Projet Microinformatique

e-cat



Fahradin Mujovi 274633 Marwan El Chazli 272319

Le 8 Avril 2020



Table des matières

[1 Introduction 3](#_Toc39928510)

[2 Principe de fonctionnement 3](#_Toc39928511)

[2.1 Analyse de l’environnement 3](#_Toc39928512)

[2.1.1 Analyse sonore 3](#_Toc39928513)

[2.1.2 Analyse visuelle 4](#_Toc39928514)

[3 Modes de fonctionnement 5](#_Toc39928515)

[3.1 Comportement aléatoire périodique 5](#_Toc39928516)

[3.2 Modes d’interactions 5](#_Toc39928517)

[3.2.1 Mode indifférent : 5](#_Toc39928518)

[3.2.2 Mode amical : 5](#_Toc39928519)

[4 Organisation du code 5](#_Toc39928520)

[4.1 Modules 5](#_Toc39928521)

[4.1.1 Module de détection d’obstacles 5](#_Toc39928522)

[4.1.2 Module de mouvement 5](#_Toc39928523)

[4.1.3 Module d'émission sonore 5](#_Toc39928524)

[4.1.4 Module de traitement du son 6](#_Toc39928525)

[4.2 Interaction des threads 6](#_Toc39928526)

[5 Discussion 6](#_Toc39928527)

[6 Conclusion 7](#_Toc39928528)

[6.1 Problèmes constatés 7](#_Toc39928529)

[6.2 Améliorations possibles 8](#_Toc39928530)

[6.3 Commentaires 8](#_Toc39928531)

# Introduction

Notre e-cat simule le fonctionnement d’un chat. Il intègre différentes actions : mouvements aléatoires, miaulements, réactions aux obstacles selon son humeur et réaction au son. Nous l’avons nommé « e-cat », voulant dire *electronic-cat.*

# Principe de fonctionnement

## Analyse de l’environnement

Deux analyses de l’environnement sont effectuées continuellement et parallèlement par l’e-cat : une analyse sonore et une visuelle. Si l’une d’entre elles détecte un élément “critique” (une fréquence spécifique pour l’analyse sonore ou un obstacle trop proche pour l’analyse visuelle), elle fait basculer un booléen qui met en pause le fonctionnement des autres modules et fait réagir l’e-cat en conséquence

### Analyse sonore

Nous utilisons la phase détectée par le microphone de gauche et de droite pour déterminer la direction du son. L’e-cat se tourne dans la direction opposée du son et “s’échappe” (cf. Figure 1). Cette partie du projet a été une des plus compliquées à mettre en place, notamment à cause des problématiques suivantes :

1. **Limitation de la bande sonore interprétée**. Nos premiers tests avec les microphones, notamment dans le calcul de la phase, ont montré qu’il y a beaucoup d’imprécisions. Sans filtrage conséquent, l’e-cat réagissait par exemple facilement au son d’une conversation. Dans l’objectif de notre démonstration (où, dans le contexte d’un “robot chat”, nous souhaitons simuler une réaction à l’équivalent d’un aboiement de chien ou d’un autre son hostile), nous avons limité la détection et la réaction au son à un signal sonore de 250 Hz. Seul un signal aux environs de cette fréquence peut déclencher la réaction de l’e-cat. Pour la démonstration du projet, nous utilisons un signal sonore généré par un smartphone.
2. **Intervalle de confiance pour l’atteinte de l’objectif**. Une autre problématique majeure est la détection de l’arrivée à “l’objectif”. En effet, les imprécisions font que l’e-cat peut “penser” à un moment donné être arrivé à l’objectif (correspondant à une différence de phase nulle entre les deux microphones que nous utilisons) car les microphones détectent régulièrement des valeurs aberrantes. Afin de nous assurer que l’e-cat soit arrivé à destination, nous utilisons une variable de type *systime\_t* pour acquérir le temps au début de notre boucle while{1} dans le thread *speed\_processing*. Ensuite, nous faisons simplement en sorte de vérifier que l’e-cat soit resté dans l’angle GOAL\_ANGLE assez longtemps, en l’occurrence 500 ms, (valeur trouvée expérimentalement) pour considérer qu’il oscille bien autour de l’objectif et non pas à cause de valeurs issues de bruit.
3. **Symétrie gênante avec seulement deux microphones utilisés**. (cf. Figure 2) La dernière grande problématique que nous avons eu à régler était le fait qu’il existe des symétries dans les phases entre certains positionnements dans l’espace des sources de sons (cf. Figure 2). Nous avons réussi à régler ce souci en utilisant le module *speed\_regulator*. Celui-ci a pour rôle de simplement dire à l’e-cat de tourner CW quand il y a un déphasage positif, et de tourner CCW quand il y a un déphasage négatif. Le fait de n’utiliser que deux microphones implique qu’avec une condition posée sur l’objectif d’arrivée du type « rendre le déphasage analysé égal à 0 », nous avons deux positions finales possibles. Cependant une des deux positions est instable et ne sera pas considérée par l’e-cat comme objectif atteint (cf. Figure 2 pour de meilleures explications).

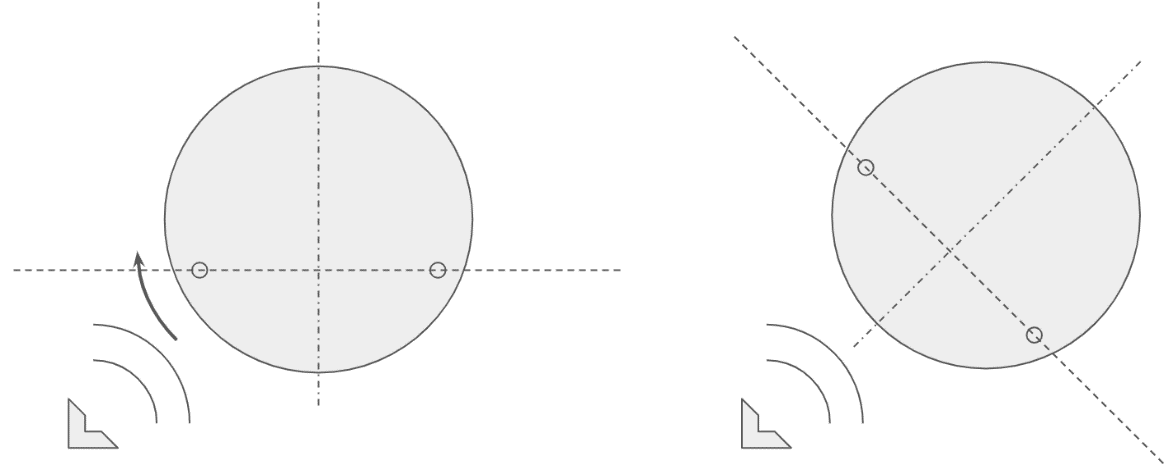


Figure 1 - Illustration d’une rotation de l’e-cat en provenance d’un son. Les deux petits cercles représentent les deux micros utilisés, et le symbole en bas à gauche représente une source de son. Deux axes sont importants à remarquer : L’axe de symétrie entre les deux micros, et l’axe commun entre ceux-ci. L’axe de symétrie défini la limite entre les phases positives et les phases négatives. L’axe commun définit une symétrie dans les phases obtenues (mieux expliqué dans la Figure 2).

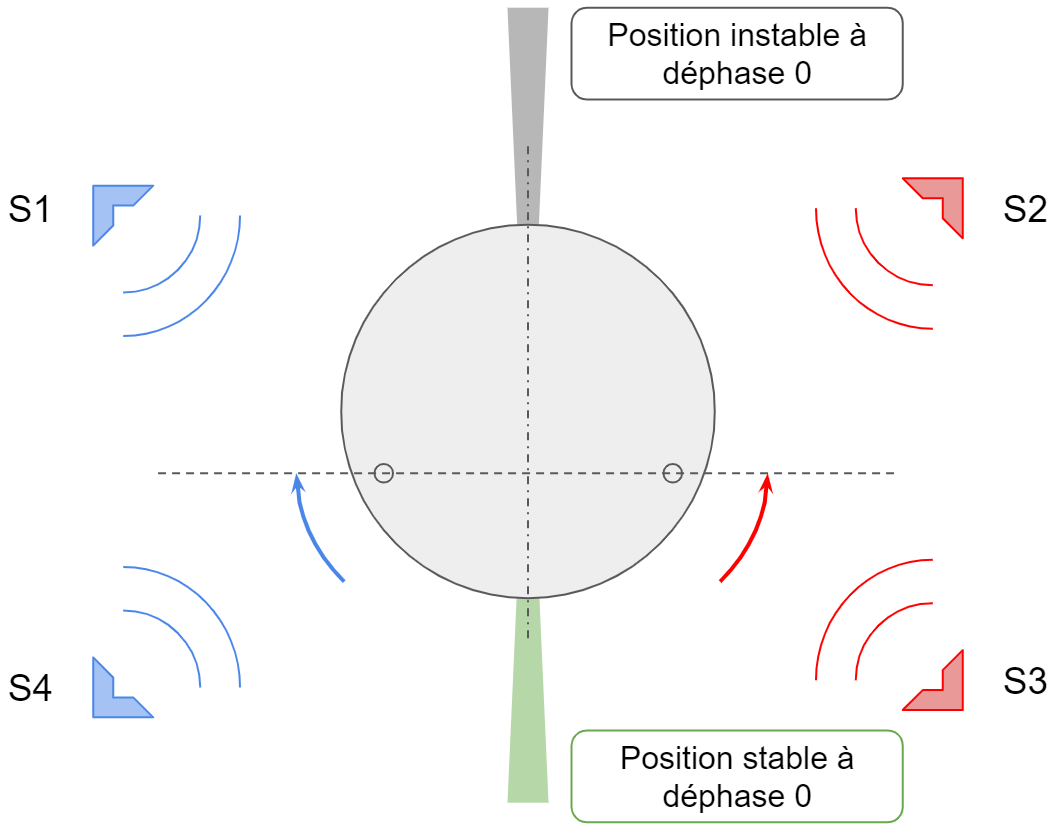


Figure 2 - Illustration plus détaillée du comportement selon la direction de la provenance du son. L’axe de symétrie illustre bien le fait que S1 = -S2 et S4 = -S3. Alors que l’axe commun montre bien le fait que S4 = S1 et S3 = S2. Les cônes gris et vert représentent la zone où le déphasage serait nul. Cependant, avec le type d’implémentation que nous avons, le cône gris est une zone de positionnement instable alors que la zone verte devient une zone de positionnement stable. Les flèches illustrent le comportement de l’e-cat pour les deux sources de son de leur couleur. Pour rappel, il faut que la source de son arrive dans un des deux cônes pendant plus de 500 ms pour que l’e-cat considère que l’objectif est atteint. Étant donné que le cône gris est une position instable, l’e-cat ne restera pas sur celui-ci plus de 500 ms et ne le considèrera donc pas comme objectif atteint.

### Analyse visuelle

**Détection d’objets** : nous utilisons les 8 capteurs IR pour détecter les obstacles. Un capteur détecte un obstacle si la distance retournée est en dessous d’une certaine constante ; les valeurs de chaque capteur sont traitées afin de déterminer non seulement lequel est face à un obstacle mais aussi lequel est le plus proche d’un obstacle. À ce moment, le robot se tourne dans la direction opposée à ce capteur pour s’éloigner (dans le mode “indifférent” décrit plus loin) ou cherche à se coller à l’obstacle (dans le mode “amical”).

**Problème de détection des obstacles bas**. Un problème que nous avons rencontré dans la détection d’obstacles est le fait que les capteurs sont directionnels et ne peuvent pas détecter des obstacles qui ne sont pas dans leur ligne de champ : l’e-cat se retrouvera donc à tenter sans succès de passer outre cet obstacle jusqu’à ce qu’un mouvement aléatoire du thread *Mover* dans une autre direction n’intervienne. Nous avons remarqué que nos prédécesseurs ont tenté de régler ce problème en pliant les pins de connexion de certains des capteurs, pratique que nous n’avons pas souhaité perpétuer pour ne pas endommager lesdits capteurs.

# Modes de fonctionnement

## Comportement aléatoire périodique

L’absence d’interventions extérieures (obstacles ou signaux sonores), le thread *Mover* contrôle le déplacement du robot, en implémentant une suite de mouvements prédéfinis (avancer, tourner, s’arrêter quelques secondes) qui s'exécutent d’une manière aléatoire (une fonction *randomizer* choisit une action à chaque fois qu’elle est appelée). Ces mouvements visent à simuler le comportement d’un chat sans interactions, qui peut parfois sembler aléatoire.

## Modes d’interactions

Selon le mode, l’e-cat se comportera d’une manière différente à la détection d’obstacles. La détection de son se fera en parallèle et monopolisera l’e-cat si un son à 250 Hz est détecté.

### Mode indifférent :

Avec le *selector* en position 1, l’e-cat cherchera à éviter les obstacles. Dans ce mode de fonctionnement, les interactions possibles seront de le faire fuir en approchant un obstacle de lui, et de le “pousser” par derrière pour le faire miauler et “sauter” en avant.

### Mode amical :

Avec le *selector* en position 2, l’e-cat veut de l’attention et des câlins : il cherche donc à se frotter aux obstacles qu’il détecte. En postant la main à proximité, l’e-cat va se coller à celle-ci et faire de petits mouvements de rotation.

# Organisation du code

## Modules

### Module de détection d’obstacles

Le thread “*Eyes*” utilise la fonction du module *proximity* qui permet d’obtenir les valeurs retournées par les 8 capteurs IR. Selon la position du *selector*, le comportement est tour à tour axé sur l’évitement d’obstacles ou au contraire la recherche d’obstacles. Le thread “*Eyes*” est HIGHPRIO*,* donc la plus grande priorité dans notre programme, ceci afin d’éviter que le robot n’engage un mouvement aléatoire alors qu’il devrait être en train de réagir à un obstacle.

### Module de mouvement

Le principe de fonctionnement est basé sur une fonction nommée *randomizer*, qui comme son nom l’indique retourne une valeur au hasard entre 0 et la valeur entrée en paramètre. Cette fonction utilise le module RNG de l’e-cat. Grâce à elle, nous simulons le comportement d’un chat en faisant passer dans un ordre aléatoire un certain nombre de mouvement préprogrammés.

### Module d'émission sonore

Notre robot émet plusieurs sons caractéristiques d’un chat. Ces sons sont enregistrés sur une carte SD et utilisés grâce au module *audio\_emitter*. Ces enregistrements sont donc ensuite ponctuellement joués par différentes parties du programme, selon les réactions du robot.

### Module de traitement du son

Nous souhaitons que notre robot détecte certaines fréquences : nous avons donc recyclé les modules *FFT* et *audio\_process* des TPs afin de les adapter à nos besoins. Le principe est de capter si un son à 250 Hz est présent et d’analyser le déphasage entre différents microphones. Ce déphasage est mis en lien avec les roues de l’e-cat afin de permettre à celui-ci de le minimiser en tournant sur lui-même (cf. Figure 1 pour plus d’explications).

## Interaction des threads

1. Mover : gère les mouvements aléatoires prédéfinis en cas d’inactivité du robot dans les autres threads. Pour ne pas stopper l’activité des autres threads (ex : éviter un obstacle), ce thread a une priorité plus basse que *Eyes* et *SpeedRegulator* et est donc paramétré en NORMALPRIO.
2. SpeedRegulator : gère la direction dans laquelle le robot se tournera, selon le déphasage perçut par les microphones gauche et droit.
3. Eyes : gère la détection d’obstacles puis la réaction de l’e-cat à ces obstacles (soit sous le mode “amical” ou le mode “indifférent”). Ce thread est le plus important car dans le mode “indifférent” il permet au e-cat d’éviter de se heurter, parfois assez rapidement, à des obstacles ; il est donc en HIGHPRIO.

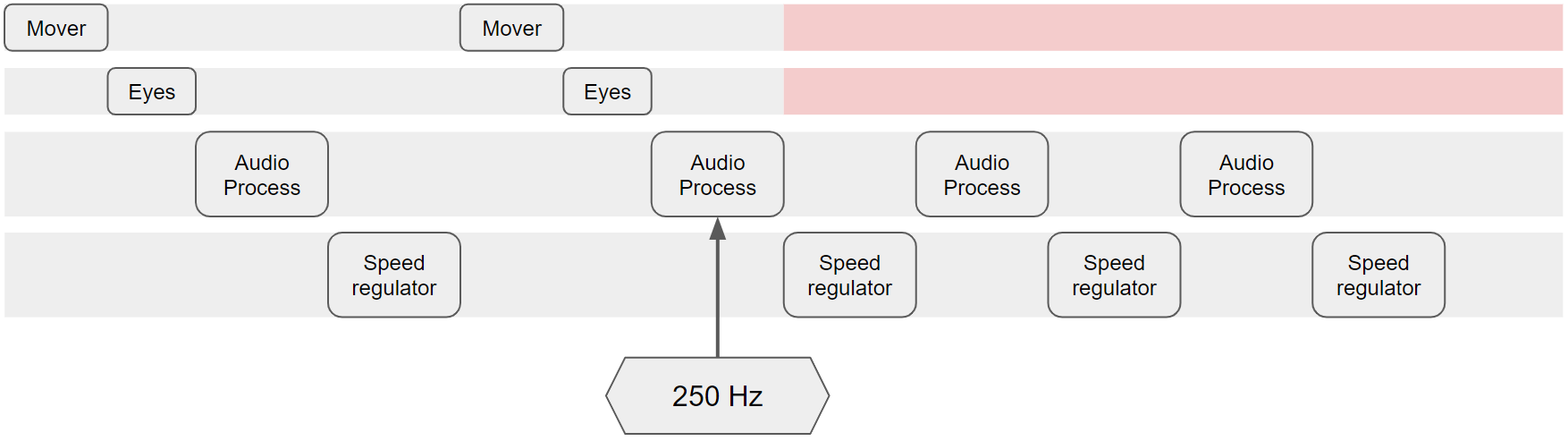


Figure 3 - Gestion des threads Audio Process et Speed regulator (sans la prise en compte de la boucle while dans le main). Les threads tournent normalement jusqu'à que le thread Audio Processing détecte un son à une fréquence 250 Hz. À ce moment, les threads Moves et Eyes sont arrêtées jusqu’à ce que Audio Process finisse la réorientation de l’e-cat selon la provenance du son.

# Discussion

**Détection d’obstacles** : plusieurs enjeux étaient présents lors de la mise en place de ce module. Notamment, il y a le fait que la détection et la réaction aux obstacles devait être prioritaire sur les autres modules, dont celui des mouvements aléatoires. Afin d’y parvenir, nous avons d’abord tenté d’utiliser les fonctions de *chThreads* avec des pointeurs sur les threads : celles-ci étaient toutefois problématiques car, plutôt que de mettre la priorité sur un thread, elles annulent dans une grande mesure le fonctionnement de ce dernier. Pour remédier à cela, nous avons fait le choix de mettre d’abord le thread d’obstacles en HIGHPRIO, puis de créer une variable statique dans le module de la détection qui permet de savoir à tout moment si une réaction à un obstacle est en cours. Si c’est le cas, les conditions qui permettent au robot de faire quoique ce soit d’autre ne sont pas validées et les autres actions sont donc mis en *stand-by*.

**Traitement du son** : ce module a été le plus compliqué à mettre en place. Pour cause, la grande quantité d’imprécision des capteurs sonores, qui détectent forcément n’importe quel bruit (cf. Figure 4) et donc peuvent à tout moment recevoir des fréquences dans des intervalles très grands. Pour remédier à cela, nous avons dû mettre en place un filtrage des valeurs reçues par les micros. Un premier filtre nous a permis d’éliminer les mesures de phases issues de différentes fréquences entre les micros, par exemple, si le micro droit analysait un son à 400 Hz alors que celui de gauche analysait un son à 250 Hz, cette mesure de déphasage était automatiquement ignorée. En combinant ce dernier filtre avec un autre filtre qui va globalement nettoyer les fréquences trop différentes de 250 Hz, nous avons pu faire en sorte de laisser passer les sons qui étaient à 250 Hz avec un certain intervalle de confiance absolu et un autre intervalle de confiance relatif entre les deux microphones. Un dernier filtre nous a permis d’éliminer les valeurs hasardeuses dans les déphasages entre micros calculés. Après expérimentation et sous les conseils des assistants, nous éliminons tous les déphasages en dehors de [-0.5, 0.5] radians. Tous ces filtres se résument à des conditions “if” qui changent l’état d’un booléen nommé SPEED\_PROCESS\_BOOL.

Quelques imprécisions subsistent toutefois encore et un des impacts les plus problématiques des imprécisions de mesure est l’apparition d’oscillations du e-cat à l’approche de l’objectif. Pour limiter cette oscillation, au moins à l’arrivée du robot à son objectif, nous avons mis un intervalle relativement large de GOAL\_ANGLE : même si nous perdons quelque peu en précision, nous nous assurons qu’une fois l’e-cat arrivé à son objectif d’angle, il ne bougera pas de manière incessante. Cette solution nous paraît adéquate, étant donné que l’objectif principal de ce module était d’approximativement orienter l’e-cat à l’opposé d’une source de son.

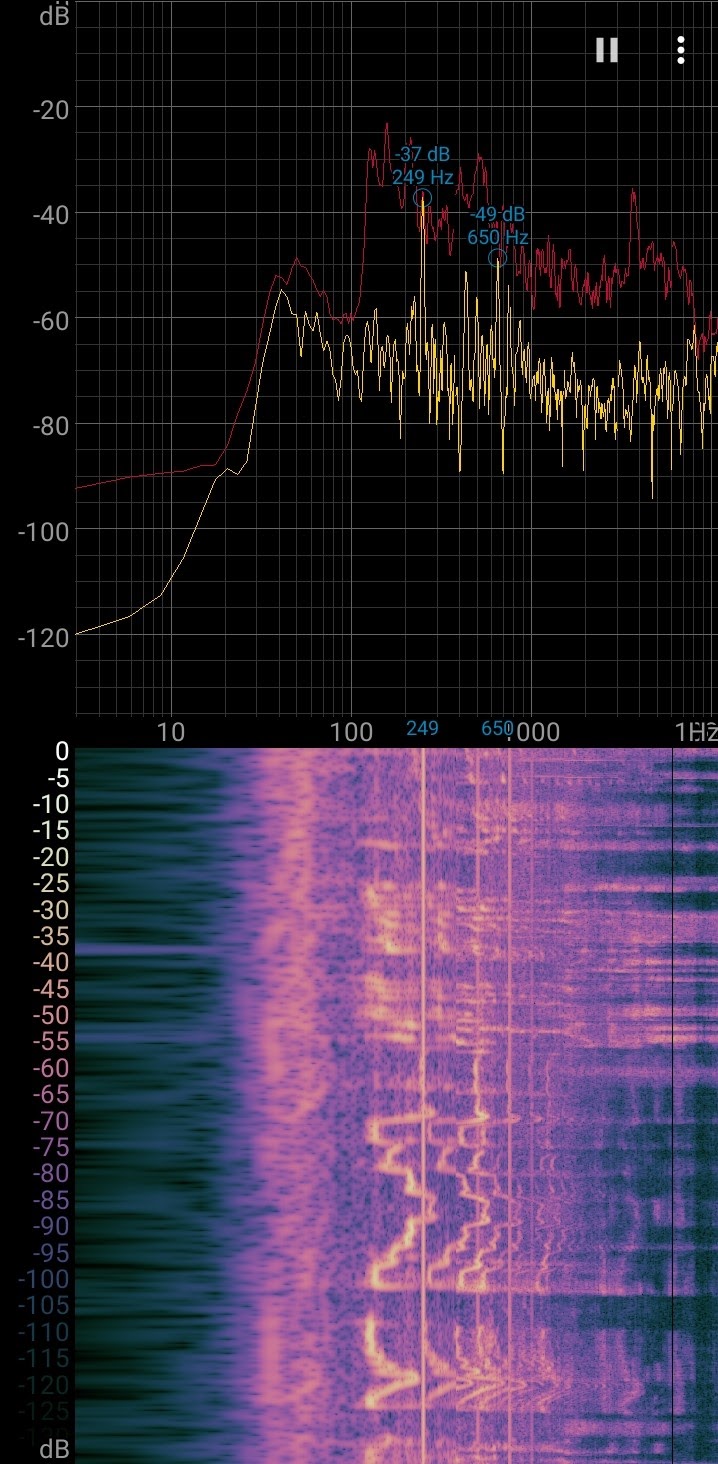
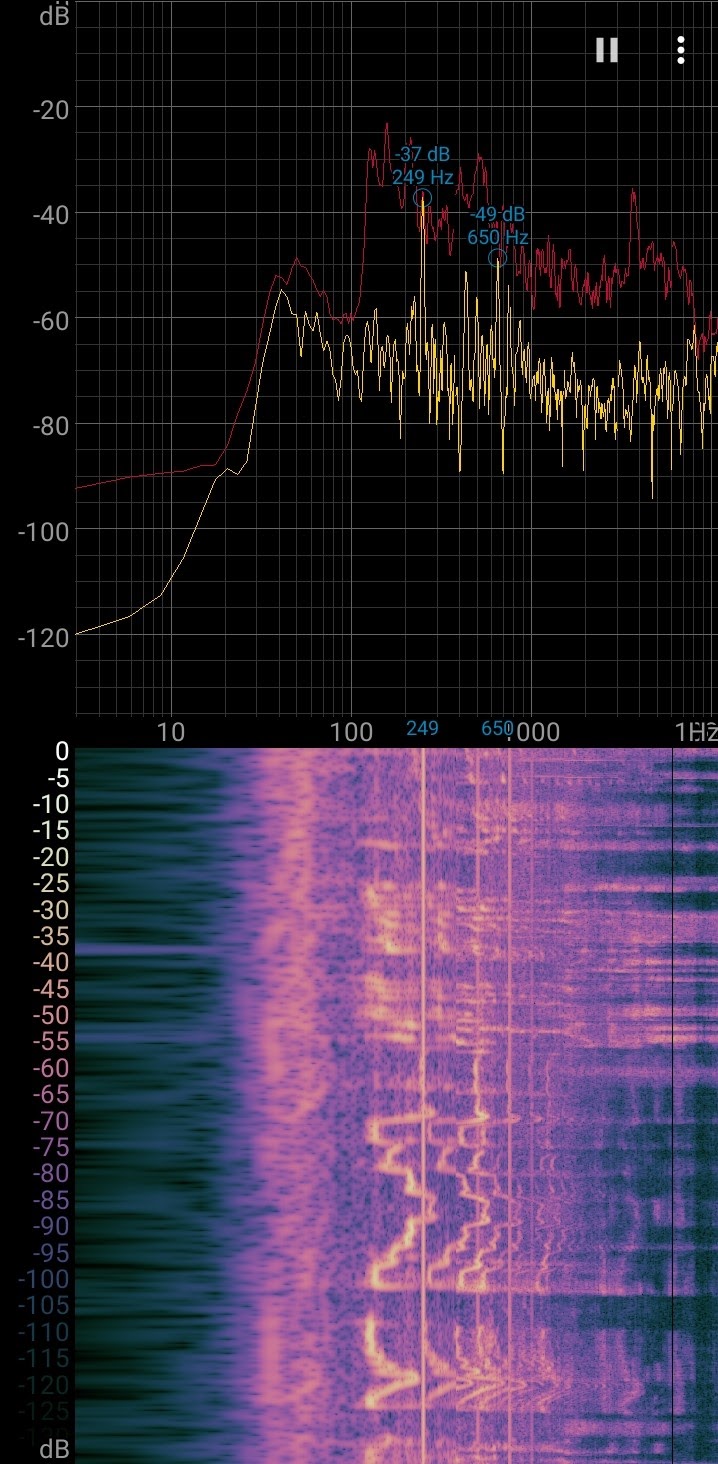


Figure 4 - Analyse du spectre sonore fait avec l’application Android Spectroid.

La partie du gauche montre, avec la couleur jaune, une représentation directe du spectre analysé, alors que la partie rouge montre les plus hautes valeurs. On note qu’il y a eu du bruit dépassant l’amplitude maximale du son à 250 Hz.

La partie de droite montre le spectrogramme du son analysé pendant un certain temps. Il est important de noter qu’il y a du bruit comparable en amplitude à notre son à 250 Hz. L’échelle horizontale est partagée entre les deux graphes et représente les fréquences des sons analysés et les échelles verticales représentent l’amplitude des sons analysés.

En constante présence de la fréquence à 250 Hz (249 Hz, certainement dû à des imprécisions dû à des limitation matérielles soit des mesures, soit de génération du son), on voit qu’il y a beaucoup de bruit malgré un environnement calme. Cela prouve qu’il était bel et bien nécessaire d’implémenter des filtres.

# Conclusion

## Problèmes constatés

Nous avons constaté que les filtres n’étaient peut-être pas optimaux. En effet, après tests, le filtre de fréquences relatif entre les deux microphones n’est peut-être pas nécessaire. Un filtre ne laissant passer que les fréquences à environ 250 Hz aurait certainement suffit.

La détection de son n’est pas aussi fiable que nous l’aurions voulu : la déduction de direction à prendre nous est satisfaisante, mais la portée de cette détection est assez petite. Nous n’avons pas trouvé comment améliorer ce point.

## Améliorations possibles

L’intégration d’un contrôleur PID gérant la vitesse des roues est une addition qui aurait rendu plus fluides et surtout naturels les mouvements de l’e-cat.

Une autre amélioration aurait été de rendre le changement de mode de comportement plus interactif, par exemple, il aurait été possible de changer de mode en présence d’un son spécifique.

L’utilisation des pointeurs de threads aurait été plus efficace, nous aurions pu mieux optimiser les relais entre les threads et limiter notre utilisation de la fonction *ThdSleep*.

Il aurait été idéal de trouver un moyen d’optimiser (si possible) le système de traitement audio afin d’augmenter sa fréquence de répétition. Cela nous aurait permis d’améliorer la qualité et la précision de notre tracking.

## Commentaires

Nous avons essayé d’intégrer un contrôleur PID pour la gestion de la rotation à la réaction des 250 Hz, mais nous avons dû abandonner l’idée à cause de la présence constante de valeurs aberrantes malgré les filtres mis en place. Cela causait des accélérations intempestives.

Nous avons intégré le calcul de différents déphasages avant de réaliser qu’ils ne seraient pas forcément utiles. Nous avons décidé de les laisser dans le code pour l’aspect démonstratif du projet.

Finalement, nous avons songé à optimiser la taille allouée aux différents threads. Par exemple, pour le thread *Eyes*, il nous est possible de réduire la taille de 256 à 128 bytes sans problèmes. Nous avons cependant décidé de ne pas le faire étant donné que nous n’étions pas proches de la limite de mémoire de la pile.