

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE



Section de Microtechnique  
Printemps 2024

---

## Systèmes Embarqués et Robotique Miniprojet : MazeRover

---



*Groupe 23 :*

Michel ABELA      339421  
Rim EL QABLI      340997

# Table des Matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Capteurs utilisés et principe de fonctionnement du MazeRover</b>	<b>3</b>
2.1	Capteurs de proximité Infrarouges . . . . .	3
2.2	Accéléromètre intégré à l'IMU . . . . .	4
<b>3</b>	<b>Structure, modularisation et multithreading</b>	<b>4</b>
3.1	Schéma bloc de la logique du multithreading de notre système . . . . .	4
3.2	Thread Main . . . . .	5
3.3	Module infrarouge et thread IRSensorReading . . . . .	5
3.4	Module de contrôle moteur pas à pas et thread MotorController . . . . .	6
3.5	Module IMU et thread IMUReading . . . . .	6
<b>4</b>	<b>Stratégies d'optimisation et implémentation de la navigation du robot dans le labyrinthe</b>	<b>7</b>
4.1	Gestion et régulation de la trajectoire dans le Labyrinthe - Module infrarouge . . . . .	7
4.2	Contrôle d'écart de Vitesse pour Mancœuvres Précises - Module Moteur . . . . .	7
4.3	Gestion Optimisée de l'activation de l'IMU - Module IMU . . . . .	8
<b>5</b>	<b>Difficultés rencontrées et pistes d'amélioration</b>	<b>8</b>
5.1	Stratégies de Debugging pour les Modules du MazeRover . . . . .	8
5.2	Adaptations et Modification de la stratégie de multithreading pour la Stabilité Système . .	8
5.3	Calibration des Capteurs et Adaptation aux Fluctuations de Batterie . . . . .	9
5.4	Création d'un Labyrinthe en MDF pour effectuer des Tests Réalistes sur le MazeRover . .	9
<b>6</b>	<b>Gestion du projet et collaboration Git</b>	<b>10</b>
<b>7</b>	<b>Conclusion</b>	<b>10</b>
	<b>Références</b>	<b>11</b>

# 1 Introduction

Ce document détaille le Miniprojet réalisé dans le cadre du cours de Systèmes Embarqués et Robotique pour la section de Microtechnique.

Notre projet, intitulé MazeRover, utilise le robot e-puck que nous avons programmé pour fonctionner comme un rover semi-autonome capable de naviguer dans un labyrinthe. La conception évoque l'image d'une boule se déplaçant librement à travers un parcours. Le robot, propulsé par le système d'exploitation temps réel ChibiOS [5], avance dans un labyrinthe portable jusqu'à rencontrer un obstacle frontal. La direction à prendre pour continuer son parcours est alors déterminée par l'inclinaison du labyrinthe, ajustée manuellement par l'utilisateur.

L'objectif principal du MazeRover est de pouvoir naviguer d'un point A à un point B selon 3 itinéraires possibles (explicités dans la Figure 1 par les traces Rouge, Vert et Bleu), sans collisions, en gérant de manière autonome les ajustements nécessaires pour éviter les parois du labyrinthe.

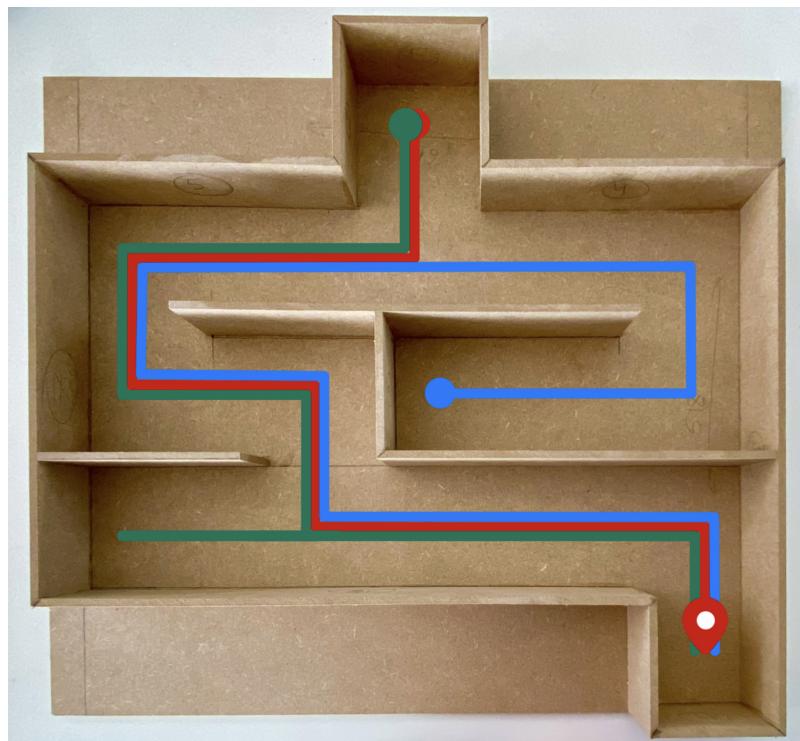


FIGURE 1 – Itinéraires possibles au sein du labyrinthe (Rouge, Vert et Bleu)

# 2 Capteurs utilisés et principe de fonctionnement du MazeRover

## 2.1 Capteurs de proximité Infrarouges

Afin d'assurer une détection efficace des obstacles tout au long de son itinéraire, nous avons choisi d'utiliser le capteur infrarouge (IR) frontal de notre robot MazeRover, en guise de capteur de proximité. Ce capteur joue un rôle crucial, car il permet au robot de s'arrêter automatiquement dès qu'un obstacle est détecté à une distance seuil prédéfinie. Par ailleurs, nous nous sommes également appuyés sur les capteurs

IR sur les côtés droit et gauche du robot. Ces capteurs latéraux nous permettent d'ajuster précisément la position du robot, garantissant ainsi qu'il ne rentre pas en collision avec les parois du labyrinthe. De plus, si le robot détecte une paroi trop proche sur un côté, ou rencontre un obstacle frontal, les LEDs rouges associées aux trois capteurs de proximité s'illuminent, signalant une détection de proximité à l'utilisateur.

En outre, le choix du bois MDF pour la construction du labyrinthe (se référer à la section 5.4) s'est avéré judicieux dans le cadre de notre utilisation des capteurs IR, notamment en raison de sa couleur claire qui favorise une bonne réflexion des rayons infrarouges émis par les capteurs, mais aussi pour la bonne adhérence des roues en caoutchouc du robot au contact du bois MDF qui l'empêche de déraper lors des différentes inclinaisons. Bien que nous ayons envisagé de peindre les surfaces du labyrinthe en blanc pour maximiser cette réflexion, les tests préliminaires ont montré que les capteurs IR fonctionnaient déjà de manière optimale avec la couleur naturelle du MDF, nous permettant ainsi de conserver l'aspect originel du matériau sans modification supplémentaire.

## 2.2 Accéléromètre intégré à l'IMU

Afin de mesurer avec précision l'inclinaison du labyrinthe, ajustée manuellement par l'utilisateur, nous avons intégré l'accéléromètre de l'Inertial Measurement Unit (IMU) [3] du robot. Ceci nous permet de remplir notre objectif de répondre rapidement aux variations d'orientation causées par l'inclinaison. Comme pour les capteurs infrarouges, un système de signalisation via des LEDs a été mis en place pour offrir un feedback visuel à l'utilisateur. Lorsque l'accéléromètre détecte une inclinaison notable du labyrinthe, soit d'un angle approximatif de 10 degrés, une LED rouge s'allume du côté de l'inclinaison détectée.

# 3 Structure, modularisation et multithreading

Dans le domaine des systèmes embarqués, l'utilisation de systèmes d'exploitation temps réel (RTOS) comme ChibiOS [5] sont essentiels pour garantir que les processus internes respectent les contraintes de temps réel imposées par le matériel et le logiciel. À la différence des systèmes d'exploitation traditionnels qui privilégient la quantité de tâches exécutées, ChibiOS donne ainsi la priorité à la fiabilité du timing des tâches, ce qui est crucial pour des systèmes comme notre robot MazeRover. [2]

Notre programme repose sur 3 modules organisés autour de 4 threads, que nous allons détailler par la suite : le thread main, le module infrarouge (thread IRSensorReading), le module de contrôle moteur (thread MotorController), et enfin le module IMU (thread IMUReadings).

## 3.1 Schéma bloc de la logique du multithreading de notre système

Afin de mieux expliciter la dynamique du multithreading au sein de notre projet, nous avons réalisé un schéma bloc à la Figure 2, mettant en avant les relations entre les différents threads et topics de notre système.

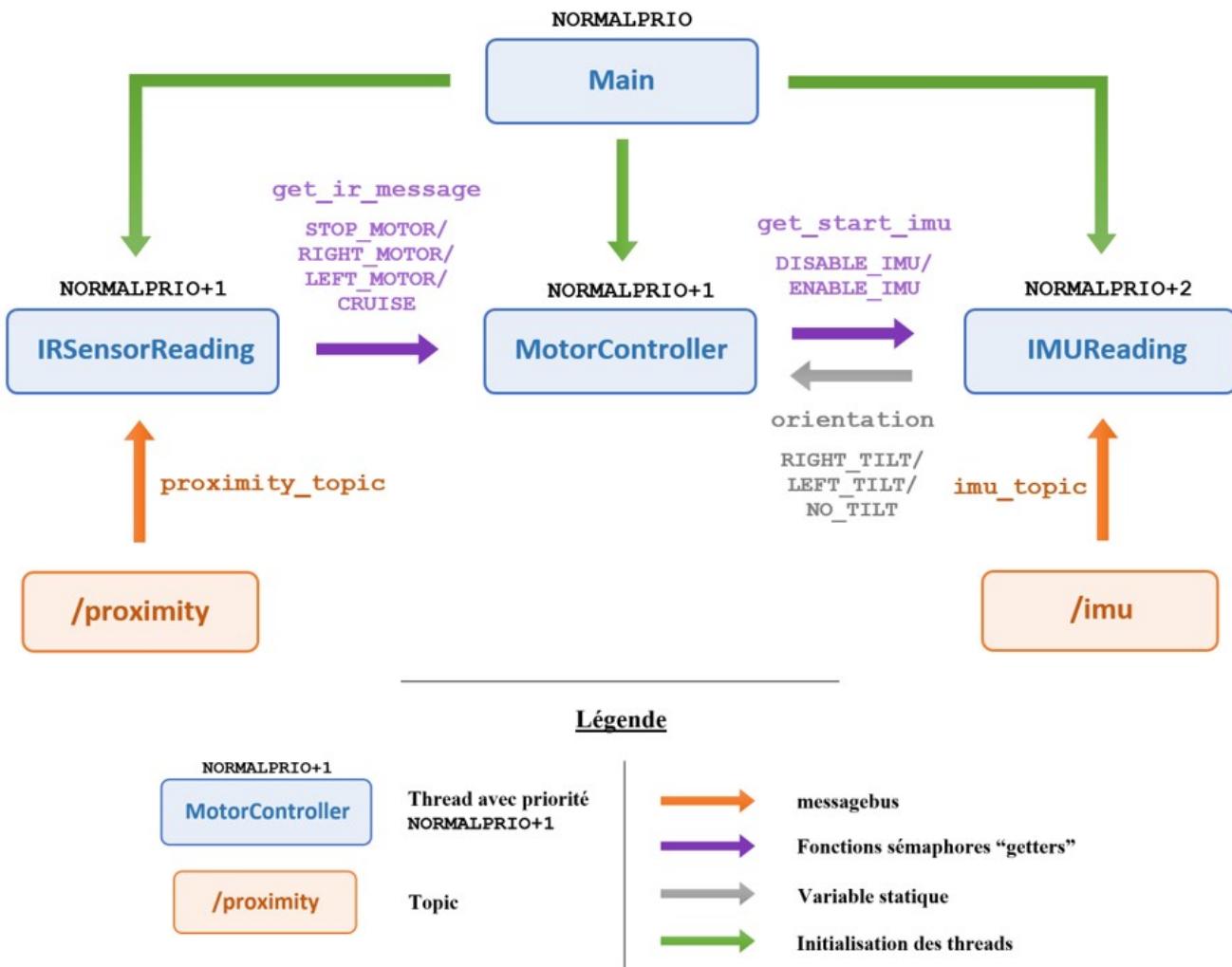


FIGURE 2 – Schéma bloc du multithreading de notre projet, muni de sa légende

### 3.2 Thread Main

Le thread Main, de priorité NORMALPRIORITY, consiste en linitialisation de divers threads, modules et périphériques de la librairie e-puck2\_main-processor. En effet, après initialisation du système et des dépendances nécessaires à l'aide de halInit et chSysInit, ce thread configure un messagebus pour faciliter la communication inter-modules, qui opère de manière conjointe aux fonctions sémaphores getters. Les différents périphériques et modules de notre robot MazeRover, à savoir les moteurs, les capteurs de proximité, et lInertial Measurement Unit (IMU) sont initialisés successivement. Ensuite, les threads spécifiques à chaque module sont lancés pour gérer les périphériques respectifs ainsi que la logique de notre système. Enfin, la boucle principale du main exécute une pause de 200 ms, permettant ainsi une consommation réduite des ressources en attendant l'exécution du reste du programme.

### 3.3 Module infrarouge et thread IRSensorReading

Le module infrarouge, exécuté sur un thread nommé IRSensorReading avec une priorité définie à NORMALPRIORITY+1, permet la détection des obstacles et l'envoi des commandes de navigation du robot au module de contrôle moteur. Ce thread surveille continuellement les données de proximité fournies par les trois capteurs infrarouges (frontal, droit et gauche) à travers le messagebus proximity\_topic, en s'abonnant, par le biais de la fonction messagebus\_topic\_wait, au topic /proximity pour s'assurer

de recevoir les mesures les plus récentes. Selon les valeurs lues (`front_value`, `right_value`, `left_value`), différentes actions sont prises : en cas de détection d'un obstacle frontal à partir d'un seuil prédéfini, les moteurs sont arrêtés (`ir_message = STOP_MOTOR`), si un obstacle est détecté à droite ou à gauche, la vitesse du moteur correspondant est augmentée (`ir_message = RIGHT_MOTOR` ou `ir_message = LEFT_MOTOR`), ou une vitesse de croisière est maintenue en l'absence de proximité d'obstacles (`ir_message = CRUISE`).

En effet, la détection d'obstacles à droite et à gauche de notre robot permet de réguler sa position à l'approche des parois du labyrinthe, et ainsi permet d'empêcher une potentielle collision. Le thread commande également l'état des LEDs correspondants à l'emplacement des trois capteurs infrarouges pour refléter visuellement l'orientation du mouvement. Enfin, afin de ne pas monopoliser les ressources, il se met en pause pour 50 ms après chaque cycle de lecture et de décision, permettant ainsi aux autres modules de fonctionner efficacement.

### 3.4 Module de contrôle moteur pas à pas et thread MotorController

Le module de contrôle moteur pas à pas, intégré dans un thread dénommé `MotorController` avec une priorité définie à `NORMALPRI0+1`, s'occupe de la gestion dynamique des vitesses des moteurs en fonction des commandes reçues du module infrarouge et des orientations déterminées par l'Inertial Measurement Unit (IMU). Dans son fonctionnement, le thread évalue d'abord si l'utilisation de l'IMU est nécessaire (`start_imu`). En mode de croisière normal (`DISABLE_IMU`), il récupère les commandes infrarouges (`ir_command = get_ir_message()`), ajustant les vitesses des moteurs droit et gauche (`speed_right`, `speed_left`) afin de naviguer efficacement sans intervention de l'IMU. Nous avons alors parmi les commandes effectuées, l'arrêt complet des moteurs, l'accélération d'un moteur pour corriger la trajectoire, ou le maintien d'une vitesse constante.

Dans le cas où le module de contrôle moteur choisit d'activer l'IMU (`ENABLE_IMU`), le thread `MotorController` se met en attente des orientations de l'IMU pour alors potentiellement exécuter un virage précis de 90 degrés selon l'accélération détectée par l'IMU (correspondant à une inclinaison induite du labyrinthe). Après chaque action, une mise à jour des vitesses est effectuée à l'aide de la fonction `motors_speed_update`, tandis que le thread se met en pause pour 50 ms, permettant ainsi aux autres modules de fonctionner sans interférence.

### 3.5 Module IMU et thread IMUReading

Le module IMU, opérant sous un thread intitulé `IMUReading` avec une priorité `NORMALPRI0+2`, constitue le module de détection de l'inclinaison du labyrinthe pour orienter avec précision les ajustements des moteurs. Ce thread interagit activement avec le topic `/imu` sur le messagebus `imu_topic` afin de recevoir les données de l'Inertial Measurement Unit (IMU). Lorsque la fonction IMU est activée par le contrôleur moteur (`start_function = get_start_imu()`), il récupère les valeurs mesurées de l'IMU et détermine la direction de l'inclinaison à l'aide de la fonction `show_gravity` [4]. Cette fonction analyse l'accélération mesurée, identifie l'orientation de l'appareil (droite ou gauche), et met à jour visuellement l'état des LEDs correspondantes en fonction de l'inclinaison détectée. Nous avons par ailleurs pu établir une équivalence empirique entre le seuil de détection de l'IMU que nous avons prédéfini, et l'angle d'inclinaison du labyrinthe, qui est d'environ 10 degrés.

Si une inclinaison est détectée, le thread écrit cette orientation (`RIGHT_TILT` ou `LEFT_TILT`) dans une variable statique (`orientation`), accessible par le module contrôleur moteur afin d'ajuster la navigation du robot. Dans le cas contraire, si l'IMU n'est pas requis ou si après avoir traité les données nécessaires aucune inclinaison n'a été détectée, l'orientation est réinitialisée à `NO_TILT`. Enfin, ce cycle de détection

et de mise à jour se déroule à intervalles réguliers de 50 ms, garantissant une réactivité suffisante sans surcharger les autres processus du système. En effet, le choix de la mise en place d'une priorité de  $N_{JORMALPRI0+2}$  pour ce thread reflète le besoin pour notre système de s'assurer que le module IMU soit exécuté sans risque d'interruption lorsqu'il est appelé par le module de contrôle moteur (de priorité plus faible), pour être sûr de détecter l'inclinaison engendrée par l'utilisateur.

## **4 Stratégies d'optimisation et implémentation de la navigation du robot dans le labyrinthe**

### **4.1 Gestion et régulation de la trajectoire dans le Labyrinthe - Module infrarouge**

Pour optimiser la navigation du robot dans le labyrinthe tout en assurant une régulation précise de sa trajectoire, une condition logique a été implémentée dans le module de détection infrarouge. Cette stratégie est cruciale lorsque le robot, se rapprochant trop d'un mur sur un côté, doit ajuster sa position pour rester plus ou moins centré dans le couloir. Par exemple, si le robot détecte qu'il est trop proche du mur droit, il tentera naturellement de se décaler vers la gauche pour maintenir une distance sécuritaire. Cette logique de navigation engendre un bon nombre de situations critiques qu'il faut prévoir dans le code du module infrarouge.

Premièrement, lors de la navigation près d'une intersection de deux couloirs, le robot doit éviter un décalage excessif qui pourrait l'orienter droit dans le mur d'en face, le positionnant ainsi de travers par rapport au chemin voulu. Pour s'occuper de cette condition, la fonction de navigation permet au robot de corriger sa trajectoire uniquement s'il est assez proche d'un mur (condition déterminée avec le threshold CRUISING\_THRESHOLD\_RIGHT / CRUISING\_THRESHOLD\_LEFT) et pas trop loin du mur opposé (à l'aide du MIN\_THRESHOLD\_RIGHT / MIN\_THRESHOLD\_LEFT, un threshold qui représente la distance dans le cas d'une intersection). Cette approche permet donc au robot de prendre en compte les deux murs du couloir avant de choisir sa nouvelle trajectoire.

Deuxièmement, pour assurer une navigation fluide et continue, le robot utilise des micro-corrections de vitesses sur chaque moteur afin d'assurer des petits rayons de courbure et ainsi éviter de se voir propulsé d'un mur à l'autre pendant son passage dans un couloir. Ceci se fait en incrémentant la valeur du SPEED\_INCREMENT à chaque itération tant que le nombre maximal de boucles MAX\_CORRECTION\_NBR n'a pas été atteint. Lorsque celui-ci est atteint, la vitesse des deux moteurs est alors remise en vitesse de croisière MOTOR\_CRUISING\_SPEED.

Enfin, afin d'éviter que le robot ne rentre en collision avec les murs latéraux dans le cas où il se trouve trop proche, une condition de navigation a été rajoutée, permettant au robot de corriger sa trajectoire dans n'importe quelle situation dans le cas où l'un des deux capteurs de proximité (droit/gauche) affiche une valeur plus grande que MAX\_THRESHOLD\_RIGHT / MAX\_THRESHOLD\_LEFT, assurant son repositionnement vers le chemin voulu.

### **4.2 Contrôle d'écart de Vitesse pour Manœuvres Précises - Module Moteur**

Pour améliorer la précision et la fluidité de la navigation du robot dans le labyrinthe, une limitation spécifique a été mise en place concernant l'écart de vitesse entre les deux moteurs. Cette mesure est cruciale pour éviter des manœuvres brusques ou des virages avec un rayon trop grand, qui pourraient résulter d'une différence excessive de vitesse entre les côtés droit et gauche du robot.

La fonction motor\_speed\_increment ajuste la vitesse du moteur tout en vérifiant que l'écart (de  $1ta$ ) ne dépasse pas une valeur limite (DELTA\_LIMIT). Si cet écart est trop important, la vitesse ajustée

est ramenée à sa valeur précédente pour maintenir une conduite plus douce et plus contrôlée, évitant ainsi des réactions imprévues qui pourraient perturber son parcours à travers le labyrinthe.

### **4.3 Gestion Optimisée de l'activation de l'IMU - Module IMU**

Dans le cadre de l'optimisation de la navigation de notre robot MazeRover au sein du labyrinthe, une attention particulière a été portée à la gestion efficace des ressources système, en particulier concernant l'utilisation de l'IMU. Afin d'éviter une consommation inutile de ressources et une sollicitation excessive du système, l'activation de la détection par l'IMU est conditionnée par l'état ENABLE\_IMU. Dans le cas contraire, l'IMU est à l'état DISABLE\_IMU. Ce mécanisme assure que les informations issues du `imu_` topic sur l'accélération et l'inclinaison du labyrinthe ne sont collectées et traitées par le système qu'au moment opportun, réduisant ainsi le temps de traitement et la charge sur le messagebus.

## **5 Difficultés rencontrées et pistes d'amélioration**

### **5.1 Stratégies de Debugging pour les Modules du MazeRover**

Dans le développement de notre projet MazeRover, nous avons porté une attention particulière à la configuration et à l'évaluation individuelle de chaque module, des étapes cruciales pour garantir la performance optimale de l'ensemble du système. Pour y parvenir de manière efficace, nous avons adopté une approche de debugging traditionnelle, en intégrant dans des versions initiales du code des instructions de type `chprintf` à travers le code. Cette stratégie nous a permis de surveiller en direct les valeurs et les états des différents paramètres des modules, ce qui a grandement facilité l'identification et la résolution des problèmes rencontrés.

### **5.2 Adaptations et Modification de la stratégie de multithreading pour la Stabilité Système**

L'intégration du multithreading et des techniques avancées de gestion des threads s'est avérée complexe dans le cadre de notre miniprojet de robotique.

Initialement, nous avons tenté d'implémenter une logique de Round Robin [6] à l'aide du `chThdYield` pour orchestrer l'exécution des threads. Cependant, cette méthode a conduit à divers problèmes, notamment des kernel panics et des dysfonctionnements du capteur infrarouge, attribuables à une compréhension limitée des interactions entre les différents threads. La difficulté à visualiser le flux de contrôle et à déterminer comment les ressources étaient allouées à chaque thread a considérablement réduit notre capacité à diagnostiquer et résoudre les problèmes.

Face à ces obstacles, nous avons opté pour des techniques de multithreading plus traditionnelles, telles que l'utilisation de `chThdSleepMilliseconds` pour la gestion des délais, `chSysLock` pour les sections critiques, et la définition de priorités spécifiques entre les threads, afin de simplifier la gestion et d'améliorer la traçabilité du code. De plus, pour contourner les complications liées à l'usage intensif de messagebus et de divers topics, qui exacerbait la complexité, nous avons implémenté des sémaphores via des fonctions "getters" qui facilitent la synchronisation entre les threads.

En outre, nous avons décidé de simplifier la structure globale du code en éliminant le module de navigation, qui était initialement prévu pour contenir toute la logique de navigation du robot. Cette logique a été redistribuée de manière plus modulaire entre les différents composants, améliorant ainsi non seulement la maintenabilité du système mais aussi sa robustesse face aux défis de la navigation dans un environnement imprévisible. Cette réorganisation nous a permis de mieux isoler les responsabilités, facilitant la détection d'erreurs et les ajustements nécessaires à l'amélioration continue du projet.

### 5.3 Calibration des Capteurs et Adaptation aux Fluctuations de Batterie

Lors de la mise en œuvre de notre projet robotique, nous avons rencontré des difficultés liées à l'imperfection inhérente des capteurs et à l'influence de la charge de la batterie sur leurs performances. En effet, chaque capteur, bien que similaire en type et en fonction, possède des caractéristiques légèrement différentes, ce qui a été le cas pour nos trois capteurs infrarouges. Ces variations nécessitent un ajustement empirique des seuils de détection pour chaque capteur afin d'assurer une réponse uniforme et précise du système.

De plus, nous avons observé que les valeurs préétablies pour ces capteurs deviennent imprécises lorsque la batterie du robot n'est pas complètement chargée. La tension plus basse affecte directement la performance des capteurs, entraînant des lectures variables qui peuvent compromettre la navigation et la détection d'obstacles du robot. Nous avons ainsi été contraints de toujours tester notre robot dans les conditions de charge les plus optimales. Au vu des difficultés techniques engendrées, une potentielle intégration et utilisation de capteurs davantage stables et performants pourrait constituer une piste d'amélioration possible pour notre projet.

### 5.4 Crédit d'un Labyrinthe en MDF pour effectuer des Tests Réalistes sur le MazeRover

Pour mettre notre projet à l'épreuve dans des conditions qui reflètent de véritables défis liés à l'environnement du MazeRover, nous avons conçu un labyrinthe en utilisant des plaques en bois Medium Density Fiberboard (MDF), visible à la Figure 4. Tandis que le design du labyrinthe s'est fait sur le logiciel de conception Fusion 360 comme démontré à la Figure 3, c'est par le biais des ressources disponibles au SKIL (Student Kreativity and Innovation Lab [7]), que nous avons pu matérialiser notre conception.

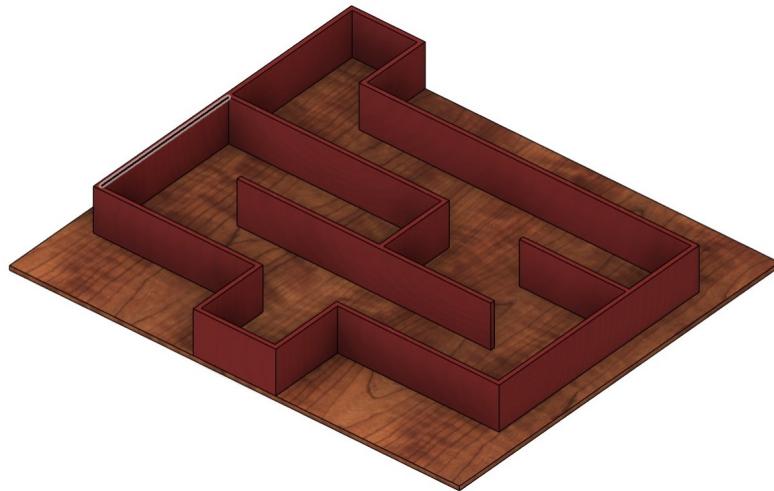


FIGURE 3 – Design du labyrinthe sur le logiciel Fusion 360

Quant au matériau choisi, le MDF, celui-ci nous a paru tout à fait pertinent pour notre application, et ce notamment en raison de sa robustesse (par rapport à d'autres alternatives comme le carton) et sa facilité de manipulation, permettant des modifications au besoin. Ce cadre en MDF a été extrêmement utile, car il nous a permis de simuler des scénarios complexes et de tester efficacement les capacités de navigation et de détection d'obstacles de notre robot, affinant ainsi nos approches et nos technologies en situation quasi-réelle. De plus, ayant initialement prévu de tapisser le sol du labyrinthe par une matière en caoutchouc afin d'améliorer l'adhérence des roues du robot, nous n'avons finalement pas opté pour cette option vu que le bois adhère déjà bien au roues du MazeRover.

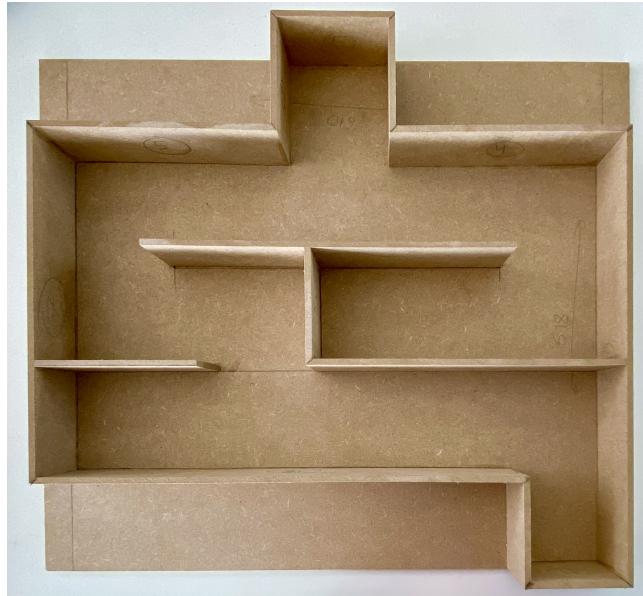


FIGURE 4 – *Labyrinthe réalisé en Medium Density Fiberboard (MDF)*

## 6 Gestion du projet et collaboration Git

Au début de notre projet, nous avons opté pour une planification initiale « papier et crayon » afin de définir clairement les modules nécessaires et les threads associés, ce qui nous a aidés à appréhender efficacement le multithreading.

Après avoir esquissé cette structure, nous nous sommes répartis les tâches pour élaborer un premier brouillon de nos différents modules, travaillant initialement de manière indépendante et en alternance, ce qui a conduit à des commits séparés dans notre dépôt Git (se référer à notre code sur Github [8]).

Par la suite, durant la phase de debugging, nous avons trouvé plus productif de collaborer en binôme sur un même ordinateur, facilitant l'échange d'idées et l'optimisation de notre approche de debugging, d'où des commits effectués depuis un seul poste.

Enfin, la rédaction du rapport et la préparation de la présentation finale du projet ont également été menées en collaboration, consolidant ainsi notre travail d'équipe.

## 7 Conclusion

Le robot MazeRover que nous avons conçu et développé a pleinement atteint les objectifs que nous nous étions fixés en début de projet, à savoir être en capacité de naviguer de manière semi-autonome d'un point A à un point B dans le cadre des différents itinéraires possibles au sein du labyrinthe.

Malgré les défis techniques rencontrés, notamment les difficultés matérielles et les complications liées au multithreading, nous avons réussi à mettre en place un système robuste. Les tests approfondis dans tous les scénarios possibles, réalisés dans le labyrinthe en MDF que nous avons construit, ont été cruciaux. Ils nous ont non seulement permis de valider la fiabilité de notre robot dans des conditions réalistes, mais aussi de peaufiner sa capacité à naviguer et à détecter les obstacles de manière efficace, tout en révélant des pistes d'amélioration potentielles.

Ce projet a aussi marqué notre première expérience avec l'utilisation de ChibiOS, un système d'exploitation temps réel (RTOS), ainsi que des outils de gestion de version tels que Git et GitHub. Ces outils ont considérablement amélioré notre gestion du code et notre capacité à collaborer efficacement.

## Références

- [1] Page de garde : photo prise par les membres du groupe
- [2] Cours et Travaux Pratiques du cours MICRO-315 - Systèmes Embarqués et Robotique du professeur Francesco Mondada, Printemps 2024, École polytechnique fédérale de Lausanne (EPFL).
- [3] Datasheet de l'IMU MPU9250.  
<https://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf>
- [4] Inspirée de la solution du TP3 du cours MICRO-315 - Systèmes Embarqués et Robotique du professeur Francesco Mondada, Printemps 2024, École polytechnique fédérale de Lausanne (EPFL).  
[https://github.com/EPFL-MICRO-315/TPs-Student/tree/TP3\\_Solution](https://github.com/EPFL-MICRO-315/TPs-Student/tree/TP3_Solution)
- [5] Site internet de la documentation ChibiOS.  
<https://www.chibios.org/dokuwiki/doku.php?id=chibios:documentation:start>
- [6] Source d'inspiration et documentation pour l'implémentation initiale du Round Robin en tant que logique de yielding pour le multithreading.  
<https://www.playembedded.org/blog/the-complete-reference-for-multithreading-in-chibios->
- [7] Site internet du SKIL (Student Kreativity and Innovation Laboratory)  
<https://www.epfl.ch/labs/skil/>
- [8] Repository Github contenant le code source du projet sur la branche "RENDU\_MINIPROJET\_2024" :  
<https://github.com/EPFL-MICRO-315/tps-2024-group-abela-elqabli>