RAPPORT SAE5

Céleste KOLOUSSA, Vincent ARIMON

Préparation pour la course F1tenth

Sommaire

I. Cahier des charges

Le projet vise à développer et intégrer de nouvelles fonctionnalités pour une voiture autonome participant au concours F1Tenth, en se concentrant sur la planification de trajectoire, le dépassement des concurrents et la sélection de la vitesse optimale. Ces fonctionnalités viendront compléter les algorithmes existants de décision sur l'angle de rotation et le freinage d'urgence. Les solutions développées devront être robustes, adaptées à des scénarios variés de course, et optimiser les performances dans des environnements avec obstacles statiques, dynamiques, ou en interaction avec d'autres voitures autonomes. Le développement sera principalement réalisé en C/C++ avec l'utilisation du système ROS (Robot Operating System) pour l'orchestration logicielle, bien que certaines parties puissent être réalisées indépendamment de ROS.

De plus, ce projet sera mis en jointure avec un autre projet complémentaire, dont l'objectif est de permettre à la voiture autonome de communiquer avec des feux tricolores via des requêtes HTTP ou des analyses vidéo à l'aide d'une caméra. Cette collaboration permettra de développer des interactions avancées entre la voiture et son environnement, renforçant ainsi ses capacités d'adaptation et sa compétitivité dans des contextes réels.

II. Environnement de travail

Nous avons travaillé dans un environnement Linux configuré sur une Jetson Nano, un micro-ordinateur performant dédié aux applications embarquées. Le développement du projet a été réalisé à l'aide de Visual Studio Code (VSCode), un éditeur de code polyvalent et adapté à nos besoins en programmation. Pour faciliter le travail à distance et l'accès à la Jetson Nano, nous avons établi une connexion SSH via Ethernet depuis une autre machine. De plus, nous avons utilisé GitLab comme plateforme de gestion de version et de collaboration, ce qui nous a permis de centraliser notre code, de suivre les évolutions du projet, et de travailler efficacement en équipe tout en maintenant une traçabilité rigoureuse des modifications. Cette configuration a assuré une coordination fluide entre le développement, les tests, et le déploiement des algorithmes.

Figure 1 - Jetson Nano

Figure 2 - Logo Gitlab

Figure 3 - Logo VsCode

III. Découverte de ROS

III.I Définition

Le ROS (Robot Operating System) est un ensemble de bibliothèques et d'outils logiciels open source conçus pour faciliter la conception et l'intégration de systèmes robotiques complexes. Plutôt qu'un « système d'exploitation » à proprement parler, il s'agit d'une infrastructure de communication et d'organisation modulaire où chaque fonctionnalité (ex. gestion des capteurs, calcul de trajectoire, pilotage de moteurs) prend la forme d'un nœud indépendant. Ces nœuds peuvent échanger des informations via des canaux nommés « topics » (système de publication-abonnement), ce qui permet de séparer clairement le code et de favoriser la réutilisation de composants logiciels entre différents projets. ROS inclut également des outils de visualisation, des simulateurs, des bibliothèques de calcul et des pilotes pour une large gamme de capteurs et d'actionneurs. Grâce à sa nature modulaire et extensible, ROS offre un cadre unifié pour développer, tester et déployer rapidement de nouvelles fonctionnalités robotiques, rendant le travail d'équipe et la maintenance du code plus efficaces.

Avant l'émergence de ROS, l'architecture la plus courante pour concevoir et déployer des applications robotiques s'appuyait souvent sur une logique client-serveur classique. Dans ce modèle, un programme central (le serveur) assurait la gestion et l'exécution des tâches critiques, tandis que les autres composants (les clients) envoyaient des requêtes et attendaient des réponses au fur et à mesure de l'avancement de la mission. La communication s'effectuait généralement via des protocoles réseau ou des sockets, et chaque client gérait une fonction ou un capteur spécifique. Cette approche impliquait de lourdes dépendances entre les différentes briques logicielles, avec une configuration parfois complexe pour coordonner l'accès aux capteurs et aux actionneurs. Du fait de cette topologie centralisée, il était aussi plus difficile d'étendre le système ou de réutiliser des briques logicielles existantes, car les applications n'étaient pas conçues pour être découplées et fonctionnelles indépendamment. L'architecture client-serveur, bien que fonctionnelle dans de nombreux cas, offrait donc moins de flexibilité et de modularité que les paradigmes plus récents comme ROS, où chaque composant s'exécute dans un nœud autonome et communique via un mécanisme de publication-abonnement.

III.II Prise en main

La prise en main de ROS s'est effectuée à travers différents travaux pratiques disponibles sur la plateforme Madoc. Ces exercices ont permis de se familiariser avec la création de packages, la mise en place de nœuds, la gestion de topics et l'utilisation de messages personnalisés (custom messages). Les TP ont également porté sur l'exploitation de fichiers de paramètres et de lancement (launch files) afin de configurer le système de manière fine, ce qui se révèle crucial pour gérer efficacement le pilotage et la communication entre les différentes composantes de la voiture. Cette phase d'apprentissage a posé les bases nécessaires à une bonne compréhension de l'infrastructure ROS et à la mise en place d'une architecture modulaire.

IV. Création d’un code fonctionnel pour la voiture

Dans un premier temps, l’équipe a analysé et repris le travail du groupe précédent en étudiant le code déjà existant. Cette démarche a permis de comprendre l’architecture globale et les fonctionnalités qui avaient déjà été implémentées. La création de deux projets distincts a alors été entreprise : l’un reposant sur le code hérité, et l’autre développé à partir de zéro. L’objectif est de comparer et d’évaluer différentes approches, afin de mutualiser à terme ce qui se révèle le plus robuste et le plus performant.

Le schéma global du projet s’appuie notamment sur l’exploitation des données issues d’un LiDAR, permettant de détecter l’environnement immédiat. En complément, un package de pilotage a été mis en place pour assurer le contrôle de la voiture via un clavier (téléop) ou une manette, tandis qu’un MUX (multiplexeur) gère la sélection du mode de conduite, qu’il soit manuel ou autonome. Pour assurer la sécurité, une première version de l’arrêt d’urgence utilise le LiDAR pour détecter un obstacle et stopper immédiatement le véhicule. Par la suite, une carte Ultrason conçue sous KiCad et fabriquée à l’IUT est venue enrichir le dispositif de détection. Les données de distance, fournies par quatre capteurs ultrason, sont transmises à l’ordinateur via une liaison série, et la suite du travail consiste à intégrer cette acquisition dans un nœud ROS afin de publier les mesures sur un topic dédié. Cela permet une fusion de l’information entre capteurs LiDAR et Ultrason, renforçant la robustesse de la détection d’obstacles. Enfin, l’implémentation de Hector SLAM est prévue pour réaliser un mapping et une localisation plus précis, renforçant l’efficacité de la navigation autonome et préparant le terrain pour des fonctionnalités avancées telles que la planification de trajectoire en temps réel.

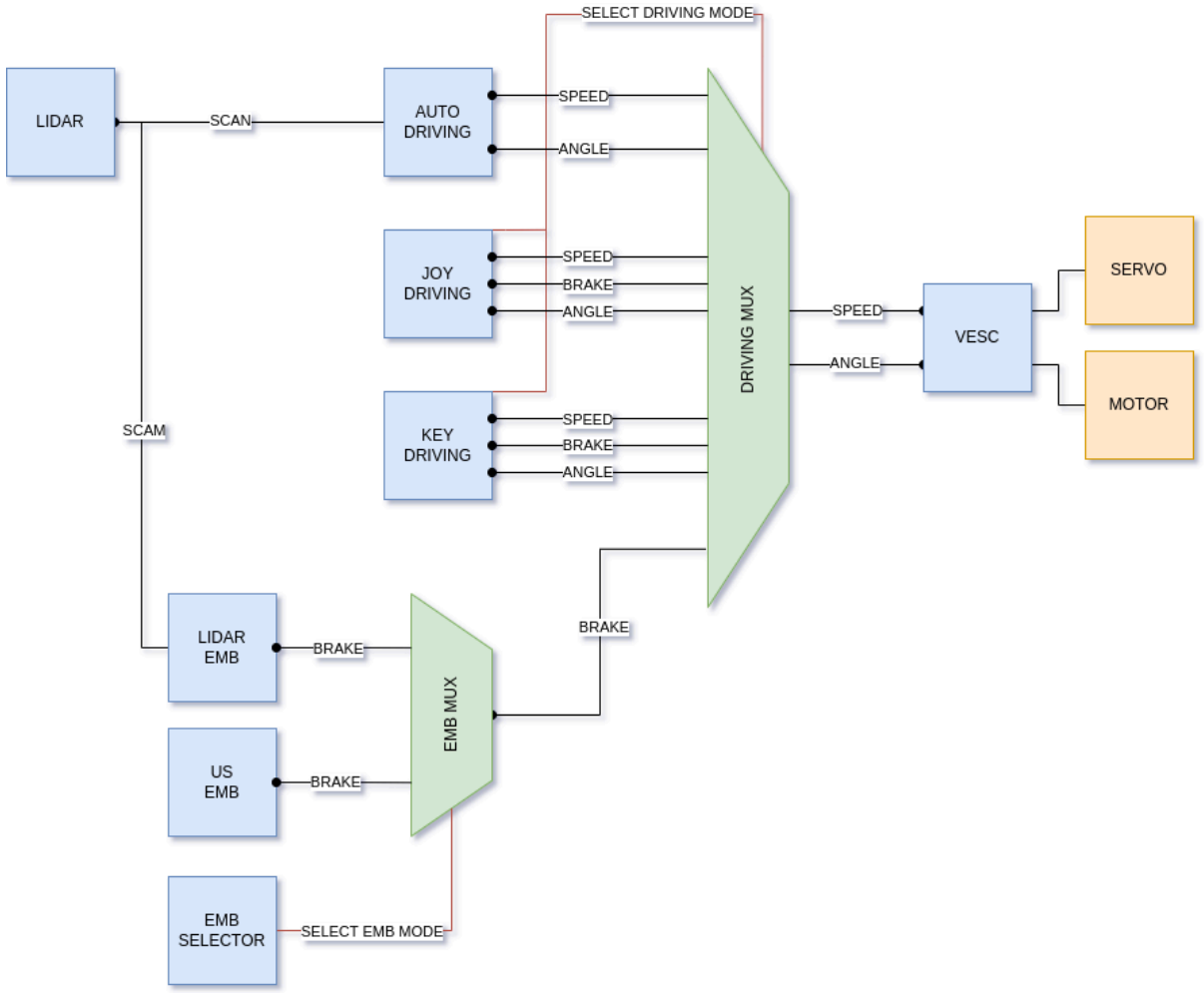

Figure 5- Diagramme fonctionnel du projet ROS

Voici une description détaillée de chaque nœud du système, en précisant leur rôle, leurs entrées et leurs sorties :

1. Nœud RPLIDAR (Package ROS RPLIDAR)

- Rôle :

Ce nœud récupère les données du capteur LIDAR, effectue le scan laser en temps réel et publie les informations sous forme de cloud de points.

- Entrées :
  - Données brutes provenant du capteur RPLIDAR connecté.
- Sorties :
  - /scan : Topic où sont publiées les données des scans laser
- Utilisation :
  - Les données publiées dans /scan sont utilisées par le nœud AUTO DRIVING pour la conduite autonome.
  - Elles sont également exploitées par le nœud EMB MUX pour la logique de freinage d'urgence via le lidar.

2. Nœud AUTO DRIVING

- Rôle :

Assure la conduite autonome en utilisant les données publiées sur le topic /scan par le nœud RPLIDAR.

- Entrées :
  - /scan : Topic contenant les données du LIDAR.
- Sorties :
  - VITESSE (SPEED) : Commande de vitesse pour le moteur.
  - ANGLE : Commande de direction pour le servo.
  - Ces sorties sont envoyées vers le DRIVING MUX.

3. Nœud JOY DRIVING

- Rôle :

Contrôle manuel du robot via une manette (joystick).

- Entrées :
  - /joy : Topic contenant les axes et boutons du joystick.
- Sorties :
  - VITESSE (SPEED) : Commande de vitesse pour le moteur.
  - ANGLE : Commande de direction pour le servo.
  - Ces sorties sont également envoyées vers le DRIVING MUX.

4. Nœud KEY DRIVING

- Rôle :

Gère le contrôle manuel via clavier pour piloter la vitesse et la direction du robot.

- Entrées :
  - Commandes clavier (interprétées par le programme).
- Sorties :
  - VITESSE (SPEED) : Commande de vitesse.
  - ANGLE : Commande de direction.
  - Ces commandes sont envoyées au DRIVING MUX.

5. Nœud DRIVING MUX

- Rôle :

Multiplexeur central pour sélectionner le mode de conduite actif (AUTO, JOY ou KEY) et envoyer les commandes finales aux actionneurs.

- Entrées :
  - VITESSE et ANGLE des trois modes :
    - AUTO DRIVING
    - JOY DRIVING
    - KEY DRIVING
  - Signal de sélection du mode de conduite.
- Sorties :
  - VITESSE : Commande finale pour le moteur.
  - ANGLE : Commande finale pour le servo.
  - Ces sorties sont envoyées au nœud VESC.

6. Nœud EMB MUX

- Rôle :

Combine les informations de détection d'obstacles pour activer le freinage d'urgence.

- Entrées :
  - LIDAR EMB : Données de détection d'obstacles via le topic /scan du nœud RPLIDAR.
  - US EMB : Données provenant des capteurs ultrasons.
  - EMB SELECTOR : Sélectionne la logique de freinage active.
- Sorties :
  - BRAKE : Signal de freinage envoyé au DRIVING MUX pour arrêter le robot.

7. Nœud VESC

- Rôle :

Exécute les commandes finales envoyées par le DRIVING MUX pour contrôler les actionneurs.

- Entrées :
  - VITESSE : Commande pour contrôler la vitesse du moteur.
  - ANGLE : Commande pour contrôler la direction via le servo.
- Sorties :
  - Contrôle du MOTEUR pour la vitesse.
  - Contrôle du SERVO pour la direction.

Résumé de l'Architecture

- Le nœud RPLIDAR collecte les données du capteur LIDAR et les publie dans le topic /scan.
- Les nœuds AUTO DRIVING, JOY DRIVING et KEY DRIVING génèrent des commandes de vitesse et angle selon le mode de conduite actif.
- Le DRIVING MUX sélectionne les commandes actives et les envoie au nœud VESC pour contrôler le moteur et le servo.
- Le EMB MUX gère la logique de freinage d'urgence en combinant les données LIDAR et ultrasons pour assurer la sécurité du robot.

Cette architecture modulaire permet d'assurer une gestion claire et efficace de la navigation, tout en offrant un mode d'arrêt d'urgence robuste.

V. Conception de la Carte Électronique pour le Traitement des Données des Capteurs Ultrasons

La carte électronique a été conçue pour récupérer les données des capteurs ultrasons, les traiter via un microcontrôleur ESP32 WROOM32E de chez Espressif et les envoyer via une liaison série vers un nœud ROS qui publiera ces données dans des topics.

La conception de la carte électronique a été relativement simple puisqu'elle se limite principalement à du routage. Aucun composant complexe n'est embarqué en dehors des connecteurs et du microcontrôleur.
Les éléments principaux de la carte sont les suivants :

1. 4 ports pour les capteurs ultrasons :
   Ces ports permettent le branchement physique des capteurs pour mesurer des distances.

2. Écran OLED connecté en I2C :
   L'écran sert à afficher localement les données mesurées par les capteurs.

3. Microcontrôleur ESP32 WROOM32E :
   Le microcontrôleur est le cœur du système. Il récupère, traite et envoie les données.

Le microcontrôleur ESP32 a pour rôle de collecter les données des capteurs ultrasons, de les organiser dans une structure puis de les envoyer via une liaison série. Le fonctionnement repose sur une architecture multi-thread pour garantir l'efficacité du processus.

V.I Structure des Données

Les données collectées sont stockées dans une structure avant d'être transmises. Cette structure permet d'organiser clairement les informations mesurées pour faciliter leur traitement côté nœud ROS.

V.II Organisation des Threads

Le processus est découpé en trois threads distincts pour répartir les tâches et éviter les blocages :

1. Thread de Mesure :
   Lit les données des capteurs ultrasons périodiquement.
   Stocke les données dans la structure dédiée.
2. Thread d'Affichage :
   Affiche les données en temps réel sur l'écran OLED via l'interface I2C.
3. Thread de Publication Série :
   Envoie les données sous forme structurée via la liaison série vers un nœud ROS.

V.III Programme test

Un programme de test a été réalisé afin de valider le bon fonctionnement de la carte et de la liaison série. Ce programme se connecte à la liaison série, reçoit les données structurées envoyées par le microcontrôleur ESP32, et les affiche dans le terminal. Cette étape a permis de confirmer que les mesures des capteurs ultrasons sont correctement récupérées et transmises. Il ne reste plus qu'à implémenter ce programme dans un nœud ROS afin de publier les données dans un topics, rendant celles-ci accessibles aux autres composants du système.

V.IV Conclusion

La carte électronique conçue est donc simple et efficace. Elle permet, grâce à l'ESP32, d'assurer une récupération fluide des données des capteurs ultrasons et leur publication via un nœud ROS. L'utilisation de threads permet d'assurer une gestion concurrente des tâches de mesure, d'affichage et de publication série, optimisant ainsi la performance du système global.

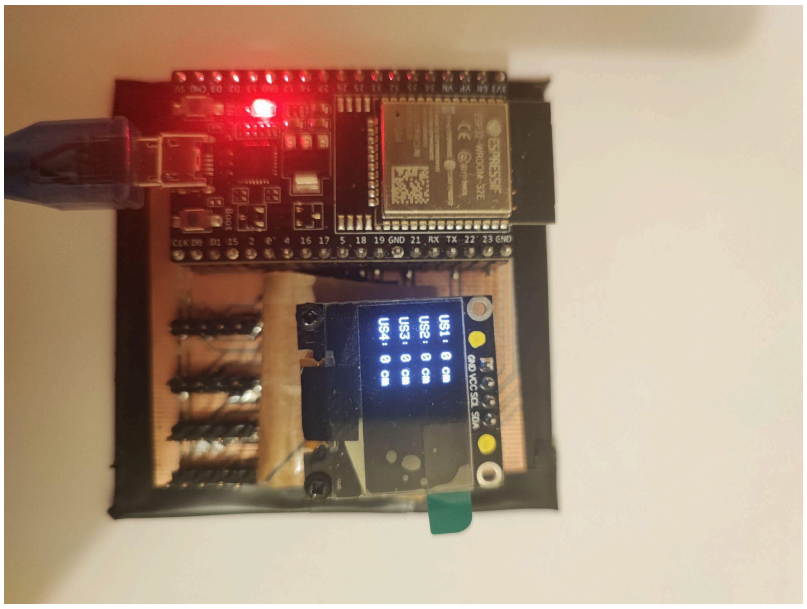

Figure 5- Carte US complète

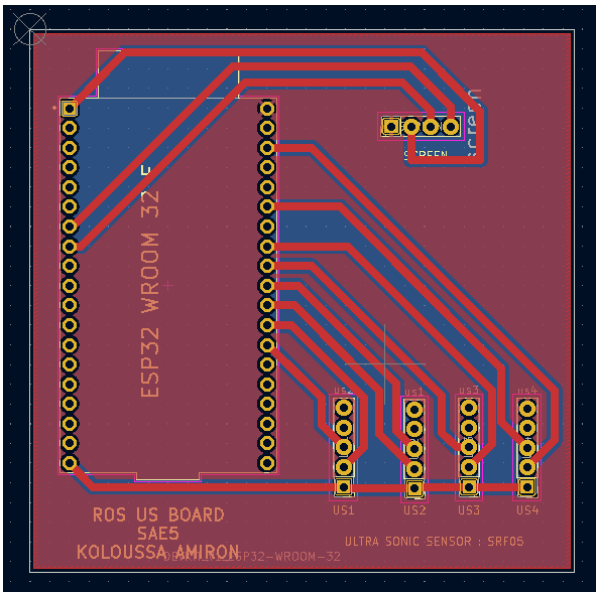

Figure 6 - Routage de la carte

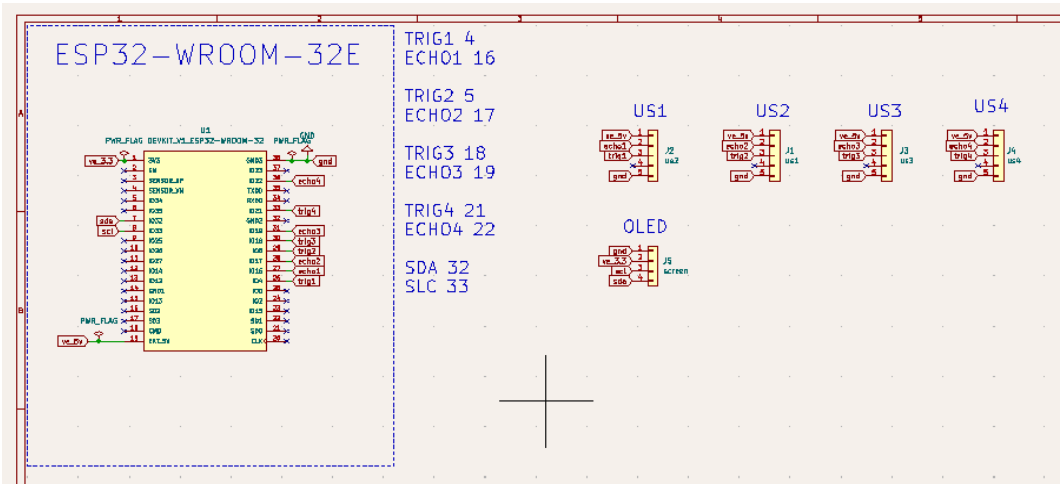

Figure 7 - Schéma kicad de la carte

VI. Objectifs futurs

À l'avenir, le projet entend finaliser l'intégration de Hector SLAM pour le mapping, ce qui améliorera la navigation et la reconnaissance de l'environnement. Il s'agira également de fusionner et de traiter les données du LiDAR et de la carte Ultrason dans le même système ROS, dans le but de perfectionner les algorithmes de détection, d'évitement d'obstacles et de dépassement. Une autre étape cruciale consiste à valider l'algorithme de dépassement en conditions de course réelles, après avoir mené une série de tests sur simulateur et sur piste F1Tenth. Par ailleurs, l'ajout de fonctionnalités de communication avec des feux tricolores — via requêtes HTTP ou analyse vidéo — offrira à la voiture des capacités d'interaction avancées avec son environnement, la rendant plus proche de scénarios de conduite autonomes dans le monde réel. L'ensemble de ces développements reposera sur ROS pour assurer la modularité, l'évolutivité et la maintenance du code à long terme, tout en permettant d'intégrer facilement de nouveaux capteurs ou algorithmes si nécessaire. L'objectif ultime est de disposer d'un véhicule autonome fiable et performant, capable de se déplacer à des vitesses élevées, d'interagir efficacement avec son environnement et de relever les défis d'une compétition comme la F1Tenth.

VII. Sources

VII.I ROS

- generationrobots.com
- digitalcorner-wavestone.com

VII.II Figures

- Figure 1 : Amazon.fr
- Figure 2 : Stickpng.com
- Figure 3 : logos-marques.com
- Image de la page de garde : roboracer.ai