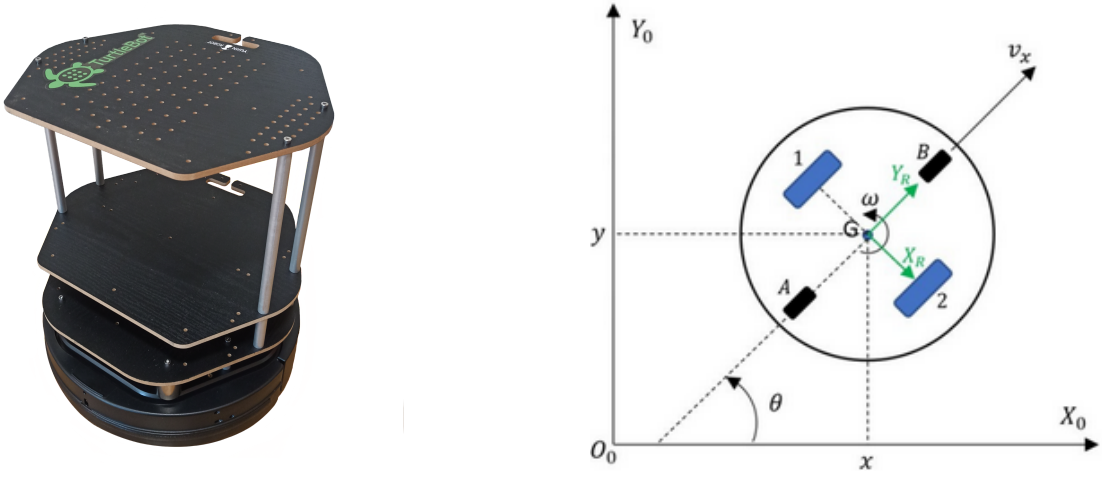
article
Le robot que nous avons utilisé pour ce projet est un TurtleBot 2, qui, comme les autres TurtleBot, est un robot à entraînement différentiel, ce qui signifie que chaque moteur peut recevoir des commandes indépendantes pour générer un déplacement. Les TurtleBots possèdent des châssis cylindriques, deux roues et peuvent embarquer une multitude de capteurs différents. Dans le cadre du projet, nous n'utilisons aucun des capteurs embarqués sur le robot, puisque le retour des caméras Qualisys forme un capteur de position absolu.

Pour chaque contrôleur programmé, nous avons systématiquement commencé par la création d'un simulateur. Cela permet de tester nos programmes sans prendre le risque de dégrader le matériel mis à disposition. De plus, une simple compilation du programme est beaucoup plus rapide que le lancement de l'interfaçage entre ROS et Qualisys. Pour les premières phases de programmation et de test du contrôleur, l'utilisation d'un simulateur est un gain de temps conséquent dans le projet.

Cependant, pour pouvoir simuler le robot, il faut pouvoir modéliser les équations dynamiques qui le régissent et les résoudre pour chaque itération dans la boucle de simulation. C'est pourquoi nous avons utilisé un modèle très simple et général de robot à deux roues cylindriques à entraînement différentiel.

0.1 Modèle d'un robot à entraînement différentiel



(a) Vue isométrique du TurtleBot 2

(b) Schéma de la cinématique du TurtleBot

FIGURE 1

La modélisation de la figure 1.b permet très facilement de déterminer les équations cinématiques du robot. On note :

- $O_0X_0Y_0$ est le repère absolu.
- GX_RY_R est le repère local du robot avec G le centre instantané de rotation du robot qui correspond aussi à son centre de gravité.
- θ l'orientation du robot par rapport au repère absolu
- x et y les coordonnées du centre G dans le repère absolu.
- ω la vitesse de rotation

- v_x la vitesse d'avance linéaire du robot

On en déduit le système suivant :

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{1}$$

Cette équation présente en vecteur de sortie les dérivées premières de position et d'orientation, et en vecteur d'entrée la vitesse de rotation ω et la vitesse linéaire v. Cette formalisation est adéquate pour notre situation puisque nous allons par la suite programmer des contrôleurs qui génèrent des commandes en vitesse de rotation et de translation, entraînant une modification du vecteur de position et d'orientation du robot simulé par le système cinématique 0.1.
La résolution numérique de ce système cinématique à chaque itération avec de nouvelles commandes et de nouvelles conditions initiales dans la boucle de simulation permet alors de reconstruire la trajectoire du robot.

0.2 Premier contrôleur à machine à états

Notre premier objectif était de réaliser un contrôleur simple pour asservir le robot en position en générant des consignes de vitesse trapézoïdales en translation et en rotation pour garantir une certaine fluidité dans le déplacement et pour ne pas détériorer le TurtleBot. Pour cela, nous avons utilisé un contrôle fonctionnant avec une machine à état qui découple le système cinématique du robot. Avec ce contrôleur, le robot effectue successivement une rotation pour se placer en direction d'un point cible (waypoint) puis une translation pour se déplacer jusqu'au point cible.

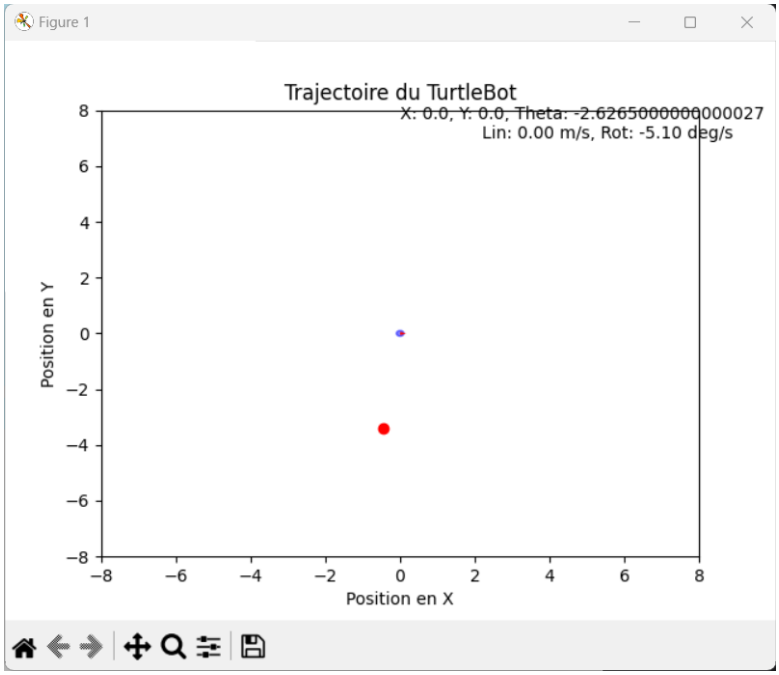
0.2.1 Création d'un simulateur

Nous avons alors implémenté rapidement un petit simulateur sur MATLAB pour tester le bon fonctionnement du système, puis nous avons retranscrit le code en Python, car l'environnement ROS nécessite des codes en Python ou en C++.

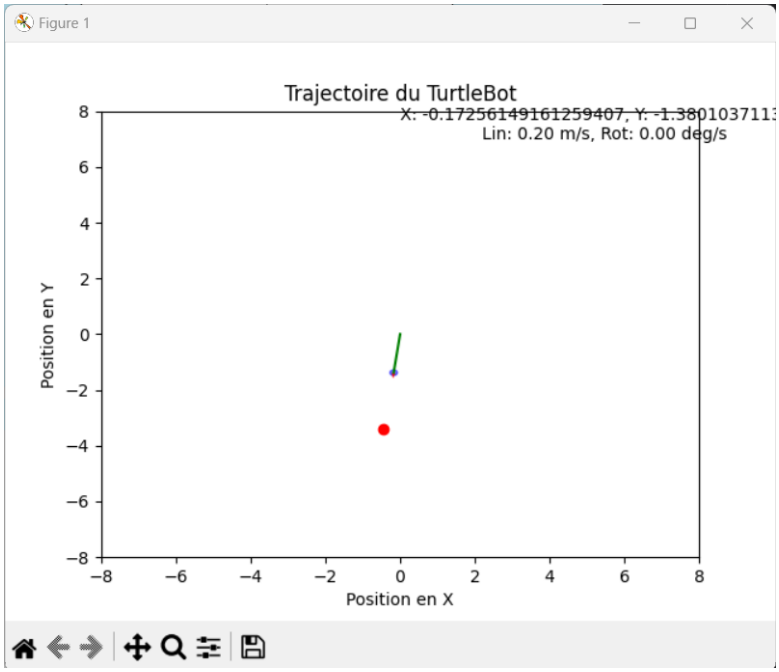
Les deux simulateurs utilisent un solveur (odeint pour Python) qui permet de résoudre numériquement le système cinématique (1) et qui prend en argument la fonction décrivant le modèle (qui renvoie la variation du vecteur de position et d'orientation du robot par rapport au temps), les arguments de cette fonction (dans notre cas les commandes de vitesse) ainsi que les conditions initiales et le vecteur temporel pour générer la trajectoire.

La boucle de simulation discrétise le temps avec un pas Δt. Pour chaque itération dans la boucle principale de simulation, la fonction solveur est appelée avec les nouvelles commandes du contrôleur et une nouvelle condition initiale ; dans notre cas, le vecteur de position et d'orientation précédent.

Les algorithmes de contrôle des phases de rotation et de translation, la machine à état et la structure du programme sont les mêmes pour les simulateurs et le nœud ROS. Cependant, la simulation nous dispense de toute utilisation des données de position enregistrées par le système Qualisys, et inversement, le nœud de contrôle réel nous dispense de l'utilisation d'un solveur.



(a) Phase de rotation



(b) Phase linéaire

FIGURE 2 – Graphique interactif du simulateur

0.2.2 Structure du nœud de contrôle réel

Le nœud ROS nommé "linear_control.py" permet donc de générer des commandes de vitesse en rotation et en translation trapézoïdale pour suivre et rejoindre des waypoints en ligne droite. La figure 4 présente la structure globale du programme. Le détail du fonctionnement des algorithmes de contrôle de la rotation, de la translation et de la machine à état est expliqué dans des sous-parties ultérieures.

Lors de son initialisation, le nœud s'abonne au topique 'qtm/turtle6dof_pose' sur lequel le nœud d'interfaçage 'qtm_interface.py' publie en temps réel les positions et l'orientation du centre du rigid body correspondant au robot. Les données sont publiées dans un message de type Pose. Le nœud initialise aussi un publieur qui envoie les commandes de vitesse sous forme de messages de type Twist() sur un topique de commande du TurtleBot.

Le programme vérifie en continu si le robot se trouve bien dans la zone calibrée avec la fonction 'acquisition(self, Pose)'. Si l'un des attributs des messages est un NaN, la fonction acquisition met à jour un indicateur d'urgence (emergency) et appelle une fonction 'tracking_lost()' qui publie une commande de marche arrière tant que emergency vaut True, et modifie le waypoint qui a entraîné la perte de contrôle.

Quand le drapeau emergency est à False, le nœud est dans son fonctionnement normal, le robot passe par les différentes étapes de la machine à état, et des commandes sont générées à partir des algorithmes de contrôle en rotation et en translation : 'rotate_tb()' et 'linear_tb()', qui utilisent des fonctions auxiliaires de calcul : 'ModuloByAngle()' et 'DistancePointToSegment()'.

Le code fonctionne tant que le nœud n'a pas été arrêté dans le terminal Linux grâce à une boucle sur le booléen 'rospy.is_shutdown()'.

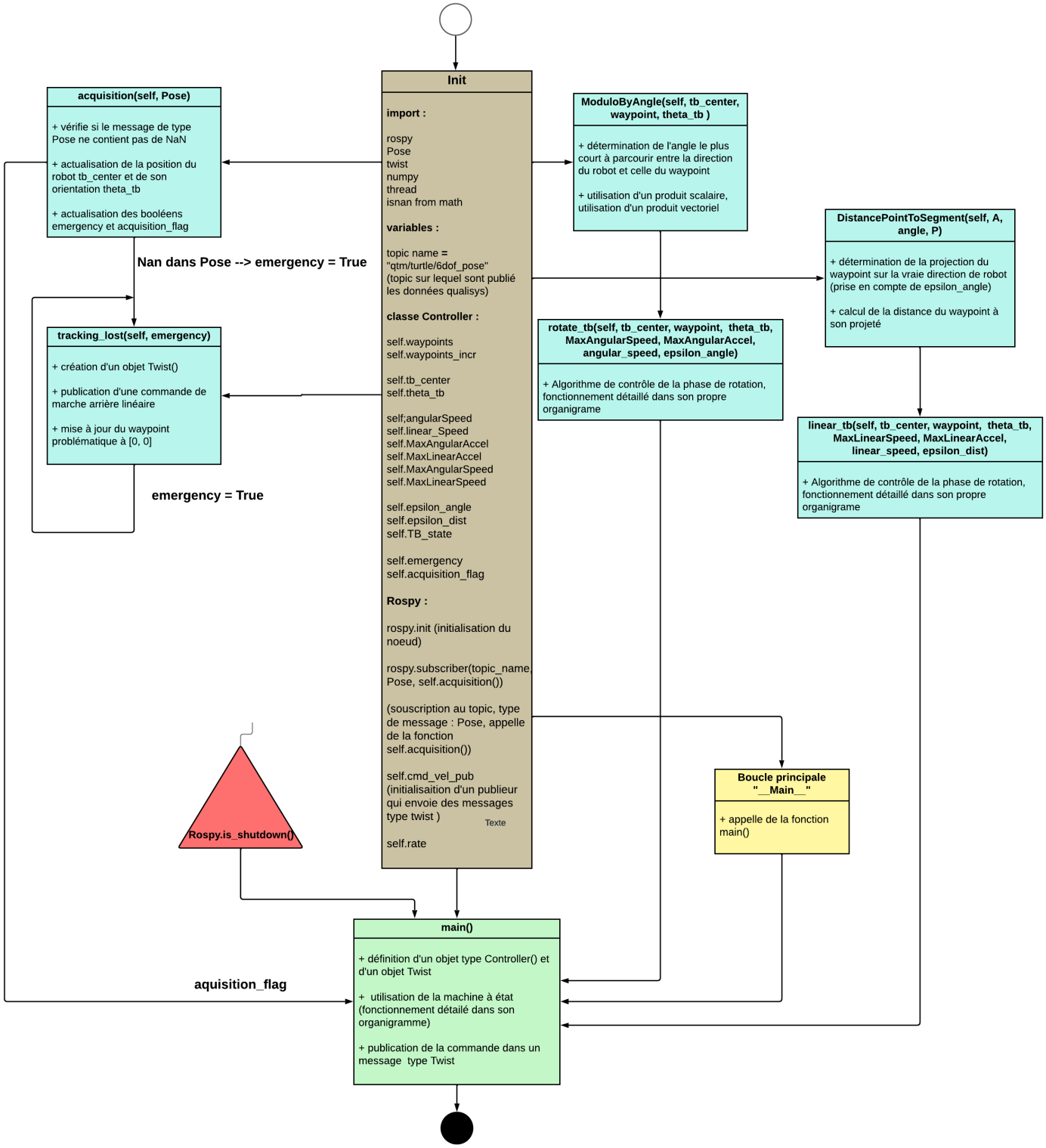


FIGURE 3 – Organigramme du nœud "linear_controler.py"

0.2.3 Fonctionnement des algorithmes de génération de commandes

Les algorithmes de génération de commandes de vitesse de rotation et de translation sont extrêmement proches et permettent de générer des consignes trapézoïdales en vitesse en envoyant des créneaux d'accélération. Les figures 5 et 6 sont les organigrammes de ces algorithmes.

L'algorithme 'rotate_tb()' prend en argument :

- le waypoint actuel
- la position du centre du robot
- son orientation
- la valeur d'accélération angulaire maximale
- la valeur de vitesse angulaire maximale
- un angle d'erreur acceptable
- la variable d'accélération angulaire courante

Il utilise la fonction 'ModuloByAngle()' pour déterminer le plus petit angle de rotation nécessaire pour que le waypoint soit dans la direction du robot.
Cette fonction calcule le vecteur directeur de la droite passant par le centre du robot et le waypoint $\overrightarrow{T_bW_p}$ ainsi que la direction actuelle du robot $\overrightarrow{T_b}$.
Elle détermine ensuite l'angle entre ces deux vecteurs à partir de la relation suivante :

$$\boxed{\theta_{\text{restant}} = \arccos\left(\frac{\overrightarrow{T_bW_p}.\overrightarrow{T_b}}{||\overrightarrow{T_bW_p}||.||\overrightarrow{T_b}||}\right)}$$

Finalement, on calcule le produit vectoriel entre ces deux vecteurs pour savoir de quel côté est le waypoint par rapport au robot. En fonction du signe du produit, la fonction renvoie $\pm\theta_{\text{restant}}$

Cette valeur de θ_{restant} est comparée avec un autre angle : θ_{stop} qui est l'équivalent d'une distance de freinage, mais pour une vitesse de rotation.

Pour la déterminer, on veut savoir combien de temps il faut pour que la vitesse angulaire passe d'une vitesse nulle à la vitesse V_{max}.

On a donc

$$t = \frac{V_{\text{max}}}{A_{\text{Max}}}$$

Or par intégration, on a aussi :

$$\theta_{\text{stop}} = \frac{1}{2}V_{\text{max}}t$$

Soit finalement :

$$\boxed{\theta_{\text{stop}} = \frac{V_{\text{max}}^2}{2 * A_{\text{Max}}}}$$

Cette valeur a ensuite été surdimensionnée pour s'assurer que le robot décélère assez pour arriver à une vitesse quasi nulle sur la direction souhaitée. Le même procédé est utilisé pour l'algorithme de translation.

θ_{stop} et θ_{remain} sont ensuite comparés dans l'algorithme : si l'angle restant est supérieur à 0 alors, il faut tourner dans le sens trigonométrique. Si ce n'est pas le cas, c'est anormal, donc on s'arrête en freinant.

Ensuite, si l'angle restant est supérieur à l'angle de freinage, on accélère si la vitesse maximale n'est pas déjà atteinte, autrement, on maintient la vitesse. Si l'angle restant est supérieur à l'angle de freinage, alors il faut freiner pour s'arrêter avec la bonne orientation.

Le cas "angle restant inférieur à 0" est parfaitement symétrique, mais il faut prendre les inégalités de conditions en valeur absolue et inverser les signes de l'accélération.
La phase de rotation se termine quand l'angle restant est inférieur en valeur absolue à notre tolérance angulaire 'epsilon_angle', le drapeau bool_val passe alors à True.

Pour l'algorithme de génération d'accélérations linéaire la structure est exactement la même, mais les éléments de comparaison sont différents. Tout d'abord, une fonction 'DistancePointToSegment()' est appelée pour calculer le projeté de la position du Waypoint sur l'axe de direction du robot (qui ne passe pas forcément par le waypoint à cause de la tolérance angulaire).

Ce calcul est effectué à partir d'un produit scalaire entre le vecteur direction et le vecteur de l'axe passant par le waypoint et le centre du robot comme pour 'ModuloByAngle()'.

Le point projeté devient alors notre nouveau Waypoint. La distance restante à parcourir correspond à la distance euclidienne entre le centre du robot et ce point projeté.

La suite de l'algorithme est similaire au précédent. La phase de translation est terminée quand la distance restante est inférieure en valeur absolue à la tolérance linéaire 'epsilon_dist'.

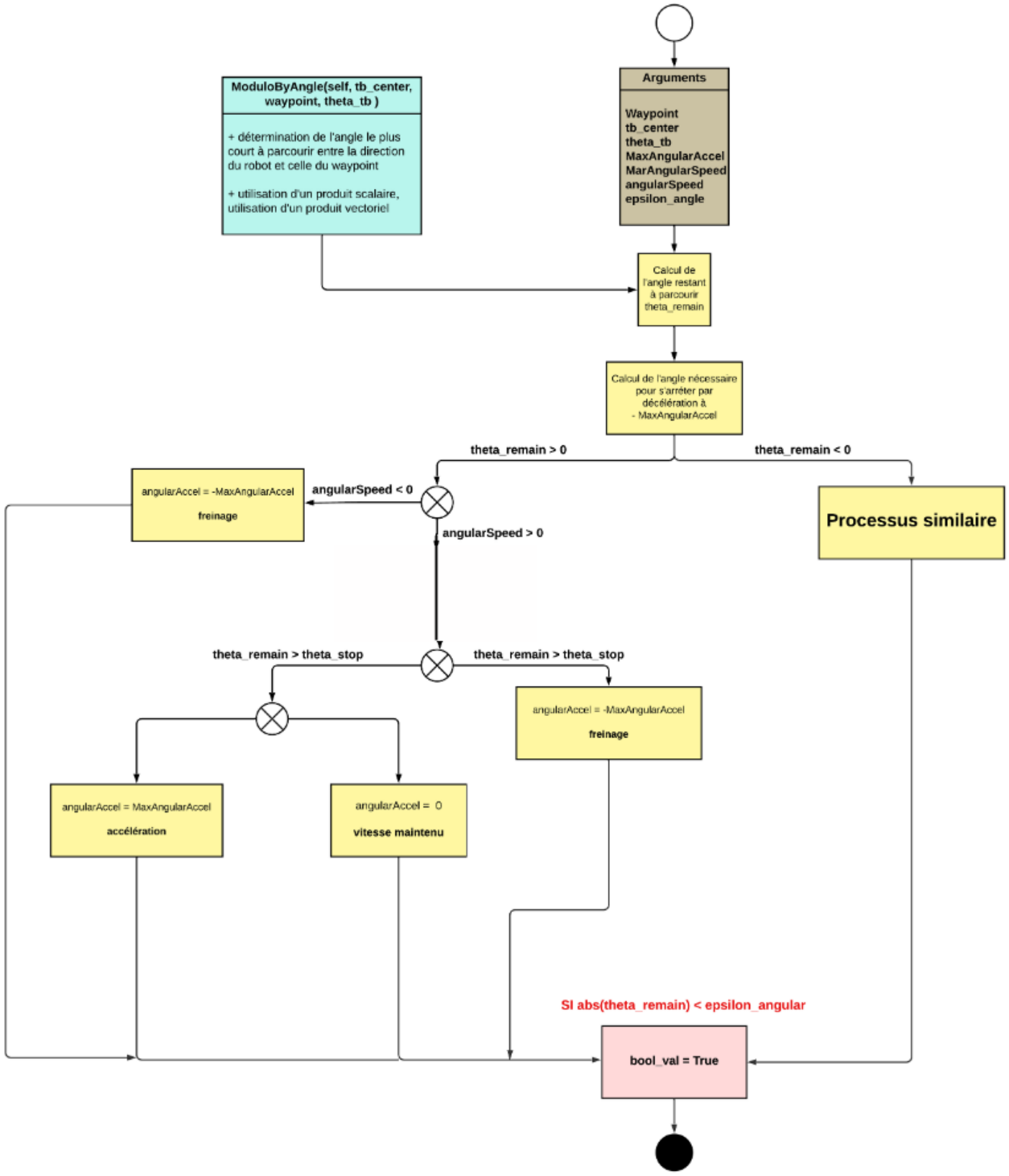


FIGURE 4 – Organigramme de l'algorithme de contrôle angulaire

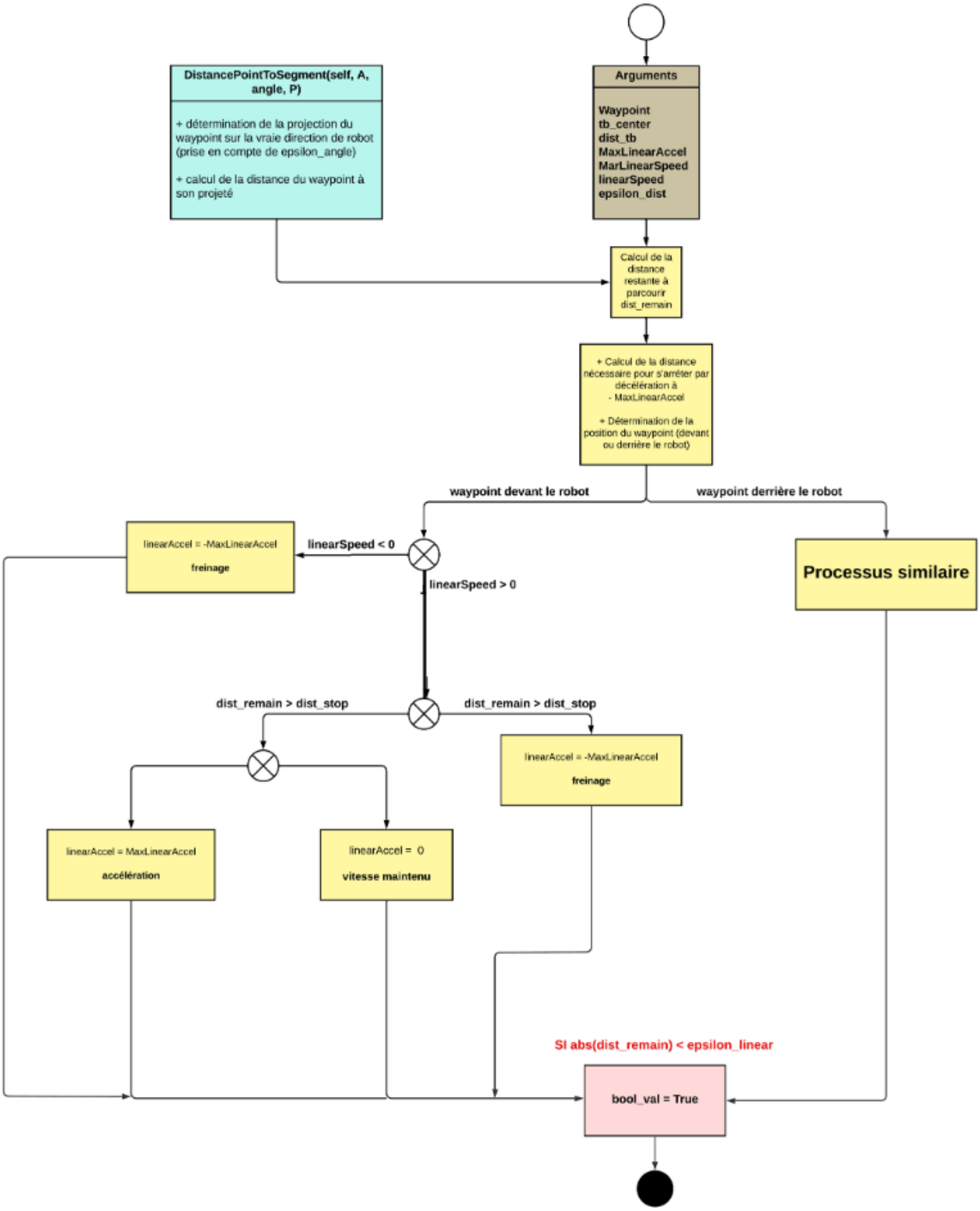


FIGURE 5 – Organigramme de l'algorithme de contrôle linéaire

0.2.4 Description de la machine à état

La machine à état est constituée de trois états :

- Un état correspondant à la phase de rotation (état initial) qui récupère la variable angularAccel et le drapeau de fin de phase bool_val à partir de l'algorithme rotate_tb() puis calcule la commande de vitesse angulaire par intégration numérique (formule de rectangle à gauche d'une largeur correspondant au pas de temps de $\frac{1}{100}$ s).
- Un état correspondant à la phase de translation. Elle fonctionne de la même façon que la phase précédente
- Un état idle qui permet de passer au prochain waypoint à la fin de la phase de translation. Si le robot est passé par tous les Waypoints, le robot boucle dans cet état en attendant l'arrêt du nœud par l'utilisateur.

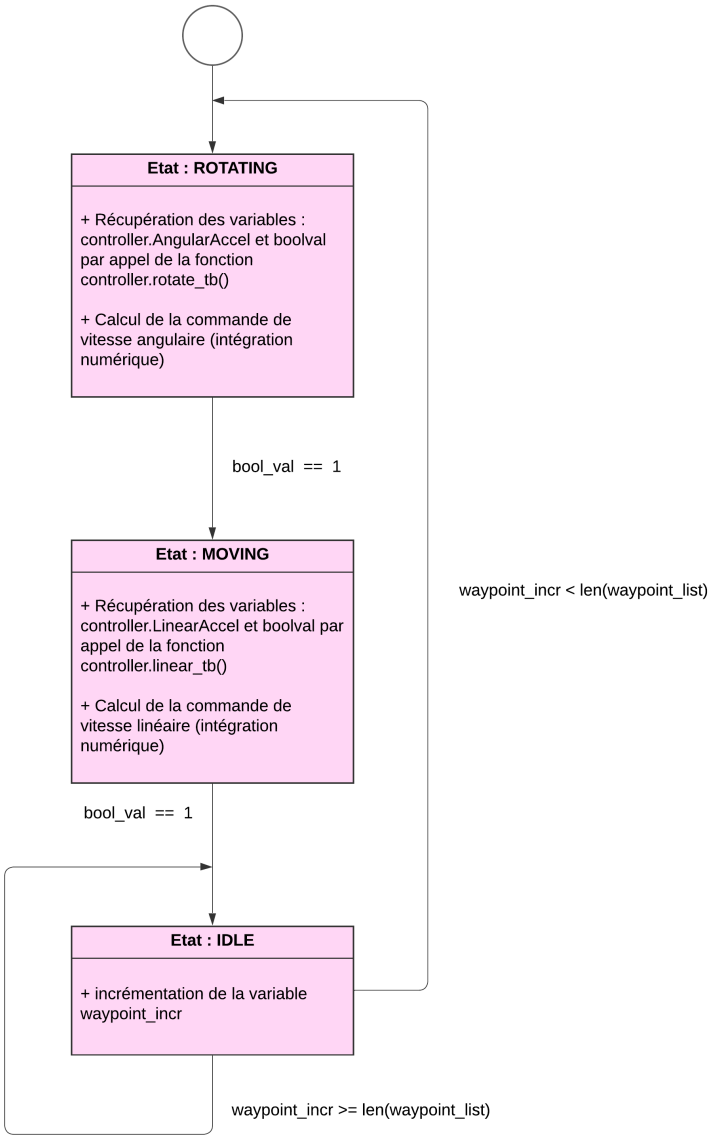


FIGURE 6 – Organigramme de la machine à état

0.2.5 Limites de la stratégie de commande

Cette méthode d'asservissement a pour avantage d'être simple à mettre en œuvre, le découplage des degrés de liberté de translation sur le plan et de rotation facilite grandement le suivi de la trajectoire demandé, et la génération de consigne trapézoïdale permet d'avoir des freinages et des accélérations fluides.

Cependant, le robot a un déplacement haché et lent quand de nombreux changements de direction sont demandés. Le fonctionnement est robuste, mais pour augmenter la rapidité et la fluidité du déplacement, la génération d'une trajectoire curviligne et la création d'un nouveau contrôleur pour la suivre sont nécessaires.

Enfin, les deux algorithmes des phases de rotations et de translations auraient pu être remplacés par un unique PID. L'utilisation d'algorithmes permet d'éviter une phase de réglage des gains par tâtonnement, mais peut s'avérer fastidieuse à comprendre et à programmer.

0.3 Suivi d'un itinéraire curviligne

Pour créer des itinéraires curvilignes à partir d'une liste de points cibles (Waypoint), nous avons décidé d'interpoler entre chaque point un polynôme cubique (cubic spline interpolation). Cela permet d'avoir un chemin curviligne fluide (on expliquera pourquoi par la suite). Finalement, les itinéraires tracés et suivis par le premier contrôleur étaient le résultat d'une interpolation linéaire déguisée entre chaque waypoint. Nous allons maintenant arrêter de découpler les degrés de libertés de notre robot pour pouvoir passer par les waypoints de manière beaucoup plus fluide et donc rapide.

L'avantage de la méthode d'interpolation spline est au niveau de sa complexité. En effet, effectuer une interpolation entre deux points ne nécessite pas un polynôme d'un degré important (il peut même être d'ordre 1 pour les splines linéaires), alors qu'une méthode d'interpolation globale nécessite un ordre très élevé pour trouver un unique polynôme qui passe par tous les points demandés.

0.3.1 Théorie de l'interpolation par spline cubique

Pour bien comprendre le mécanisme d'interpolation par spline cubique, nous allons d'abord détailler la méthode mathématique pour arriver au système linéaire d'une interpolation quadratique entre plusieurs points sur un plan (x0y).

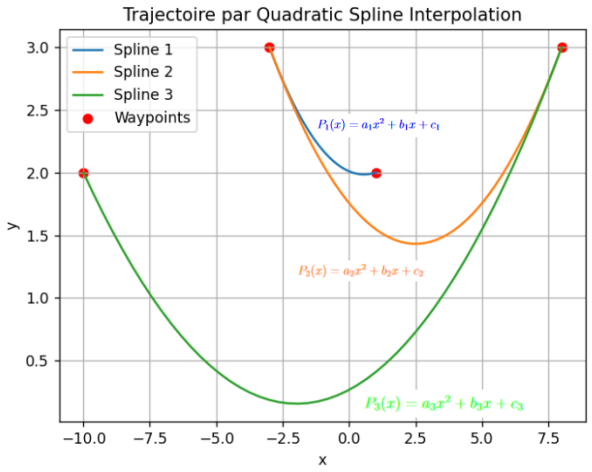
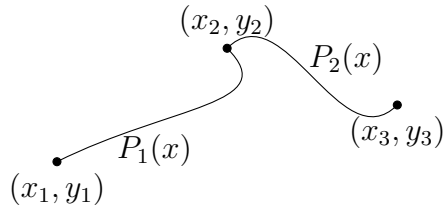


FIGURE 7 – Exemple d'interpolation par spline quadratique

Un polynôme du second degré a une forme générale : $P_i(x) = a_ix^2 + b_ix + c_i$. Pour chaque spline, l'objectif est de trouver les trois coefficients a_i, b_i et c_i. Il y a trois inconnues par spline.
Il faut donc trouver trois relations par spline pour déterminer les paramètres. Autrement dit, pour n+1 waypoints, il faut n splines et donc résoudre un système de 3n équations.

Prenons l'exemple simple suivant :



L'évaluation de chaque spline à leurs extrémités permet d'avoir deux équations par spline. De plus, on peut ajouter une contrainte de continuité pour les points à l'intérieur du chemin. Pour cela, les dérivées de deux splines accolées doivent être égales en leur point commun.

Dans notre cas, on aurait :

$$\begin{cases} P_1(x_1) = y_1, & P_2(x_2) = y_2 \\ P_2(x_2) = y_2, & P_2(x_3) = y_3 \\ \\ \frac{dP_1(x_2)}{dx} = \frac{dP_2(x_2)}{dx} \end{cases}$$

Pour les points aux extrémités du chemin, et les splines les reliant, une dernière contrainte peut être ajoutée pour rendre le problème solvable. On peut par exemple demander une certaine pente au point (x_1, y_1) ou imposer une dérivée seconde nulle par exemple.
En posant les différentes équations, on arrive à un système linéaire d'équation qui permet de trouver le vecteur formé des paramètres des splines d'ordre 2. On peut le résoudre avec un algorithme de pivot de Gauss ou de factorisation LU par exemple.

La construction d'itinéraire par interpolation de spline d'ordre 3 fonctionne de la même façon, mais quatre équations sont maintenant nécessaires pour trouver les coefficients d'une spline. Il faut

donc utiliser pour les points internes une égalité des dérivées secondes, ce qui permet en plus de la continuité d'une interpolation par spline quadratique d'avoir une trajectoire décorrélée du temps beaucoup plus fluide.

Pour réaliser les interpolations par spline cubique dans le constructeur de trajectoire, nous avons utilisé le module SciPy et n'avons pas nous-mêmes créé une fonction qui le faisait. Le problème, c'est que cette fonction nous imposait d'avoir une liste de waypoints ordonnés par coordonnées sur x croissantes. Nous sommes donc passés par une variable intermédiaire en interpolation $t \in [0, 1]$ pour faire une interpolation avec les coordonnées x des waypoints et une interpolation avec les coordonnées y des waypoints. Ensuite, l'affichage de $y(t)$ en fonction de $x(t)$ permet d'avoir le chemin voulu. Le graphique suivant clarifie le fonctionnement de notre générateur de chemin :

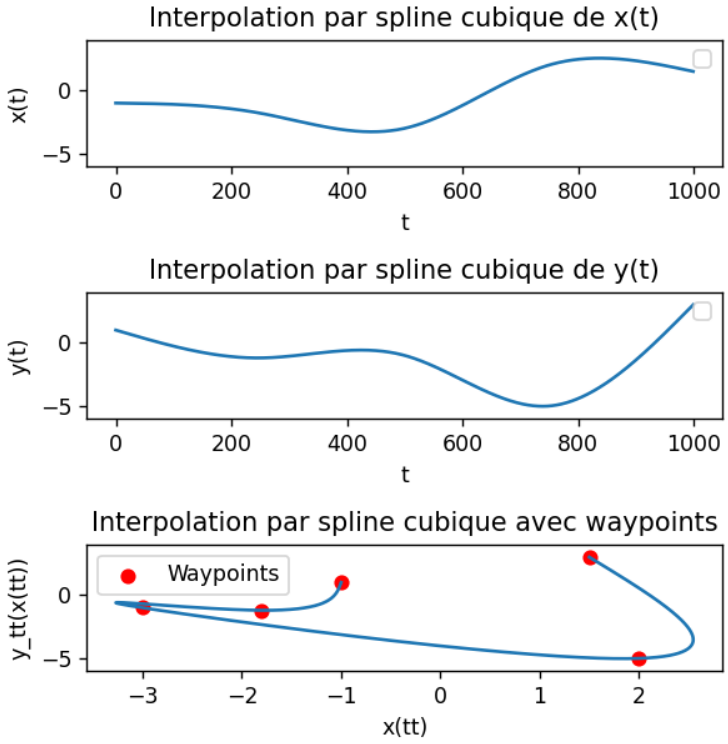


FIGURE 8 – Génération d'un itinéraire en utilisant un paramètre intermédiaire

0.3.2 Description de l'algorithme

Pour suivre une trajectoire de manière fluide, nous avons utilisé l'algorithme "pure pursuit" qui est un algorithme de suivi de trajectoire. Il prend en entrée une liste de points discrets qui définissent une trajectoire et une vitesse linéaire constante. En sortie, l'algorithme nous retourne la vitesse angulaire qui nous permettra de contrôler le turtleBot.

Connaissant la position et l'orientation courantes du TurtleBot, l'algorithme calcule un point cible se trouvant sur la trajectoire à une distance d_l (distance de prévision). Ensuite, il détermine l'arc de cercle optimal passant par le point courant et le point cible par rapport à son orientation actuelle. Enfin, il calcule la vitesse de rotation appropriée pour parcourir l'arc de cercle.

On considère un repère orthonormé (O, i, j) dans le plan du robot. La position du centre du robot est définie par le point de coordonnées (R_x, R_y), et P_d est un point de la trajectoire de coordonnées (g_x, g_y) se trouvant à la distance d_l du robot.

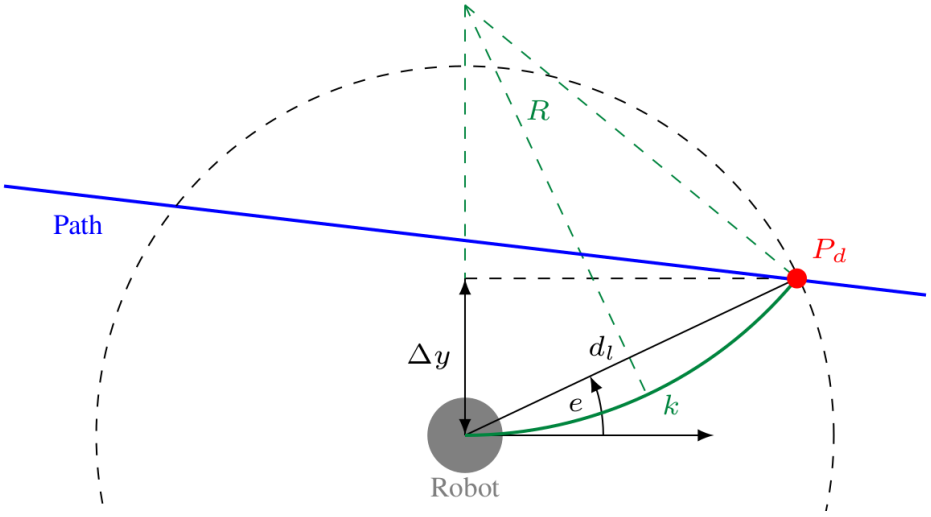


FIGURE 9 – Pure pursuit

On définit d_l la distance euclidienne entre le point courant et le point cible, ω la vitesse angulaire du robot, et α l'angle de cap du robot par rapport à la droite (d_l).

$$r = \frac{d_l^2}{2 \times |g_y|}$$

$$\alpha = \frac{1}{r} = \frac{2 \times |g_y|}{L^2}$$

$$\omega = \frac{2 \times V \times |g_y|}{d_l^2}$$

0.3.3 Schéma blocs de l'algorithme

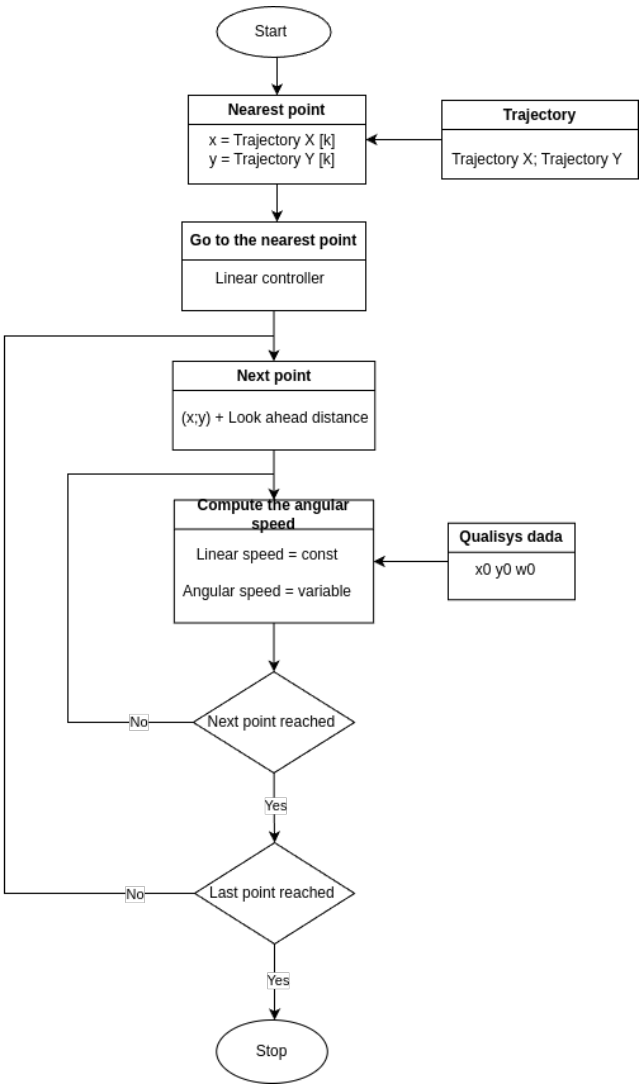


FIGURE 10 – Pure pursuit algorithm

L'algorithme prend en entrée la trajectoire, puis calcule le point le plus proche de cette trajectoire par rapport à sa position courante. Pour déterminer ce point, nous calculons la distance euclidienne entre tous les points discrets de la trajectoire et le centre du robot, puis sélectionnons la plus petite distance. Une fois le point le plus proche de la trajectoire déterminé, le robot se positionne dessus en utilisant un algorithme d'asservissement en position, qui est dans notre cas un contrôleur PID.

Après que le robot s'est positionné sur un point de la trajectoire, l'algorithme calcule ensuite le point cible sur la trajectoire qui se trouve à une distance d_l (distance de prévision). Cette distance d_l est définie comme constante dans le programme.

En fonction de l'angle courant du robot, l'algorithme calcule le rayon de courbure optimal en appliquant la formule du rayon de courbure définie ci-dessus, puis détermine la vitesse angulaire correspondante pour suivre la courbure calculée.

À chaque calcul du point cible, l'algorithme vérifie si le prochain point est le dernier de la tra-

jectoire. Si tel est le cas, le robot s'arrête dès qu'il atteint ce point. Sinon, il se rend à ce point et recalculera le prochain point de la trajectoire.