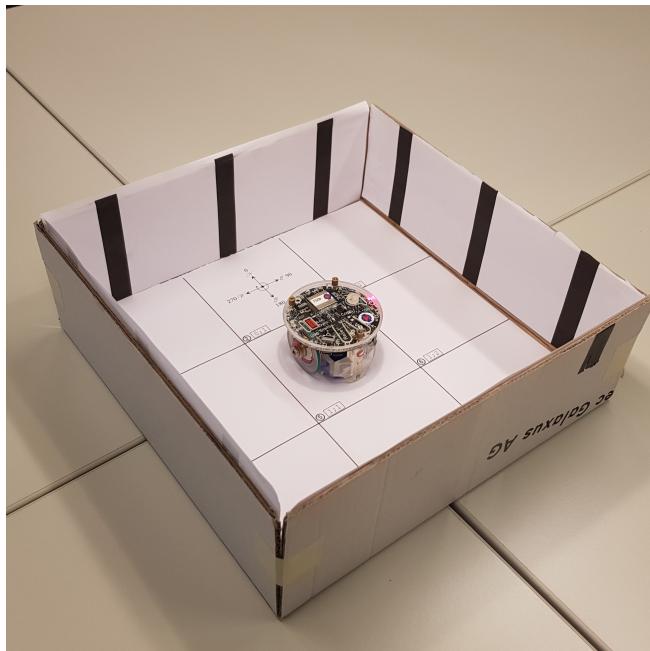


# Le Plotbot

Groupe 38 : Gieruc Théo, Krafft Guillaume

**Résumé:** Dans le cadre du cours *MICRO-315 Systèmes embarqués et robotique*, nous avons conduit un projet dont le but est de partir sur la base des éléments vu lors des TPs 1-5 pour créer plusieurs tâches plus complexes à résoudre par le robot e-puck [5]. Nous avons essayé de monter un projet complet utilisant un grand nombre de périphériques. Nous employons la caméra DCMI, les moteurs, les capteurs de proximité, le capteur TOF, les LEDs, le microphone, la carte microSD ainsi que le haut-parleur.



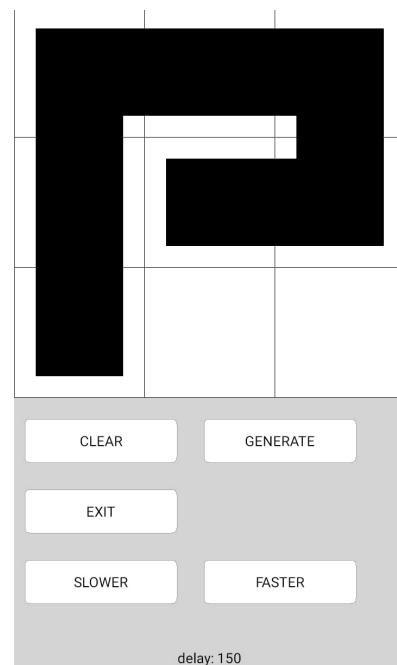
**Fig. 1:** La version finale du plan de jeu

## 1 Introduction

Notre projet PlotBot permet au robot e-puck<sup>2</sup> de reproduire des trajectoires sur un quadrillage 3x3. La trajectoire initiale est définie sur une application Android ou web (codée en javascript avec la librairie p5.js) qui transforme ensuite les coordonnées en signaux sonores (chaque fréquence représente une case). Le robot enregistre ces données, puis à l'aide de la caméra frontale, du capteur *time-of-flight*, des capteurs de proximité infrarouge et du microphone arrive à reproduire la trajectoire sur le quadrillage physique.

## 2 Mode d'emploi

- 1.Dessiner un chemin sur l'application, comme sur la Fig 2. Le robot peut parcourir un maximum de 15 cases et ne peut faire de déplacement en diagonale, comme la tour au jeu d'échecs.
- 2.Placer le robot à la position de départ du chemin en direction du nord sur le plateau de jeu (voir Figure 1).
- 3.Mettre le smartphone à proximité de l'e-puck, appuyer sur *GENERATE* afin de commencer la transmission de la séquence de position. Celle-ci se faisant de manière audio, il est important d'avoir un environnement silencieux.



**Fig. 2:** Capture d'écran de l'application Android, avec un chemin dessiné

- 4.Le robot avertit de manière sonore si la transmission s'est effectuée sans problème ou non. S'il n'y a pas eu d'erreur, l'e-puck reproduit le tracé sur le plateau de jeu.

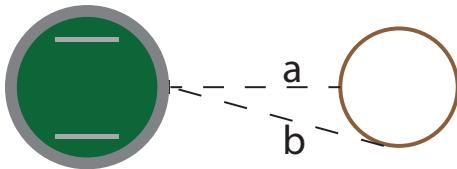
## 3 Application de traçage de chemin

Une application a dû être développée afin de transmettre le chemin voulu.

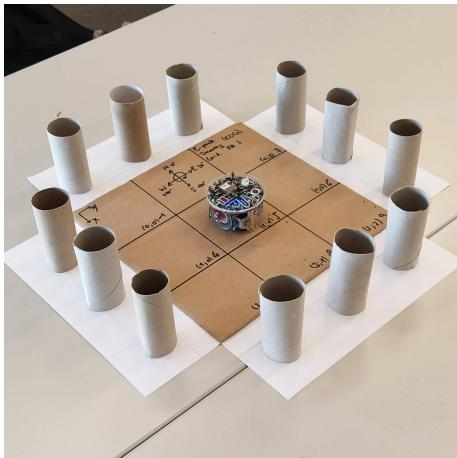
La première version était une application bureau sur Processing [4], un environnement de développement adapté à la création artistique. Nous avons ensuite décidé de l'adapter sous forme de webapp [3], permettant plus de flexibilité quant à l'appareil utilisé.

Finalement, à cause du manque de stabilité pour la génération de son du navigateur web, nous avons décidé d'en faire une application Android, visible à la Figure 2, à nouveau à l'aide de Processing.

Le principe de fonctionnement de l'application est simple : il faut tracer le chemin que l'on veut que l'e-puck suive puis appuyer sur *GENERATE*. L'application génère alors la suite de fréquences.



**Fig. 3:** La distance *a* est plus petite que la *b*



**Fig. 4:** La première version, avec les tubes en carton

## 4 Évolution du projet

Notre projet a sensiblement évolué depuis sa première itération. C'est plus précisément les méthodes et capteurs utilisés pour le repérage du robot qui ont été modifiés.

Notre première idée consistait à utiliser des tubes de carton placés à chaque extrémité des cases pour le repérage du robot comme illustré à la figure 4. Le TOF était utilisé et permettait à l'e-puck de s'arrêter sur le point le plus proche (d'où l'intérêt des cylindres, voir Figure 3) à une distance voulue. Cette méthode assez intéressante d'un point de vue technique et conceptuel n'a pas été retenue car l'utilisation seule du capteur *TOF* n'était pas suffisamment précise.

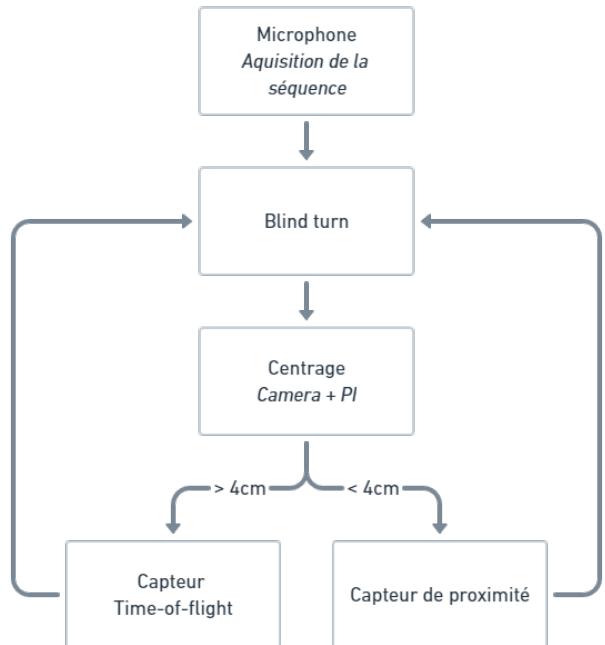
Nous avons donc décidé de remplacer chaque cylindre par une bande noire verticale. Ce n'est alors plus le TOF qui est utilisé par le robot pour s'orienter mais la caméra (comparable à ce qui a été fait dans le TP4 CamReg). Cela permet au robot de s'orienter beaucoup plus précisément. La figure 1 montre le plan de jeu final. Le TOF reste cependant utilisé pour permettre au robot de s'arrêter lorsqu'il se situe à une case du mur, là où les capteurs infrarouges seraient hors de portée. Ces derniers sont utilisés pour permettre au robot de s'arrêter proche des murs (env. 1cm), ils sont ici plus adaptés que le TOF, car ce dernier donne des résultats incohérents à partir d'environ 4cm.

## 5 Fonctionnement

Le fonctionnement du programme est illustré dans la Figure 5. En premier, l'e-puck attend de recevoir la séquence de positions. Une fois reçue, il effectue une rotation à l'aveugle puis utilise la caméra pour se centrer. Il avance ensuite droit jusqu'à la prochaine case. Il s'arrête soit en utilisant le capteur *TOF*, soit les capteurs de proximité.

### 5.1 Acquisition des positions

L'acquisition des positions a un thread dédié. On utilise la fonction `arm_cfft_f32()` vue lors du TP5 pour transformer ce que le microphone mesure en fréquence. Cette fonction utilise la partie



**Fig. 5:** Flowchart du fonctionnement du programme

Position	Fréquence uint8_t	Fréquence réelle [Hz]
1	17	265.625
2	20	312.5
3	23	359.375
4	26	406.25
5	29	453.125
6	32	500
7	35	546.875
8	38	593.75
9	41	640.625

**TABLE 1** Tableau des fréquences relatives aux positions

DSP du STM32F407, beaucoup plus optimisée pour ce type de calcul. Par souci de simplification et d'optimisation, nous utiliserons cette définition de la fréquence :

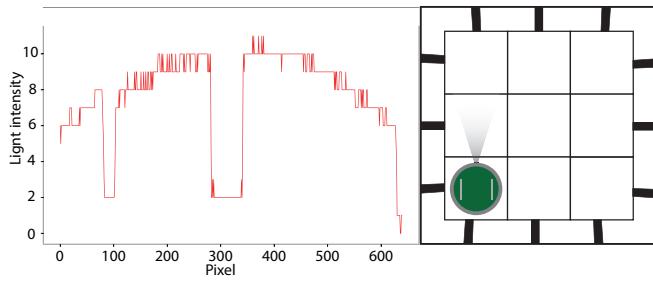
$$fréquence = \frac{fréquence\ réelle}{15.625}$$

Cela nous permet de stocker les fréquences dans des `int8_t`. Nous allons utiliser cette définition des fréquences dans la suite des explications.

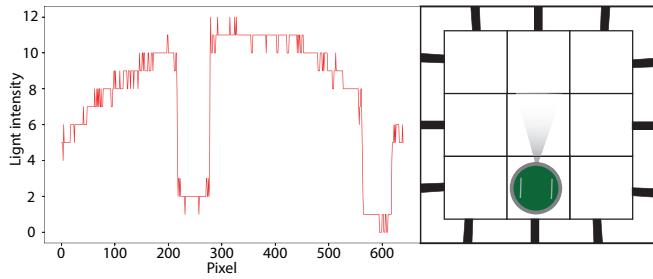
L'acquisition des positions se passe en deux phases : en premier l'e-puck attend de recevoir la séquence de départ. Une fois reçue, il enregistre chaque fréquence perçue jusqu'à ce qu'il reçoive la séquence de fin. Les fréquences assignées aux différentes cases vont de 17 à 41 et sont à chaque fois distantes de 3 afin d'avoir une meilleure robustesse, comme on peut le voir dans la Table 1. Les fréquences enregistrées sont alors transformées en position grâce cette formule qui tire profit de l'arrondi inférieur des `uint` :

$$position = \frac{freq + 5}{3} - 6$$

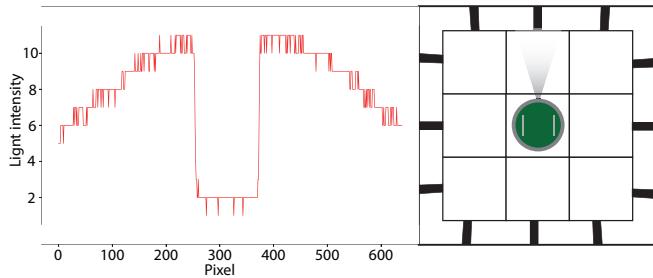
Nous utilisons dans la séquence de départ et celle de fin des fréquences en dehors de celles utilisées par les positions afin qu'il n'y ait pas de confusion possible.



**Fig. 6:** L'e-puck voit la ligne en face de lui ainsi que celle sur le côté.



**Fig. 7:** Même étant légèrement mal orienté, la ligne le plus au milieu est la bonne



**Fig. 8:** L'e-puck est suffisamment proche pour ne voir qu'une ligne, mais suffisamment éloigné pour la voir entièrement

## 5.2 Mouvements

Tout ce qui a trait au mouvement est dans un thread dédié. Un algorithme transforme chaque changement de position en un angle à effectuer et la distance que le robot aura par rapport au mur lorsqu'il aura fini son mouvement.

**5.2.1 Blind turn:** L'e-puck commence par tourner sur lui-même afin d'être dans la direction de sa prochaine position. Le nombre de pas à effectuer par moteur est calculé de la manière suivante :  $\frac{\pi \cdot \text{perimetre roues}}{\text{distance entre roues} \cdot 360} \cdot 1000$ . Il a néanmoins été nécessaire d'ajuster légèrement le ratio pour s'adapter à la réalité matérielle.

**5.2.2 Centrage:** L'e-puck se centre en utilisant la ligne noire face à lui (voir Figure 1), qu'il voit avec sa caméra DCMI. Dans le cas où il voit plusieurs lignes, il essaiera toujours de se centrer sur celle la plus au centre comme dans la Figure 7. Il est donc important que le blind turn effectué précédemment soit suffisamment précis. L'algorithme ne prend pas en compte les lignes moins larges que 30 pixels, comme dans la Figure 6. Dans le cas où le robot se trouve à une distance d'une case du mur, il ne verra qu'une seule ligne mais sera suffisamment éloigné pour la voir dans toute sa largeur, comme dans la Figure 8.

Un contrôleur PI avec un AWS est utilisé pour envoyer les commandes aux moteurs.

**5.2.3 Forward:** Le centrage est fait de manière suffisamment précise pour que l'e-puck puisse simplement avancer d'une case en ligne droite. Il s'arrête lorsqu'il est à une distance voulue du mur d'en face. S'il y a une case entre le but et le mur, l'e-puck utilise le capteur *Time-of-Flight*. Si son but est directement face à un mur, il utilise les capteurs de proximité. Parfois, un des deux capteurs de devant sera face à une ligne noire et recevra ainsi que peu de lumière. C'est pourquoi nous faisons la moyenne entre les deux capteurs de proximité de devant.

**5.2.4 Fin du mouvement:** Lorsque le chemin a été entièrement parcouru par l'e-puck, le robot s'arrête et un signal sonore indique la fin du programme.

## 5.3 Critique et points faibles

Il y a deux problèmes apparents dans la mise en oeuvre du projet : premièrement, la dépendance aux conditions de luminosité. Selon la lumière, les capteurs de proximité seront plus ou moins sensibles. Afin de contrer ce problème, nous plaçons une forte lampe au-dessus du plan de jeu afin d'avoir globalement toujours les mêmes conditions.

Le second défaut de notre projet est qu'une erreur de déplacement a un impact sur toute la suite des déplacements. Une possibilité d'amélioration serait d'utiliser un système de position absolu, à l'aide d'une caméra au-dessus du plan de jeu par exemple. Néanmoins, cela serait moins intéressant pour ce projet car l'e-puck n'aurait dès lors plus besoin d'utiliser sa caméra, ses capteurs de proximité et son capteur TOF.

## 6 Organisation du code

Le code est divisé en cinq fichiers source, chacun avec leur header. Toutes les fonctions uniquement utiles au fichier dans lequel elles se trouvent sont déclarées en local. L'utilisation des variables statiques globales est réduite au maximum, à savoir uniquement lorsque plusieurs fonctions et threads se partagent des variables au sein d'un même fichier, comme dans la librairie pour le capteur TOF. Nous avons également fait attention à utiliser le type de variable adapté, afin de ne pas utiliser inutilement trop de mémoire.

### 6.1 main.c/.h

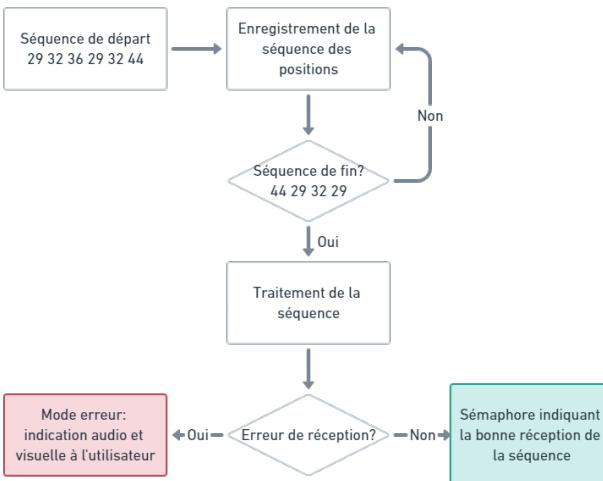
L'initiation des périphériques et threads se fait dans la fonction `main()`. Dans `main.h`, nous définissons si nous voulons que l'e-puck envoie des messages d'information par Serial à l'aide de `#define DEBUG`.

### 6.2 audio\_processing.c/.h

Le thread `ThdGetAudioSeq` s'occupe de toute la partie acquisition de la séquence de position. Le fonctionnement de ce thread est illustré à la Figure 9. Le thread attend la réception de la séquence de départ, puis enregistre chaque nouvelle fréquence jusqu'à la réception de la séquence de fin. Dans ce cas, il va vérifier si la réception s'est bien passée, c'est-à-dire si la séquence de position respecte les règles édictées au point 1 de la section 2. S'il n'y a pas d'erreur, le thread active le sémaforo `sequAquired`, qui indiquera aux autres threads que l'acquisition s'est terminée sans erreur.

### 6.3 process\_image.c/.h

Deux threads sont dans cette partie : `CaptureImage` et `ProcessImage`, comme dans le TP4. Le code diffère du TP dans `ProcessImage`, qui va non pas calculer la position de la barre noire mais calculer la position de la barre la plus au centre, tout en filtrant les barres trop fines. Il va ensuite activer le sémaforo `position_ready_sem` afin d'indiquer aux autres threads que l'image a été traitée.



**Fig. 9:** Flowchart du thread d'aquisition

#### 6.4 smartmove.c/h

Dans ce fichier se trouve le thread ThdSmartMove qui s'occupe du déplacement. Le fonctionnement global des déplacements a déjà été expliqué à la section 5.2. Ici, nous allons nous intéresser davantage aux interactions avec les autres threads.

Il attend d'abord que la séquence de positions ait été acquise par le thread ThdGetAudioSeq. Il l'importe à l'aide de la fonction publique de audio\_processing get\_sequ() qui passe la séquence par pointeur. Pour le calcul de l'erreur de centrage pour le PI (expliquée à la section 5.2.2), il attend d'abord de recevoir le sémaphore position\_ready\_sem, lui indiquant que la position vient d'être calculée.

La réception des valeurs du capteur *Time-of-Flight* se fait directement avec la fonction VL53L0X\_get\_dist\_mm(), tandis que la communication avec les capteurs de proximité se fait à l'aide du *message bus*.

#### 6.5 leds\_animation.c/h

Cette partie s'occupe de l'animation des leds selon les cinq modes définis dans leds\_animation.h :

1. *IDLE* : l'e-puck attend de recevoir la séquence de départ.
2. *LISTENING* : la séquence de départ a été reçue, la séquence des positions est en train d'être enregistrée.
3. *MOVING* : l'e-puck est en mouvement.
4. *DONE* : le chemin a été parcouru.
5. *ERROR\_MODE* : une erreur a été détectée dans l'acquisition de la séquence. L'e-puck reste dans ce mode et ne bouge plus.

Un thread est nécessaire afin d'avoir des animations dynamiques.

## 7 Conclusion

Nous avons réussi l'objectif que nous nous étions fixé au départ : tracer un chemin sur une application, la transférer à l'e-puck afin qu'il la reproduise. Nous avons identifié les points faibles de notre projet et proposé une manière d'y remédier.

Ce projet nous a permis de nous familiariser avec la programmation d'un système embarqué ainsi qu'avec l'utilisation d'un RTOS à l'aide de la librairie ChibiOS. Nous avons également utilisé notion.so [2], dont la fonctionnalité la plus intéressante est la

roadmap\* qui nous a grandement aidés à nous organiser. D'autres compétences transversales ont également été acquises : l'utilisation de Git [1] ainsi que la création d'applications avec Processing [4] et p5.js [3].

## 8 Références

- 1 [n.d.]. *GKrafft2/Projet-robotique-BA6*. <https://github.com/GKrafft2/Projet-robotique-BA6>
- 2 [n.d.]. *Notion – The all-in-one workspace for your notes, tasks, wikis, and databases*. <https://www.notion.so>
- 3 [n.d.]. *p5.js Web Editor | path generator*. <https://editor.p5js.org/tgieruc/sketches/f21FWcYWD>
- 4 [n.d.]. *Processing.org*. <https://processing.org/>
- 5 Francesco Mondada and Daniel Burnier. [n.d.]. Introduction aux miniprojets. ([n.d.]), 22.

\*. <https://www.notion.so/fe5f1266cde64489876d4f5c7741366av=4ed3c727ff5845d0b8ac1f45e488ccdd>