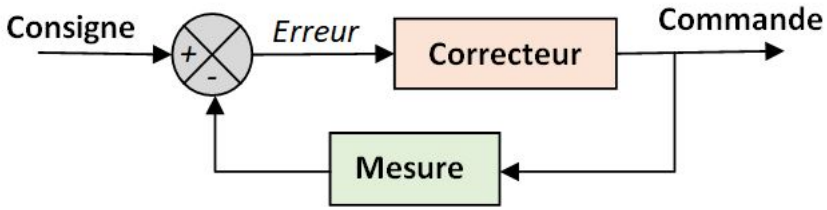
Atelier Asservissement CUBOT.FR

L'**asservissement** ? On en a partout dans nos robots ! C'est ce qui va permettre à n'importe quel système d'atteindre son objectif en un minimum de temps et en amortissant les perturbations extérieurs.

Voici la modélisation d'un système asservis :



- La **Consigne** c'est l'entrée : ce qu'on demande au système
- La **Commande** c'est la sortie : la réponse du système à l'instant t
- La **Mesure** c'est nos capteurs
- Le **Correcteur**, c'est des maths : une fonction « magique » qui calcule la Commande en fonction de l'erreur actuelle. On y trouve la plupart du temps un PID… mais on verra tout ca plus tard !

Il y à beaucoup de théorie et des heures de cours sur le sujet. Mais ici on va présenter du concret, sur nos applications de robotique mobile, avec comme objectif : **déplacer un robot avec précision et efficacité.**

Alors pas de panique, les maths serons simples, un code clair et des étapes… sans précipitation !

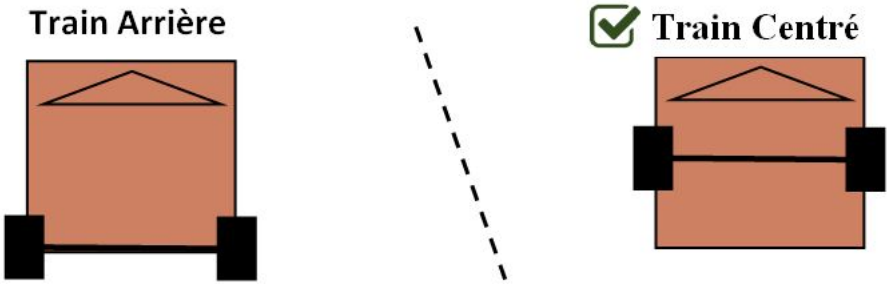
Chapitre 1 : cas de charge

Robot avec 2 roues motrices

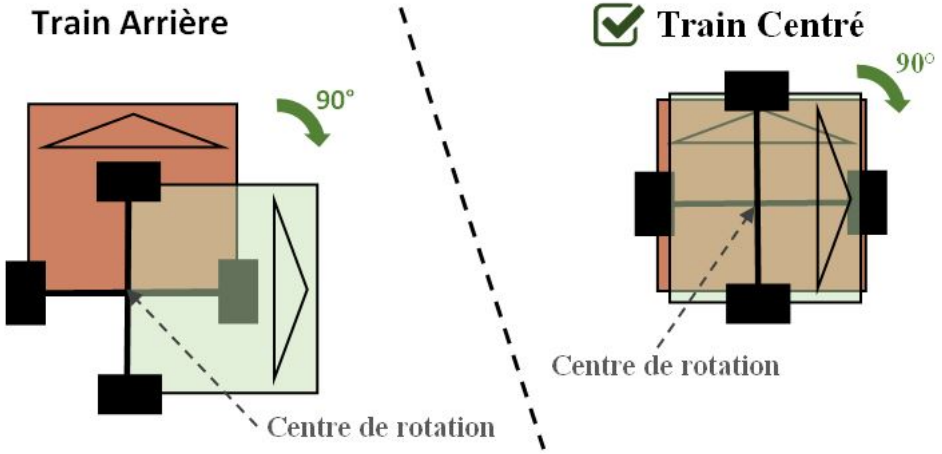
Commençons par le commencement : **la mécanique**

Il n'y a pas de débat, la solution la plus simple pour déplacer un robot, c'est de le munir de 2 roues motrices… et croyez-moi, **simplicité = efficacité** !

Alors, où mettre ces roues motrices sur notre base roulante ? Deux possibilités :



Nous préférons centrer au mieux le train sur la base du robot : principalement pour faciliter les rotations et donc simplifier les trajectoires et l'évitement. Exemple pour une rotation simple de 90° à droite :

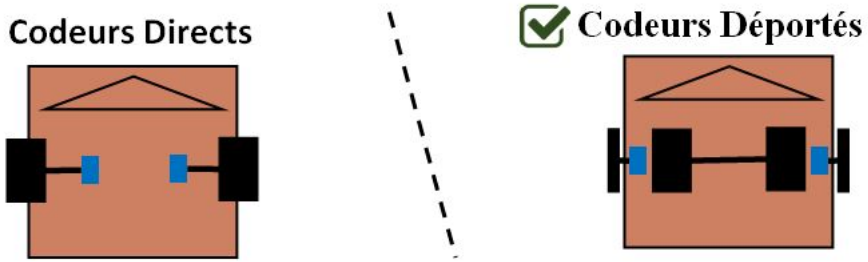


La seule raison qui pourrait nous amener à décentrer le train, serait d'avoir une obligation mécanique vis-à-vis de l'action à réaliser par le robot. Mais nous avons toujours réussi à éviter ce cas.

Le robot a maintenant 2 belles roues, prêtes pour se déplacer sur le plan XY, tourner, avancer, reculer et faire de belles trajectoire. Mais il lui manque une info importante avant ça : **Où est-il ?**

Pour réponde à cette question, il faut munir le robot de capteurs. Il y a beaucoup de possibilités (capteur optique de souris, triangulation…) mais une fois de plus nous avons choisi la « simplicité » : l'utilisation de codeurs (mesure angulaire).

Deux choix s'offrent à nous :

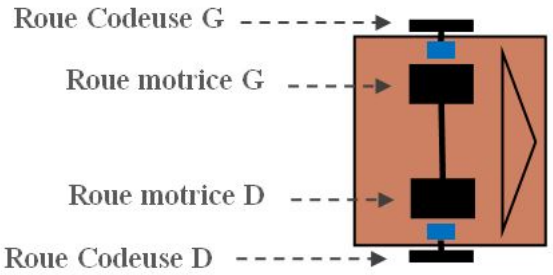


- **Codeurs Directs :** c'est le plus intuitif, on met un codeur sur chaque moteur et on mesure le déplacement de chaque roue motrice.
- **Codeurs Déportés :** légèrement plus complexe, ce système consiste à ajouter les codeurs sur deux roues libres (alignées aux roues motrices). On mesure alors le déplacement des roues libres.

La solution la plus simple (en codeurs directs) fonctionne, mais uniquement en condition idéal : au moindre choc ou glissement divers, la mesure sera décalée du déplacement réel. En pratique, si on veut avoir une vitesse intéressante en acceptant du glissement des roues motrices, il est indispensable d'avoir des codeurs déportés : les roues codeuses ne sont pas motrices, elles ne glissent pas et mesurent quoi qu'il arrive (ou presque) leurs avances sans la surestimer.

Conclusion : mécaniquement, notre système à asservir est une plateforme avec deux roues motrices et deux roues codeuses déportés :

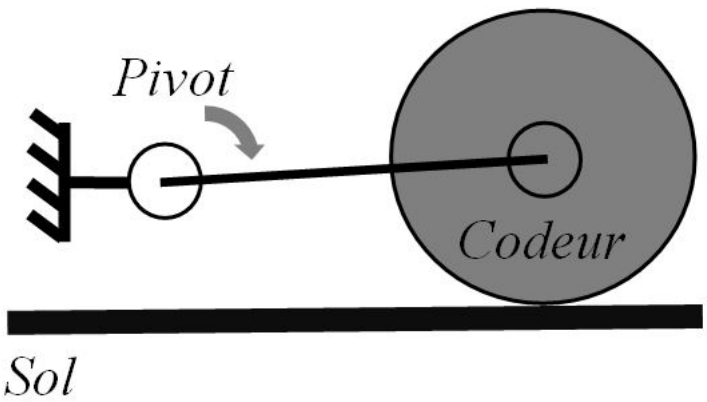
Base roulante étudiée :



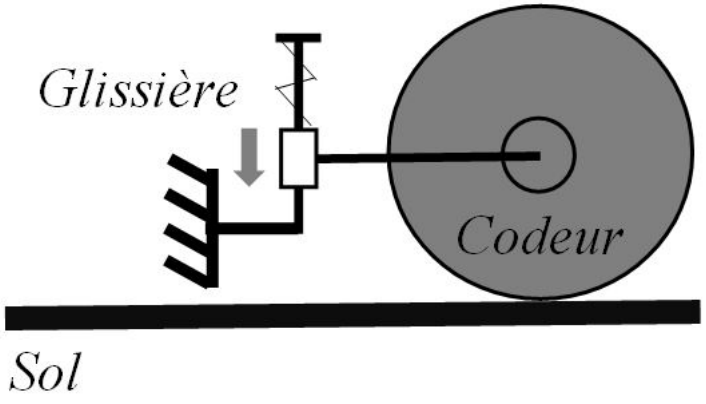
Quelques conseils de conception pour cette base roulante :

- Utiliser des matériaux lourds (comme le laiton) pour garder au plus bas **le centre de gravité** du robot final.
- Découper le plus possible **les coins de la base**, pour faciliter les rotations.
- **Aligner les roues codeuses** dans le plan des roues motrices (pour éviter du glissement tangentiel des roues codeuses)
- Plus les roues codeuses serons sur l'extérieur (**entraxe important** entre les deux roues) plus la précision de mesure de déplacement du robot sera importante. Il faut donc mettre les roues codeuses sur l'extérieur sans les rendre vulnérables d'un choc.
- Il est nécessaire d'assurer un plaquage des roues codeuses sur le sol. Je conseil toujours l'utilisation d'une liaison pivot pour réaliser ce plaquage, plutôt que la liaison glissière (qui est toujours moins bonne mécaniquement qu'un pivot), même si la translation du codeur est théoriquement meilleure pour la mesure, en pratique la liaison pivot est plus efficace et l'erreur de mesure théorique (dû au décalage lors de la rotation) est négligeable :

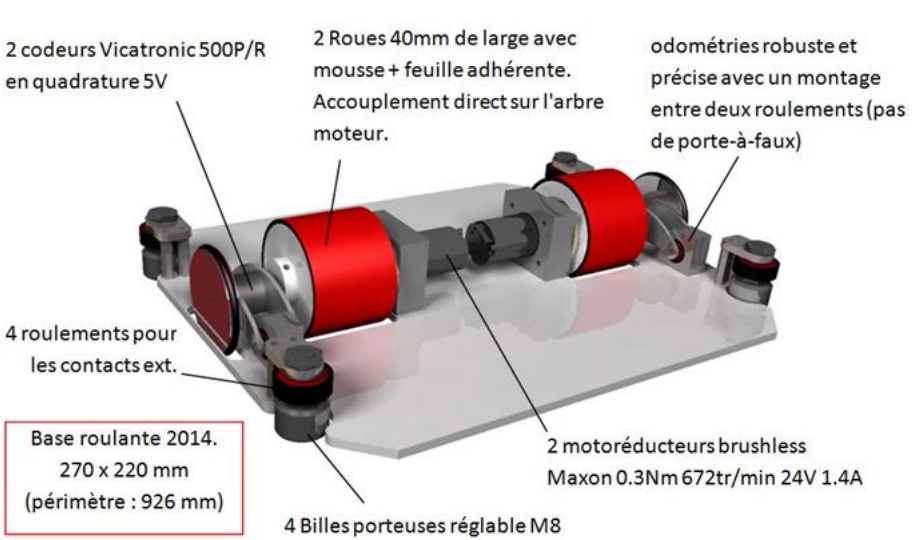
Codeur en rotation

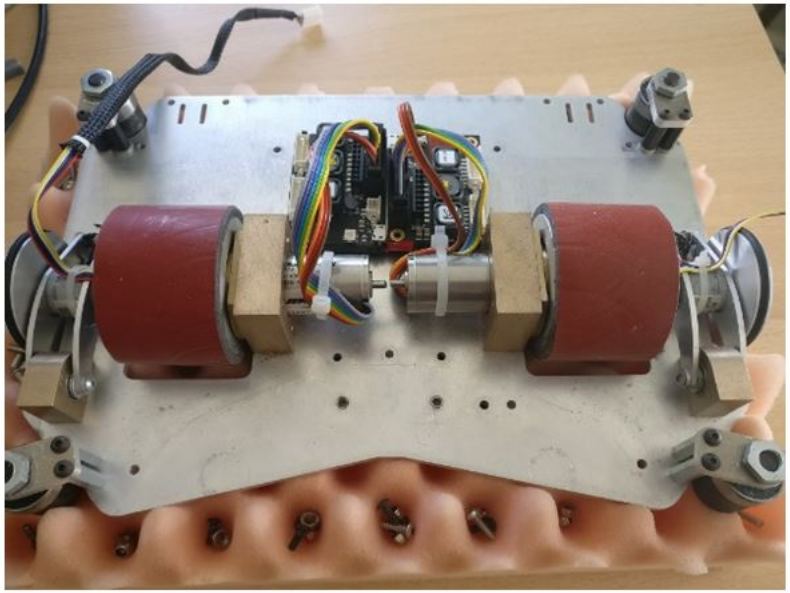


Codeur en translation



Concrètement et sur ce principe voici notre meilleure base roulante actuelle :



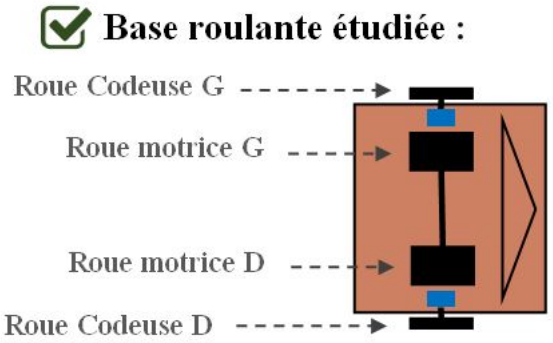


Préc.
Suite

Chapitre 2 : odométrie

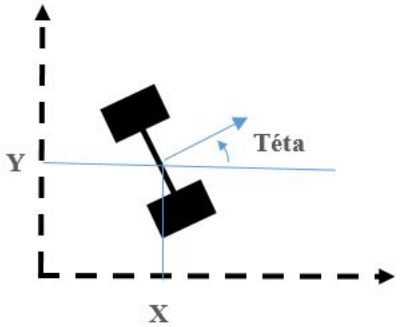
La mesure de déplacement

Dans le chapitre précédent, nous avons construit notre base roulante et nous l'avons modélisé de la façon suivante :



Intéressons nous maintenant à la mesure du déplacement du robot : l'**odométrie**.

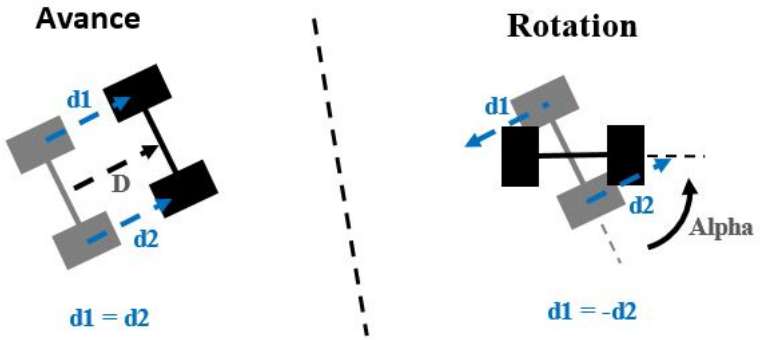
Le but est d'utiliser les deux roues codeuses pour en déduire un déplacement global du robot et définir sa position en tout temps (X, Y et Téta) :



Sur notre châssis, il existe deux types de mouvements élémentaires :

Au vue de ces deux mouvements, on ne cherchera pas à asservir le robot en cartésien, mais en polaire (cela ne nous empêche pas de convertir la position polaire en cartésien, car on ne va pas se mentir : on travaillera mieux la stratégie avec des coordonnées cartésiennes par la suite).

Observons le déplacement élémentaire du point de vue des codeurs :

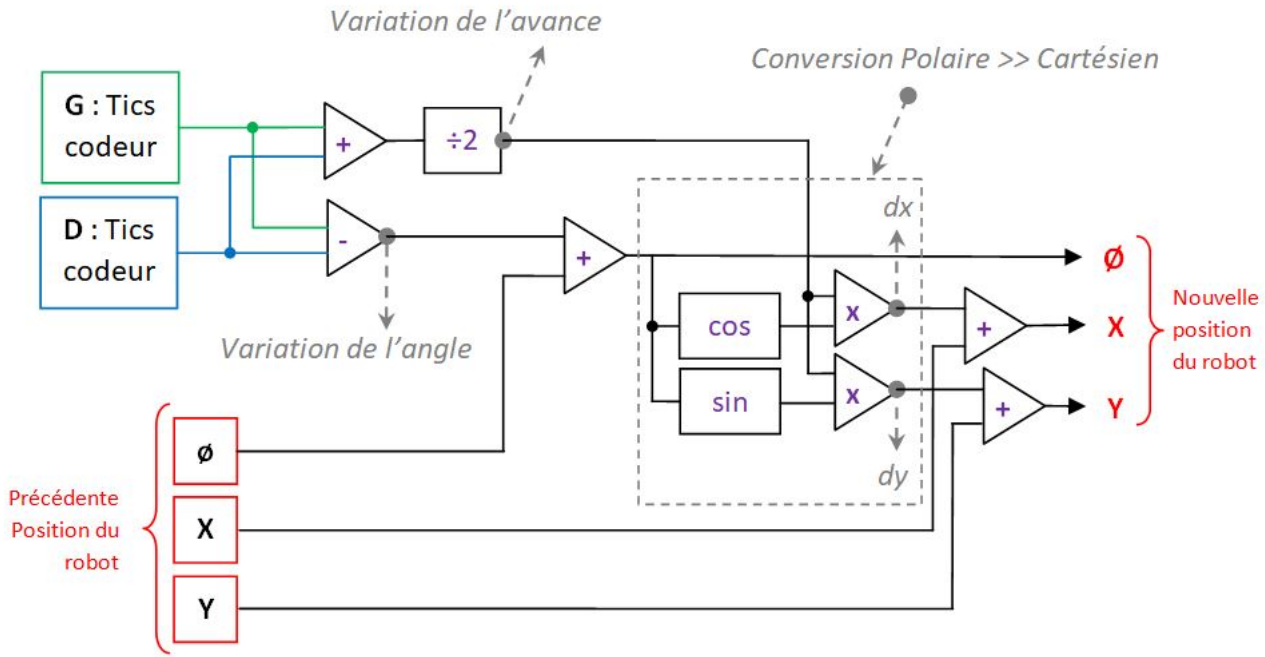


Sur ce constat simple :

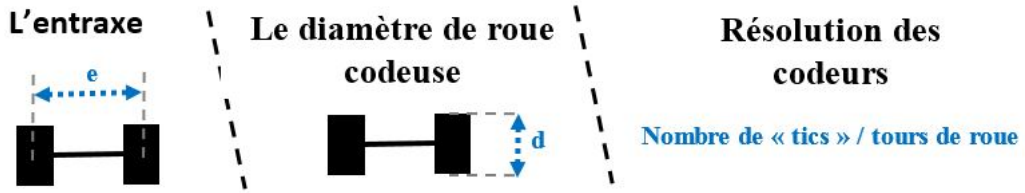
- Pour obtenir l'**avance du robot**: on somme la distance parcouru par la roue 1 et 2 (**D = (d1+d2)/2**)
- Pour obtenir la **rotation du robot**: on soustrait la distance parcouru par la roue 1 et 2 (**Alpha = d1-d2**)

Et oui, rien de plus compliqué, à chaque cycle d'asservissement, la fonction d'odométrie actualise la position du robot par une simple addition et une soustraction… puis il ajoute les deux résultats à la position précédente.

En ajoutant le passage en coordonnées Cartésiennes, voici le détail de l'algorithme :

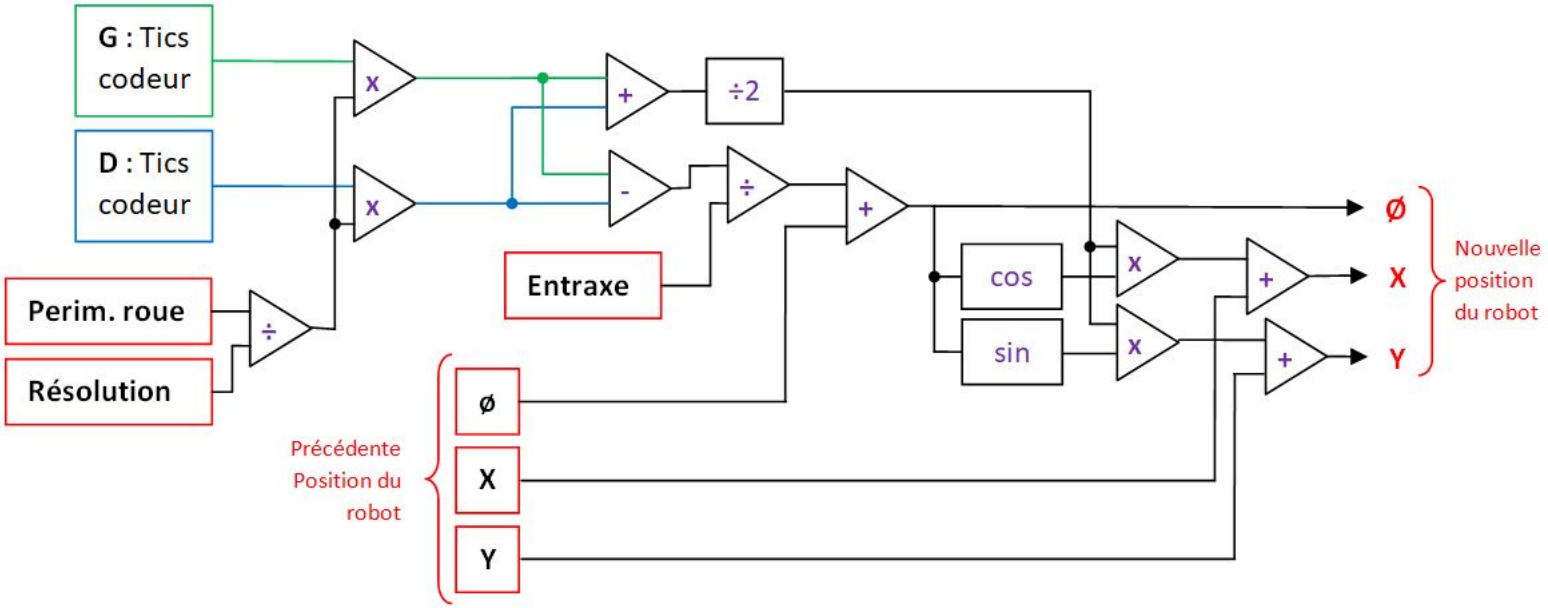


Pour mettre des unités à ces coordonnées, il ne reste plus qu'à prendre en compte les dimensions du robot avec ces 3 constantes :



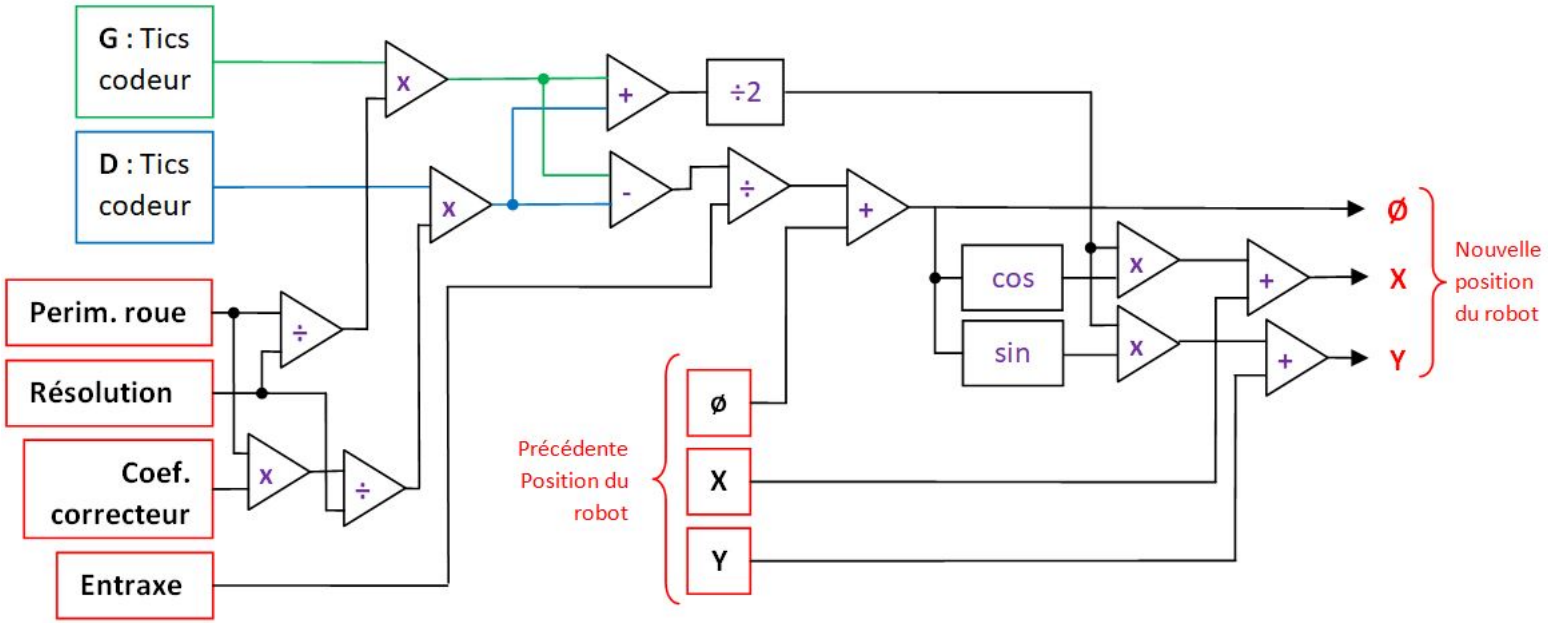
Pour être exact, on n'a pas besoin du diamètre de la roue, mais de son périmètre. Mais est-ce utile de préciser que **Perim. Roue** = **Pi** x **Diamètre Roue** ? Bon… bah c'est fait !

En ajoutant les 3 constantes, l'algorithme devient :



Enfin, il y a un dernier paramètre à ne pas négliger pour avoir une odométrie précise : un coefficient correcteur de diamètre entre la roue codeuse Droite et Gauche. Ce paramètre permet de corriger l'infime défaut mécanique.

L'algorithme final pour le calcul d'odométrie est alors le suivant :



L'angle en sortie est en radian et les X et Y sont en mm.

Pour la suite, à partir de ce premier programme, on peut facilement y ajouter :

- une conversion de l'angle du robot en degré.
- faire intervenir « ***dt*** » (le temps du cycle) pour le multiplier à :
  - l'avance pour avoir la **vitesse linéaire** du robot.
  - la rotation qui nous donnera la **vitesse angulaire** du robot.
- Pour Eurobot : **symétriser l'aire de jeu** (choix de la couleur)… tout est ici : Symétrie de la table.

Préc.
Suite

Chapitre 3 : calibration de l'odométrie

Ou "étalonnage" pour les puristes

Dans le chapitre précédent, nous avons écrit l'algorithme pour le calcul d'odométrie et nous avons défini 4 constantes à déterminer :

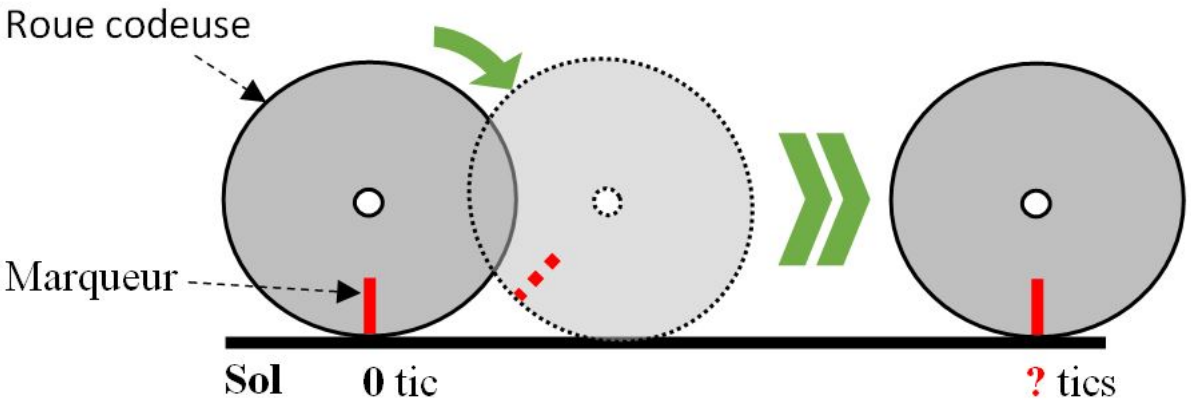
Alors comment définir ces variables ?

Et non, toujours pas de calculs compliqués ! Pour être précis, il faut faire quelques essais avec le robot et dans le bon ordre :

1 - Résolution des codeurs

C'est le paramètre le plus simple : il suffit de lire la datasheet du codeur pour y trouver sa résolution : le nombre de pas mesuré sur un tour de codeur.

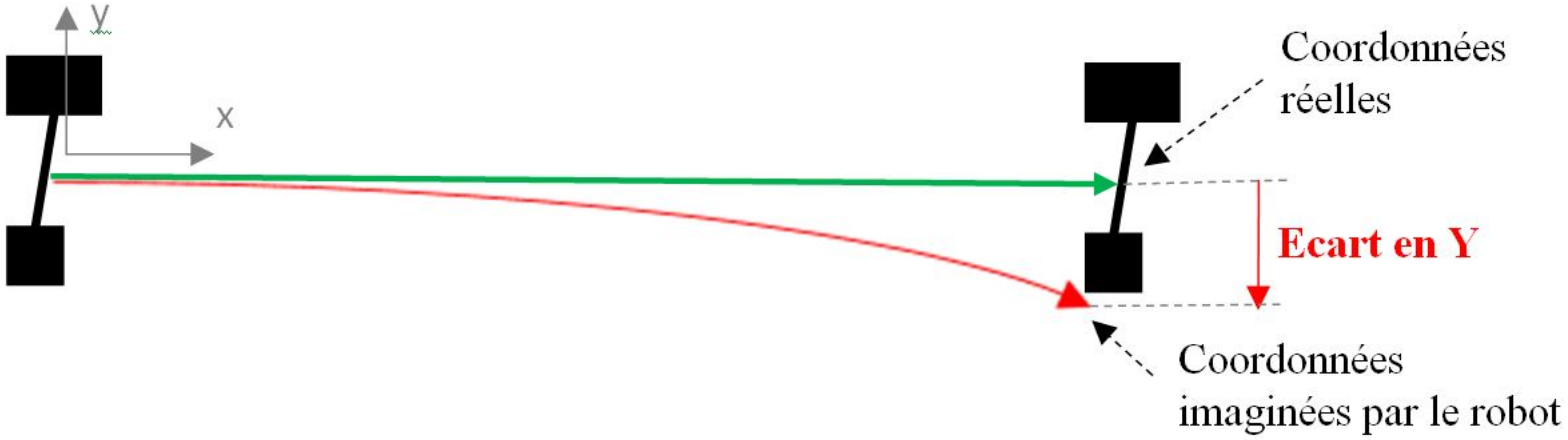
C'est facilement vérifiable en affichant le compteur de tic du codeur et en réalisant un tour de roue codeuse (pas besoin de chercher à être précis au tic près : la constante qui définit le diamètre de roue corrigera s'il y a une erreur sur ce coefficient) :



2 – Coef. Correcteur de diamètre de roue

Même si ce n'est pas intuitif la première fois, l'ordre de calibration des coef. est le bon. Je conseil quand même de mettre un coef. d'entraxe et de diamètre de roue approximatif, mais leurs valeurs ne changerons pas le résultat de l'essai.

La manipulation, consiste à réaliser une ligne parfaitement droite avec le robot et de mesurer la dérive : par exemple le faire avancer (à la main) en X sur 3 ou 4 mètres et de noter l'écart en Y… il faut alors corriger le coef. et refaire l'essai jusqu'à obtention de 0 en Y.



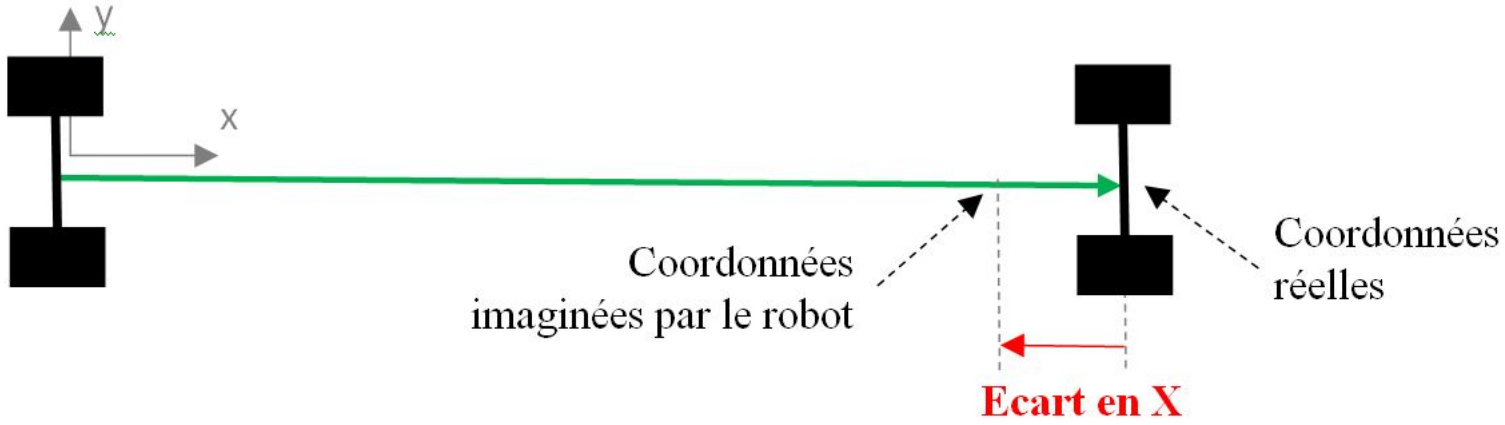
Sans affichage des coordonnées X et Y, un autre moyen de déterminer ce coefficient consiste avec la même manipulation d'obtenir le même nombre de « tic » codeur D et codeur G. En effet, dans l'exemple ci-dessus, la roue D étant plus petite, elle donnera plus de tics que la roue G sur une ligne droite sans le coefficient correcteur.

Un bon résultat pour ce coefficient est d'être précis à 1 mm de dérive pour 4 mètres linéaire.

3 – Le diamètre des roues codeuses

NB : On parle bien des roues codeuses depuis le début : l'odométrie se fiche de la propulsion, puisqu'elle est déportée.

Maintenant que le robot sait évaluer une vrai ligne droite, on va continuer à lui en faire faire, mais en lui demandant d'être précis sur son avance en X :



- **Ecart négatif**: on augmente la constante du diamètre de roue

- **Ecart positif**: on diminue la constante du diamètre de roue

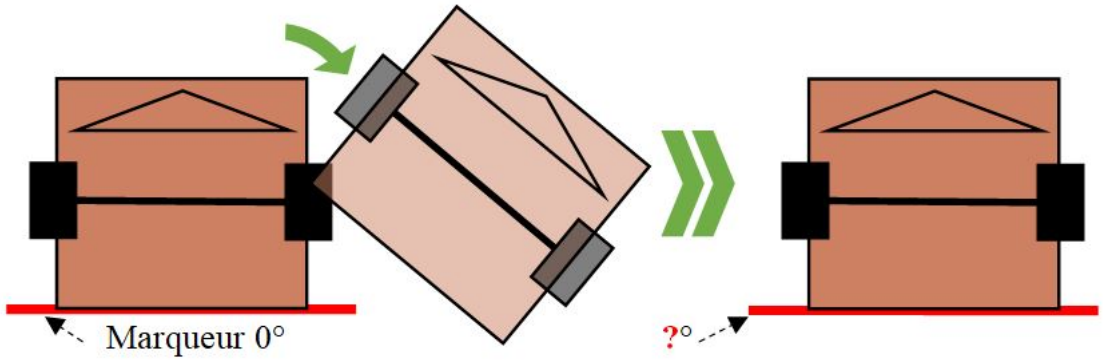
Un bon résultat pour ce coefficient est d'être précis à 1 mm de d'écart pour 4 mètres linéaire.

4 – L'entraxe

On termine avec le paramètre le plus important : l'entraxe entre les deux roues codeuses.

C'est le paramètre qui va définir la rotation du robot, et la précision angulaire fait toute la précision de l'odométrie (car un tout petit écart d'angle engendre rapidement une dérive sur les calculs).

Pour ce paramètre, on va tourner le robot sur lui-même un grand nombre de fois… à l'arrivé sur l'angle initial, on doit retrouver les 0° du début.



- **Ecart négatif**: on augmente la constante de l'entraxe
- **Ecart positif**: on diminue la constante de l'entraxe

Un bon résultat pour ce coefficient est d'être précis à 1° d'écart pour 40 tours (14400°).

Normalement après ça l'odométrie est bien réglé, mais je conseil de refaire à partir de l'étape 2 (coef. de diamètre de roue), car on est jamais à l'abri d'une erreur de manipulation.

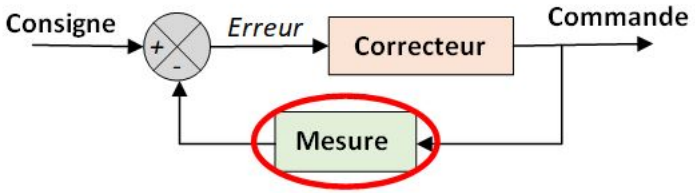
Préc.
Suite

Chapitre 4 : consigne

Le calcul des erreurs avant d'asservir

Dans les chapitres précédents, nous avons écrit l'algorithme pour le calcul d'odométrie et nous l'avons calibré.
Notre robot connait donc maintenant sa position XY et son angle Téta à tout instant.

Dans notre système à asservir, les précédents calculs d'odométrie correspondent à la mesure :

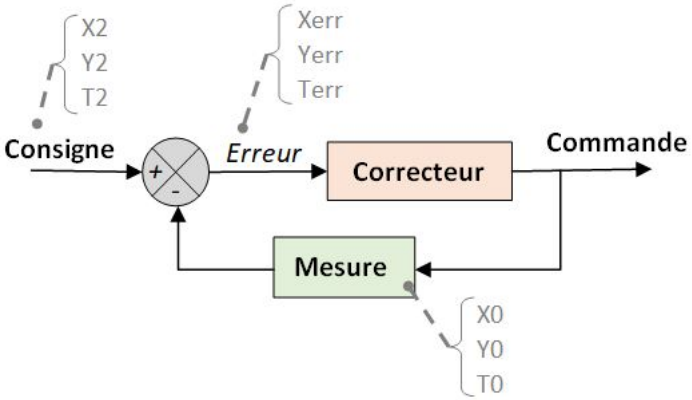


Cette mesure donne le point X0, Y0, T0.

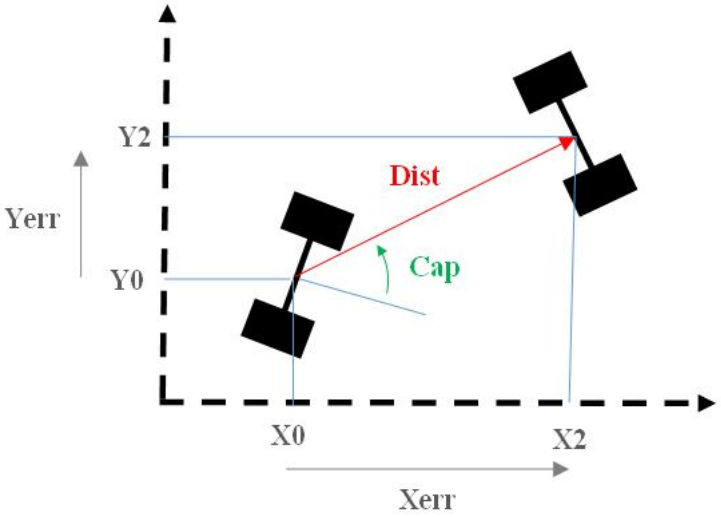
La consigne, c'est la destination du robot (demandé par le programme de stratégie) >> On souhaite aller au point X2, Y2, T2

L'erreur du système est donc très simple à calculer :

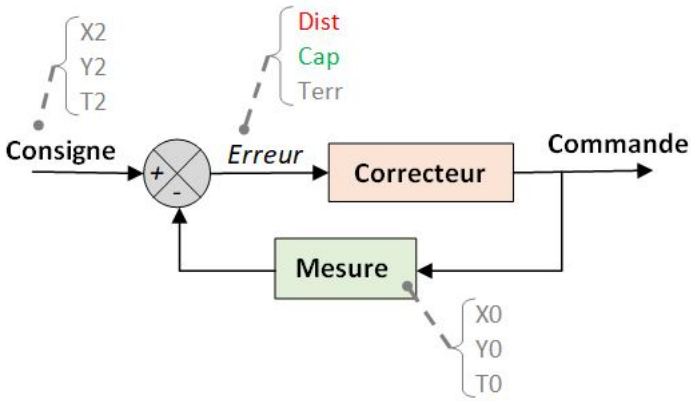
- Xerr = X2-X0
- Yerr = Y2-Y0
- Terr = T2-T0



On va commencer par mettre de côté l'angle final Terr, il se fera « tout seul » une fois la position XY obtenue. On se penche alors sur le calcul de la consigne en position :



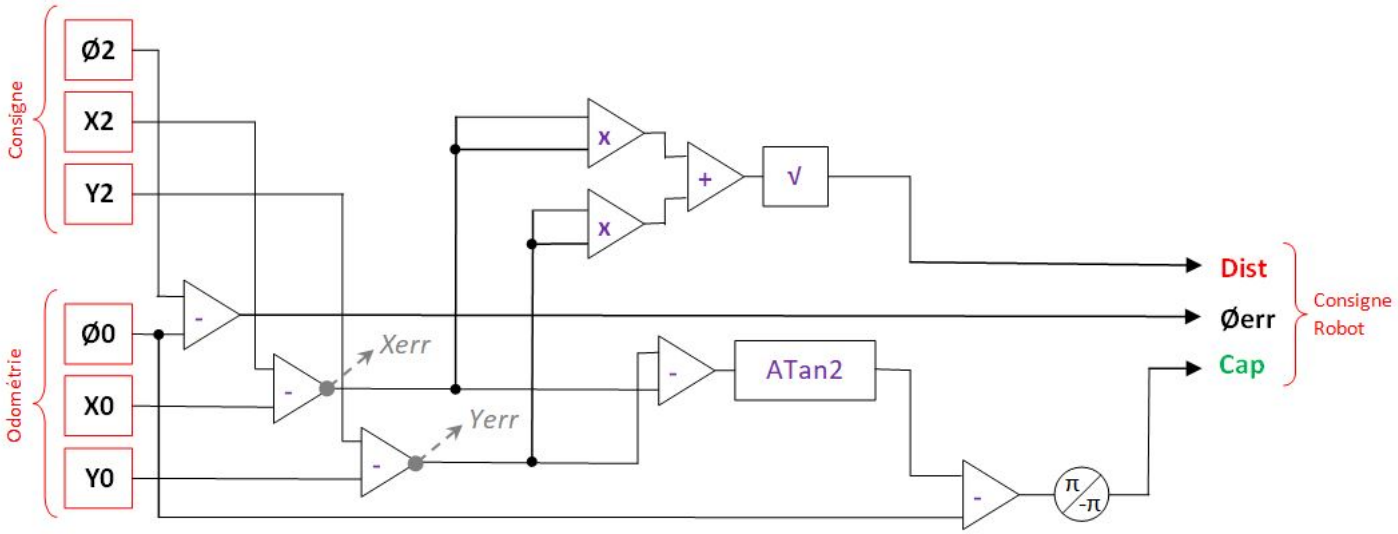
Comme le robot fonctionne en polaire, on ne va pas travailler avec l'erreur cartésienne, mais avec une erreur polaire :



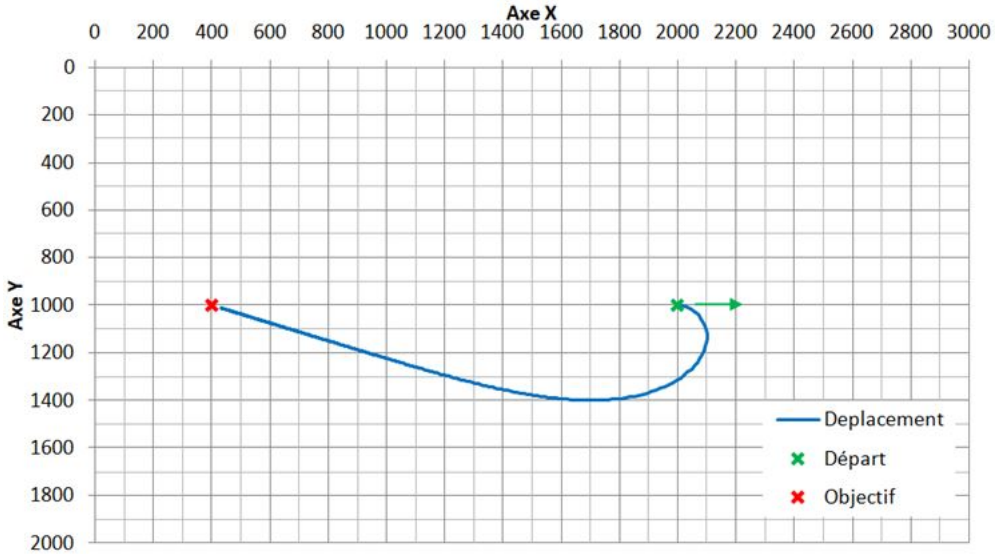
- **Dist**: c'est la distance entre la position du robot et son objectif. Un calcul du niveau collège (comme disait mon prof de maths), grâce à Pythagore : **Dist = RACINE(Xerr²+Yerr²)**
- **Cap**: comme son nom l'indique, c'est l'orientation que doit prendre le robot pour atteindre son point. Toujours un peu de trigonométrie : **Cap =ArcTan(Yerr/Xerr) – T0**

Il faut être très vigilent avec les angles. Le calcul du cap peut amener l'ArcTan à jongler autour de PI et –PI… et donc (avec la soustraction de T0) calculer un Cap qui fait plus d'un tour. Il est extrêmement important de borner la consigne en angle. Je conseille donc de ramener immédiatement le calcul du Cap entre PI et –PI avec une fonction récursive (un simple -2PI si le Cap est supérieur à PI ou +2PI si le Cap est inférieur à –PI).

Avec tout ça, on peut déjà commencer à écrire un morceau d'algorithme pour calculer la consigne pour le robot (l'erreur du système à asservir) :

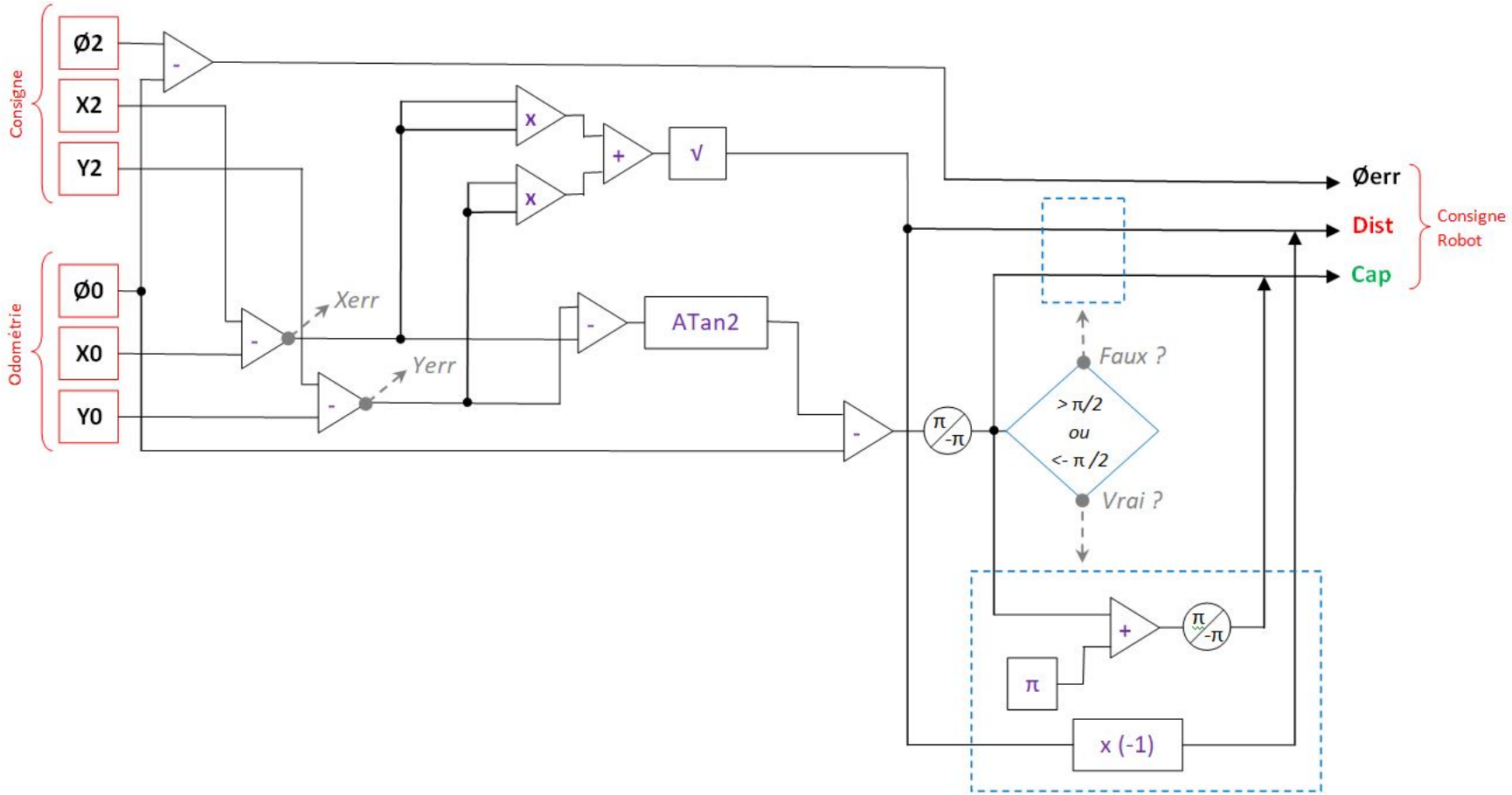


Avec ça, on a déjà une consigne pas trop mal pour déplacer le robot. Pour clôturer ce chapitre, on va simplement ajouter un tout petit bout de code pour gérer la marche arrière. En effet, avec le précédent calcul **Dist** est toujours positif (on demande toujours au robot d'avancer), alors si son objectif est derrière lui il va y aller de cette façon :



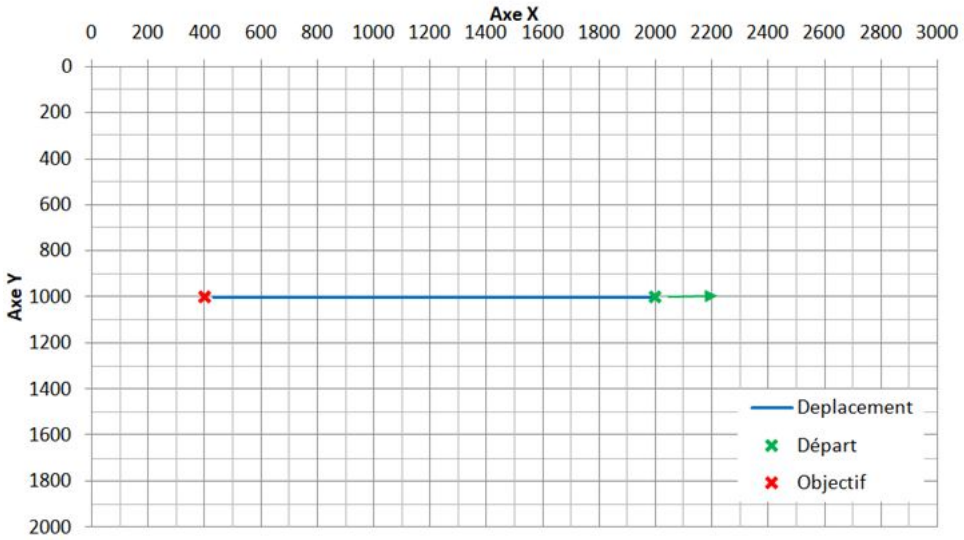
Alors qu'une simple marche arrière toute droite serait tellement plus efficace !

Pour s'occuper de cette marche arrière, on va inverser le calcul de la consigne si l'objectif dépasse la moitié du robot (90° ou -90°). Concrètement, si le **Cap** dépasse PI/2 ou est inférieur à –PI/2 on inverse le signe de **Dist** et on ajoute PI au calcul du **Cap**.
L'algo devient alors :



Si on est en marche arrière, il faut de nouveau penser à placer le Cap entre PI et –PI.

Avec cette simple condition de marche arrière, on obtient bien cette ligne droite au plus court :



Préc.
Suite

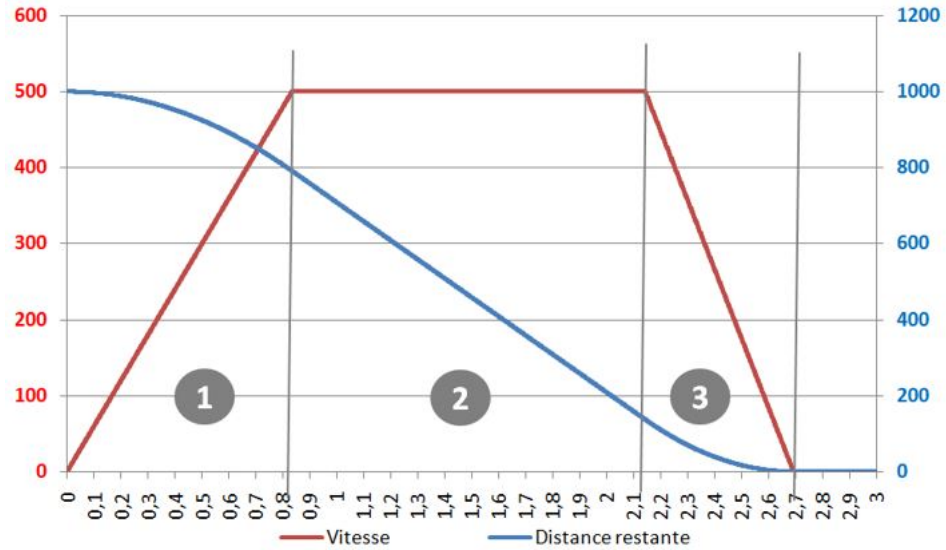
Chapitre 5 : rampe

La gestion de la vitesse, de l'accélération et du freinage

Avant de s'occuper de la régulation par PID des consignes de distance et d'angle, on va se pencher sur l'algorithme qui contrôlera la vitesse max et les accélérations/décélérations du robot.

L'algorithme est à utiliser deux fois (une fois pour la consigne d'avance et une fois pour la rotation), avec des constantes distinctes pour les deux cas (la vitesse de rotation n'aura rien à voir avec la vitesse linéaire). Comme l'algorithme est un peu plus complexe que les autres, je ne vais détailler ici qu'un des deux (la rampe pour l'avance dans l'exemple ci-dessous), mais c'est exactement le même algorithme à appliquer sur la consigne « **Cap** ».

L'idée de cette étape, va être de contrôler le profil de vitesse en fonction de la distance à parcourir et des constantes (vitesse max, accélération et freinage) de façon à obtenir la rampe suivante :



- **En bleu**: c'est la distance qu'il reste à parcourir du chapitre précédent (l'erreur qu'on souhaite amener à 0)
- **En rouge**: le profil de vitesse que l'on souhaite piloter et qui correspond à la dérivé de la position en bleu.
- **En abscisse**, le temps en seconde.

On peut aussi représenter les phases d'accélération en dérivant la vitesse : on obtient un escalier (une phase 1 positive, une phase 2 à zéro, et une phase 3 négative.

Dans cet exemple on a initialement 1000mm à parcourir. Ils vont être parcourus en 2,7s :

- **Phase 1 :** c'est la phase d'accélération qui correspond à la pente de la vitesse (la dérivée), dans l'exemple 600mm/s²
- **Phase 2 :** représente le régime établi, avec une vitesse max de 500 mm/s. Si la distance est trop faible, cette phase peut ne pas apparaitre.
- **Phase 3 :** on anticipe l'arrivée par un freinage, une décélération de 1000mm/s² dans l'exemple (toujours la dérivé de la vitesse)

Pour programmer ça, il faut distinguer ces 3 phases dans l'ordre suivant :

Phase 3, freinage :

En entrée de cette phase de freinage, on a besoin des informations suivantes :

- **Temp d'échantillonnage *(une boucle d'aserv)*:** dt = 0,005
- **La vitesse du robot :** Vrob = (dist(t-1)-dist(t0))/dt
- **La décélération maximale *(une constante)* :** Afrein = 1000

Avec ces informations, on peut calculer la distance de freinage nécessaire pour s'arrêter à cette vitesse :

- **Dfrein = (Vrob²) / (2 x Afrein)**

Si dist < Dfrein alors on entre dans cette phase de freinage :

- **Vconsigne = Vrob – (Afrein x dt)**

Phase 1, l'accélération :

En entrée de cette phase d'accélération, on a besoin des informations suivantes :

- **Temp d'échantillonnage *(une boucle d'aserv)*:** dt = 0,005
- **La vitesse du robot :** Vrob = (dist(t-1)-dist(t0))/dt
- **La vitesse maximale *(une constante)* :** Vmax = 500
- **L'accélération maximale *(une constante)* :** Amax = 600

Si on n'est pas en phase de freinage et que V < Vmax, alors on est en phase d'accélération :

- **Vconsigne = Vrob + (Amax x dt)**

Phase 2, vitesse max :

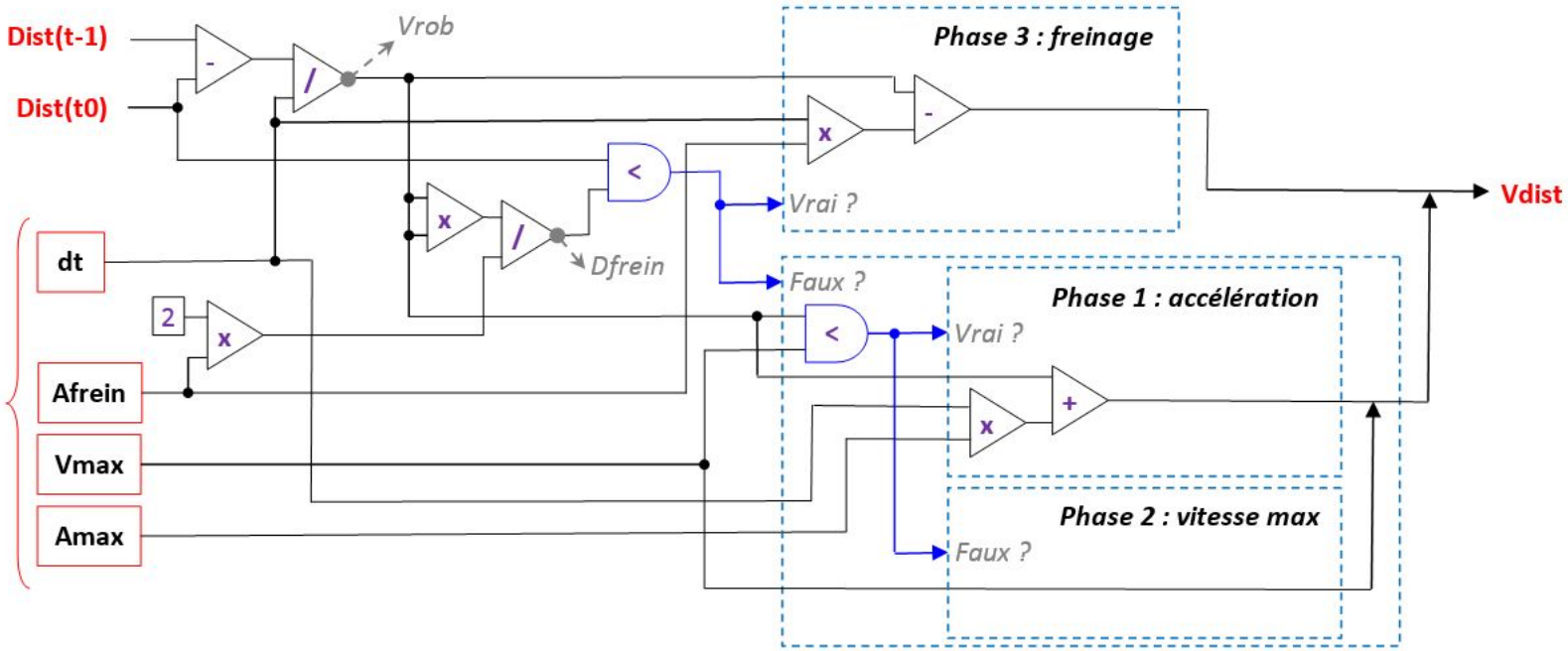
Une seule information est nécessaire pour cette phase :

- **La vitesse maximale *(une constante)* :** Vmax = 500

Si on n'est pas en phase de freinage et ni en phase d'accélération, alors on est en régime établi :

- **Vconsigne = Vmax**

L'Algorithme finale pour la rampe de vitesse d'avance est alors :



Et faire de même pour l'angle afin d'obtenir Vcap.

Pour vous aider à bien comprendre cet algorithme, on vous met à disposition ce fichier Excel qui réalise tous ces calculs :

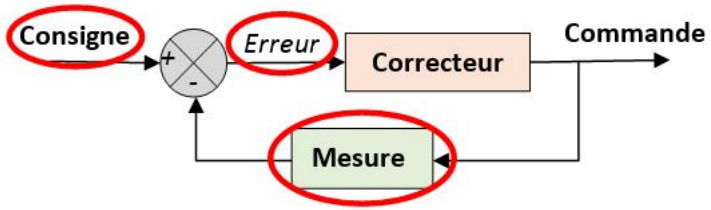
Calcul Profil de Vitesse

Telecharger
Préc.
Suite

Chapitre 6 : PID

entrons dans le cœur de l'asservissement

Dans les chapitres précédents, nous avons écrit les algorithmes pour calculer la mesure, la consigne et l'erreur :



Dans ce chapitre, nous allons corriger cette erreur pour l'amener à « 0 » (ou presque) en un minimum de temps et en un minimum d'oscillation. Ce correcteur, est un PID (Proportionnel, Intégrale et Dérivée). Pas d'inquiétude, c'est encore plus simple comme calcul que les précédents. Dans le monde informatique discrétisé :

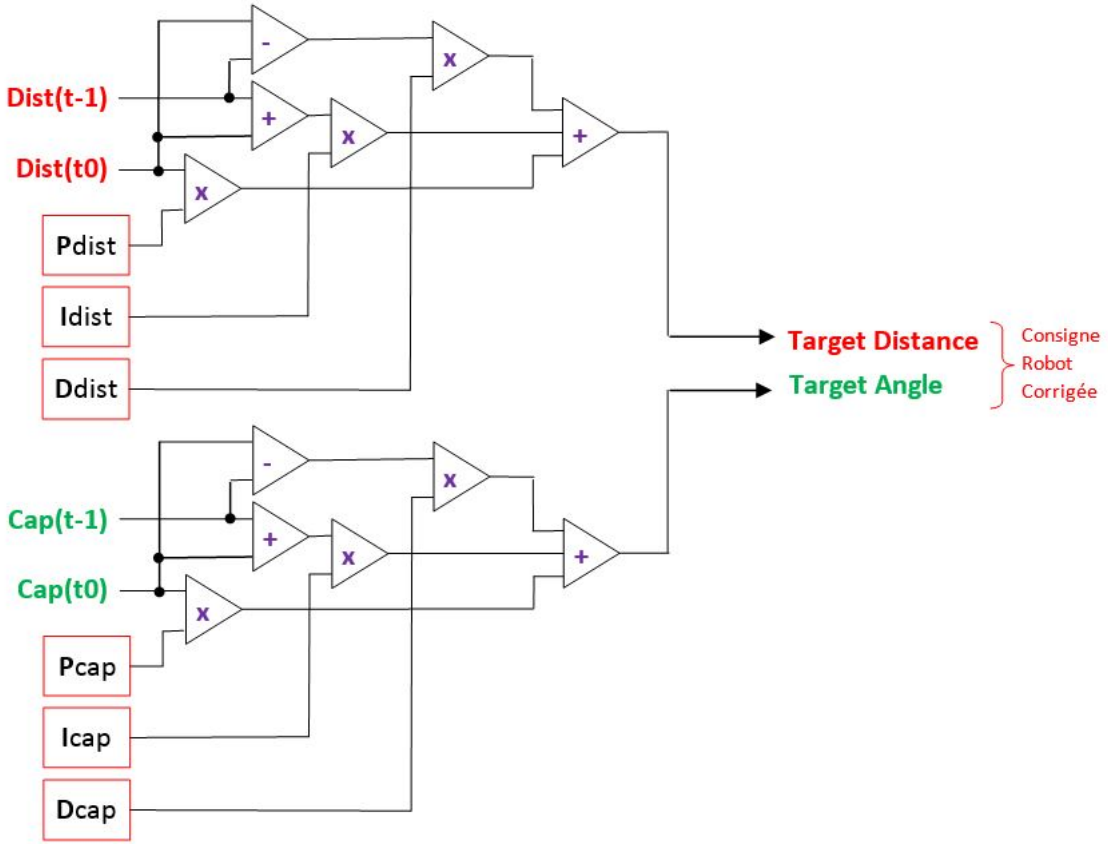
- *pour intégrer on additionne,*
- *pour dériver on soustrait.*

Il y a deux erreurs à corriger : la consigne en distance et la consigne en rotation. Le prochain algorithme demande alors 2 PID, soit 6 constantes à définir : Pcap , Icap, Dcap, Pdist, Idist, Ddist.

Pour revenir dans le détail du calcul du PID, c'est une addition des 3 calculs suivant :

- **Proportionnel** : Erreur x P
- **Intégrale** : (Erreur + Erreur précédente) x I
- **Dérivée** : (Erreur – Erreur précédente) x D

L'algorithme pour les deux correcteurs est alors :



Rien de plus !

Pour définir les 6 constantes, comme pour la calibration de l'odométrie : il faut faire des tests avec le robot. Le réglage des deux PID (Dist et Cap) se fait séparément, mais dans les deux cas la procédure est la même :

- **Réglage proportionnel :**

Pour commencer on met à « 0 » les « I » et « D » : Il faut trouver les « P » qui vont bien pour les moteurs. Pour cela, on augmente le gain « P » jusqu'à obtenir une oscilation.

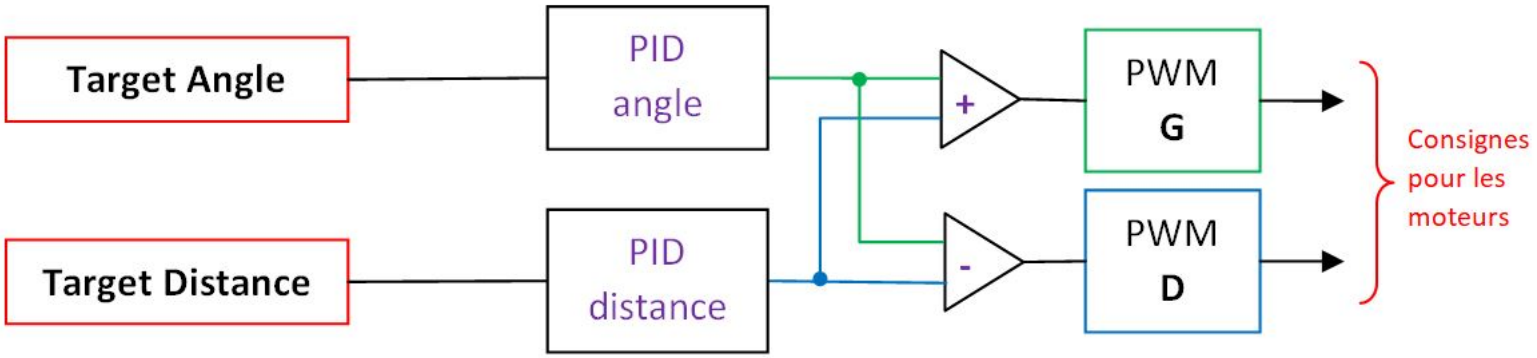
- **Réglage de la dérive :**

Quand on commence à avoir des oscillations uniquement avec du gain « P », on redescend légèrement le « P » et on augmente la dérive « D » jusqu'à éliminer totalement les oscillations.

- **Réglage de l'intégrale :**

Je ne donnerais pas de conseil sur l'intégrale « I », car nous la laissons toujours à 0 (elle ne nous apporte rien).

Pour terminer la boucle d'asservissement, on converti les « target » d'angle et de distance en PWM pour les moteurs (à Droite et à Gauche) :



La boucle est bouclé ! Le robot est prêt pour rouler.

Bien joué si vous êtes arrivé jusqu'ici ! pensez à fêter les premiers tours de roues avec une bonne bière : Pour l'occasion, je conseil une classique mais, toujours aussi savoureuse : **Triple Karmeliet** !

PI : *Le terme triple utilisé dans son nom fait référence aux trois grains (orge, avoine et froment) employés pour son brassage et non à sa fermentation. D'ailleurs le terme triple fermentation est à réserver aux bières d'abbayes.*

Préc.
Suite

Synthèse

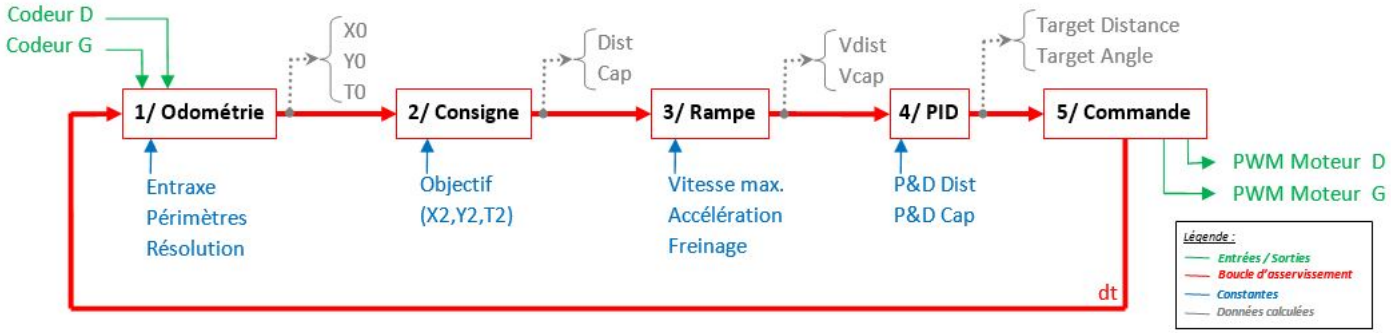
avec en cadeau : un exemple d'application sous Excel

On y est !

Quand on arrive là, sans « tricher » c'est que le robot roule… là où on lui demande de rouler. Félicitation !

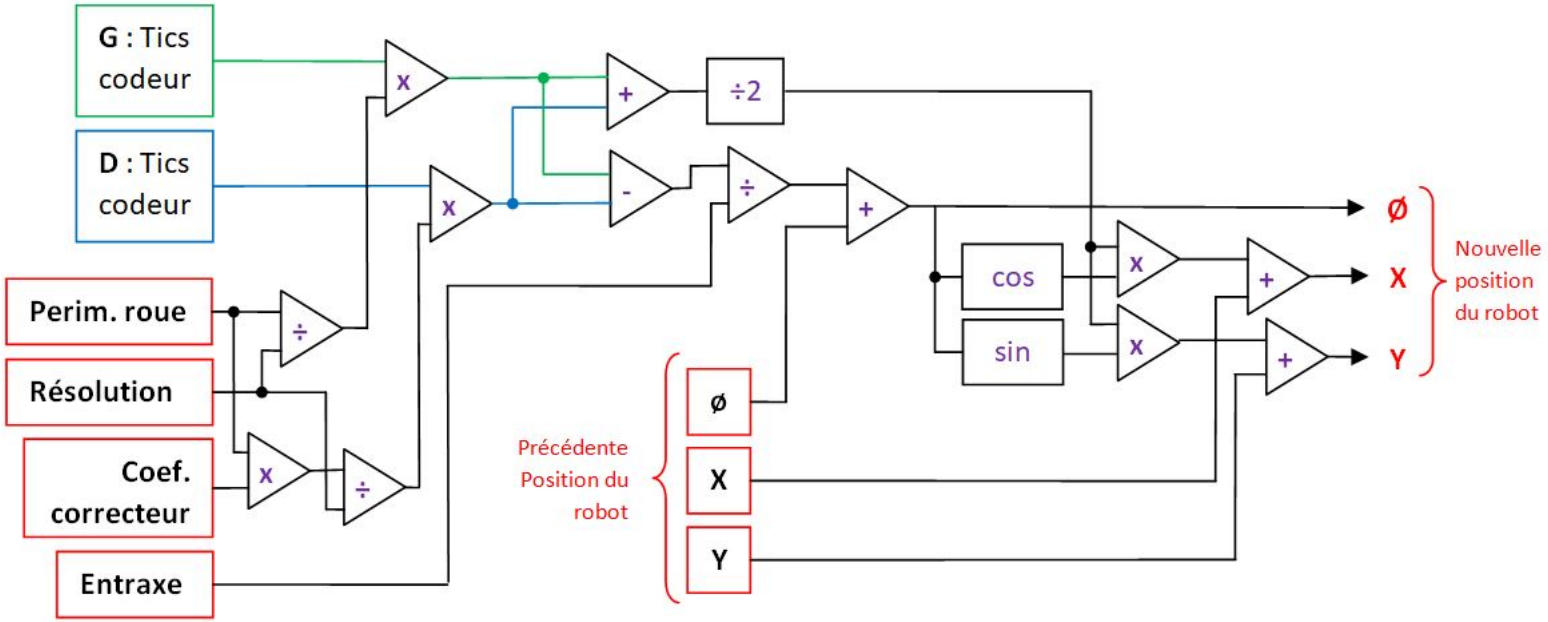
Tous les ans, plusieurs équipes nous montrent à la coupe de France de Robotique, qu'avec uniquement une bonne base roulante fiable (additionné d'un bon évitement), les 16èmes de finales sont accessibles : d'où l'importance de bien travailler cet atelier avant de concevoir de nouveaux systèmes ultra complexe.

Comme l'atelier asservissement est assez gros, un résumé s'impose pour garder l'essentiel en tête :



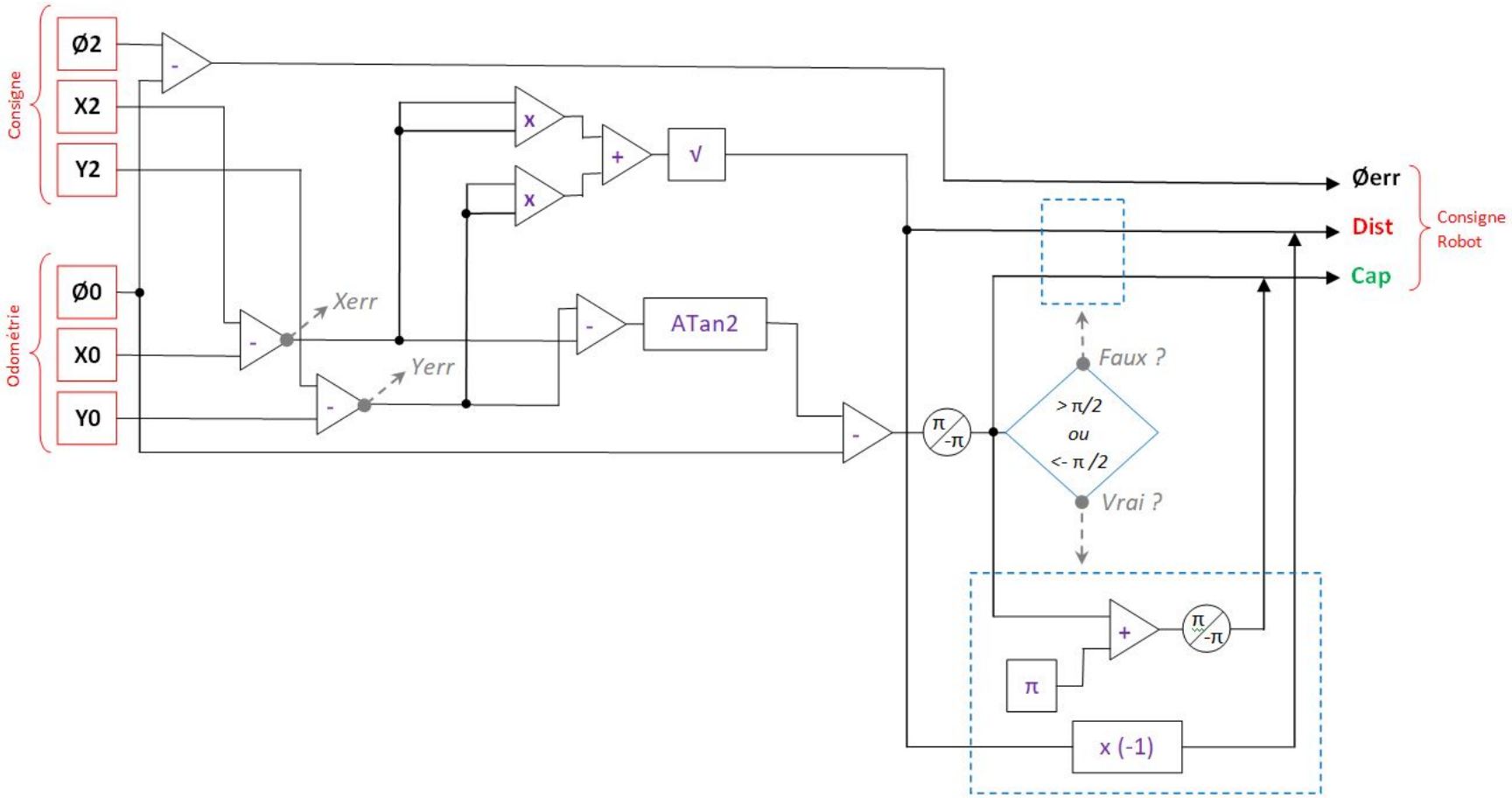
1/ Odométrie :

L'algorithme calcul la position du robot en fonction de l'avance des codeurs et des constantes topologique du châssis.



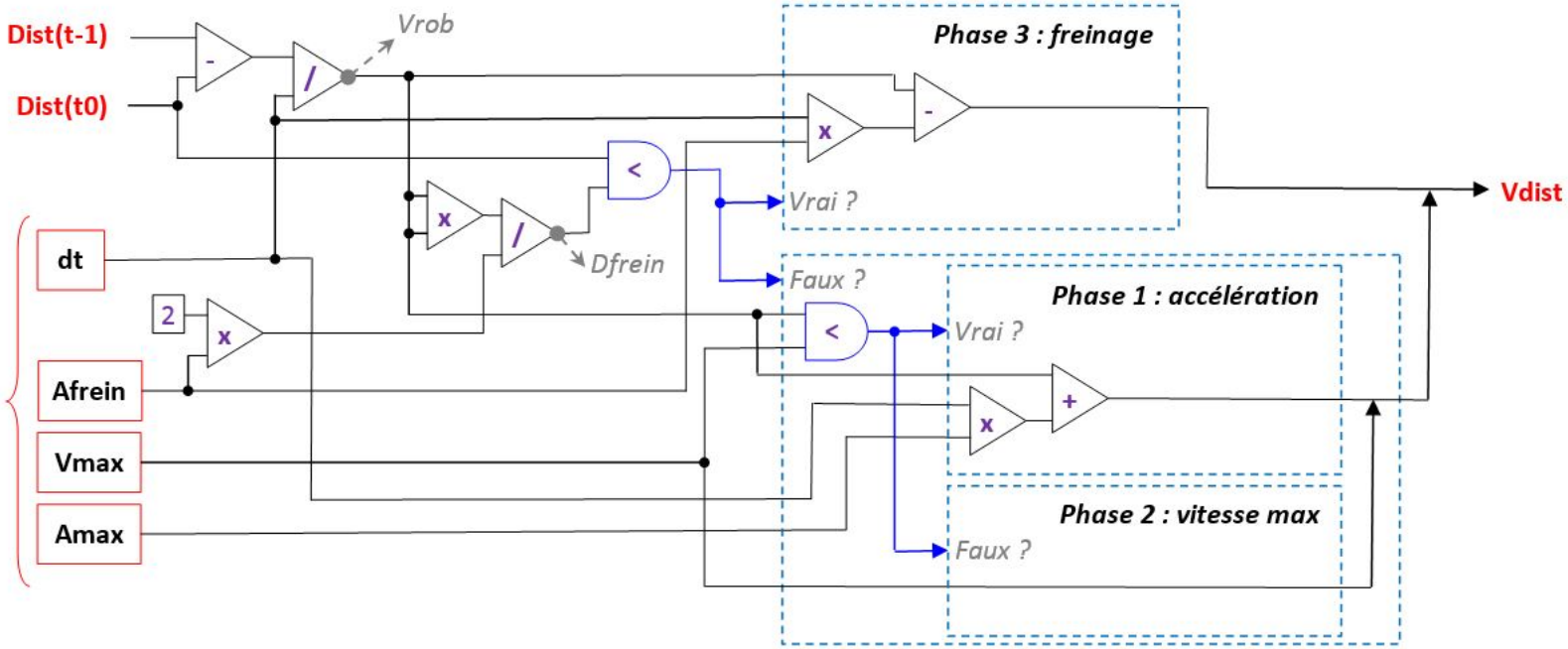
2/ Consigne :

C'est le calcul de l'erreur entre la position souhaitée et la position actuelle du robot :



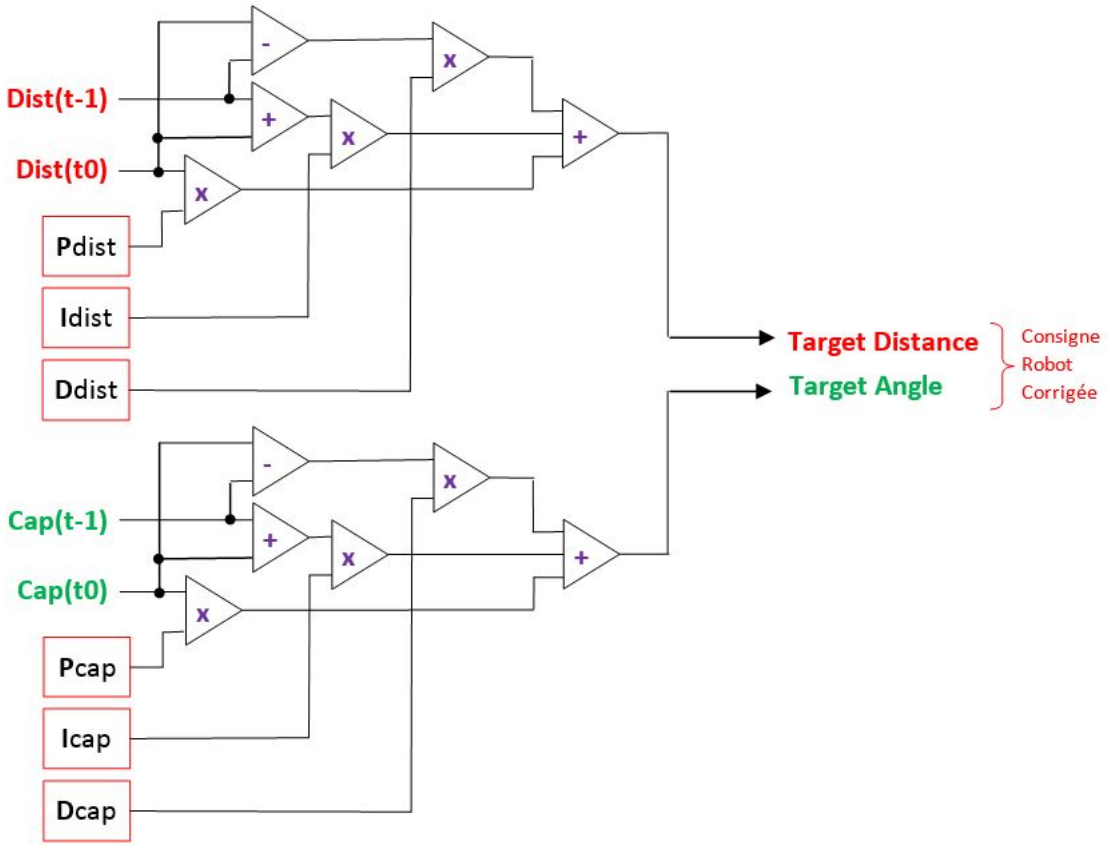
3/ Rampe :

La rampe permet de contrôler la vitesse max (de rotation et d'avance) et de définir une accélération et une décélération (freinage).



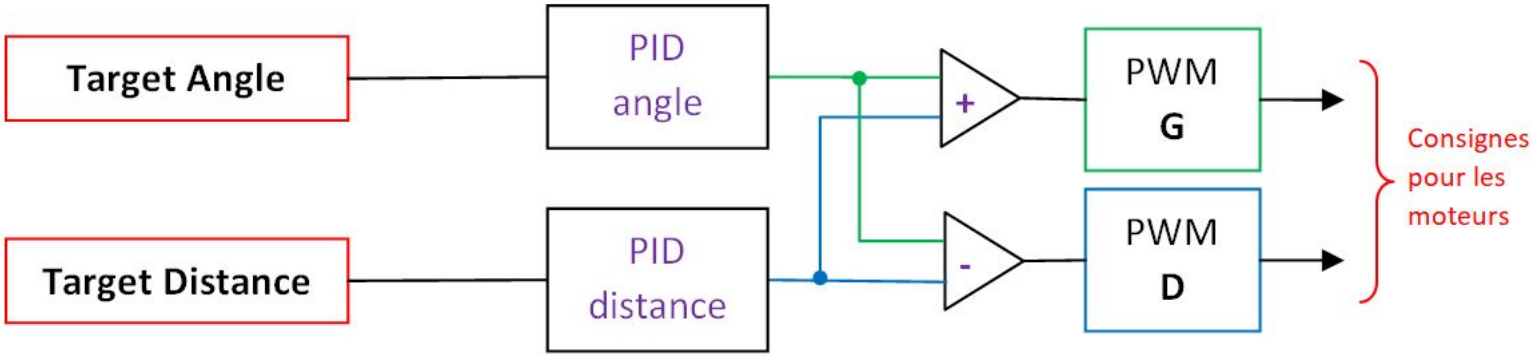
4/ PID :

C'est la régulation de l'asservissement, pour s'assurer de réduire l'erreur et donc atteindre l'objectif en un minimum de temps tout en limitant les oscillations :



5/ Commande :

Ce petit algorithme permet de convertir la consigne d'avance et de rotation en PWM pour chacun des moteurs :



Si on ajoute à ces algorithmes un pilotage de l'objectif on peut rapidement et simplement commencer à contrôler le robot sur des belles trajectoires. Dans un premier temps je conseil de se contenter de déplacer le robot par étapes : **prise du cap** >> **avance jusqu'à XY** >> **prise de l'angle final**.

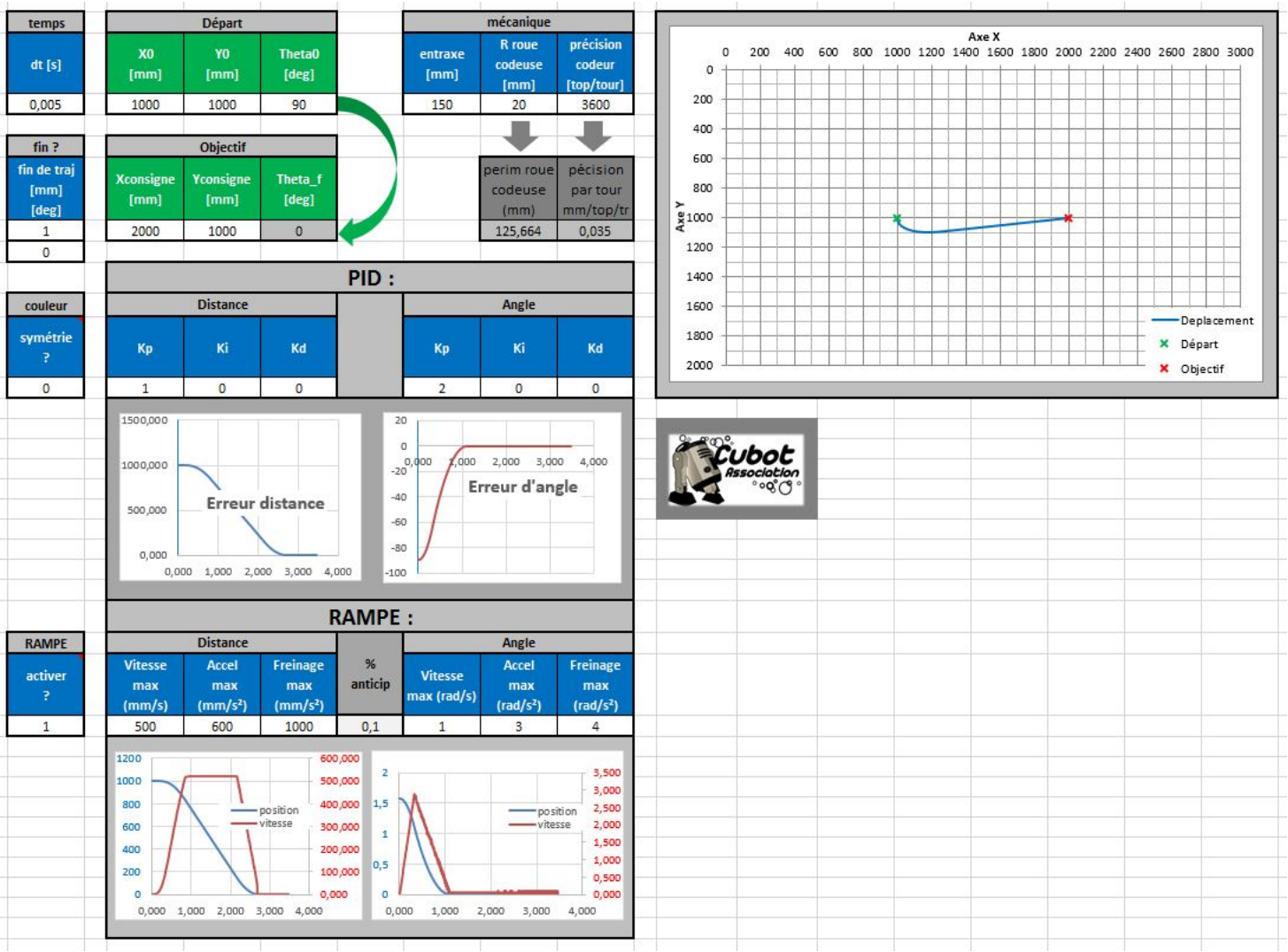
Mais la gestion de trajectoire sera le sujet d'un autre atelier.

Comme on aime partager dans l'équipe Cubot (et pas que des bières), voici un fichier Excel qui inclut tous ces algorithmes et permet de tester les déplacements en fonction de toutes les constantes en jeu :

Calcul Asservissement

Telecharger

NB : Pour nous remercier : on accepte de la bière, de l'argent ou des massages à nos stands en compétition !



Attention tout de même avec ce fichier Excel théorique : il n'y a pas d'oscillation, car le comportement des moteurs, l'inertie du robot et les glissements ne sont pas inclus dans l'équation de la réponse. Donc inutile d'imaginer régler vos coefficients avec ce fichier : rien de mieux qu'un réglage avec des essais physique.

Préc.