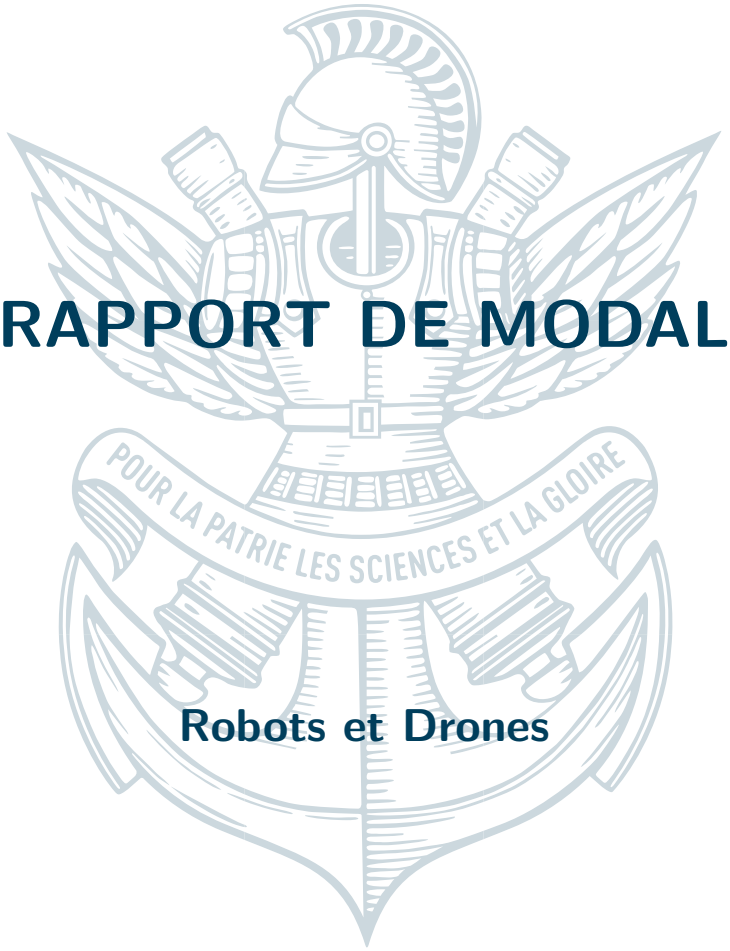

RAPPORT DE MODAL

Robots et Drones

Février 2025

Roméo Nazaret

TABLE DES MATIÈRES

Mon dépôt GitHub

1
UN PREMIER ALGORITHME DE NAVIGATION : LE SUIVI DE MUR

1.1 Présentation générale de l'algorithme

L'algorithme présenté dans le TP est relativement simple : suivre le mur de gauche. Pour ce faire, il suffit de calculer la distance de la voiture au mur de gauche, ce qui peut être fait facilement avec un peu de trigonométrie et les données du Lidar. Ensuite, l'erreur est passé à contrôleur PID qui contrôle la voiture par un asservissement de l'angle de braquage. La vitesse quand à elle s'adapte selon la valeur de l'angle de braquage.

1.2 Choix d'implémentation

Après avoir fait fonctionner le suivi de mur et l'avoir tester en simulation et sur la F1tenth je me suis rendu compte que ce dernier était relativement sensible à la forme du circuit notamment à haute vitesse. Un virage un peu trop serré et la voiture part dans le mur. Pour remédier à cela j'ai décidé d'implémenter un suivi reposant sur les deux bords du circuit, on cherche toujours à positionner la voiture au centre de la piste. Ainsi quelque soit la largeur du circuit, la distance désirée au bord s'adapte. Cela peut être très pratique dans le cas de circuit dont la largeur varie. Voici un pseudo code très simple du fonctionnement de l'algorithme :

Algorithm 1 Wall follow :

```
function LIDAR_CALLBACK() :
    dg ← calculer_distance_mur_gauche()
    dr ← calculer_distance_mur_droite()
    expected_dist ← (dg + dr)/2          ▷ On veut positionner la voiture au centre de la piste
    error_g ← dg − expected_dist
    error_r ← dr − expected_dist
    if abs(error_g) > error_d then       ▷ On corrige toujours l'écart le plus grand
        pid(−error_g)    ▷ On apelle le controlleur PID, le - vient juste de la convention pour les angles
    else
        pid(error_d)
```

Malgré son apparente simplicité, l'algorithme est très performant en pratique au moins tant qu'il n'y a pas d'aléa sur le circuit. En optimisant les paramètres du PID et en poussant la vitesse maximale jusqu'à $7m/s$. L'algorithme parvient à réaliser des temps d'environ $17.5s$ sur le circuit *track.yaml* (Voir les vidéos associées au *follow_the_wall* sur le github).

2
UN ALGORITHME SIMPLE QUI PREND EN COMPTE L'ENVIRONNEMENT : REACTIVE GAP FOLLOW

2.1 Présentation générale de l'algorithme

L'algorithme présenté dans le TD fonctionne de manière générale de la façon suivante :

Les données du LiDar sont analysées pour déterminer un espace suffisant où la voiture pourrait s'embarquer. Voici le pseudo code de l'algorithme implémenté pour réaliser cela :

Algorithm 2 Reactive gap follow :

1: **function** LIDAR_CALLBACK($lidar\_data$) :
2: $preprocess(lidar\_data)$ ▷ On ne garde que les données dans un cône devant la voiture, 90°par exemple
3: $remove\_close\_points(lidar\_data)$ ▷ On met à zéro toutes les valeurs du Lidar sous un certain seuil
4: $safety\_bubble(lidar\_data)$ ▷ On met à zéro les valeurs autour du point le plus proche
5: $direction \leftarrow find\_center\_biggest\_gap(lidar\_data)$ ▷ On cherche le milieu de la plus longue suite non nulle dans les données
6: $go\_to(direction)$ ▷ On publie les messages nécessaires pour suivre la direction trouvée

2.2 Choix d'implémentation

L'implémentation est plutôt directe à partir du pseudo-code. Les seuls éléments que j'ai pu apporter ici sont les valeurs du seuil et de la safety bubble. L'obtention de ces valeurs est majoritairement empirique (évidement, il faut que le seuil soit inférieur à la largeur de la piste soit $1.8m$ sur $track.yaml$ et plutôt $80cm$ en salle de TP). En testant de nombreuses combinaisons grâce au simulateur, je suis venu à la conclusion que la bonne valeur de seuil était $1m$ (il faudrait évidement la diminuer sur le circuit réel), et que la safety buble n'apportait pas de grande amélioration. Ainsi même si cette dernière est implémentée, elle n'est jamais appelée par le code (cf github).

Cette algorithme de navigation est intéressant car il demande peu de temps de calcul et permet de prendre en compte la présence d'autres obstacles comme d'autres voitures par exemple. Ce point est d'autant plus intéressant que l'algorithme précédant ($follow\_the\_wall$) ne le permet pas. Il n'est donc pas possible de rouler dans un environnement qui ne se se compose pas d'un simple circuit. Cela est par contre possible avec ce nouvel algorithme.

Nous allons maintenant nous intéresser à des algorithmes qui ne se contente pas uniquement de réagir à l'environnement qui entoure la voiture mais qui sont également capables de prévoir une trajectoire.

3
UN ALGORITHME DE SUIVI DE TRAJECTOIRE : PURE PURSUIT

Afin de pouvoir utiliser un algorithme capable de prédire une trajectoire, il est nécessaire d'être en mesure de suivre une trajectoire prédéfinie. C'est précisément ce que fait le *pure_pursuit*.

3.1 Présentation générale de l'algorithme

Cette algorithme, plus complexe que les précédents, nécessite de récupérer plus d'information que les simples données du LiDar. En effet, nous avons besoin de pouvoir nous localiser par rapport à la trajectoire. Ainsi il est nécessaire de connaître la position précise de la voiture sur le circuit, ce qui peut être fait à l'aide d'un *particlefilter* déjà implémenté dans *ROS*. Ce dernier se base sur les informations du Lidar, d'une carte de l'environnement préchargée ainsi que des déplacements de la voiture pour estimer la position de cette dernière.

Voici le pseudo code de l'algorithme qui explique son fonctionnement :

Algorithm 3 Pure pursuit :

Initialisation :
waypoint_list ← $parse\_waypoints()$
function POSE_CALLBACK($position, orientation$) :
    $target \leftarrow find\_waypoint()$
    $goal \leftarrow world\_frame2car\_frame(target)$ ▷ Converti la position de la target du rapport du circuit à celui de la voiture
    $angle \leftarrow calculate\_steering\_angle(goal, position, orientation)$
    $speed \leftarrow calculate\_speed(angle)$
    $publish(speed, angle)$

L'algorithme est ici décrit de façon très succincte, détaillons un peu le fonctionnement des 2 méthodes les plus importantes : $find\_waypoints$ et $calculate\_steering\_angle$:

- $find\_waypoints$: L'idée ici est de déterminer le point de la trajectoire à distance L (choisie arbitrairement) de la voiture. Pour ce faire :
  - Nous trouvons le waypoint le plus éloigné à distance inférieur à L, et le plus proche à distance supérieure à L.
  - Puis nous cherchons sur la droite joignant ces deux points l'intersection avec le cercle de rayon L centré sur la voiture.
  - Il suffit alors de retourner ce point. Nous pouvons noter que si seulement un des point précédant existe alors c'est lui que nous retournons.

  Il est à noter ici qu'il y a pas nécessairement un unique point sur la trajectoire. Pour s'assurer de produire un résultat, il ne faut considérer les waypoints qui sont devant la voiture. (On peut autoriser dans certains cas, des waypoints légèrement derrière si le circuit contient des virages très serrés).
- *calculate_steering_angle* : Le calcul de l'angle de braquage provient de considération géométrique. Nous cherchons à déterminer la courbure du cercle passant par la voiture et le *goal*, tel que son centre soit situé sur l'abscisse du repère de la voiture.

3.2 Choix d'implémentation

Comme précédemment, les TD étant plutôt guidé et l'algorithme relativement simple, l'implémentation s'est faite de façon plus ou moins directe.

Il y a tout de même quelques points qui ont retenu mon attention :

- La méthode *parse_waypoints* a pour objectif de charger les points de passage de la trajectoire stockés dans un fichier csv. En plus de simplement lire le fichier, la fonction s'assure que les points chargés ne sont pas inutilement proche : cela ne sert à rien d'avoir un point chaque millimètre. Ainsi *parse_waypoints* retournera une liste de point de passage espacer d'une certaine distance fixée arbitrairement à $\frac{L}{3}$ afin qu'il y ait toujours un certain nombre de point à distance inférieur à L de la voiture.
- La méthode *isAhead* : comment déterminer si un point est bien devant la voiture ? Il suffit de calculer la produit scalaire, entre le vecteur $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ et le vecteur partant de la voiture vers le point ! Si ce dernier est positif alors le point est devant. Si l'on souhaite autoriser une certaine tolérance, on peut renormaliser le vecteur entre la voiture et sa cible (diviser par L). On peut alors fixer un seuil de tolérance à -0.2 par exemple.
- Optimiser la recherche des waypoints : Même si dans son utilisation actuelle l'algorithme ne présente aucun problème de performance. On pourrait imaginer qu'avec des cartes gigantesques, *waypoint_list* soit gigantesque également. Ainsi la parcourir intégralement pour déterminer les points de passage intéressants semble compliqué...

  Solution : dans notre cas précis, les points enregistrés par le *waypoint_logger* (algorithme fourni dans le TD) sont dans l'ordre de la trajectoire. Ainsi en stockant l'index du dernier point trouvé, on peut à chaque tour commencer la recherche à partir de ce point (entre deux appels consécutifs, la voiture ne s'est que très peu déplacée).

  Dans le cas où les points ne seraient pas enregistré dans l'ordre, on pourrait imaginer clusteriser ces derniers afin d'effectuer une première recherche parmi les clusters avant de s'intéresser aux points à l'intérieur. Cette idée n'a pas été implémentée directement dans le *pure_pursuit* mais sera utile pour le RRT^*.

4
UN ALGORITHME PLUS COMPLET DE PLANIFICATION DE TRAJECTOIRE : RRT/RRT*

Nous venons de nous intéresser à un algorithme capable de suivre un chemin établi. Pour construire ce chemin deux approches sont possibles :

- Enregistrer une trajectoire pendant que la voiture se déplace en étant pilotée par un autre algorithme ou par un pilote.
- Générer cette trajectoire en se basant sur les données du Lidar ou bien d'une carte préenregistrée.

Les algorithmes *RRT* et *RRT** rentrent dans la seconde catégorie.

4.1 Présentation générale de l'algorithme : RRT vs RRT*

Intéressons nous directement au fonctionnement de ces deux algorithmes. Commençons par le pseudo-code du *RRT* :

Algorithm 4 RRT :

Initialisation :
occupancy_grid ▷ une grille représentant la carte. Les cases de la grille contiennent une valeur entre 0 (vide) et 100 (mur)
tree ▷ un arbre contenant comme unique nœud le point de départ de la recherche.
while 1 : **do** ▷ Nous allons construire un arbre aléatoire pour atteindre la position désirée, matérialisée par le point target.
    $sample\_point \leftarrow sample()$ ▷ générer un point aléatoire sur la carte
    $nearest\_node \leftarrow nearest(sample\_point)$ ▷ renvoie le point de l'arbre le plus proche
    $new\_node \leftarrow steer(nearest)$ ▷ renvoie le point du segment [candidate, nearest] à distance L (arbitraire) de nearest.
    if $check\_collision(nearest\_node, new\_node)$ **then** ▷ Faux si un mur se trouve sur [candidate, nearest]
        $tree.add(new\_node)$
        if $is\_goal(new\_node)$ **then** ▷ Vrai si *new_node* est suffisamment proche de l'objectif
            $path \leftarrow reconstruct\_path(tree)$
            return path

En procédant comme décrit dans le pseudo code, l'algorithme va construire itérativement un arbre aléatoire ou chacun des nœuds est accessible depuis la racine (nous nous sommes assurés qu'il n'y avait pas de collisions avec aucun mur entre deux nœuds). Ainsi une fois avoir ajouté suffisamment de nœud, l'arbre devrait finir par contenir un chemin partant de la racine et allant jusqu'à l'objectif.

Cet algorithme est relativement efficace pour trouver des chemins et peut fonctionner directement sur la voiture pour éviter des obstacles, il suffit de construire la carte *occupancy_grid* avec le Lidar en temps réel (ce point sera présenté plus tard).

Cependant l'algorithme présente également quelques problèmes. Par exemple, une fois ajouté dans l'arbre, les nœuds et les arrêtes ne sont jamais modifiés. Ainsi si l'algorithme commence par explorer de façon hasardeuse, la trajectoire qui en ressortira sera hasardeuse elle aussi. Voici un exemple de trajectoire générée par le RRT : le résultat est loin d'être mauvais mais nous pouvons observer tout de même de fortes ruptures dans la trajectoire.

FIGURE 1 – Un exemple de trajectoire proposée par le RRT

Présentons maintenant le *RRT** :

Algorithm 5 RRT* :

Initialisation :
occupancy_grid ▷ une grille représentant la carte. Les cases de la grille contiennent une valeur entre 0 (vide) et 100 (mur)
tree ▷ un arbre contenant comme unique nœud le point de départ de la recherche.
while *stop_condition* **do** ▷ Nous allons construire un arbre aléatoire pour atteindre la position désirée, matérialisée par le point target.
    $sample\_point \leftarrow sample()$ ▷ générer un point aléatoire sur la carte
    $nearest\_node \leftarrow nearest(sample\_point)$ ▷ renvoie le point de l'arbre le plus proche
    $new\_node \leftarrow steer(nearest)$ ▷ renvoie le point du segment [candidate, nearest] à distance L (arbitraire) de nearest.
    if $check\_collision(nearest\_node, new\_node)$ **then** ▷ Faux si un mur se trouve sur [candidate, nearest]
        $tree.add(new\_node)$
        $cost[new\_node] \leftarrow cost[nearest\_node] + line\_cost(candidate, nearest)$
        $rewire(tree)$ ▷ on cherche à modifier l'arbre pour minimiser les coûts de chaque nœud
        if $is\_goal(new\_node)$ **then** ▷ Vrai si *new_node* est suffisamment proche de l'objectif
            $path \leftarrow reconstruct\_path(tree)$
        end if
    end if
end whilereturn path =0

Les modifications ont été affichées en rouge. On se rend ici compte que l'algorithme n'est pas très différent. Il ne présente pour ainsi dire qu'une méthode majeure de plus : la méthode *rewire*. C'est cette dernière qui est chargée de modifier la structure de l'arbre afin de minimiser le coût de la trajectoire de l'origine à chacun des nœuds. Contrairement à précédemment, l'arbre est sans cesse amélioré, ainsi même si l'on trouve une trajectoire valide au temps t, on devrait en trouver une encore meilleure en attendant plus longtemps. Nous ne sommes donc plus obligés de renvoyer le premier chemin mais nous pouvons attendre qu'une certaine condition d'optimalité soit vérifiée ou qu'un grand nombre d'itérations ait eu lieu.

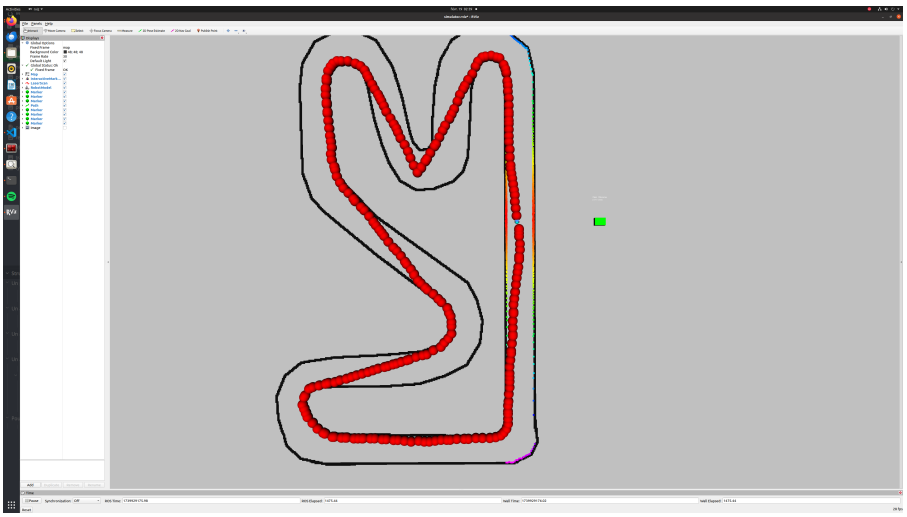

FIGURE 2 – Un exemple de trajectoire proposée par le RRT*

Rentrons maintenant un peu plus dans les détails et intéressons nous à l'implémentation des méthodes principales de l'algorithme.

4.2 CHOIX D'IMPLÉMENTATION ET OPTIMISATION

4.2.1 • STRUCTURE GÉNÉRALE

La structure générale est la suivante : Nous avons une classe *Node* qui nous permet de représenter les nœuds de l'arbre et de leur associer : une position réelle, une positon sur la grille, un parent, un coût, et un booléen indiquant si le nœud est la racine de l'arbre.

La classe *Node* contient deux autres attributs : *bubbles* et *neighbor* dont nous expliquerons l'utilité dans la partie suivante.

A cela s'ajoute une classe *RRT* qui contient l'ensemble des méthodes évoquées plus haut dans le pseudo-code ainsi que quelques autres pour faciliter le code. Cette classe contient également des attributs utiles au fonctionnement de l'algorithme, comme *waypoints* qui contient la liste des *target* que va viser le RRT ou L évoqué plus haut dans le pseudo code. La classe contient aussi 2 grilles : une première permanente qui correspond à la carte du circuit. Une seconde, calculée en temps réel qui correspond à ce que voit le Lidar et qui permet donc de réagir aux obstacles.

4.2.2 • FONCTIONNEMENT ET OPTIMISATION DES MÉTHODES

Afin de fournir une trajectoire proche de l'optimale, il est nécessaire de générer un grand nombre de nœuds dans l'arbre. Ce processus devant permettre d'éviter des obstacles, il faut qu'il soit très efficace pour que l'algorithme soit réactif. Nous allons maintenant nous intéresser aux méthodes principales ainsi qu'aux choix qui ont étés faits pour les optimiser :

- **sample** : la méthode est assez basique. Il suffit de générer aléatoirement un point sur la grille, ce qui est facilement faisable avec *random.randint*. Cependant, la taille de la grille ne correspond pas nécessairement à la taille du circuit. Or si la grille est bien plus grande que le circuit, la plupart des points seront générés en dehors du circuit, ce qui entraînera nécessairement un grand nombre de collision et donc un grand nombre de rejet. De plus les trajectoires proposées auront tendance à se cantonner aux bords extérieurs du circuit. A l'inverse, si la taille de la grille se limite à celle du circuit, très peu de point seront générés sur les bords du circuit et il peut être difficile de progresser dans les coins de la piste par exemple.

Solution : parcourir la carte à l'initialisation pour récupérer les bords du circuit : *find_map_corner* puis ajouter une petite marge sur chaque côté. Il s'agit des attributs $gauche, droite, haut, bas$ de la classe RRT.
Pour optimiser la recherche, nous pouvons aussi décidé de choisir régulièrement la *target* comme point. Ainsi régulièrement, l'arbre évoluera vers l'objectif. Cette vitesse est caractérisé par le paramètre $TEMPERATURE$. Une température de 0 entraîne toujours le choix de *target* tandis qu'une température de 1 entraîne un choix complètement aléatoire. Une fois un premier chemin trouvé, il n'est plus nécessaire de continuer à privilégier autant la *target*, on peut donc augmenter la température.

- **line_cost** : Cette méthode renvoie le coût pour passer d'un nœud à un autre. Dans un premier nous pouvons considérer ce coût comme la distance entre ces deux points que l'on peut facilement calculer. Cependant, à cause de la méthode *rewire* que nous décrirons juste après, il est nécessaire de calculer le coût entre deux nœuds voisins très régulièrement. Ainsi plutôt que de réaliser plusieurs fois ce calcul pour une même paire de voisins, la donnée de la distance entre deux voisins est stockée dans l'attribut *neighbor* de la classe de *Node* qui contient une liste de couple (Nœud, distance), qu'il suffit alors de retourner.

- **rewire** : Il s'agit de la méthode principale qui permet de distinguer le RRT^* au RRT. Pour réagencer l'arbre, nous allons parcourir l'ensemble des voisins du nœud (nœud à une distance inférieur à *neigbor_dist*) que nous venons d'ajouter et pour chacun de ces voisins, il faut vérifier si le nouveau nœud est un meilleur parent. Dans ce cas, on modifie le parent du voisin. Un meilleur parent est désigné si le coût du noeud diminue en changeant de parent. Puis si un noeud a changé de parent, son coût a été modifié et nous pouvons alors essayer de réarranger l'arbre autour de ce dernier également. Etc... J'ai donc décidé d'implémenter *rewire* de façon récursive, cependant pour éviter de "réarranger l'arbre à l'infini", j'ai ajouté une profondeur maximal matérialisé par le paramètre $DEPTH$. C'est ce processus de réarrangement emboîté qui nécessite de très nombreux appels à *line_cost* et qui a nécessité l'optimisation précédente.

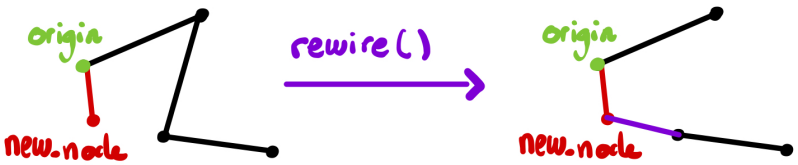

FIGURE 3 – Principe de fonctionnement de la méthode rewire

- **near** : Il s'agit de la méthode qui étant donné un noeud N, renvoie son voisinage. Une implémentation naïve serait de parcourir l'ensemble des nœuds de l'arbre et de regarder si la distance avec N est inférieure à *neighbor_dist*. Cela fonctionne évidemment mais peut être très coûteux si nous commençons à avoir beaucoup de noeud. Pour donner un ordre de grandeur, nous créons environ 6000 nœuds par parcours. Cela fait donc un total de $6000 * 5999/2 = 17997000$ calculs de distance. C'est malheureusement trop coûteux.
Solution : Partitionnons la grille en disque de rayon *neighbor_dist* disposés sur les intersections d'une grille carrée de maille *neighbor_dist*. Pour chaque noeud, gardons en mémoire la liste des disques auxquels il appartient dans l'attribut : *bubble*. Pour chaque disque, gardons en mémoire l'ensemble des nœuds qu'il contient et stockons tous ces disques dans l'attribut *bubbles_list* de RRT. Ainsi pour déterminer le voisinage d'un noeud, il suffit de regarder dans les nœuds situés dans les 4 disques voisins !

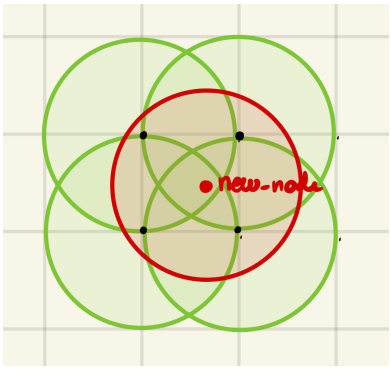

FIGURE 4 – Principe de fonctionnement de la recherche parmi les 4 disques voisins

- **nearest** : Comme présenté dans le pseudo code, *nearest* renvoie le noeud de l'arbre le plus proche d'un point donné. Comme précédemment, une implémentation naïve pourrait être de parcourir l'ensemble des noeuds pour trouver le plus proche. A nouveau on se retrouve avec 17997000 calculs de distance. C'est encore trop coûteux.

  Solution Profitons du découpage de la grille en disque. En effet, nous pouvons commencé par parcourir les disques de façon concentrique jusqu'à trouver un disque non vide D. Ensuite il suffit de calculer les distances au point contenu dans des disques à distance inférieur à D. Notons que lorsque le nombre de point est petit, il est plus efficace de parcourir directement la grille. Mais dès que la grille commence à être remplie, la probabilité que le disque dans lequel se trouve le point aléatoire est élevée, et il ne faudra alors parcourir uniquement les points de ce disque, c'est beaucoup plus efficace !

- **scan_callback** : Il s'agit de la méthode qui construit la carte local en fonction des données du Lidar pour prendre en compte les obstacles et les variations de l'environnement. Afin de ne pas essayer de passer au travers des obstacles, ces derniers sont épaissis d'un certain rayon proportionnel à la taille de la voiture.

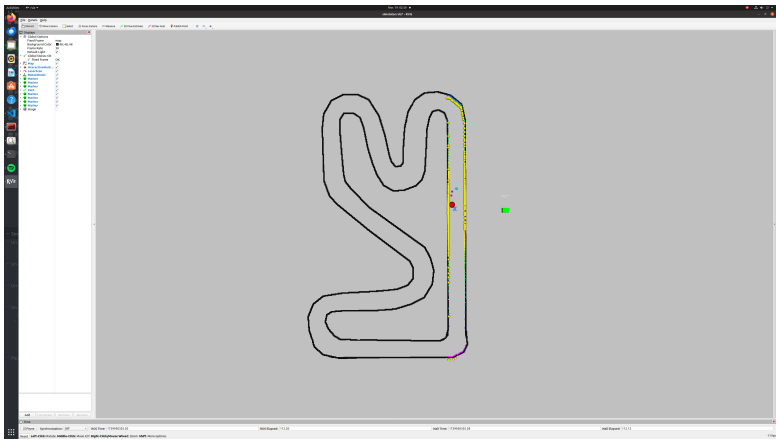

FIGURE 5 – Simulateur : les markers jaunes représentent les obstacles vus au LiDAR

- **check_collision** : Il s'agit de la méthode qui détecte si deux points peuvent ou non être reliés sans croiser d'obstacles. Pour une implémentation efficace, j'ai repris l'algorithme de tracé de segment de Bresenham.

4.3 Améliorer les trajectoires

Nous avons jusqu'à présent cherché à optimiser chacune de nos méthodes pour minimiser le temps de chaque itération et ainsi avoir plus de chance de converger vers une solution optimale ou presque.

Une autre vision pourrait être d'améliorer la qualité des solutions trouvées à chaque itération. Pour cela, j'ai trouvé essentiellement une piste : ajouter une heuristique à *line_cost*.

- **line_cost** : Il est clair que les trajectoires rectilignes sont préférables aux trajectoires qui zigzaguent. Ainsi j'ai décidé de valoriser les points d'autant plus qu'ils sont alignés avec leur parent et grand-parent. (la qualité de l'alignement se mesure avec le produit entre le vecteur (grand-parent, parent) et le vecteur (parent, point) (que l'on renormalise). Ainsi si l'alignement est parfait on obtient une valeur de 1 et pour le pire une valeur de -1. Cette valeur est pondérée par un facteur : $STRAIGHT\_LINE\_COEFF$ est soustraite à la distance entre le point et son parent.
- **thicken_map** : Enfin, on peut remarquer que les trajectoires ont tendance à longer les murs. Pour éviter que la voiture ne rentre en collision avec ces derniers, j'ai implémenté une fonction qui élargi artificiellement l'épaisseur de ces murs pour que la voiture ne se les prenne pas.

Cette approche du RRT^* consistant à calculer en temps réel sur la voiture une trajectoire présente un dernier problème. D'une itération à l'autre les solutions proposées peuvent être radicalement différentes.

Solution : Les premiers *sample_point* ne sont pas générés aléatoirement mais sont en fait les points du chemin trouver précédemment que l'on ajoute un à un. Ainsi si une autre solution est proposée, cela signifie qu'elle est strictement meilleure.

4.4 Communication avec le pure pursuit

Une fois la trajectoire construite par le RRT^*, il faut encore pouvoir la communiquer au *pure_pursuit*. Une première approche pourrait être d'intégrer directement le pure pursuit dans le fichier du RRT. Je ne trouvais pas cela très élégant.

J'ai donc décidé de communiquer le chemin du RRT^* au *pure_pursuit* via un topic ROS dédié : 'waypoints'. Il fallait pour cela convertir une liste de $Node$ en un message ROS.

C'est le rôle de la méthode **path2msg** :

Cette fonction convertit la liste de $Node$ en suit une string sous le format suivant : "(Node0.x, Node0.y)(Node1.x, Node1.y)...(NodeN.x,NodeN.y))". String qui est ensuite convertit en une liste de waypoints par le pure pursuit qui fonctionne alors comme précédemment.

5
POUR ALLER PLUS LOIN : AUTOMATISATION DES PROCESSUS

Initialement, je voulais profiter des dernières semaines pour développer une intelligence artificielle basée sur un réseau de neurones et un apprentissage de type *DQN* pour piloter la F1tenth. Cependant, j'ai préféré optimiser le plus possible le RRT^* afin d'espérer pouvoir le porter sur la vraie voiture. Toutefois j'ai tout de même entrepris de développer un script pour automatiser le lancement des différents programmes python et espérer ainsi automatiser l'apprentissage.

5.1 Le scheduler

Il s'agit du script responsable de déclencher et de stopper d'autres scripts python comme le RRT^* ou le $follow\_the\_wall$ par exemple.
Son fonctionnement est le suivant :

La classe scheduler possède une liste de tâche à effectuer : $task\_queue$. Les tâches suivent le format suivant : (2, WALL_FOLLOW, TIME_LAP, WAYPOINT_LOGGER). Le 2 représente le nombre de tour durant lequel la tâche est effectuée, la suite représente les scripts à démarrer au début de ces tours et à kill à la fin.

J'ai pu utiliser ce script pour réaliser la séquence suivante (la vidéo est disponible sur le github) :

- Faire un premier tour de circuit avec le $wall\_follow$ pour réaliser une nouvelle carte.
- Faire un second tour avec la $wall\_follow$ pour logger des waypoints grâce au $waypoint\_logger$
- Faire une série de deux tours avec le RRT^* grâce aux waypoints précédents.