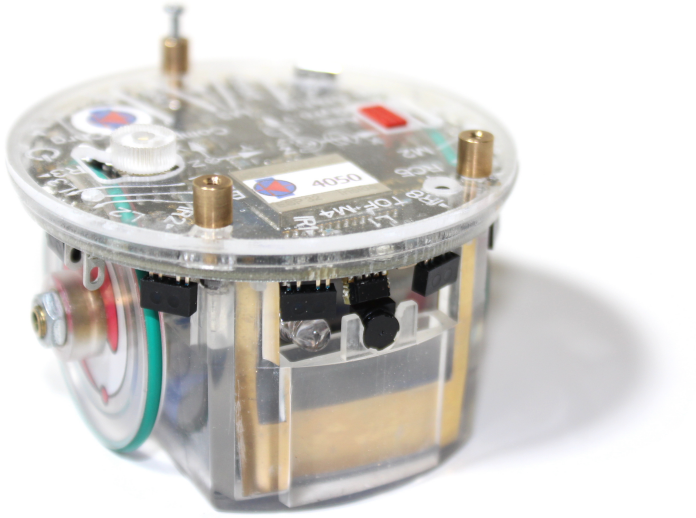
Microinformatique - Mini Projet

Professeur : Francesco Mondada

Simulation d'une bulle dans un liquide



Groupe 34

Aladin Guillaume — Sciper: 250361
Hémon Christopher — Sciper: 250668

Semestre académique - Printemps 2019

Contents

1 Présentation du projet

1.1 Cahier des charges

Ce projet a pour objectif de mettre en pratique les différentes bases de programmation du robot *E-puck 2* acquises lors des travaux pratiques du cours de Micro-informatique. Le cahier des charges impose l'utilisation de certains modules du robot, à partir desquels les étudiants sont libres d'implémenter une idée.

Les éléments imposés par le cahier des charges sont les suivants :

- Le projet doit être construit sur la base de la librairie e-puck2_main-processor vue lors des travaux pratiques 4 et 5.
- Les deux moteurs pas-à-pas doivent être utilisés.
- Un des deux types de capteurs de distance, à savoir, infrarouges ou time-of-flight, doit être utilisé.
- Un capteur parmi ceux utilisés durant les travaux pratiques doit être implémenté. À savoir, la caméra, les microphones ou le module *IMU*.
- Chaque capteur ou actuateur doit être géré par un thread dédié.
- Le code doit être rendu sous la forme d'une librairie de fichier .c et .h qui s'intègrent avec la librairie du système*Chibios*.

1.2 Notre idée: *Bubble*

Nous avons choisi d'utiliser le robot afin de lui faire simuler les mouvements d'une bulle qui remonterait dans un liquide. Après avoir été initialisé, le robot est placé sur un plan incliné qu'il remonte en contournant des obstacles. Le robot doit être conçu de manière à rester coincé lorsque l'obstacle qu'il rencontre est concave. De la même manière, lorsque le robot rencontre une structure convexe, la manœuvre d'évitement doit ressembler au glissement que ferait une bulle le long d'un obstacle, au détail prêt que le robot ne touche jamais les obstacles mais se contente de les frôler.

Une fonction supplémentaire à été ajoutée à cette simulation. Le robot doit être capable de réagir à un son et s'en éloigner. Cela correspondrait à un courant dans le liquide, qui viendrait affecter les mouvements de la bulle. L'idée est de pouvoir « souffler » sur le robot, ce qui permet de le forcer à redescendre légèrement la pente afin de se décoincer d'un obstacle concave.

Nous utilisons les données de l'accéléromètre pour diriger le robot vers le haut de la pente, les capteurs infrarouges pour détecter des obstacles, ainsi que les microphones pour détecter la provenance d'un son.

2 Implémentation

2.1 Modélisation

Notre première approche consistait à imaginer la physique qui agit sur une bulle dans un fluide. Nous avons commencé par définir un vecteur *gravité inverse* qui donne à la bulle la direction du haut du fluide. Puis nous avons rajouté un *vecteur obstacle* qui représente la force de répulsion perpendiculaire que subit la bulle lorsqu'elle atteint un obstacle. De plus, nous voulions ajouter une fonction de déviation à la bulle lorsqu'elle détecte la provenance d'un son. Cette fonctionnalité se traduit par un troisième élément: *le vecteur son*. La somme de ces vecteurs donne la *direction à suivre* au robot.

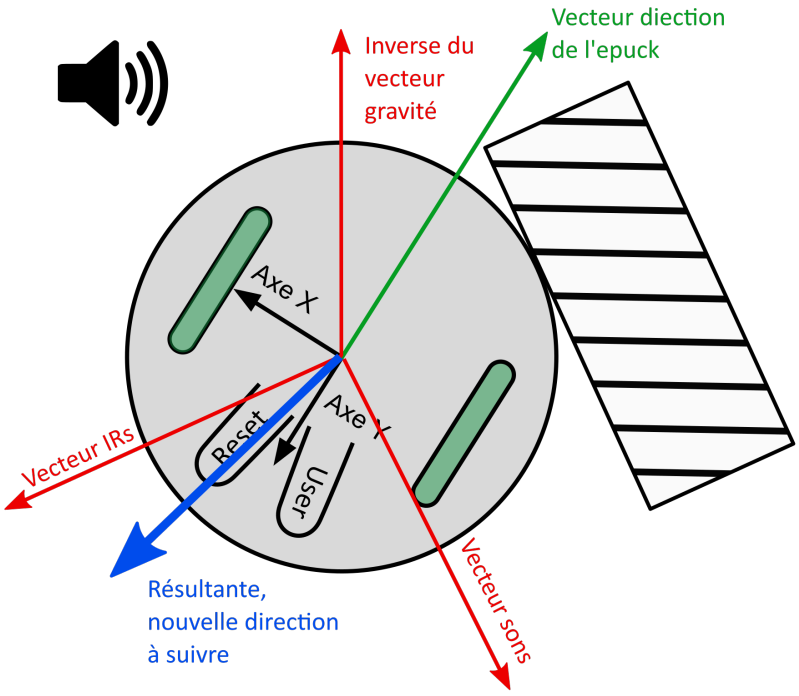


Figure 1: Modélisation d'une bulle soumise aux trois contraintes

La figure 1 présente la modélisation physique d'une bulle dans un fluide que nous avons imaginée pour construire notre projet.

Une fonction de contrôle des moteurs a ensuite été imaginée. Son rôle était de traduire la différence entre le vecteur *résultante* (en bleu sur la figure 1) et la *direction actuelle de l'e-puck* (en vert sur la figure 1) en terme de vitesses moteur pour réorienter en permanence l'e-puck et le faire suivre la trajectoire prévue.

Le problème majeur de cette approche était qu'il impliquait de calculer des tangentes pour générer les vecteurs. Le programme devait ensuite prendre l'inverse de ces tangentes pour calculer les vitesses moteur correspondantes. Cette manoeuvre impliquait l'utilisation de *LookUp Tables* et augmentait le coût calcul de notre algorithme.

Afin de palier à ce défaut, nous avons choisi de projeter les vecteurs imaginés précédemment sur l'axe x. En effet, du point de vue des moteurs, seules des directions gauche/droite sont nécessaires pour corriger sa trajectoire. Plutôt que de calculer un vecteur avec un angle, nous séparons les influences des événements qui ont lieu à gauche et à droite, et nous corrigeons séparément les vitesses des deux moteurs. Par exemple, un obstacle détecté par les infrarouges de droite impliquerait un ralentissement du moteur gauche de l'e-puck afin de le faire pivoter à gauche pour éviter l'obstacle.

Pour implémenter ceci, nous avons choisi une approche *« divide and conquer »*. Nous avons séparé la simulation de la bulle en plusieurs comportements et avons créé les fonctions nécessaires pour chaque comportement.

Le premier comportement est la montée de la bulle vers le haut du plan incliné. Il faut donc que l'e-puck puisse reconnaître une pente, s'orienter correctement et avancer contre la pente. Nous utilisons les mesures de

l'accéléromètre du module IMU et des fonctions de logique pour orienter l'e-puck en direction de la pente. Pour obtenir un mouvement fluide, nous calculons la nouvelle vitesse des moteurs de manière incrémentale. Nous effectuons une somme pondérée de l'ancienne vitesse des moteurs avec la nouvelle vitesse calculée. De cette manière, le changement de vitesse s'effectue sur plus de cycles et donne un mouvement fluide au robot. Pour assurer une vitesse nulle lorsque l'e-puck est à plat, nous devons définir des tolérances où le robot ne réagit pas. En effet, en utilisant *RealTerm* on observe que les valeurs oscillent entre plus et moins 0.7 m/s^2 lorsque le robot est posé sur une surface horizontale. Cela correspond à peu prêt au basculement de l'e-puck autour de l'axe des roues. Nous imposons une tolérance légèrement supérieure à cette valeur de basculement sur l'axe x. Nous avons choisi 10% de la valeur de la gravité pour les deux axes, soit 0.981 m/s^2. La figure 2 illustre cette tolérance autour des deux axes de mesures de l'e-puck, qui correspond à environ 9° de jeu. Tant que le robot se trouve à l'intérieur de ces tolérances, la vitesse des moteurs est fixée à zéro.

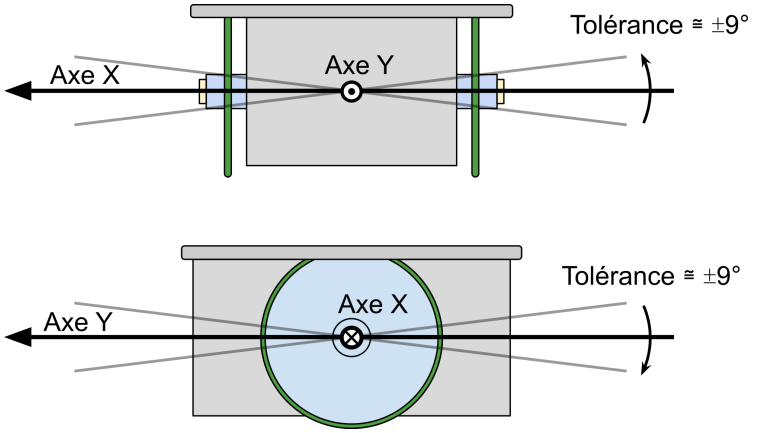


Figure 2: Angles de basculement de l'e-puck

Le deuxième comportement est celui de l'évitement d'obstacles. Les événements détectés par les capteurs infrarouges sur le côté droite du robot engendrent un ralentissement du moteur gauche de ce dernier et vice-versa. Les capteurs situés à l'avant ont un facteur de freinage plus important que ceux situés à 45° et que les capteurs latéraux. Tous les capteurs situés à l'arrière ne sont pas utilisés. Un rayon de détection de 1 cm autour de l'e-puck est souhaité. Pour ce faire, nous avons créé une fonction de détection d'obstacles avec une valeur de seuil mesurée empiriquement à l'aide de *RealTerm*.

Dans la réalité, une bulle ne peut pas revenir en arrière lorsqu'elle rencontre un obstacle. Elle ne peut que s'y coincer ou l'éviter en glissant dessus. Pour cette raison, nous avons implémenté une fonction qui se charge de surveiller le robot, afin de garantir que ce dernier ne se retourne pas complètement et reparte vers le bas, même si les capteurs de proximité venaient à lui dicter de le faire.

Le dernier comportement correspond au cas où un son vient dévier le déplacement du robot. Nous simulons cela avec les microphones de l'e-puck. Nous utilisons la fréquence de 406 Hz pour le faire entrer dans le mode de déviation sonore. La fréquence choisie et une analyse spectrale constante permettent d'éviter des irrégularités dans la détection. Pour déterminer l'orientation de la source sonore et ainsi déterminer la direction dans lequel le robot doit bouger, nous implémentons une fonction simplifiée de comparaison de l'amplitude sonore sur les quatre microphones.

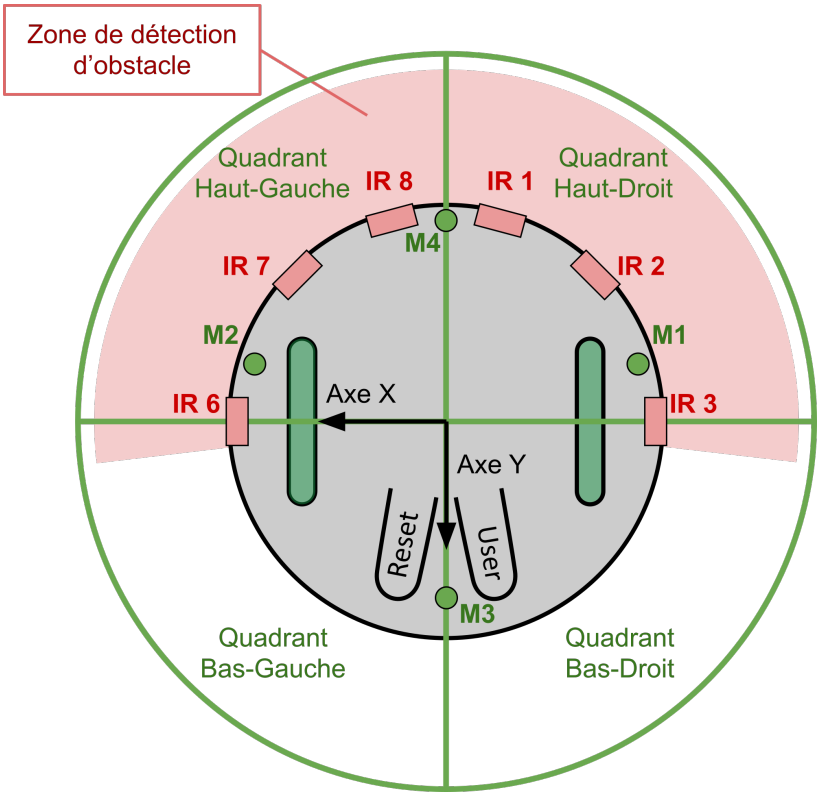


Figure 3: Schéma des senseurs de l'e-puck

La figure 3 illustre la position des six capteurs infrarouges utilisés, la zone de détection d'objets, les quatre quadrants de détection sonore ainsi que la position des microphones et les axes des accéléromètres.

2.2 Architecture logicielle

2.2.1 Définitions des Threads

Voici la liste des Threads et des fonctions utilisés dans ce projet:

- Thread waBubble : ce thread est le thread principal. Il gère toute la simulation de l'e-puck. Il fonctionne à une fréquence de 100 Hz et fait le lien entre les capteurs en entrée et les actuateurs en sortie du robot. La mémoire réservée à ce thread est de 128 octets car elle contient l'espace mémoire nécessaire à toutes les autres fonctions de notre code.

- Threads secondaires : ces threads sont au nombre de trois et ont été nommés « threads internes » (déjà définis dans la librairie *e-puck2_main-processor*). Ils assurent la mesure fréquente des capteurs. Nous avons proximity.c, imu.c et microphones.c qui, lorsqu'ils sont initialisés, lancent leurs threads respectifs.

- Fonction bubble_motion_start : cette fonction lance le thread de simulation waBubble.

- Fonction motion_ascend : cette fonction prend les mesures de l'accéléromètre et réoriente le robot afin qu'il soit en face de la pente. Cette fonction peut également ajouter l'élément de mouvement linéaire

si nécessaire, e qui fera monter le robot vers la pente lorsqu'il lui fait face s'il respecte les tolérances définies sur l'axe x.

- Fonction motion_avoid : cette fonction prend les mesures des capteurs infrarouges et freine les moteurs proportionnellement aux valeurs de ces mesures. Cette fonction est appelée seulement si un obstacle est présent dans la zone de détection lorsque le robot remonte la pente. Elle est également appelée lorsque le robot est « soufflé » dans une direction, après détection sonore.

- Fonction motion_blowturn : cette fonction tourne le robot de 45° ou 135° à gauche ou à droite dépendant du quadrant de détection de son.

- Fonction motion_facing : cette fonction répond une valeur logique positive ou négative si l'e-puck est face à la pente ou non. Elle est utilisée dans différents cas pour faire entrer et sortir le robot du mode « bloqué » et/ou du mode « soufflé ».

- Fonction motion_obj_alert : cette fonction logique répond *oui* ou *non* lorsqu'un objet est perçu ou non dans la zone de détection d'obstacle à l'avant de l'e-puck. Si la réponse est *oui*, le robot sort du mode « acsension » et entre en mode « évitement ».

- Fonction processAudioData : cette fonction est appelée par main.c lors de l'initialisation des microphones. Il s'agit majoritairement de la fonction d'analyse audio du TP5, mais c'est également dans cette fonction que toute l'évaluation de la position de la source du son a lieu. Une comparaison entre la magnitude des données des micros gauche/droit et avant/arrière permet de déterminer le quadrant source. Un élément de filtrage est également mis en place. Il assure que la fréquence de 406 Hz est émise pendant plus de 1 seconde avant d'évaluer le quadrant. Cela permet de rejeter une grande majorité des détections accidentelles.

- Fonction audio_get_status : cette fonction retourne le nouveau statut du robot ainsi que le quadrant source lors du cas où le robot est en mode « soufflé ».

- Fonction animation : cette fonction est purement visuelle et sans intérêt pour la simulation de la bulle. Elle anime le robot avec les LEDs rgb en fonction du mode dans lequel se trouve la simulation.

2.2.2 Organisation des fichiers du code

Notre code est séparé en quatre fichiers sources et la fonction main du fichier main.c.

Main.c est utilisé pour initialiser ChibiOS, ainsi que les communications et les threads internes des capteurs de l'e-puck. Il appelle également la fonction qui démarre le thread principal waBubble gérant toute la simulation.

Les fichiers audio.c et motion.c gèrent tous les *inputs*. Eux seuls ont accès respectivement à microphones.h, imu.h et proxitmity.h . Ils gèrent directement les mesures brutes des capteurs utilisées par l'e-puck et font tout le pré-traitement nécessaire de ces données.

Le fichier source bubble.c contient le thread principal de simulation. C'est lui qui gère tous les modes de comportement de la bulle. C'est ce fichier qui fait le lien entre les *inputs* et *l'output* du robot. C'est également

l'unique fichier à avoir accès à motors.h. C'est par conséquent lui qui met à jour la vitesse calculée du robot.

Le fichier animation.c sert simplement à ajouter un élément visuel à la simulation, en animant les diverses LEDs de l'e-puck en fonction de l'évolution du robot.

2.2.3 Synthèse graphique

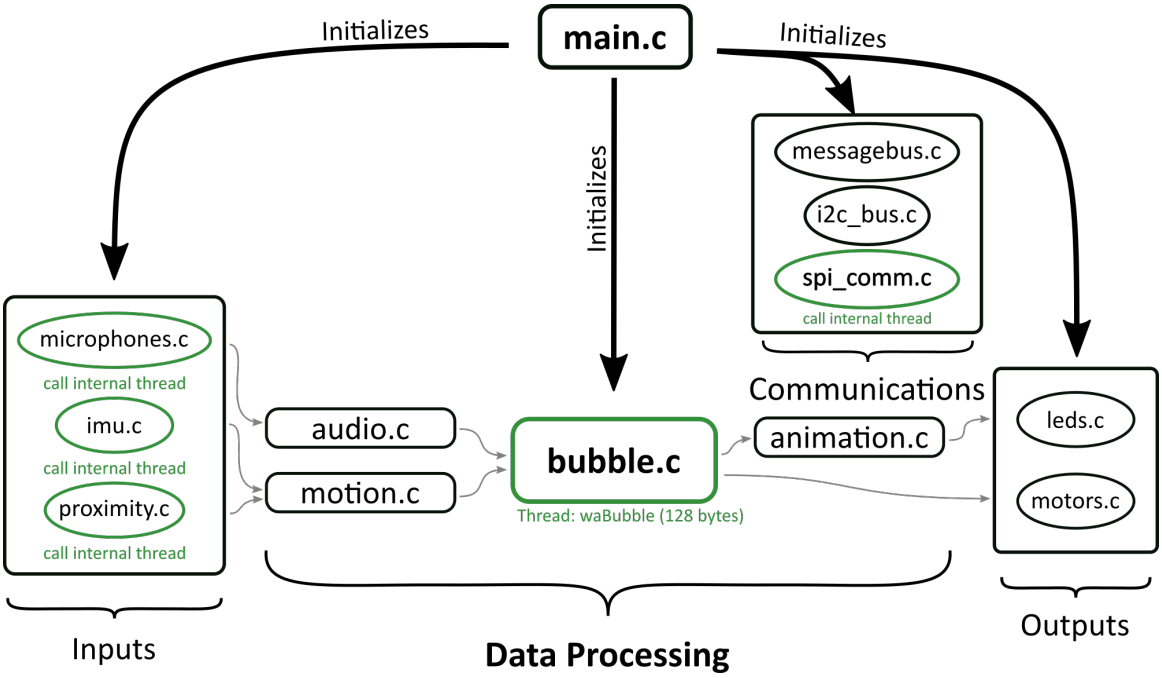


Figure 4: Arbre des appels des fichiers

La figure 4 présente l'arbre des appels de fichiers. En vert sont relevés les fichiers qui définissent ou font appels à des *threads*. La plupart d'entre eux sont déjà définis dans la libraire de base (notés *internal threads* sur cette image).

3 Résultats et Conclusion

3.1 Utilisation de la mémoire

Nous avons fait des efforts pour limiter le coût mémoire de notre code, dans le but de le rendre le plus léger possible et réduire le temps d'exécution du programme. En effet, nous utilisons un grand nombre des ressources présentes dans la librairies de l'e-puck et ne définissons qu'un seul thread pour toute l'implémentation de notre simulation. Nous avons réduit au strict minimum le nombre de variables globales par fichier source et avons obtenu un coût mémoire total de 55 octets selon nos calculs. Plus de la moitié de l'espace mémoire est utilisée pour gérer les fonctions audios qui sont imposantes en terme de coût mémoire. Néamoins, notre thread principal waBubble réserve un espace mémoire de seulement 128 octets. Une valeur inférieure engendrerait des situations de « panic » ; notamment lorsque le robot entre en mode « bulle soufflée ».

3.2 Aspect qualitatif de la démonstration finale

Notre démonstration illustre tous les aspects de notre simulation : la détection d'obstacles, la remontée du pan incliné, le blocage du robot face à des obstacles concaves ainsi que la déviation induite par un son dans la direction opposée à la source de ce dernier.

Il y a cependant plusieurs éléments de notre simulation qui ne correspondent pas exactement au comportement réel d'une bulle. D'une part, les conditions de blocage sont très sensibles aux changements faibles d'accélération qui arrivent lors du déplacement du robot; notamment lors des décélérations rapides qui ont lieu lorsque le robot arrive face à un mur. Pour corriger cela, nous avons dû augmenter la tolérance de blocage d'un facteur 3 (déterminé empiriquement). L'effet secondaire de cette correction fut que lors d'évitement d'obstacle, le robot se bloque plus tardivement dans certaines situations. De plus il ne se bloque plus systématiquement devant des obstacles concaves. D'autre part, la détection de son se faisant par quadrants, un mouvement parfaitement linéaire du robot est impossible. Ce dernier s'éloigne du son en zigzaguant; ce qui ne correspond pas exactement au mouvement linéaire qu'aurait effectué une bulle.

Néanmoins, l'utilisation d'une mise à jour incrémentale des vitesses des moteurs, pondérées par les anciennes vitesses, permet d'obtenir un mouvement assez fluide, ce qui est visuellement agréable et simule bien les mouvements d'une bulle. De plus, la transition en sortie d'obstacle paraît cohérente. En effet, le robot décroche de l'obstacle assez tôt, ce qui ressemble relativement bien à une bulle.

3.3 Améliorations

Comme mentionné dans le paragraphe *2.1 Modélisation* , nous avons choisi de ne pas utiliser des vecteurs pures mais seulement une projection de ces derniers. Cette décision a permis d'éviter de travailler avec des *LookUp Tables*. Elle a cependant engendré d'autres complexités de code qui auraient pu être évitées avec des vecteurs. En effet, une multitude de conditions ont dû être rajoutées pour traiter des singularités provenant de l'axe y. (Exemple, si le robot se retrouve parfaitement tête en bas, il faut le forcer à tourner). Si ce projet était à refaire, il serait plus judicieux de s'en tenir à l'utilisation de vecteurs.

Une autre amélioration serait à faire au niveau de la correction des vitesses des moteurs. En effet, il serait envisageable de freiner un moteur et d'accélérer le deuxième de la même valeur. Cela permettrait de donner au robot un fonctionnement antagoniste. Il pivoterait alors plus sur son centre.

3.4 Conclusion

En conclusion, ce projet nous a permis de découvrir la programmation du point de vue de la robotique de manière très concrète. Cela a été pour nous une occasion de synthétiser les différentes connaissances acquises jusqu'alors en microtechnique à travers toutes les étapes de fonctionnement d'un système. Nous avons ainsi pu développer une vision globale de tous les éléments qui interviennent lorsqu'on réalise ce type de système, à savoir, capteurs, électronique, encodage, traitement des données, modélisation et compréhension de la physique, algorithmes et programmation, commande, régulation, et finalement, résultats dans le monde réel.

De plus, ce projet nous a permis de réaliser qu'il est tout à fait possible de créer des systèmes complexes, interagissant avec de nombreuses données, à condition de bien décomposer le problème en sous-fonctions et modules, représentés sur papier, comme le propose la méthode *Divide and Conquer* acquise en première année.