique, elle a le mérite d'être la plus simple mathématiquement, ce qui a le double avantage de ne pas être prise de tête et d'être réalisable en temps réel.

Il y a un réel problème sur le fait qu'on ne puisse spécifier une destination.

5.3.2 L'interpolation cubique pour arriver à destination

La construction d'un chemin de la manière ci-dessus ne permet pas de spécifier où on veut aller. Or, nos tests nous ont montré [6] qu'on ne peut pas juste demander au pathfinding de se rapprocher

6. dans des circonstances tout à fait malheureuses

autant qu'il le peut : on arrive à des situations absurdes où le robot enchaîne les marches avant et les marches arrière pour se rapprocher assez de la destination [7].

Le dernier arc du pathfinding est donc particulier. Dans l'idéal, on aimerait pouvoir générer une courbe avec une courbure continue, qui permette la continuité de certaines grandeurs avec les arcs précédents et qui arrive à la bonne destination.

On peut chercher à calculer le paramètre κ' qui donnerait la clothoïde qui arrive pile là où on veut. Mais un tel calcul n'est pas si simple ; je vous propose une autre méthode. Une interpolation cubique [8].

Une interpolation cubique est une interpolation (trouver une trajectoire qui passe par certains points) qui utilise une polynôme du troisième degré (d'où le mot "cubique"). D'ailleurs, la clothoïde s'approche très bien par un polynôme du troisième degré pour des courbures assez faibles. De plus, un polynôme étant C^∞, on aura continuité de la position, de l'orientation et de la courbure durant tout l'arc.

Dans la suite de cette section, $'$ désignera la dérivée par rapport à t.

On va chercher un polynôme $P : [0,1] \longrightarrow \mathbb{R}^2$,

$$P(t) = \begin{pmatrix} P_x \\ P_y \end{pmatrix}(t) = \begin{pmatrix} a_x \\ a_y \end{pmatrix} t^3 + \begin{pmatrix} b_x \\ b_y \end{pmatrix} t^2 + \begin{pmatrix} c_x \\ c_y \end{pmatrix} t + \begin{pmatrix} d_x \\ d_y \end{pmatrix} \tag{1'}$$

$$P'(t) = \begin{pmatrix} P'_x \\ P'_y \end{pmatrix}(t) = 3\begin{pmatrix} a_x \\ a_y \end{pmatrix} t^2 + 2\begin{pmatrix} b_x \\ b_y \end{pmatrix} t + \begin{pmatrix} c_x \\ c_y \end{pmatrix} \tag{2'}$$

$$P''(t) = \begin{pmatrix} P''_x \\ P''_y \end{pmatrix}(t) = 6\begin{pmatrix} a_x \\ a_y \end{pmatrix} t + 2\begin{pmatrix} b_x \\ b_y \end{pmatrix} \tag{3'}$$

On cherche à interpolation entre deux points A et B, de telle manière qu'il y ait continuité de position, d'orientation et de courbure en A et qu'on arrive bien en B.

La contrainte de la continuité de la position en A donne :

$$P(0) = \begin{pmatrix} d_x \\ d_y \end{pmatrix} = \begin{pmatrix} A_x \\ A_y \end{pmatrix}$$

En notant θ_A l'orientation du robot en A et α la norme de $P'(0)$, la continuité de l'orientation en A donne :

$$P'(0) = \begin{pmatrix} c_x \\ c_y \end{pmatrix} = \alpha \begin{pmatrix} \cos(\theta_A) \\ \sin(\theta_A) \end{pmatrix}$$

Pour la continuité de la courbure, on utilise la formule suivante [9] :

$$\kappa(t) = \frac{P'_x(t)P''_y(t) - P''_x(t)P'_y(t)}{(P'_x(t)^2 + P'_y(t)^2)^{3/2}} \tag{4'}$$

En notant κ_A la courbure du robot en A, la continuité de la courbure en A donne donc :

$$\kappa_A = \frac{c_x \times 2b_y - c_y \times 2b_x}{\alpha^3}$$

La continuité de la position en B donne :

$$\begin{pmatrix} P_x \\ P_y \end{pmatrix}(1) = \begin{pmatrix} a_x \\ a_y \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix} = \begin{pmatrix} B_x \\ B_y \end{pmatrix}$$

7. ce qui annule complètement l'avantage des trajectoires courbes
8. En ce moment même, la clothoïde crie à l'infidélité.
9. non, cette formule ne se devine pas

Ce qui nous donne comme solution, sachant que α est un paramètre libre :

$$\begin{cases} d_x &= A_x \\ d_y &= A_y \\ c_x &= \alpha \cos(\theta_A) \\ c_y &= \alpha \sin(\theta_A) \\ b_x &= -\frac{\kappa\alpha^2}{2} \sin(\theta_A) \\ b_y &= \frac{\kappa\alpha^2}{2} \cos(\theta_A) \\ a_x &= B_x - d_x - c_x - b_x \\ a_y &= B_y - d_y - c_y - b_y \end{cases} \quad (***)$$

A vrai dire, cette équation n'est pas très satisfaisante. On ne contrôle pas l'orientation et la courbure à l'arrivée, alors qu'on pourrait parfaitement en termes d'expressivité de l'interpolation cubique : en effet, on aurait huit paramètres (a_x, a_y, b_x, b_y, c_x, c_y, d_x, d_y) et huit équations (deux fois deux pour la continuité en position en A et en B, deux pour la continuité en orientation en A et en B, deux pour la continuité en courbure en A et en B).

Remarque : cette section étant écrite peu de temps avant la coupe, je n'ai pas eu le temps d'approfondir cette idée. Contactez-moi si vous avez avancé sur la question !

Pour notre part, nous utilisons la solution ci-dessus en faisant varier α afin d'obtenir une courbe qui satisfasse les contraintes en courbure.

Dans le cadre du système proposé par ce rapport, on souhaite connaître la position, l'orientation et la courbure en tout point pour suivre correctement la trajectoire. Pour la position, il suffit d'utiliser (1'). L'orientation en un point t se déduit de la vitesse (2') et vaut $\theta(t) = \arctan(P'_y(t), P'_x(t))$. La courbure est directement donnée par l'équation (4').

Ainsi, l'A* arrivera à sa destination tout en conservant la continuité de la position, de l'orientation et de la courbure.

5.3.3 En urgence : le chemin partiel

On a à présent un pathfinding qui nous trouve une trajectoire courbe. Mais faire ces calculs prend du temps. Mais alors, que faire si je fonce à toute vitesse dans un robot adverse ? [10] Comment obtenir rapidement un itinéraire de secours ? [11]

Il y a la solution de calculer à tout moment un tel itinéraire de secours. Mais on va faire plus malin, vous et moi.

On va demander au pathfinding de nous donner un itinéraire partiel, qui n'est pas encore arrivé à destination mais qui nous en rapproche. Ainsi, même quand le pathfinding n'a pas encore fini de calculer le chemin, l'asservissement pourra déjà commencer à suivre ce chemin.

Mais comment faire ça ? On peut avoir à tout moment le meilleur chemin partiel trouvé jusque là… mais rien ne dit qu'on ne trouvera pas mieux bientôt, auquel cas le chemin change et le précédent "meilleur chemin partiel" ne l'est plus. Or, il faut que le chemin partiel qu'on donne à l'asservissement pour amorcer le mouvement doit être présent dans l'itinéraire final ! On ne peut pas juste dire "va à droite ! Heu… non, à gauche en fait !".

La solution, c'est la liste ouverte. À tout instant, dans l'algorithme A*, la liste ouverte contient la liste des nœuds non visités les plus intéressants. Si on retire tous les nœuds de la liste ouverte sauf le plus intéressant, alors le chemin final sera obligé de passer par ce nœud (car toutes les alternatives auront été retirées). Ça revient à relancer le pathfinding avec comme nœud de départ la fin de l'itinéraire partiel.

Ainsi, l'itinéraire partiel sera forcément dans l'itinéraire final. Ouf ! Il ne reste plus qu'à programmer le pathfinding de telle manière qu'une fois un danger imminent détecté, il réponde au bout d'un

10. Ce qui, soyons-en conscients, arrivera fatalement

11. Dans le doute, ayez un robot solide. De toute façon, dans le repère de votre robot c'est l'ennemi qui vous a foncé dessus. Et la priorité à droite, hein ?

certain temps au maximum [12], soit avec un itinéraire complet soit un chemin partiel, afin d'être sûr mécaniquement de pouvoir réagir à temps. [13]

5.3.4 Les limites du chemin partiel

Il y a tout de même une importante limite à la construction de ce chemin partiel. En effet, que se passe-t-il si le chemin partiel n'est pas bon et nous emmène dans un cul-de-sac ? En effet, A* utilise une heuristique pour savoir où se diriger en priorité ; mais celle-ci cherche à minimiser la distance à vol d'oiseau, et n'est pas fiable (dans le sens où elle peut aboutir à des cul-de-sac) en cas d'obstacles.

Pour expliquer cela, prenons un cas concret.

FIGURE 5.3 – Rainbow Dash suit un chemin modélisé par une suite d'anneaux en vitesse supersonique

FIGURE 5.4 – Un pikachu sauvage est apparu ! Vite, il faut recalculer une trajectoire en urgence !

5.4 Pathfinding D*Lite : la rapidité incarnée

5.4.1 Un pathfinding à rebours

Lorsqu'on demande à l'algorithme D*Lite [KL05] de calculer l'itinéraire entre X et A (A pour "arrivée"), il exécute un simple A*. À une différence près : au lieu de calculer de chemin de X vers A, il calcule le chemin de A vers X. Mais pourquoi ?

Dans le cas classique, quand le robot veut se rendre en A depuis X, il calcule A*(X, A) et bouge. Si à un moment, à cause d'un obstacle, il doit rechercher un itinéraire, ça se fera depuis sa situation courante jusqu'en A, c'est-à-dire A*(Y, A) : en fait, on va toujours sur un chemin recalculer des itinéraires vers A. Et c'est embêtant, car les calculs de A* concernent la distance entre le point de départ

12. qui dépend de la vitesse du robot, de la distance de l'obstacle, et du taux de réussite au jet d'esquive que vous désirez
13. et même si ça ne marche pas et qu'il y a collision… au moins, on pourra pas dire que vous avez pas tout fait pour l'éviter. Dans le doute, ayez un robot solide !

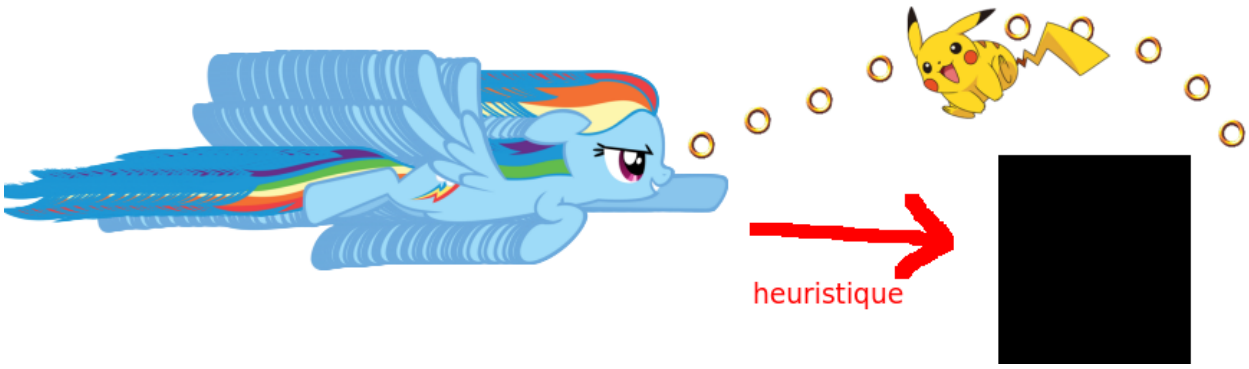


FIGURE 5.5 – Oh non, le chemin partiel trouvé en se basant sur l'heuristique va mener Rainbow Dash droit dans le mur ! Quel suspense insoutenable !

et les autres points. Or, comme le point de départ change, on ne peut conserver aucun calcul entre A*(X, A) et A*(Y, A). Par contre, si on cherche un itinéraire depuis la fin, A* va stocker les distances entre A et les autres points, et ces distances pourront être réutilisées. Habile.

5.4.2 La replanification rapide

Il n'y a pas que ça. L'algorithme est un peu compliqué, mais en gros il ne recalcule que ce qu'il faut. En plus, plus les obstacles qui apparaissent sont proches du robot, plus le recalcul est rapide : génial pour les robots qui utilisent des capteurs de proximité (infrarouge, ultrason) et qui donc voient ce qui est proche d'eux !

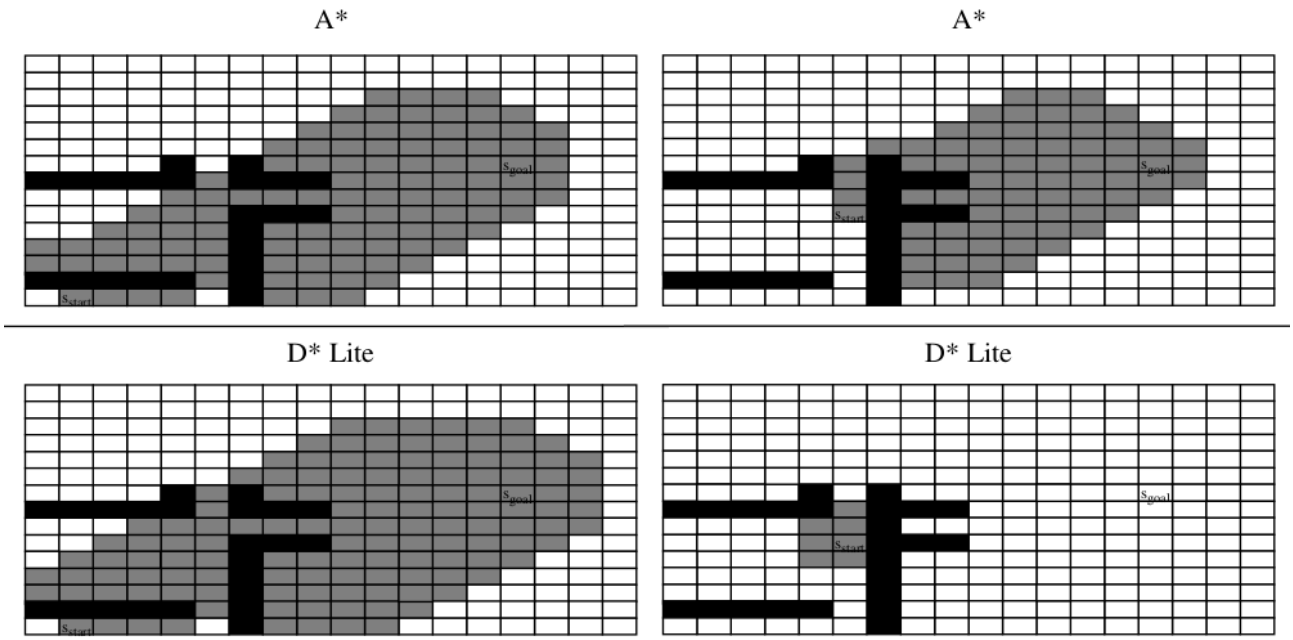


FIGURE 5.6 – Comparaison entre A* et D*Lite. A gauche, les nœuds calculés pour une première planification. A droite, les nœuds recalculés pour une replanification.

Le gros avantage du D*Lite est donc sa faculté à recalculer très rapidement un itinéraire (une fraction d'un calcul complet). Par contre, comme le A* il fournit une ligne brisée comme itinéraire.

5.5 FUUUU-SION !

Génial, on a un pathfinding qui fait des trajectoires courbes mais qu'on peut facilement mettre en défaut et un pathfinding qui est très rapide mais qui fournit des trajectoires en ligne brisée.

Or, de la même façon qu'on ne peut pas hybrider un oiseau et un zombie pour créer une armée de zombies volants [14], on peut se demander comment il est possible d'allier ces pathfindings de manière à arriver à quelque chose qui conviennent à nos besoins.

La réponse tient en un mot :

L'HEURISTIQUE !

En effet, quand le A* se plante, c'est que l'heuristique lui disait qu'il cherchait du bon côté alors qu'il arrivait à un cul-de-sac. Car les heuristiques classiques ne prennent pas en compte le terrain et les obstacles.

Mais si l'heuristique du A* est la distance du chemin de P à l'arrivée A calculée par le D*Lite, alors A* va chercher dans la direction où il y a *vraiment* une sortie. Il ne pourra pas être piégé. Et s'il n'y a aucun moyen de parvenir à l'arrivée, le D*Lite le saura déjà.

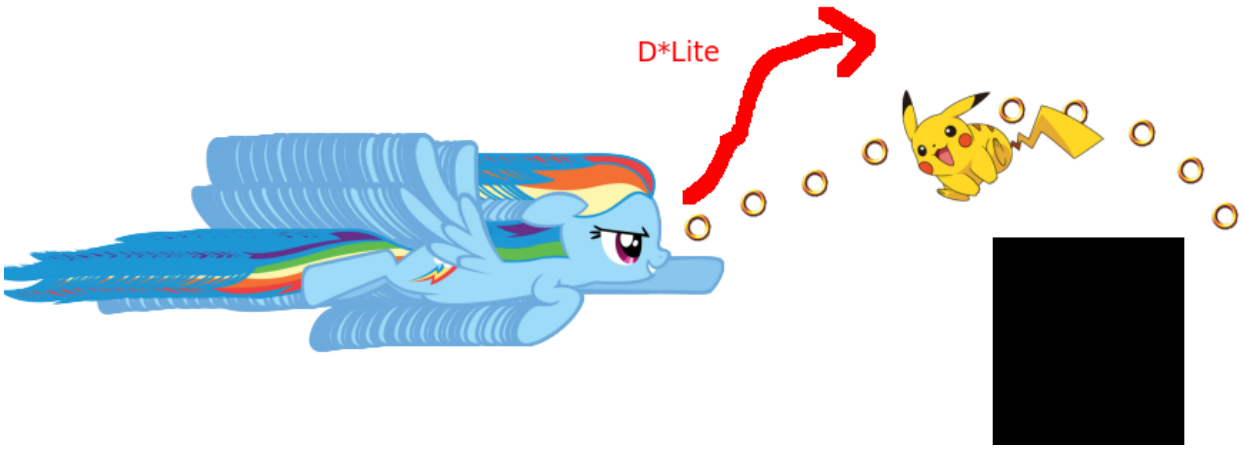


FIGURE 5.7 – Ouf ! Dès qu'elle a vu Pikachu, Rainbow Dash a exécuté son D*Lite. Grâce à cette nouvelle heuristique, Rainbow Dash peut réévaluer rapidement sa trajectoire et éviter l'obstacle. Bravo Dashie !

Concrètement, l'algo se fait donc en deux étapes :

— calcul des distances des points à l'arrivée par le D*Lite ;

— calcul de trajectoire avec A* dont l'heuristique est la distance calculée par le D*Lite.

Le gros avantage est qu'en cas d'urgence, le chemin partiel du A* ne nous mènera pas à un cul-de-sac : la trajectoire reste optimale.

14. et c'est pas faute de pas avoir essayé

Chapitre 6

L'implémentation des trajectoires courbes

Où y a (enfin !) du code

6.1 Ce qu'on cherche à obtenir

On va détailler dans ce chapitre les algorithmes permettant la mise en place des trajectoires courbes utilisant les clothoïdes. Une trajectoire courbe étant composée d'une infinité de points, notre premier travail va donc être de la discrétiser.

D'une manière générale, on va chercher à obtenir la chose suivante : on va calculer une succession régulière [1] de points qui approcheront la trajectoire courbe qu'on cherche à suivre.

Plus exactement, on aimerait pour chacun de ces points connaître :

— leur position sur la table ;
— l'orientation de la trajectoire en ce point ;
— la courbure de la trajectoire en ce point.

Ces informations sont en effet nécessaires à l'asservissement (cf. l'équation (★)) du robot à la trajectoire.

Cette succession de points ne sera pas une seule clothoïde mais un enchaînement, du genre "pendant trois points on prend une dérivée de la courbe de 3, puis pendant 10 points on ne modifie pas la courbure puis on finit avec 5 points avec une dérivée de courbure de -1". On aura donc des morceaux de clothoïdes de paramètre différent.

Mais ça fait beaucoup, beaucoup de possibilités de trajectoire courbes. Afin de simplifier le travail du pathfinding de trajectoire courbe (le A*), on va simplifier ceci et fixer un nombre de points pendant lesquels la dérivée de la courbure sera constant. Et étant donné que les points sont régulièrement espacés, ça veut aussi dire qu'on va fixer la dérivée de la courbure pour une certaine longueur (par exemple 20 cm). Ceci permettra au pathfinding de manipuler directement des arcs de longueur respectable et accélérera son calcul.

6.2 Les équations utiles…

6.2.1 …concernant la clothoïde unitaire

Je vous rappelle qu'on a déjà vu à quoi ressembler la clothoïde unitaire à la page 9, et qu'on dispose d'un développement limité pour calculer ses points : (**)).

La clothoïde unitaire est celle pour laquelle $\kappa'(s) = 1$. Or, comme $\kappa(0) = 0$ (par symétrie), on a :

$$\kappa(s) = s \tag{1}$$

C'est très pratique : si vous cherchez le point s associé à une certaine courbure κ, alors vous savez que $s = \kappa$. De plus, on aura besoin de connaître l'orientation θ qu'aura le robot à une certaine abscisse

1. et je veux dire par là que la distance entre deux points successifs est constante

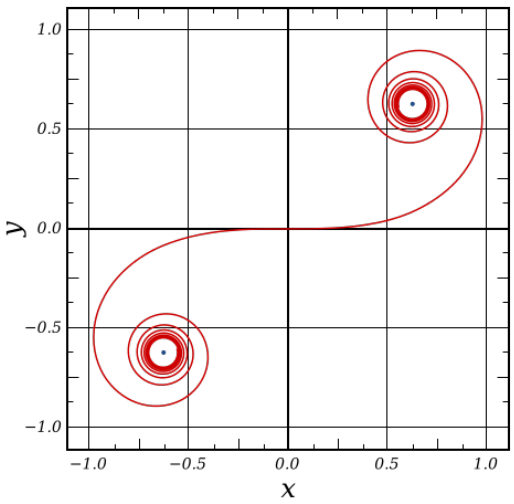


FIGURE 6.1 – Une clothoïde. $s < 0$ donne la partie gauche, $s > 0$ donne la partie droite

curviligne s (en supposant que le robot à l'avant dirigé vers s croissant). Alors, pour la clothoïde unitaire on a :

$$\theta(s) = s^2 \tag{2}$$

On a alors un résultat en radians ; ceci découle directement de l'équation paramétrique de la clothoïde (*).

Et la dernière équation à connaître est la longueur l de l'arc de la clothoïde unitaire entre s et s' :

$$l = |s' - s| \tag{3}$$

Voilà pourquoi je disais plus tôt que la clothoïde est simple mathématiquement : les grandeurs qui nous intéressent sont toutes simples à calculer.

6.2.2 ...concernant les clothoïdes non-unitaires

On a vu des propriétés intéressantes pour la clothoïde unitaire, mais qu'en est-il des autres clothoïdes ?

Les clothoïdes à κ' strictement positif

Quand on a $\kappa' > 0$, cela signifie qu'on tourne le volant vers la gauche. La clothoïde unitaire rentre dans cette catégorie. En fait, toutes les clothoïdes $\kappa' > 0$ sont égales à une homothétie près. Pour passer d'un point $P_1(s)$ de la clothoïde unitaire à son homologue dans la clothoïde pour laquelle $\kappa' = C$, il faut utiliser l'équation :

$$P_C(s) = \frac{P_1(s)}{\sqrt{C}} \tag{4}$$

De plus, étant donné que $\kappa' = C \neq 1$, on n'a plus la relation $\kappa(s) = s$, mais plutôt cette relation :

$$\kappa_C(s) = s \times \sqrt{C} \tag{5}$$

On peut remarquer que l'orientation en un point s n'est pas affecté par cette homothétie.

Les clothoïdes à κ' strictement négatif

La clothoïde de paramètre $\kappa' = C$ est symétrique par rapport à l'axe des abscisses de la clothoïde de paramètre $\kappa' = -C$, c'est-à-dire :

$$\begin{cases} P_C(s).x &= P_{-C}(s).x \\ P_C(s).y &= -P_{-C}(s).y \end{cases} \tag{6}$$

De plus, on peut observer que comme on tourne vers la droite alors qu'avant on tournait vers la gauche, on a aussi symétrie de l'orientation et de la courbure :

$$\kappa_C(s) = -\kappa_{-C}(s) \tag{7}$$

$$\theta(s) = -\theta_{-C}(s) \tag{8}$$

On peut donc se ramener facilement au cas précédent.

Le cas où $\kappa' = 0$

Lorsque $\kappa' = 0$, la courbure est constante, et donc le rayon de courbure aussi. Il s'agit d'une forme dégénérée de la clothoïde : il s'agit d'une ligne droite si $\kappa = 0$ et d'un cercle de rayon $1/\kappa$ sinon. Ces cas devront être gérés manuellement.

6.3 Construction d'un chemin clothoïdique

6.3.1 Un arc pour les gouverner tous

Maintenant que les maths sont fixées, on va pouvoir répondre à la question suivante : comment obtenir n points régulièrement espacés d'un arc de clothoïde de paramètre $\kappa' = C$ sachant que le début de cet arc à une certaine position p_R, une orientation θ_R et une courbure κ_R ?

L'algorithme va fournir une trajectoire *trajectoire*, chacun de ses points ayant trois paramètres :

— *pos*, sa position ;
— *o*, son orientation ;
— *courbure*, sa courbure.

On suppose disposer d'une fonction *clotho*(*s*) qui fournit, pour une valeur de *s*, la position de la clothoïde unitaire en cette abscisse. De plus, on suppose définie la variable *distance*, qui contient la distance désirée entre deux points consécutifs dans la trajectoire générée. L'idée est la suivante : à partir de la clothoïde unitaire, on peut en déduire tous les paramètres des autres clothoïdes. Ensuite, l'objectif est de placer ces points afin d'avoir une continuité de la position et de l'orientation du robot par un changement de repère.

Je laisse au lecteur les cas où $\kappa' = 0$ (ligne droite et cercle) ainsi que de la manière dont on peut gérer un arrêt du robot pour repartir en marche arrière. Bref, je vous laisse les parties dont les maths ne sont pas censées poser problème.

6.3.2 Enchaîner les arcs

Génial, on peut construire un arc de quelques centimètres ! Mais une trajectoire digne de ce nom est une succession d'arcs de clothoïde. Comme vous allez le voir, faire se succéder des arcs de clothoïde est très simple.

Voici un petit algo qui prend en paramètre une certaine position p_R, une orientation θ_R et une courbure κ_R (comme pour l'algo précédent), ainsi qu'une succession de κ'_i avec $1 \leq i \leq n$ et qui devra fournir une trajectoire composée de n arcs qui se suivent et qui respectent les κ'.

6.4 L'implémentation de l'interpolation cubique

Pour l'interpolation cubique, il suffit d'appliquer les coefficients (***). Il reste à choisir les instants $t_1, \ldots, t_n$ de telle manière que les points $P(t_1), \ldots, P(t_n)$ soient uniformément espacés. Si on suppose que deux point consécutifs sont séparés d'une distance d :

$$d = \int_{t_{i-1}}^{t_i} \|P'(t)\| \, \mathrm{d}t \approx (t_i - t_{i-1}) \|P'(t_{i-1})\|$$

Algorithm 2: Génération de points d'un arc de clothoïde

Input: p_R, θ_R, κ_R, κ'
Output: *trajectoire*

if $\kappa' = 0$ **then**
  if $\kappa_R = 0$ **then return** *GeneratePointsLigneDroite*(p_R, θ_R, κ_R, κ') // ligne droite
  else return *GeneratePointsArcCercle*(p_R, θ_R, κ_R, κ') // arc de cercle
else
  $s_d \leftarrow \kappa_R / \sqrt{|\kappa'|}$ // par (5)
  if $\kappa' < 0$ **then** $s_d \leftarrow -s_d$ // par (7)
  $o_d \leftarrow s_d^2$ // par (2)
  if $\kappa' < 0$ **then** $o_d \leftarrow -o_d$ // par (8)
  $s \leftarrow s_d$ // s_d est constant et s va varier
  trajectoire $\leftarrow \emptyset$
  for *i from 1 to nb* **do**
    $s \leftarrow s + \sqrt{|\kappa'|} \times distance$ // par (3)
    $\Delta pos \leftarrow clotho(s) - clotho(s_d)$ // on se place dans le repère de s_d
    $\Delta pos \leftarrow \Delta pos / \sqrt{|\kappa'|}$ // par (4)
    if $\kappa' < 0$ **then** $\Delta pos.y \leftarrow -\Delta pos.y$ // par (6)
    $\Delta pos \leftarrow rotation(\Delta pos, \theta_R - o_d)$ // on assure la continuité de l'orientation
    $point.pos \leftarrow p_R + \Delta pos$ // on assure la continuité de la position
    $o_{tmp} \leftarrow s^2$ // par (2)
    if $\kappa' < 0$ **then** $o_{tmp} \leftarrow -o_{tmp}$ // par (8)
    $point.o \leftarrow \theta_R + o_{tmp} - o_d$
    $point.courbure \leftarrow s \times \sqrt{|\kappa'|}$ // par (5)
    if $\kappa' < 0$ **then** $point.courbure \leftarrow -point.courbure$ // par (7)
    trajectoire.add(*point*)
return *trajectoire*

Ce qui nous donne un moyen facile de calculer les t_i (sachant que $t_1 = 0$) :

$$t_i = t_{i-1} + \frac{d}{\|P'(t_{i-1})\|}$$

Attention néanmoins, il s'agit d'une approximation où on suppose $\|P'\|$ constant. Dans les faits, ça marche assez bien sauf, bien sûr, si $\|P'(t)\|$ est proche de 0.

Algorithm 3: Génération d'une trajectoire courbe

Input: p_R, θ_R, κ_R, $\kappa'_1, \ldots, \kappa'_n$
Output: *trajectoire*

trajectoire $\leftarrow$ GeneratePoints(p_R, θ_R, κ_R, κ'_1)
for *i from 2 to n* **do**
  last $\leftarrow$ *trajectoire*.getLast(); // le dernier point ajouté
  trajectoire.add(GeneratePoints(*last.pos*, *last.o*, *last.courbure*, κ'_i))
return *trajectoire*

Conclusion

Où y a plus grand chose à dire…

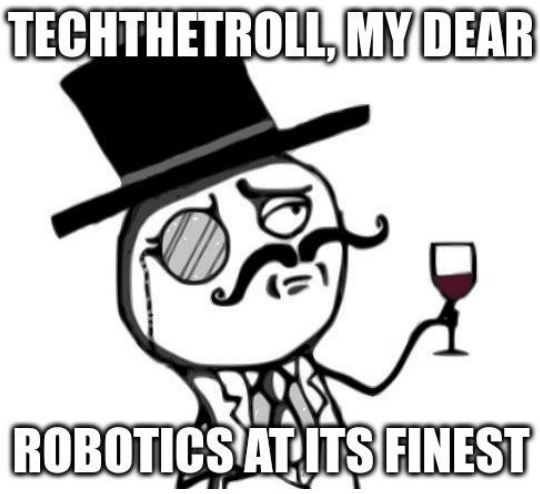
Au final, on a un robot asservi correctement une trajectoire courbe parce qu'elle est mécaniquement adaptée, qui peut calculer son chemin et surtout réagir rapidement à la moindre variation de son environnement.

Il y a encore des points à améliorer, et ce document sera modifié. N'hésitez pas à me contacter si vous avez des questions ou des idées d'amélioration. Si ce sujet vous intéresse et que vous souhaitez en savoir plus, je vous encourage fortement à consulter la bibliographie à la page suivante.

Et n'oubliez pas…

Les docs de TechTheTroll, c'est comme les AX-12. C'est fait pour tourner !

PF



Bibliographie

[Baa82] KG Baass. Use of clothoid templates in highway design. Technical report, 1982.

[BF12] Enrico Bertolazzi and Marco Frego. Fast and accurate clothoid fitting. *arXiv preprint arXiv :1209.0910*, 2012.

[BLP10] Ilya Baran, Jaakko Lehtinen, and Jovan Popović. Sketching clothoid splines using shortest paths. In *Computer Graphics Forum*, 2010.

[DKK98] Elena Degtiariova-Kostova and Vladimir Kostov. Irregularity of optimal trajectories in a control problem for a car-like robot. 1998.

[Gau99] Eric Gauthier. *Utilisation des réseaux de neurones artificiels pour la commande d'un véhicule autonome*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 1999.

[KC99] Kyoung Chul Koh and Hyung Suck Cho. A smooth path tracking algorithm for wheeled mobile robots with dynamic constraints. *Journal of Intelligent and Robotic Systems*, 1999.

[KL05] Sven Koenig and Maxim Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 2005.

[Kno06] David Knowles. Real time continuous curvature path planner for an autonomous vehicle in an urban environment. *California Institute of Technology, Tech. Rep*, 2006.

[MS93] Alain Micaelli and Claude Samson. Trajectory tracking for unicycle-type and two-steering-wheels mobile robots. Technical report, 1993.

[RCV10] RCVA. Odométrie. 2010.

[RS90] James Reeds and Lawrence Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific journal of mathematics*, 1990.

[Sam95] Claude Samson. Control of chained systems application to path following and time-varying point-stabilization of mobile robots. *Automatic Control, IEEE Transactions on*, 1995.

[SS90] Dong H Shin and Sanjiv Singh. Path generation for robot vehicles using composite clothoid segments. Technical report, 1990.

[WM05] Desmond J Walton and Dereck S Meek. A controlled clothoid spline. *Computers & Graphics*, 2005.