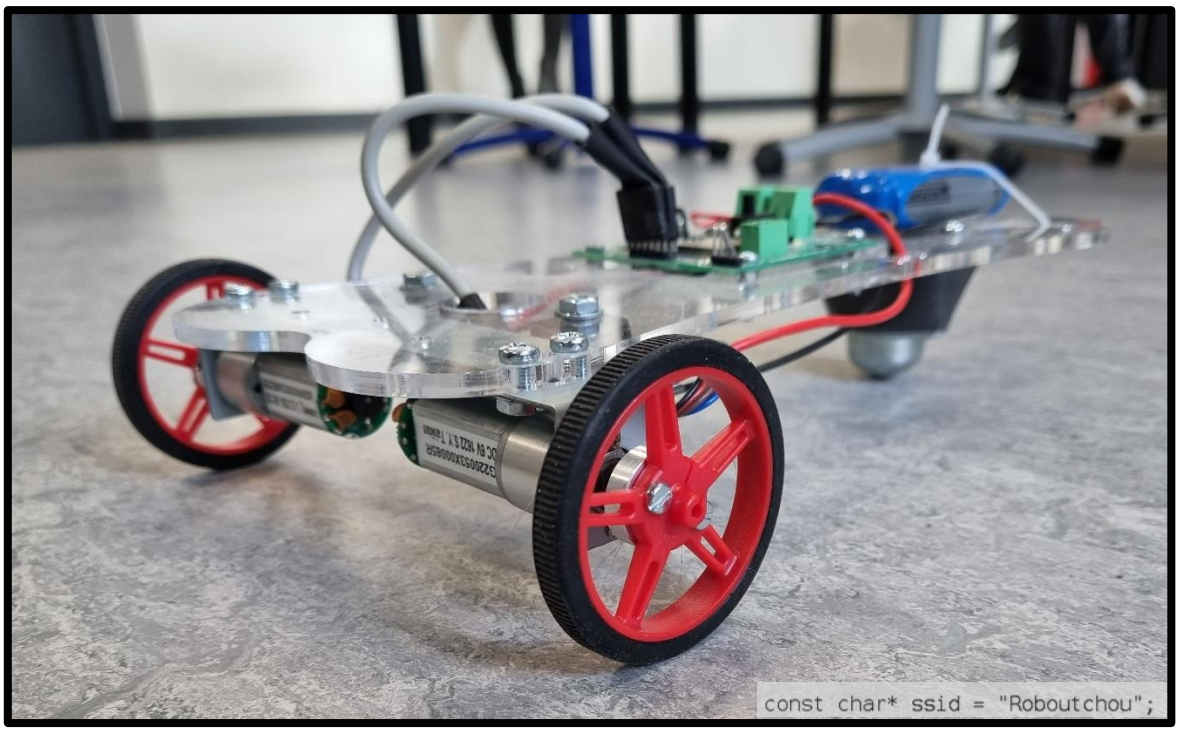
PRT n°6 : Programmation de robot

Algorithme d'évitement de trajectoire par la théorie des champs de potentiel



Réalisé par :

Mathis Louazé

Mathieu Rancé

Quentin Clément

Enseignants tuteurs :

Romain Delpoux

Rémi Chalard

SOMMAIRE

I. Introduction

La mobilité est une problématique majeure dans notre société. De nos jours, que ce soit pour se déplacer en tant que particulier ou pour gérer un flux de matière à des fins industrielles, il faut pouvoir le faire de manière efficace et optimisée. C'est dans ce contexte que se fait sentir le besoin d'automatiser certains systèmes de déplacement. Seulement, ce qui se révèle être compliqué pour l'automatisation des déplacements c'est la complexité du chemin à parcourir. En effet des obstacles peuvent apparaître sur la trajectoire de manière inopinée ou encore le chemin à parcourir peut nous être totalement ou partiellement inconnu. En somme, nous ne contrôlons généralement pas complètement le terrain. Il faut donc, dans un souci d'automatisation, être robuste face à ces aléas. Pour cela nous proposons de développer et tester une méthode d'optimisation de trajectoire capable d'éviter les obstacles.

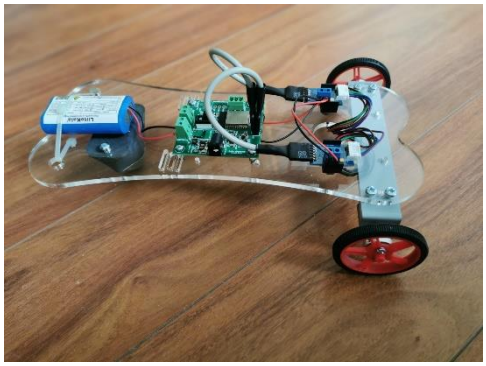
Notre projet a pour objectif d 'implémenter un tel algorithme de trajectoire sur un robot mobile préexistant. Une des premières étapes dans la mise en œuvre automatique notre trajectoire est de travailler sur la manœuvre d'évitements un itinéraire donné. De ce fait et dans un premier temps, notre algorithme doit permettre au robot d'éviter des obstacles connus par l'utilisateur, de manière à arriver à sa destination sans choc.

Nous allons donc poser plusieurs hypothèses sur l'environnement pour assurer la bonne réalisation du projet :

- Environnement contrôlé
- Le terrain est plat
- Les obstacles sont connus par l'utilisateur
- Les obstacles sont fixes
- Pas d'intempéries
- La surface de contact est homogène (pas de changement de terrain)
- Pas contrainte de contrainte de vitesse

II. Matériels à notre disposition

Nous avons tout d'abord à notre disposition un robot à deux roues motrices en traction et une roue folle à l'arrière. Le système est alimenté par une batterie lithium qui gère l'alimentation des moteurs et de l'unité de contrôle. Chaque roue motrice est pilotée par un moteur à courant continu. La commande est réalisée par une carte ESP32 avec un module WIFI. Le contrôle de la vitesse des moteurs se fait grâce à des ponts en H connectés à la carte. Nos seuls capteurs sont des codeurs incrémentaux servant à récupérer la vitesse de chaque roue.



Concernant la communication du robot, la carte ESP32 possède un module WIFI permettant une connexion à distance pour d'une part le commander et d'autre part récupérer des données utiles. Nous avions au début à notre disposition un code Arduino pour un premier pilotage de notre robot. Ce code Arduino avait déjà une librairie conséquente qui nous a servi de base. Il y avait entre autres la partie commande moteur, les protocoles de communication, la définition des encodeurs ainsi que la mise en place d'un protocole d'acquisition des vitesses et des angles de chaque moteur.

L'algorithme d'évitement d'obstacles avec lequel nous allons travailler est un code Matlab qui repose sur la théorie des champs de potentiels. Le principe est simple, dans un environnement connu, nous entrons le point de départ du robot afin d'initialiser sa position. Ensuite, le point d'arrivée est renseigné par un point attractif. Les obstacles quant à eux, sont représentés par des points répulsifs plus ou moins larges comme dans la figure ci-dessous. De ce fait, la trajectoire est le gradient descendant du champ de potentiel. Le code va donc permettre au robot d'éviter les zones répulsives pour se rendre vers son point d'arrivée.

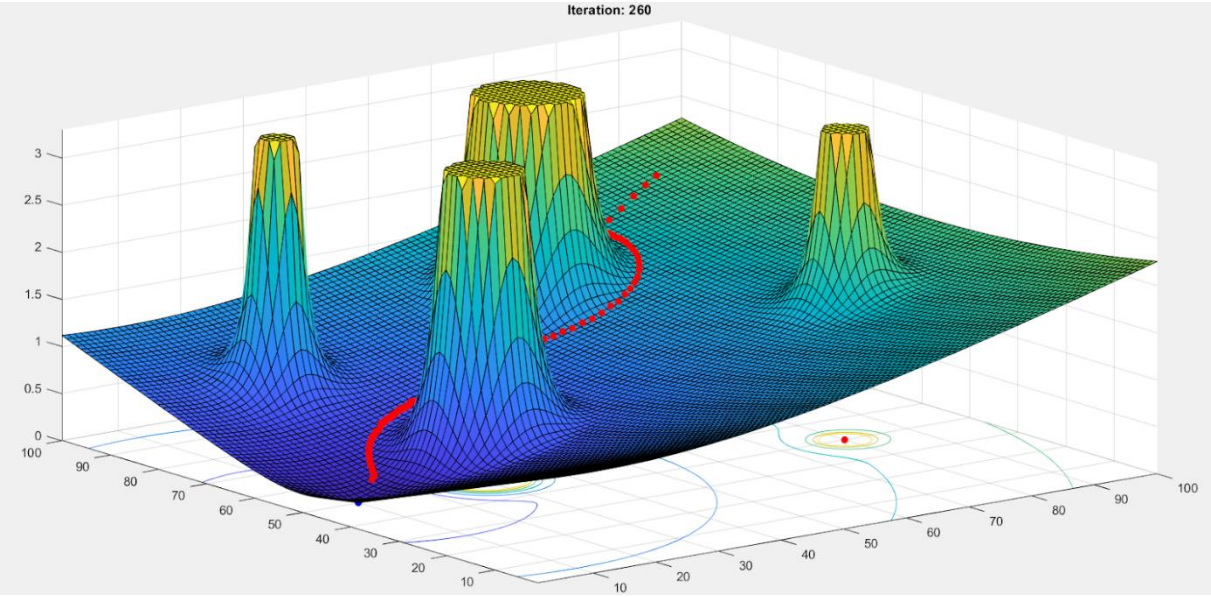


Figure 1 : Exemple de trajectoire par un champs de potentiels

III. Plan d'étude

Dans un premier temps, nous tâcherons d'asservir le robot. Pour ce faire nous allons procéder à 3 boucles d'asservissement que l'on implémente successivement :

- Une boucle d'asservissement en vitesse. C'est le premier paramètre à réguler pour pouvoir contrôler notre robot.
- Une boucle d'asservissement angulaire pour chaque roue motrice. Cette boucle d'asservissement sur le système préalablement réguler en vitesse.
- Une boucle d'asservissement en position. Cette dernière boucle nous permettra de faire en sorte que le robot suive un tracé de point successif.

Une fois que le véhicule sera capable de suivre les points qu'on lui a indiqués, on travaillera sur la génération de la trajectoire en évitant les obstacles.

Nous allons revenir sur chacune de ces parties pour les expliquer en détail et voir entre autres les difficultés que nous avons pu rencontrer durant leur élaboration.

IV. Les asservissements

a) Asservissement en vitesses des deux moteurs

Nous disposons sur notre robot de deux moteurs mcc (machine à courant continu), notre but étant d'être précis sur les commandes demandées au robot, nous avons dû les asservir. Le premier asservissement en vitesse est semblable à ceux que l'on a pu effectuer en TP d'automatisme.

Nos deux moteurs ont un comportement similaire et du 1er ordre que l'on peut voir sur la capture ci-dessous. Comme leur comportement est très proche à partir d'une certaine vitesse assez faible, nous en profiterons pour modéliser et donc implémenter un correcteur identique pour chaque moteur.

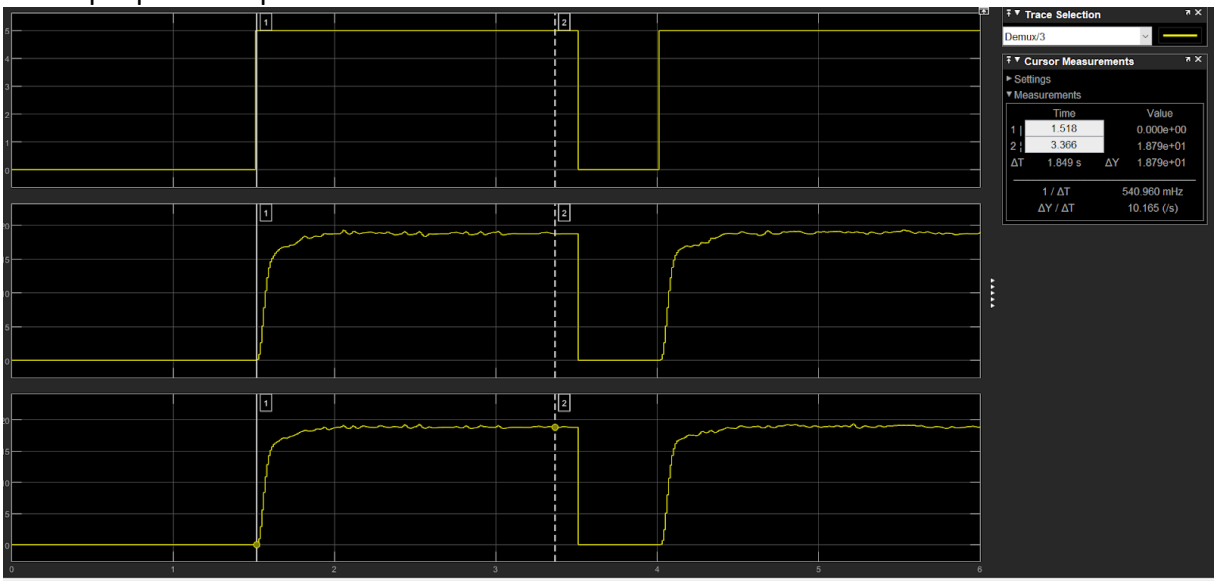


Figure 2: réponse indicielle d'un des MCC

L'analyse de la réponse indicielle en boucle ouverte de nos moteurs nous permettent de déterminer les paramètres suivants :

Le gain du système : K=3,754 rd/s/V

Ainsi que son temps caractéristique : τ=0,09s

Ces paramètres nous ont permis de modéliser sur Matlab notre processus afin de réaliser un correcteur adéquat. Nos simulations avec un correcteur PI nous donne les résultats suivants:

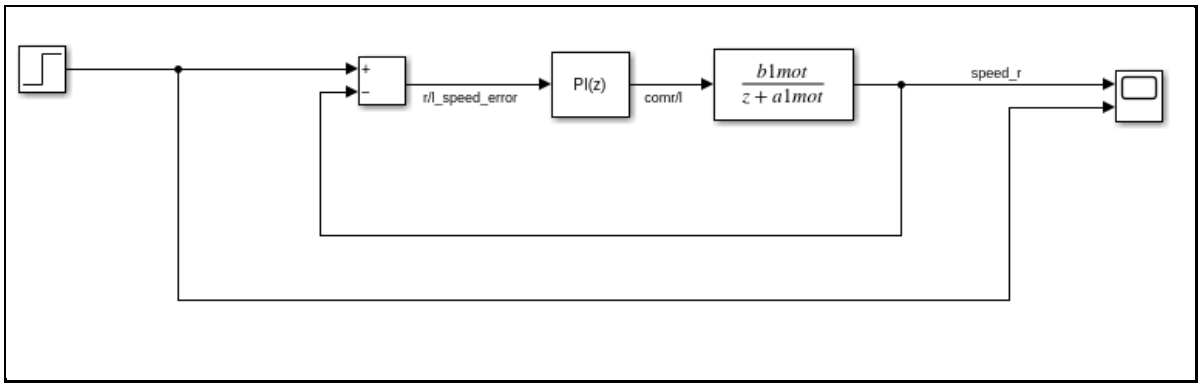


Figure 3 : Simulation Matlab de l'asservissement des MCC

D'après les mesures en réponse indicielle, notre système peut s'assimiler à un second ordre, un PI est donc suffisant. Pour le dimensionnement de notre correcteur nous avons décidé de garder une dynamique proche du système en boucle ouverte avec un écart statique nul.

Une fois le cahier des charges établi, on utilise nos annexes pour établir notre correcteur en continue, puis pour le discréditer. On obtient le résultat ci-dessous.

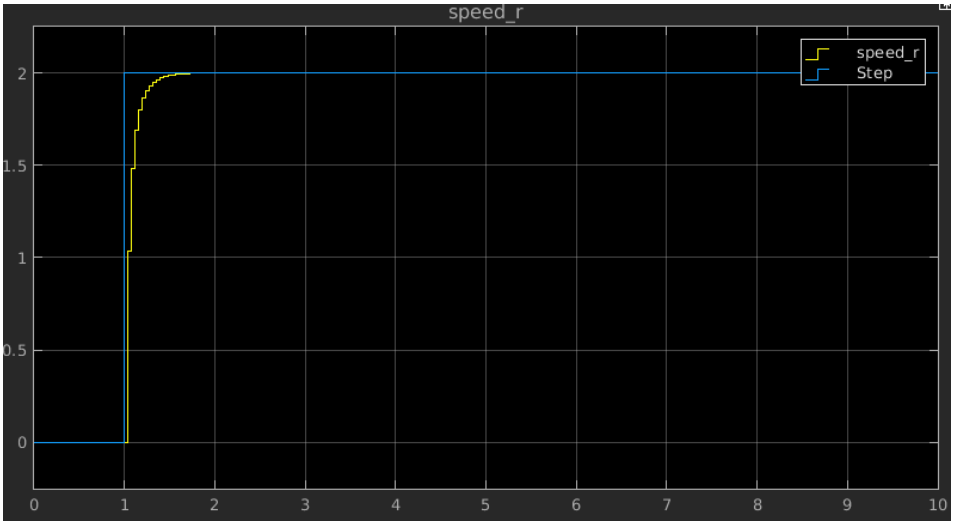


Figure 4: réponse du système discrétisé et asservis

Comme on peut le voir sur la capture de notre scope, le correcteur semble parfaitement fonctionnel. Nous avons donc choisi t0=τ=0,09s. Les paramètres du correcteur PI obtenus sont :

- Pmoteur=0,3848
- Imoteur = 2,9598

Nous pouvons maintenant que la simulation marche parfaitement nous pouvons intégrer notre correcteur dans le code du robot. A partir de la commande discrète, on extrait une équation de récurrence pour la commande que l'on implémente dans le code.

```
r_speed_error = r_speed_Command - speed_r;
l_speed_error = l_speed_Command - speed_l;

sum_r_speed_error += tsControl * 0.001 * r_speed_error;
sum_l_speed_error += tsControl * 0.001 * l_speed_error;

com_r = KpMoteur * r_speed_error + KiMoteur * sum_r_speed_error;
com_l = KpMoteur * l_speed_error + KiMoteur * sum_l_speed_error;
```

Après contrôle sur le robot, celui-ci semble bien avancer à vitesse constante selon les mesures des encodeurs. On vérifie ensuite de manière expérimentale en mesurant la vitesse du robot grâce à un pointage vidéo. On constate cependant que celui-ci à une forte tendance à glisser et ce peu importe la surface sur laquelle il se trouve. La solution la plus facile trouvée pour pallier ce problème est de poser sur ce dernier juste au-dessus des roues motrices une charge permettant d'augmenter les frottements et donc de réduire le glissement. Pour pouvoir disposer notre charge, il a fallu déplacer les ponts en H sur le châssis du robot.

b) Asservissement en angle du robot

Maintenant que l'on arrive à asservir la vitesse d'avancée de notre robot, nous souhaitons piloter ce dernier en angle afin de pouvoir lui faire effectuer des trajectoires précises.

Pour cela nous avons réalisé un nouvel asservissement mais cette fois ci non plus d'un robot seul mais du système entier (càd nos deux moteurs).

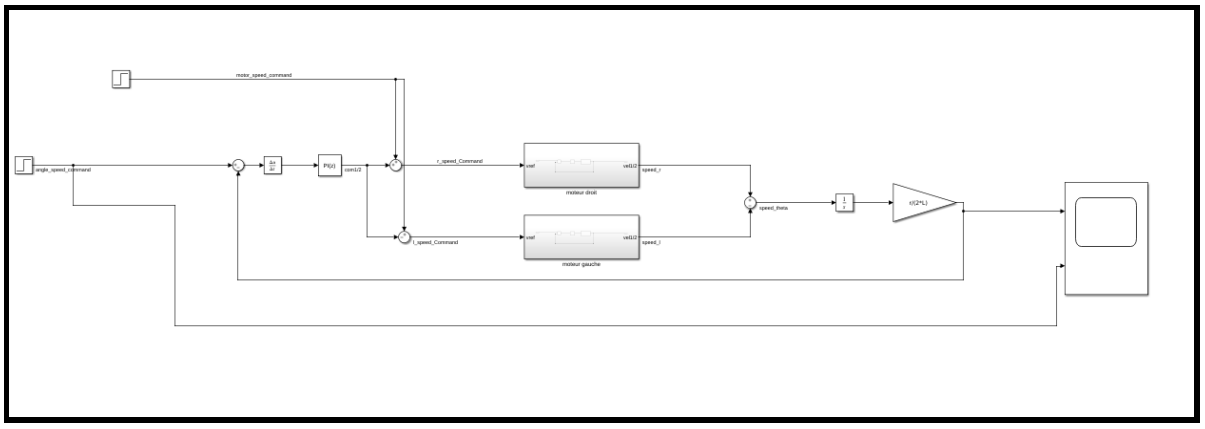


Figure 5 : Modélisation complète du système en boucle ouverte

Notre modélisation du système est la suivante, on peut voir les deux blocs qui correspondent à nos deux moteurs asservis en vitesse. Ceux-ci reçoivent en entrée la commande de vitesse moyenne (comme vu dans l'asservissement précédent) en plus d'une commande d'angle pondéré par un coefficient de +1 ou -1 suivant le côté du moteur. Cette différence sur notre entrée entraîne une variation angulaire de notre robot, celle-ci est intégrée pour obtenir l'angle en fonction du temps Θ(t). Puis Θ(t) est multipliée par un facteur r/2L avec r le rayon de nos roues et L la distance entre nos deux roues afin de passer de la variation d'angle au niveau de nos roues à la variation d'angle du robot lui-même.

Pour l'asservissement, on procède de la même manière que pour l'asservissement en vitesse. La réponse indicielle en boucle ouverte s'assimile de nouveau à un système du premier ordre et on décide de nouveau d'utiliser un correcteur PI. Cette fois-ci, on choisit une dynamique en asservissement assez lente pour éviter de déraper et d'accumuler des erreurs que l'on ne pourra pas corriger. On s'impose aussi un écart statique nul. On calcule le correcteur PI en continu en respectant le cahier des charges puis on le discrétise.

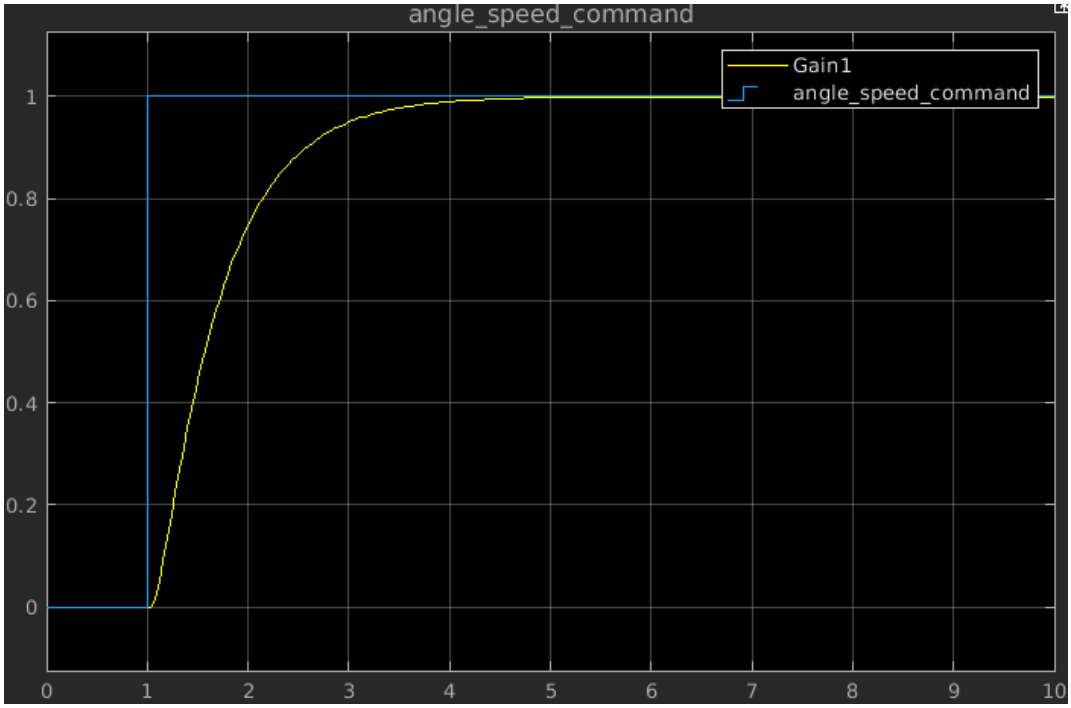


Figure 6 : réponse du système complet discrétisé et asservis

Notre asservissement en angle nous donne les résultats ci-dessus. On peut remarquer que la courbe obtenue semble particulièrement lente. Il s'agit là d'un choix volontaire (on a pris T0 Angle=4s) car cela permet d'éviter que le robot patine sur la surface, en effet avec un correcteur trop rapide les commandes évoluent trop vite et la friction sur le sol n'est pas suffisante pour effectuer les commandes.

Les paramètres de ce correcteur PI sont les suivants :

- Pangle=0,0982
- Iangle = 3,7403

<u>NB1</u> : Notre système ici n'était pas réellement un 1er ordre d'où la forme de notre courbe après asservissement. Cependant les résultats obtenus avec un correcteur PI étant fonctionnels, nous n'avons pas jugé nécessaire d'utiliser une boucle d'asservissement plus complexe.

Nous obtenons sur le robot le code suivant à partir du correcteur discrétisé :

```
void mode_cl() {

  erreur_alpha=(angle_command-pos_alpha);
  erreur_alpha_prim=(erreur_alpha-erreur_alpha_prec)/(tsControl * 0.001);

  sum_erreur_alpha_prim+= tsControl * 0.001 * erreur_alpha_prim;

  com_alpha = kpalpha * erreur_alpha_prim + kialpha * sum_erreur_alpha_prim;

  float r_speed_Command = motor_speed_command + com_alpha;
  float l_speed_Command = motor_speed_command - com_alpha;

  r_speed_error = r_speed_Command - speed_r;
  l_speed_error = l_speed_Command - speed_l;

  sum_r_speed_error += tsControl * 0.001 * r_speed_error;
  sum_l_speed_error += tsControl * 0.001 * l_speed_error;

  com_r = KpMoteur * r_speed_error + KiMoteur * sum_r_speed_error;
  com_l = KpMoteur * l_speed_error + KiMoteur * sum_l_speed_error;

  erreur_alpha_prec=erreur_alpha;
}
```

Les tests effectués sur ce dernier nous donnent une précision relative sur l'angle et la vitesse de l'ordre de 10%. Ces résultats sont toujours obtenus avec l'ajout d'une masse sur l'avant du robot. Il est possible qu'une partie de l'imprécision soit toujours dû au patinage de notre robot sur le sol malgré la charge. La précision obtenue est cependant jugée suffisante pour continuer le projet.

Nous obtenons donc une commande robuste et fonctionnelle du robot sur sa vitesse moyenne (en rad/s au niveau des roues) et sur son angle (en rad).

NB2 : Dans un premier temps nous avions mis en place un asservissement en angle et en vitesse angulaire. Bien plus simple à mettre en place car l'asservissement en vitesse angulaire ne demandait pas de correcteur, il s'est avéré trop imprécis pour la suite du projet et nous avons dû effectuer l'asservissement en angle.

V. Suivi de point (asservissement en position)

Notre robot étant correctement asservi en vitesse et angle, il devient possible de lui envoyer une suite de commande permettant de suivre une trajectoire. A ce stade du projet, nous commençons à être limité par notre interface de contrôle à distance du robot. Codée sur Matlab, nous n'avons pas la main sur cette dernière (pas de code source disponible sur le moment) et nous n'étions pas familier avec la programmation d'application sur le logiciel. Nous avons donc pris la décision de créer une nouvelle interface de commande et monitoring en C++ sur laquelle on aurait complètement la main.

Nous développons donc durant les vacances de la Toussaint une application en C++ sous QTCreator (reposant sur le système QTWidget et QMainwindows). Le travail est très laborieux notamment sur la connexion entre le robot et l'ordinateur sur laquelle nous n'avions à l'époque aucune information. Finalement, nous obtenons à la fin de la semaine un programme dont les fonctionnalités sont semblables à celles que nous avons sur l'application Matlab et avec un fonctionnement relativement similaire (communication via wifi en TCP).

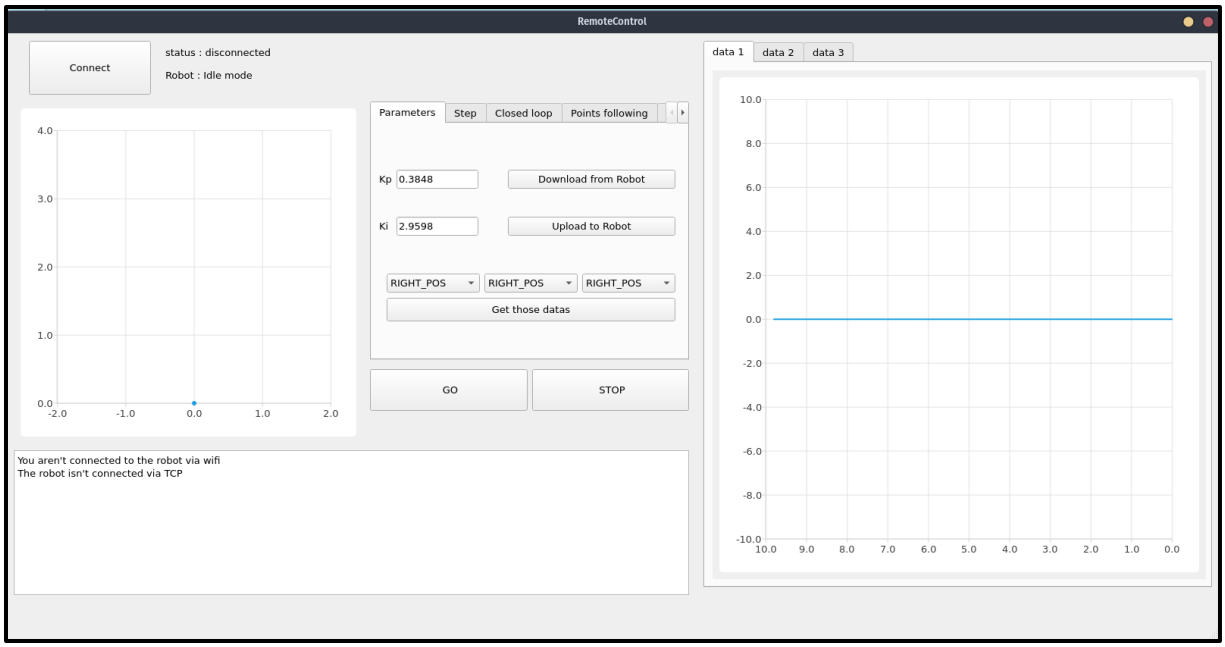


Figure 7 : Interface de commande QT

Les fonctions disponibles à l'époque sur ce dernier sont le mode « step », « closed loop » et surtout la sélection à distance des paramètres de monitoring (exemple position en x ou angle instantanée) ce qui permet un debug facile du robot. On a également ajouté un système de modification en direct du correcteur de nos moteurs (que l'on a finalement très peu utilisé).

Nous sommes venus ajouter à toutes ces fonctionnalités une zone de tracé de trajectoire où l'on peut sur une carte créer plusieurs points de passage et les envoyer au robot (sur lequel on a créé un système de réception des données adéquat).

<u>NB3</u> : Le système de commande est dès le départ considéré plutôt comme un prototype, sa programmation n'est pas la plus rigoureuse, le but recherché est uniquement son fonctionnement. A la fin on se rendra compte que ce dernier est devenu un véritable "code spaghetti". Si nous avions eu plus de temps, refaire depuis 0 celui-ci aurait été un de nos objectifs.

Il nous reste donc à réaliser un asservissement en position du robot de sorte qu'il arrive à suivre notre suite de points. Pour cela nous avons encore utilisé dans un premier temps Matlab. L'idée de l'algorithme de suivi de point est le suivant :

- On calcule l'angle entre la droite formée par la position actuelle du robot et le point ciblé et notre axe des X
- On envoie cet angle en commande au robot ainsi que la vitesse moyenne sélectionnée
- On calcule la distance entre le point actuel du robot et la position souhaitée, si elle est inférieure à une valeur spécifiée on passe au point suivant

Sur Matlab on utilise qu'un seul point donc l'étape 3 n'est pas utilisée. On obtient la simulation suivante :

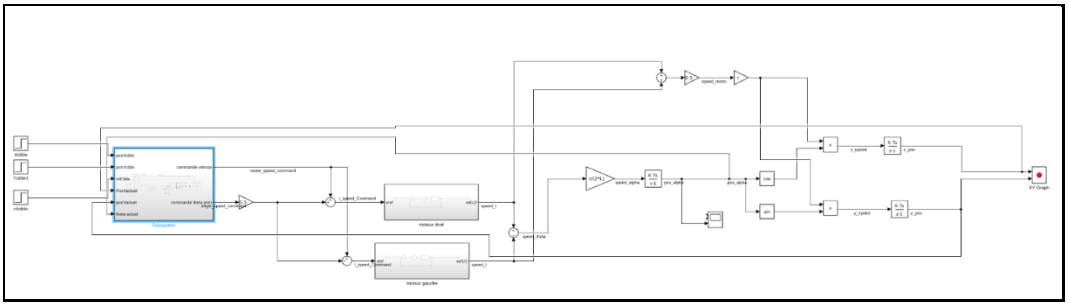


Figure 8 : Schéma du système complet asservi en position

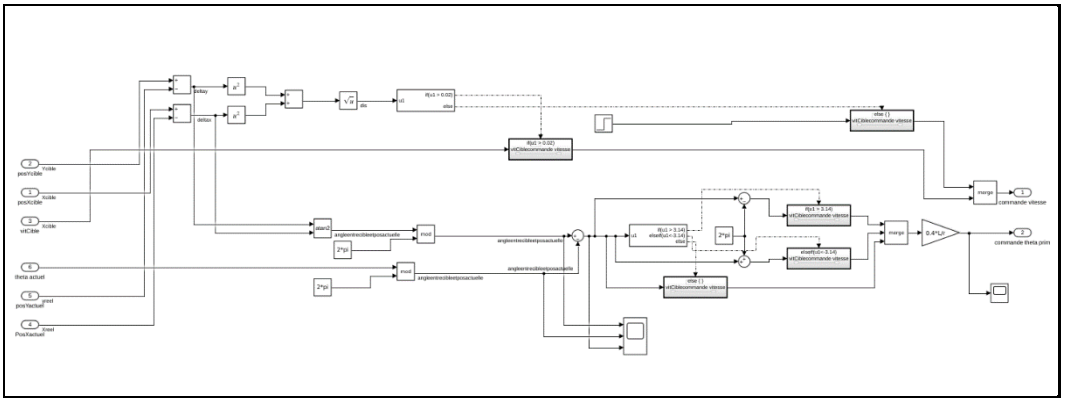


Figure 9 : Schéma du bloc de commande en position

Notre asservissement en position réside donc dans le gros bloc en bleu dont le détail est donné ci-dessous. Celui-ci paraît très compliqué mais en réalité sa complexité ne vient que du fait qu'il est nécessaire pour notre angle de commande de se trouver en -π/2 et +π/2 pour que le robot tourne correctement peu importe la direction exigée. Cela impose plusieurs blocs modulos qui complexifie le modèle Simulink.

<u>NB4</u> : Ici on a la simulation que l'on avait lorsque l'on utilisait toujours un asservissement en vitesse angulaire. Nous avons depuis modifié le schéma en utilisant l'asservissement en position angulaire (disponible dans le dossier rendu).

On obtient avec notre algorithme le résultat suivant :

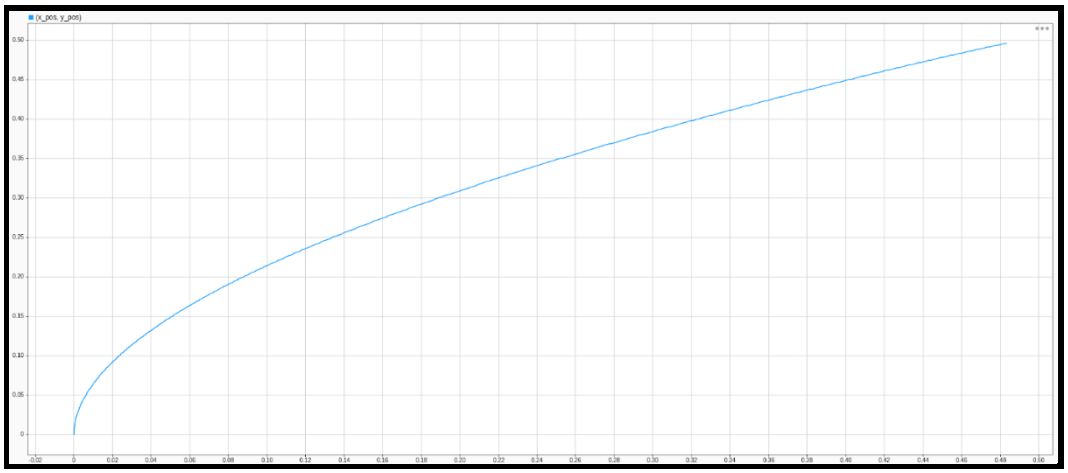


Figure 10 : Trajectoire du robot entre 2 points

On atteint bien le point cible (en haut à droite en partant de (0,0). Comme dit précédemment nous n'avons ici qu'un seul point car la modélisation Simulink de plusieurs points allait être encore plus complexe et ne nous apporterait strictement rien de plus.

Nous pouvons adapter le code sur la carte.

```
void calctraj::RecupCommandes(float &CommVitesse, float &CommAngle, const float &Px, const float &Py, const float &Alpha){
  if(!TrajEffectuee&&TrajOK){
    float deltaX=PosX[PointCible]-Px;
    float deltaY=PosY[PointCible]-Py;
    float Angle=atan2(deltaY,deltaX);
    if(Angle>3.14){
      Angle-=2*3.14;
    }
    if(Angle<-3.14){
      Angle+=2*3.14;
    }

    if(Alpha-Angle>3.1415){
      Angle+=2*3.14;
    }
    if(Angle-Alpha>3.1415){
      Angle-=2*3.14;
    }

    CommAngle=Angle;
    CommVitesse=VitComm;

    if((deltaX*deltaX+deltaY*deltaY)<pow(DISTANCE_MIN,2)){
      Serial.println("je suis arrivé");
      PointCible++;
      if(PointCible==NbrPoints){
        if(Repeat){
          PointCible=1;
        }else{
          TrajEffectuee=true;
        }
      }
    }
  }else{
    CommAngle=0;
    CommVitesse=0;
  }
}
```

Notre code est situé dans une classe appelée "CalcTraj" qui contient toutes les informations nécessaires à la réalisation de la trajectoire :

- Vitesse moyenne
- Points de passage
- Répétition du parcours à sa fin ou pas

Le code suit donc le procédé décrit précédemment, la seule différence notable est la comparaison Alpha-Angle. Celle-ci permet d'éviter que dans certains cas le robot tourne dans le sens inverse au sens le plus rapide pour aller dans la bonne direction.

Ici aussi, après beaucoup de debug nous obtenons un fonctionnement correct du système. On arrive bien à suivre notre série de points. La précision de notre robot n'est pas parfaite mais toujours de l'ordre des 10% d'erreurs vues précédemment.

<u>NB5</u> : Ici le fonctionnement était bon dès le départ mais nous a demandé beaucoup de travail pour enlever de petits bugs (départ dans le mauvais sens, rotation dans le mauvais sens dans certains cas, impossible de démarrer avec le robot placé sur l'axe des X…). Il semble cependant que nous soyons parvenus à résoudre la totalité de ces problèmes.

Nous arrivons donc désormais à suivre un parcours bien défini et envoyé dans sa totalité au robot.

VI. Implémentation d'une trajectoire

Nous avons désormais toutes les briques élémentaires nécessaire à l'évitement d'obstacles, plusieurs alternatives sont envisageables :

- Calculer une trajectoire depuis une liste d'obstacles sur l'ordinateur et envoyer la trajectoire au robot comme précédemment
- Envoyer au robot la liste d'obstacles et le laisser calculer en local la trajectoire qu'il va effectuer (donc calcul de trajectoire sur le robot et non plus sur l'ordinateur)
- Développer un système de trajectoire évolutif en temps réel qui recalcule en direct sa future trajectoire. Ce système serait celui à implémenter sur un robot qui aurait des capteurs intégrés car la carte n'est dans la réalité pas connue à l'avance mais découverte petit à petit par le robot.

La solution la plus intéressante que nous aurions dû développer est la 3ème mais faute de temps nous avons préféré développer la 1ère qui nous assure un fonctionnement correct et qui est bien plus facile à modifier/déboguer puisque le calcul à lieu sur le PC qui peut donc afficher la trajectoire calculée.

NB6 : Avec un peu plus de temps, il est certain que nous nous serions consacrés à la réalisation de la solution 3. Notre solution reste cependant intéressante si on imagine un système ou une caméra placée au plafond pour scanner la zone qui serait donc connue à l'avance.

Nous avons donc dû réaliser un calculateur de trajectoire en fonction d'obstacles. Tout d'abord on a développé un système pour rentrer nos obstacles sur le logiciel de commande à distance. On peut choisir leur position au premier clic et leur rayon (donc leur poids) au second clic. Ces informations sont enregistrées en local sur l'ordinateur qui va pouvoir effectuer le calcul de trajectoire.

Ce dernier repose sur la méthode de calcul donnée sur Matlab, quelques différences sont à noter :

- On a choisi d'avoir un pas fixe de sorte à ne pas créer une multitude de points très proches qui risquerait de perturber notre robot en le faisant zigzaguer. Pour cela on divise nos composantes de force en X et Y par la norme de la force puis on multiplie par notre pas
- On a lissé notre parcours en prenant comme direction la moyenne entre la force précédente et la nouvelle force pour éviter encore une fois un phénomène de zigzag

```
while(!Arrived&&i<maxiteration){
    XForce=0;
    YForce=0;
    for(int k=0;k<Obstacles.count();k++){ //repulsives forces
        float deltax=(posX-Obstacles.at(k).x());
        float deltay = (posY-Obstacles.at(k).y()) ;
        float dis=sqrt(deltax * deltax + deltay * deltay);
        float radius = Radius.at(k);

        XForce+=deltax/dis*radius*1.5*exp(-Radius.at(k)*dis*dis);
        YForce+=deltay/dis*radius*1.5*exp(-Radius.at(k)*dis*dis);
    }

    QPointF arrivePoint;
    arrivePoint=Dest.at(0);
    float deltax=(posX-arrivePoint.x());
    float deltay = (posY-arrivePoint.y()) ;
    float dis=sqrt(deltax * deltax + deltay * deltay);//attractive forces

    XForce-=deltax/(dis*dis)*0.9;
    YForce-=deltay/(dis*dis)*0.9;//ajouter coeff?

    XForce=0.5*(XForce+XForcePrev);
    YForce=0.5*(YForce+YForcePrev);


    NormForce=sqrt(XForce*XForce+YForce*YForce);
    Xdisplacement=XForce/NormForce*stepsize;
    Ydisplacement=YForce/NormForce*stepsize;

    newX=ListPoints.points().last().x()+Xdisplacement;
    newY=ListPoints.points().last().y()+Ydisplacement;

    posX=newX;
    posY=newY;

    NewPoint=QPointF(newX,newY);
    ListPoints<<NewPoint;

    if(dis<=stepsize){//verification distance
        Arrived=true;
    }

    XForcePrev=XForce;
    YForcePrev=YForce;

    i++;
}
```

On obtient le code ci-dessus. Ce dernier semble plutôt fonctionnel, il laisse apparaître plusieurs constantes sur le calcul des forces attractives et répulsives. Ces dernières ont été obtenues de manière empirique en testant plusieurs scénarios d'obstacles et la réaction obtenue.

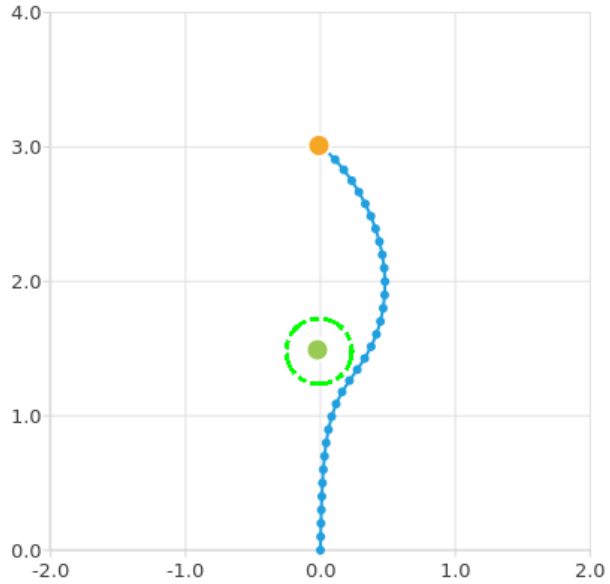


Figure 11 : Tracé de trajectoire avec obstacle

On peut voir que notre robot esquive bel et bien les obstacles posés sur son chemin. Le nuage de point obtenu est bien espacé régulièrement (de 10cm, paramètre réglé dans le code). Le comportement n'est dans la réalité pas parfait et peut amener à des résultats étranges dans certains cas. Il est assez difficile de trouver une solution qui répond parfaitement à tous les parcours d'obstacles. Nous pensons cependant avoir trouvé un bon compromis grâce à nos coefficients de réglage comme le montre la photo ci-dessous.

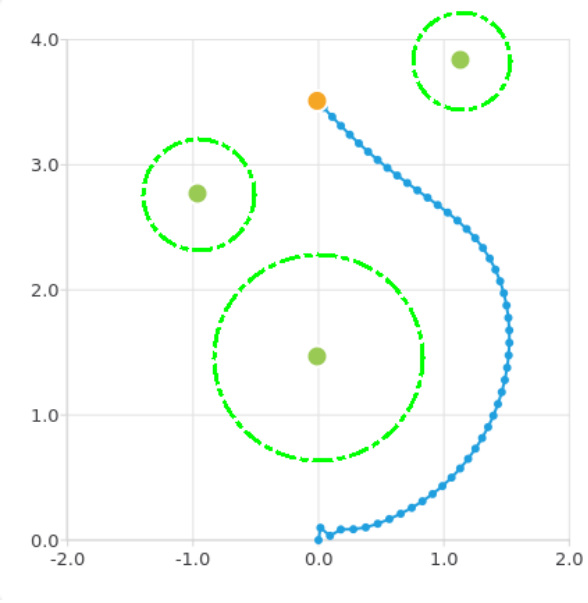


Figure 12 : Tracé de trajectoire complexe

Il ne reste plus qu'à transmettre la série de point au robot comme précédemment afin qu'il effectue la trajectoire. Les résultats obtenus sont exactement ceux espérés. Tous les éléments étudiés précédemment (calcul de trajectoire, asservissement angulaire et suivi de point) se comportent comme prévu une fois assemblés.

VII. Conclusion

Au cours de ce PRT nous avons pu répondre en grande partie à notre problématique. Nous avons à partir d'un robot existant, mis en œuvre un protocole permettant de l'asservir puis de lui faire suivre une trajectoire. Cette dernière est bien générée en fonction des obstacles présents sur le terrain. Notre projet est relativement abouti, dans son état actuel on peut :

- Générer une trajectoire sur ordinateur sur une surface de 2m par 4m.
- Placer des obstacles n'importe où et de taille variable
- Définir le pas entre chaque point
- Définir la vitesse de notre robot
- Suivre la trajectoire voulue

Bien sûr, en l'état le système n'est pas fini. Pour améliorer le système afin de l'implémenter dans n'importe quelle situation il faudrait :

-Régler le problème de glissement au niveau des roues
-Faire le calcul de trajectoire en local, sur la carte du robot
-Pouvoir détecter les obstacles et mettre à jour la trajectoire en temps réel