Université Pierre et Marie Curie

Projet SAR

Kilobot

Implémentation d'algorithmes pour les cohortes de robots

Auteurs :
Arnaud Guermont
Benjamin Bielle

Superviseur :
Dr. Swan Dubois

4 mai 2015

Sommaire

Table des figures

Chapitre 1

Introduction

Contexte

Kilobot est une plate-forme constituée de petits robots autonomes. Cette plate-forme est proposée par l'université d'Harvard [1].
Ces robots se déplacent uniquement par vibration et communiquent par infrarouges.

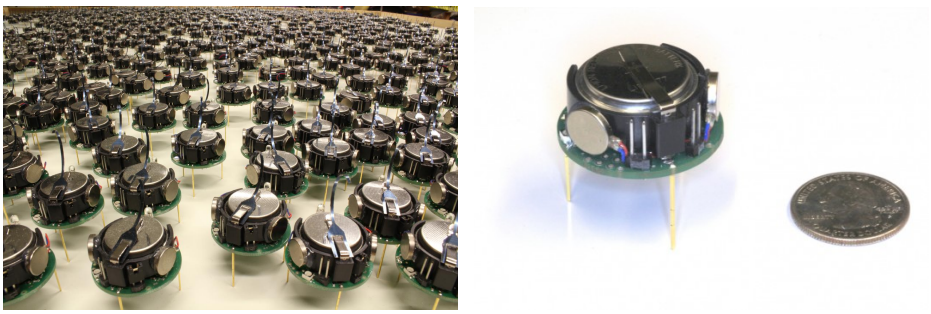

FIGURE 1.1 – Robots Kilobot

Le projet a pour objectif d'implémenter des algorithmes répartis sur un essaim de robots (la plate-forme **Kilobot**).
Il se découpe en trois phases, la première consiste en une recherche documentaire et une prise en main de la plate-forme matérielle, la seconde concerne l'implémentation d'algorithmes (dans notre cas de bio-algorithmes). Enfin la dernière phase se concentre sur la recherche des possibilités d'implémentation d'un modèle : le **modèle CORDA** [2].

L'existant

Il existe deux API de base pour la plate-forme Kilobot :

- ***K-team*** [3].
- ***Kilobotics*** [4].

1. http ://www.eecs.harvard.edu/ssr/projects/progSA/kilobot.html
2. Giuseppe Prencipe. A new distributed model to control and coordinate a set of autonomous mobile robots : The corda model, 2000.
3. http ://www.k-team.com/mobile-robotics-products/kilobot
4. https ://www.kilobotics.com/

Celle proposée par ***K-team*** (fournie par défaut avec les robots) nous permet une moins grande liberté que celle de ***Kilobotics***.
Donc notre ***API*** se basera sur celle-ci car elle nous garantit une plus grande liberté, portabilité et extensibilité.

Veille Technologique

Il existe plusieurs projets similaires aux kilobots, par exemple le projet ***I-Swarm***[5] (trop limités techniquement), mais seule la plate-forme ***Kilobot*** permet de manipuler des centaines de robots de manière réaliste hors d'un simulateur (avec un prix convenable).
Le projet ***E-Swarm***[6] propose une plate-forme intéressante mais le prix de chaque robot en fait une plate-forme non éligible pour ce projet.

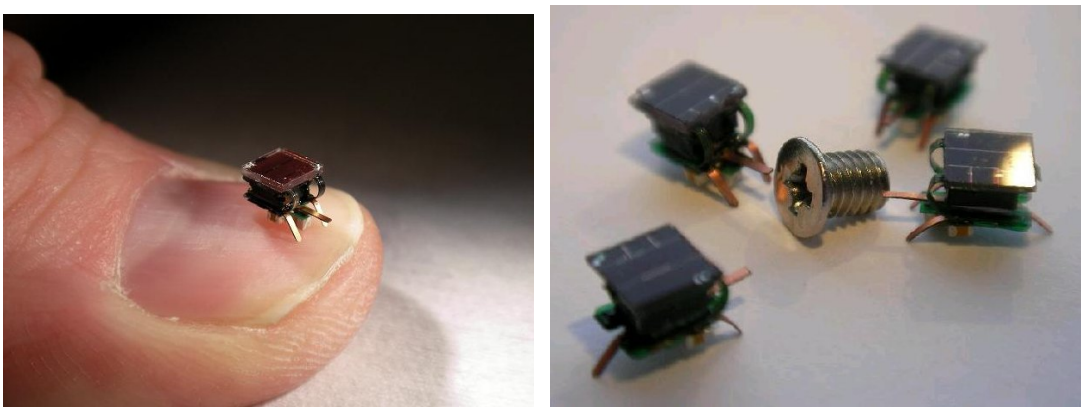

FIGURE 1.2 – Robots I-Swarm

FIGURE 1.3 – Robots E-Swarm

5. http ://www.hizook.com/blog/2009/08/29/i-swarm-micro-robots-realized-impressive-full-system-integration
6. http ://www.e-swarm.org/research$_m ain.php$

Chapitre 2

Spécifications

Matérielles

Le but du modèle est de représenter une grande quantité de robots (presque un essaim) pour pouvoir effectuer des tâches collectives qui seraient impossibles à réaliser en solitaire. De nombreux exemples dans la nature mettent en action ce principe : les fourmis peuvent explorer des zones d'un périmètre de plusieurs kilomètres par exemple.
Dans la robotique, il existe des algorithmes divers telle-que l'auto-assemblage, les constructions collectives ou encore l'exploration.
Ces algorithmes sont faits pour être testés sur plusieurs centaines voir milliers d'individus. Cependant, pour des raisons de coûts, de temps ou encore de complexité, il est parfois uniquement possible de les valider en simulation.
C'est pourquoi un modèle comme celui que les kilobots implémentent fut créé. Les robots devaient avoir plusieurs capacités : communiquer et percevoir leur environnement par l'intermédiaire de capteurs, connaître la distance entre eux et se déplacer en avant, tourner sur la gauche et sur la droite. De plus, ils devaient aussi pouvoir tous être contrôlés ensemble simultanément par un seul opérateur. Mais la principale contrainte restait le coût. En effet, il était très dur auparavant de tester des algorithmes sur des essaims de robots qui coûtaient plusieurs centaines de dollars à l'unité.

Communication et perception de l'environnement

De nombreux algorithmes collectifs utilisent la communication entre robots et la distance entre ces même robots pour piloter individuellement le comportement de chaque robot. Il est donc indispensable que les kilobots puissent communiquer entre eux ainsi que mesurer la distance qui les sépare l'un de l'autre.
Pour qu'ils puissent communiquer, chaque kilobot est pourvu d'une led pour transmettre et d'une photodiode infrarouge comme récepteur qui sont toutes les deux placées en-dessous au centre du robot. En étant placé au centre, le récepteur peut recevoir des messages d'une égale distance dans toutes les directions. Les messages sont envoyés par la led et utilisent la réflexion du support sur lequel ils évoluent pour transmettre (c'est pourquoi le support doit être réfléchissant comme un tableau blanc ou un miroir par exemple). En utilisant cette technique, un kilobot peut transmettre dans un rayon de 7 centimètres à un débit de 30 kb/s. La méthode d'accès CSMA/CA est utilisée pour éviter les collisions (tous les robots émettent sur le même canal).
Une led RGB est aussi située sur le côté pour pouvoir informer l'utilisateur.

Les kilobots sont aussi pourvus d'un capteur de lumière permettant de mesurer l'intensité lumineuse de l'environnement du robot.

Mesure de la distance

Le calcul de la distance entre un robot et un autre se fait via l'envoi et la réception du message. L'intensité de la lumière émise par la led diminue proportionnellement avec la distance qu'elle parcourt. Il ne reste alors plus qu'au récepteur à calculer la distance en fonction de l'intensité du signal reçu. Comme le signal peut être affecté par l'environnement, la précision est de +/- 2 mm.

Déplacement

L'une des principales capacités des robots est de pouvoir se mouvoir dans leur environnement. L'une des solutions les plus simples et efficaces est d'utiliser un design à deux roues, chaque roue étant contrôlée par un moteur. Mais cette solution est aussi très chère et ne permettra pas de réaliser un robot à bas coûts.
Pour y remédier, les kilobots disposent de trois jambes rigides et de deux moteurs à vibration disposés de chaque côté du robot. Lorsqu'un des moteurs est activé, il génère une force centripète qui est convertie en une force vers l'avant transmise à la jambe situé en dessous du moteur. Grâce au positionnement décentré de chaque moteur, la mise en marche de l'un ou l'autre va causer une rotation vers la gauche ou la droite en fonction du moteur. En activant les deux moteurs en même temps et en contrôlant la magnitude de leur vibration, ils est alors possible de faire avancer le robot en ligne droite vers l'avant. Il peut ainsi se déplacer à une vitesse d'environ 1 cm/s et pivoter à environ 45 $^\circ$ /s.

Contrôle des robots par l'opérateur

Lorsqu'on utilise un très grand nombre de robots, le chargement de leur programme, leur mise en marche ou encore le rechargement de leur alimentation peut vite prendre beaucoup de temps pour l'utilisateur : émettons l'hypothèse qu'il faille 3 secondes pour allumer un robot via un interrupteur. Pour un groupe de 1000 robots, cela prendrait au total 50 minutes, ce qui est bien trop long pour être efficace.
Les kilobots sont donc conçus pour être utilisés à grande échelle par une seule personne.

La communication entre l'opérateur et les robots passe par un contrôleur. Le contrôleur est une plateforme disposant d'une connexion usb pour interagir avec l'ordinateur de l'utilisateur et de multiples led pour envoyer des messages aux kilobots. Chaque kilobot dispose sur son dos d'un récepteur pour la réception des signaux. Le contrôleur est l'interface entre l'opérateur et l'essaim de robots. Disposé à une hauteur d'un mètre, il couvre un périmètre d'un mètre. Il permet d'effectuer de multiples opérations comme la mise en veille, le réveil, le chargement d'un nouveau programme ou encore l'arrêt du programme en cours d'exécution. Il est aussi utilisé pour la calibration des moteurs des kilobots.

Le problème évoqué auparavant aurait aussi pu s'appliquer aux kilobots. Pour y remédier, les robots disposent d'un mode veille. Le microcontrôleur peut désactiver le régulateur de tension pour les moteurs et aussi celui pour le système de communication (led et récepteur). Il ne reste plus que celui pour le microcontrôleur principal. Dans cet état de veille, le microcontrôleur va tester toutes les minutes pendant 10 ms s'il a reçu un message de réveil. Si c'est le cas, il réalimente alors les régulateurs de tension et le robot redevient de nouveau opérationnel. Ce mode veille permet à un kilobot d'avoir une autonomie de 3 mois. Mettre en veille un groupe ne prend que quelques secondes et le réveil s'effectue en moins d'une minute. Les kilobots disposent aussi d'un interrupteur coupant toute alimentation du robot si ces derniers venaient à ne pas être utilisés pendant une longue période.

Le rechargement de l'alimentation des robots est aussi un facteur de temps. Les kilobots disposent d'une batterie au lithium-ion de 3.4 V 160 mAh. Elle leurs assure une autonomie entre 3 et 24 heures selon l'usage qui en est fait. Le rechargement de celle-ci est aussi adapté à la manipulation d'un grand nombre de robots. Pour cela, les kilobots utilisent deux de leurs trois jambes comme conducteur : l'utilisateur place les jambes des robots en contact avec un courant électrique de 6V via une plaque métallique ou un

tube métallique et connecte le crochet situé sur le dessus du robot à la masse. Cela permet le rechargement de plusieurs kilobots simultanément sans intervention sur le robot. Par exemple, l'utilisation d'une plaque métallique comme chargeur et d'une autre plaque qui viendrait se poser au dessus des kilobots est une solution abordable et adaptable à très grande échelle. Lorsque le chargement est terminé, la led RGB des kilobots en informe l'utilisateur.

Le chargement d'un programme s'effectue via le contrôleur. Lorsque l'opérateur souhaite charger un nouveau programme, il envoie un message aux kilobots pour qu'ils placent leur pointeur de programme dans le bootloader. Ce dernier dispose d'un programme qui va alors recevoir le nouveau programme de l'utilisateur et le placer dans la mémoire primaire. Le bootloader va ensuite vérifier que ce programme ne dispose pas d'erreur puis ensuite redémarrer le kilobot qui va alors exécuter le nouveau programme. Cette méthode permet de programmer un essaim de robots en environs 35 secondes.

Coûts

La principale contrainte lorsqu'on implémente un modèle à large échelle est le coût. En effet, les robots actuels dépassent les centaines de dollars à l'unité et ne sont donc pas envisageables.
Les kilobots ont été conçus dans le but de ne pas être chères à construire. Le coût à l'unité pour 1000 robots est de 14 dollars, ce qui est environ dix fois moins que les autres robots conçus pour les algorithmes collectifs.
Pour cela, les robots disposent d'un moyen de locomotion par vibration comme vu auparavant et sont constitués de pièces simples et peut coûteuses :

- Déplacement : 3,12 $
- Alimentation : 3,61 $
- Communication/Capteur : 2,20 $
- Contrôleur : 2,83 $
- Structure : 1,55 $
- Divers : 0,74 $
- Total : 14,05 $

L'assemblage des robots est aussi très simple et rapide (la plupart des composants sont montés à la surface du PCB) et peut être réalisé par un robot pour un coût d'environ 5 $ par robot (pour 1000 robots). Cela prend donc environ 5 minutes pour obtenir un robot fonctionnel à partir des composants.

Logicielles

Simulation

Lorsqu'on implémente des algorithmes collectifs, on désire parfois les simuler avant de pouvoir les tester grandeur nature sur les robots (ici les kilobots) afin d'éviter les erreurs d'implémentation ou de se retrouver avec un programme qui implémente mal l'algorithme.

V-Rep

Pour cela, nous avons à notre disposition le simulateur ***V-Rep*** de la société ***Coppelia Robotics***[1] qui est un simulateur spécialisé dans le domaine de la robotique. Il implémente des modèles 3d de l'ensemble des robots existants et il intègre aussi un créateur de modèle 3d afin de pouvoir simuler des prototypes. Il dispose aussi d'un moteur physique permettant de vérifier la viabilité de certains robots. Il utilise un langage de script (le Lua) pour programmer les robots et ainsi les faire évoluer dans l'environnement simulé. Il peut aussi être couplé avec de nombreuses interfaces via des API remotes.

1. http ://www.coppeliarobotics.com

Le modèle 3d des kilobots fut implémenté par la ***K-Team***[2] dans le simulateur et permet la simulation de l'ensemble des fonctionnalités des robots. Il existe deux implémentations : avec et sans capteur de lumière. En effet, la simulation du capteur de lumière nécessite des calculs supplémentaires et allonge la durée de la simulation. Il n'est donc pas utile qu'il soit utilisé si l'algorithme n'en n'a pas besoin.
Le contrôleur est aussi simulé dans V-Rep via son propre modèle 3d. Il permet de lancer le programme, arrêter le programme en cours et aussi de connaître le niveau de batterie simulé.
L'ensemble de l'API de la K-Team est implémenté dans le simulateur via un squelette de codes inclu dans chaque script associé à un objet kilobot.

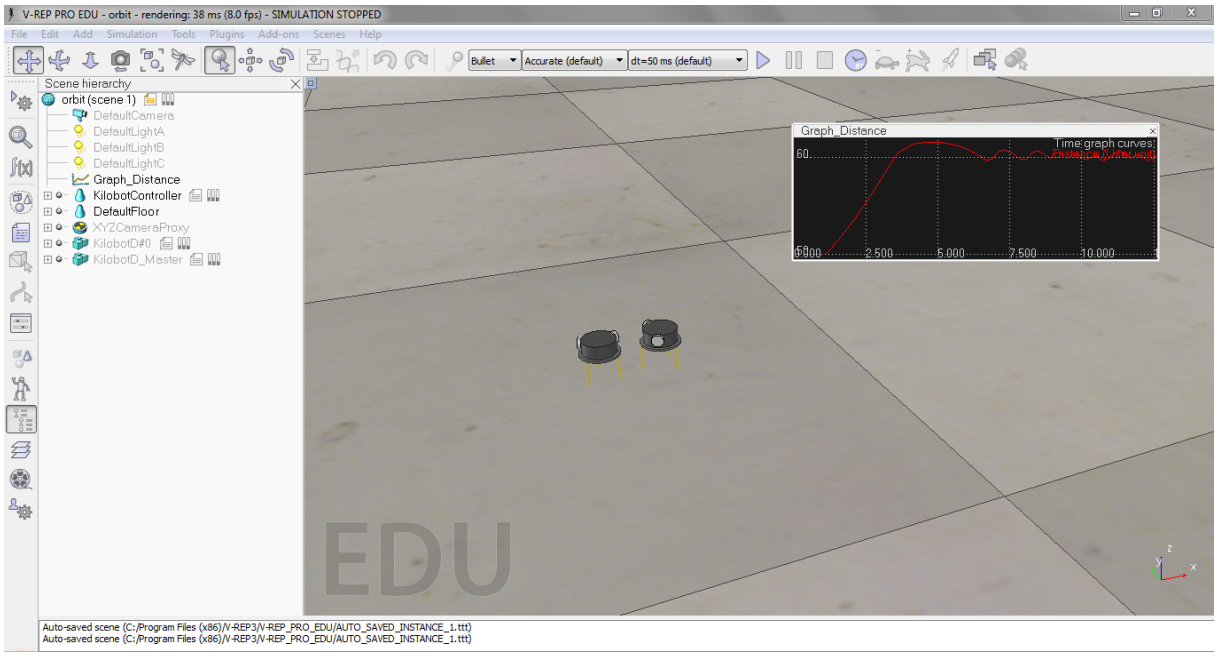

FIGURE 2.1 – Interface du simulateur V-Rep

Malheureusement, le simulateur V-Rep ne répond pas à nos besoins pour la simulation d'algorithme collectif sur les kilobots. Plusieurs problèmes furent rencontrés :

Le langage utilisé pour les script est le Lua. Bien que très similaire au langage C, il nous oblige à retranscrire chaque programme en Lua. De plus, le fichier contenant le script en lua est incorporé dans le fichier contenant la scène du simulateur (la scène comprend les modèles de robot simulé, l'environnement ainsi que l'ensemble des scripts). C'est à la fois un avantage (partage d'une simulation très facile : tout est regroupé dans un seul fichier) et un inconvénient (obliger de partager l'ensemble de la scène pour pouvoir travailler en groupe et ne pas pouvoir travailler sur le script sans installer ou exécuter le simulateur).

Le simulateur dispose lors de la simulation de plusieurs vitesses de simulation. Il est alors utile de pouvoir accélérer le temps pour voir si l'implémentation de l'algorithme est robuste et stable. Sauf que l'augmentation de la vitesse de simulation engendre des bugs dans le déroulement des scripts et ne permet pas de s'assurer du bon fonctionnement d'un programme de manière fiable.

Enfin, V-Rep est un simulateur orienté mécanique. Il dispose en effet d'un moteur physique et d'un environnement en 3 dimensions pour pouvoir simuler des robots complexes comme des bras robotiques. Nous l'utilisons pour effectuer des simulations d'algorithmes, donc comme un simulateur logique. Il n'est alors pas adapté à nos besoins (trop lourd, langage de script trop contraignant, nombreux bug durant la

2. http ://www.k-team.com/mobile-robotics-products/kilobot

simulation).

KbSim

Nous avons alors cherché un autre simulateur et avons trouvé ***KbSim***[3], un simulateur entièrement dédié aux kilobots et développé en python en utilisant la librairie pygame.
Bien que n'utilisant pas les mêmes fichiers que les kilobots, sa programmation en python est aisée et permet une implémentation rapide des algorithmes dans le simulateur. Il offre aussi d'autre fonctionnalités comme par exemple l'affichage du rayon d'émission de chaque kilobot. Sa consommation plus faible en ressource et sa plus grande portabilité en font un simulateur plus adapté pour notre tâche que le simulateur V-Rep.

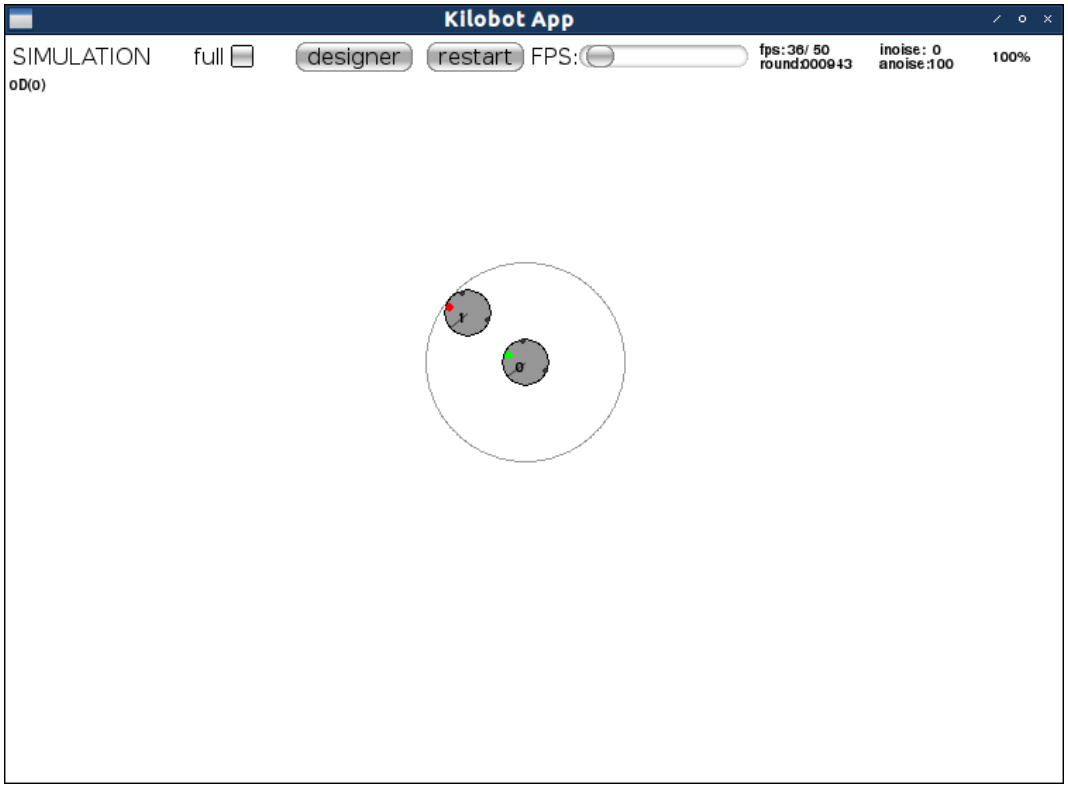

FIGURE 2.2 – Interface du simulateur KbSim

3. https ://github.com/ajhalme/kbsim

Firmware

K-Team

Les kilobots sont un modèle de robots développés par l'université d'Harvard [4]. L'entreprise suisse ***K-Team*** [5] les proposent à la vente et s'occupe de la fabrication et de l'assemblage. Ils fournissent les robots avec leur propre firmware développé en interne par la K-Team.
Ce firmware possède un inconvénient : il est compatible uniquement sous windows.
En effet, pour que l'opérateur puisse charger le programme dans les kilobots, il doit passer par le contrôleur qui est l'interface entre l'ordinateur et les robots. Or, le driver du contrôleur est uniquement disponible pour Windows et la K-Team ne fournit pas les sources pour un portage sous Linux.
De plus, nous avons rencontré des problèmes lors de la calibration des moteurs avec le firmware d'origine. Ce dernier ne permet la calibration uniquement du moteur gauche ou du moteur droit mais pas des deux lorsqu'on souhaite avancer. Or, le positionnement des jambes de chaque kilobot n'étant pas parfaitement identique, nous avons besoin de valeurs différentes pour les deux moteurs afin que le robot se déplace en ligne droite.

Kilobotics

Après recherche, nous décidâmes d'utiliser le firmware ***Kilobotics*** [6]. Kilobotics est le nom de la librairie développée par l'université d'Harvard. Il est open-source et est plus récent et dispose d'un driver pour Linux. De plus, son utilisation limite les risques pour le matériel. Le contrôleur permet de charger les programmes dans les kilobots. Mais pour effectuer cela avec le firmware d'origine de la K-Team, il faut à chaque nouveau programme reprogrammer le microcontrôleur du contrôleur, ce qui implique le risque de se retrouver avec un microcontrôleur mal programmé et alors un contrôleur potentiellement hs. Le firmware kilobotics nécessite lui de ne le reprogrammer qu'une seule fois, ce qui limite grandement les risques.
En ce qui concerne la calibration, kilobotic permet via l'interface graphique (**KiloGUI**) de calibrer chaque moteur séparément mais aussi de leur attribuer des valeurs différentes pour le déplacement en ligne droite. Kilogui propose aussi la possibilité de rentrer directement les valeurs souhaitées (entre 1 et 255) pour la puissance des moteurs. Enfin, il est possible d'affecter un identifiant à chaque robot qui sera stocké dans la mémoire EPROM.

4. http ://www.eecs.harvard.edu/ssr/projects/progSA/kilobot.html
5. http ://www.k-team.com/mobile-robotics-products/kilobot
6. https ://www.kilobotics.com/

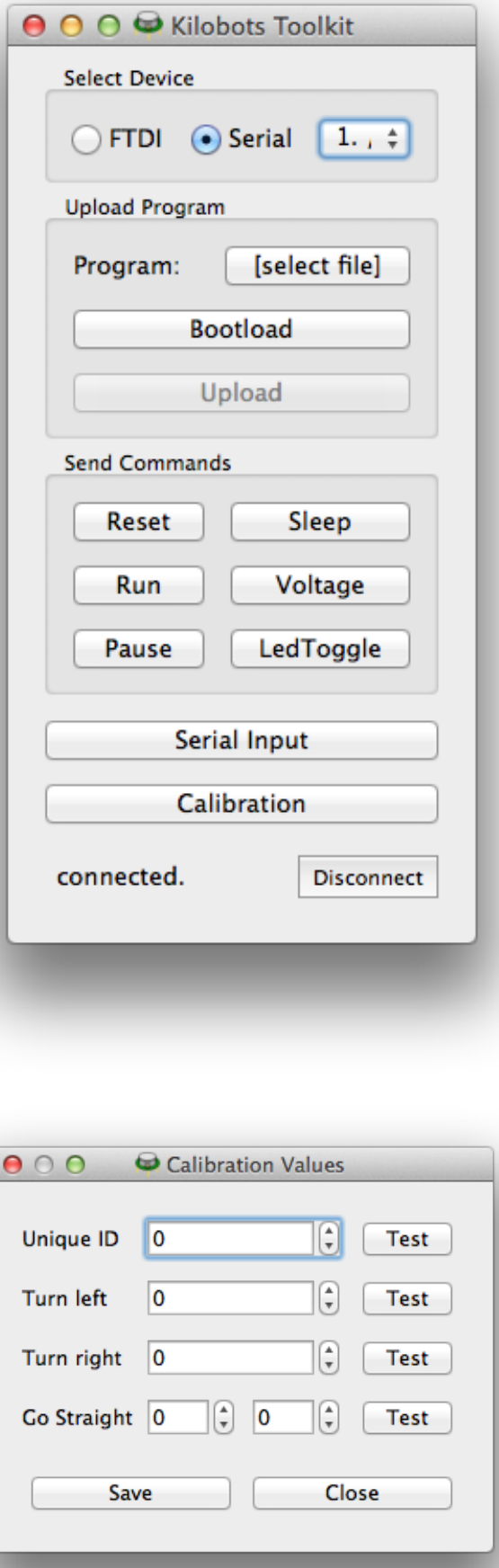

FIGURE 2.3 – Interface de l'application KiloGUI

Chapitre 3

Phase I

Ce projet se découpe en 3 phases :

Phase-1 ***Recherche Documentaire et Prise en main.***

Phase-2 ***Recherche d'un bio-algorithme et développement.***

Phase-3 ***Implémentation d'un modèle de robot avec les kilobots.***

La phase I ne comprenant que de la recherche documentaire et la prise en main de la plate-forme, la répartition se deroula ainsi, Arnaud prit en main la plate-forme et Benjamin la recherche documentaire. Pour la prise en main de la plate-forme, deux algorithmes fut choisis mettant chacun en scène une ou plusieurs capacités : la locomotion et la communication.
Le premier algorithme est celui de l'orbite. Il permet de déplacer de façon circulaire un kilobot autour d'un point représenter par un autre robot.
Le second met en avant la communication entre eux et se nomme firefly. Il réalise une synchronisation de plusieurs horloges (ici celles de plusieurs robots) via échange de messages.

Orbite

L'orbite reprend le principe des satellites naturels comme la lune qui tourne autour de la terre. Chaque astre prend ici la forme d'un kilobot. Le premier va effectuer une révolution autour du second pendant que ce dernier lui enverra des messages en continu. L'emploi d'un second kilobot est indispensable pour permettre à un autre d'avancer en formant un cercle. Sans point de repère, il est impossible d'en effectuer un correctement, les déplacements par vibrations étant trop imprécis et entraînant des mouvements non désirés. La surface sur laquelle évolue le robot rajoute encore un facteur imprévisible.

Un kilobot va rester en position stationnaire et envoyer des messages en continue. Le contenu de ceux-ci n'est d'aucune importance. Ils ne servent qu'à transmettre la distance qui sépare l'émetteur des récepteurs.
Le second kilobot va lui avancer en continu et recevoir les messages du premier. A chaque réception, il va calculer la distance qui le sépare de l'émetteur et ainsi pouvoir faire des corrections de trajectoire. Il avance ainsi tout droit tant qu'il se trouve dans les bornes définies dans le programme. S'il se trouve à une distance inférieure à celle-ci, il tournera à gauche jusqu'à de nouveau être dans les bornes. Il effectuera la même opération s'il est à une distance supérieure, mais vers la droite.
Ainsi, il va naturellement réaliser une révolution autour du kilobot fixe en tournant autour.

Orbite

Legende : **S** (Send) et **R** (Receive).

Master		*Slave*	
S :	envoie un message <>	**S :**	
R :		**R :**	réception d'un message <> Si distance >= distance_max tourner à droite avancer tout droit Sinon SI distance <= distance_min touner à gauche avancer tout droit Sinon avancer tout droit Fin Si

Le kilobot *Master* reste immobile tandis que le kilobot *Slave* orbite autour de lui.

Firefly

L'algorithme firefly est un bio-algorithme s'inspirant du comportement des lucioles dans la nature. Quand elles se retrouvent en groupe, elles sont capables de "clignoter" de manière synchronisée. Les kilobots font de même avec cet algorithme.

Pour permettre possible la synchronisation de plusieurs individus, nous implémentons une horloge logique dans chacun. La synchronisation va se réaliser en collectant les horloges contenues dans les messages des voisins afin de déterminer la moyenne du nombre de tick d'horloge qui sont en avance (ou en retard).

Le kilobot dispose donc d'un tableau dont les indexes sont le nombre de tick en retard ou en avance par rapport à ses voisins.
Exemple : Si la différence entre son horloge et l'horloge reçue dans un message est de 8 ticks, alors il incrémentera de 1 la valeur contenue à l'index 8 du tableau.
Le tableau est ainsi mis à jour à chaque réception d'un message seulement si la différence entre l'horloge de l'émetteur et celle du récepteur est inférieure à la moitié de la période d'horloge. Cette prévention permet d'éviter que deux kilobots voisins se mettent à jour en même temps.

A chaque itération de la boucle principale du programme, la moyenne de la somme des ticks enregistrés dans le tableau est effectuée (tick_offset) et celui-ci remis à zéro. La différence entre l'horloge locale (kilo_ticks) et celle reçue est calculée :

modulo_clock = ((kilo_ticks – tick_offset) / 4) % 32

Si modulo_clock est égale à zéro, le robot peut alors faire clignoter sa led. Ainsi la synchronisation se fait au fil des cycles en réduisant l'écart entre chaque horloge.

Luciole/Firefly

Legende : **S** (Send) et **R** (Receive).

S : envoie un message <Horloge>

R : réception d'un message <Horloge>
Si mon_Horloge >Horloge
décrémentation de mon_Horloge d'une demi-période
Sinon
incrémentation de mon_Horloge d'une demi-période
Fin Si

Chapitre 4

Phase II

La phase II demandant plus de réflexion, nous mîmes nos compétences acquises à la première phase en commun afin de proposer un bio-algorithme digne d'intérêt.

Phototaxis

Notre bio-algorithme se base sur le principe de phototaxis.

Réaction de locomotion d'organismes mobiles provoquée par la lumière et qui les porte soit à s'en approcher (phototaxie positive), soit à s'en éloigner (phototaxie négative).

Notre algorithme ne permet pas une entraide des kilobots, c'est à dire que chaque robot suit son programme sans se soucier des autres.
Cette version ne permet donc pas à un essaim en formation de suivre la lumière tout en gardant cette formation.
L'algorithme se déroule ainsi, chaque robot capte la lumière ambiante (la lumière ambiante doit être faible afin de ne pas gêner les capteurs des kilobots dans la détection de la source cible de lumière), si la source de lumière détectée par le capteur provient de la droite on tourne à droite et inversement, après chaque demi-tour on avance tout droit pendant un certain laps de temps.
Les kilobots ne communiquent jamais ensemble.

```
var currentLight <- indice de lumière ambiante.
var currentDir   <- indice de direction.

Tant que vrai
     r\'eception de la lumière

     Si currentLight > barrière haute
     alors
        -tourner à droite
        -currentDir <- droite;
     Fin Si

     Si currentLight < barrière basse
     alors
        -tourner à gauche
        -currentDir <- gauche;
     Fin Si
Fin Tant que
```

Gradient

L'algorithme du gradient permet de calculer sa distance par rapport à une balise fixe.
Chaque kilobot transmet à tous ses voisins la distance qui le sépare de la balise. Ainsi, chaque robot peut calculer sa propre distance par rapport à la balise sans pour autant communiquer directement avec elle. L'ensemble des robots sert de relais pour la balise fixe.
Une fois sa distance calculée, le kilobot allume sa led pour signaler sa position.

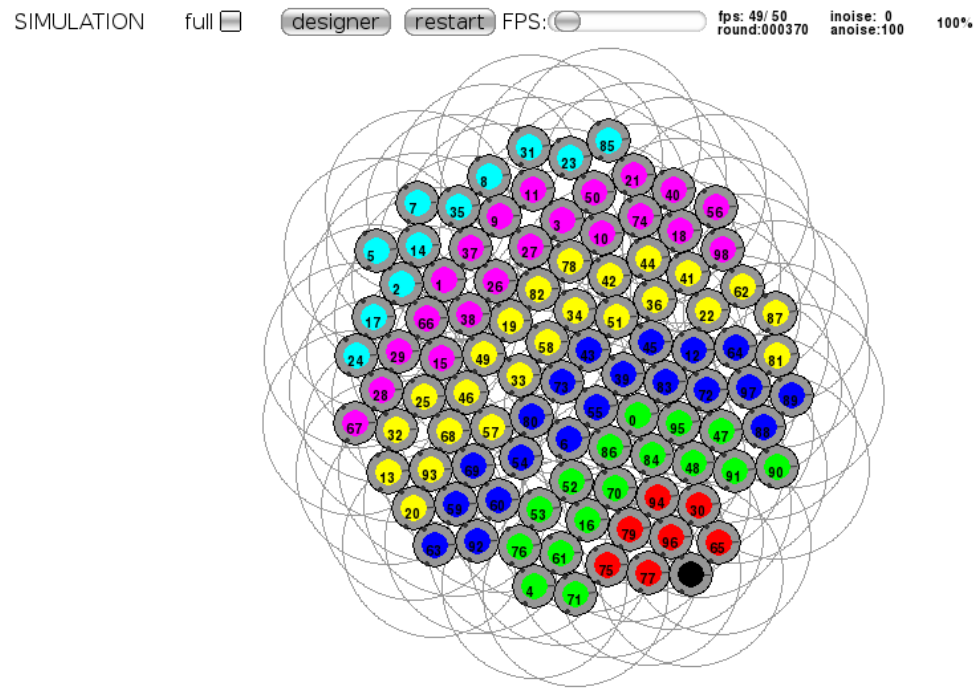

FIGURE 4.1 – Gradient à une balise

Gradient

```
var timer
var gradient_value

Tant que vrai
     attendre une seconde
     incr\'ementer timer
     Si timer >= 5000 // Actualisation toutes les 5 secondes
     alors
          Si kilo_uid == 0 // Si le kilobot a l'id 0.
          alors
               gradient_value <- 0
          Sinon
               gradient_value <- Valeur maximum
          Fin si
          timer <- 0
     Fin si

     Si nouveau message
     alors
          Si gradient_value > valeur_reçue + 1
          alors
               gradient_value <- valeur_reçue + 1
               mettre à jour le message à envoyer.
               timer <- 0
          Fin si

          Switch(gradient_value % 3)
               0 -> allumer led en rouge
               1 -> allumer led en vert
               2 -> allumer led en bleu
          Fin switch
     Fin si
Fin tant que
```

Chapitre 5

Phase III

Modèle CORDA

Cette phase vise à rechercher le moyen d'implémenter le **modèle CORDA** [1].
Nous avons choisi de nous pencher sur ce modèle précis car ce modèle est largement utilisé pour les algorithmes réparties dans le domaine de la robotique.

Il définit des caractéristiques physiques dont devront disposer les robots :
- chaque robot possède un repère orthonormé.
- chaque robot possède une vision "infinie".
- chaque robot ne possède aucun moyen de communiquer avec ses congénères.
- chaque déplacement du robot est précis.

Ces contraintes posent plusieurs problèmatiques. En effet la plate-forme Kilobot ne propose aucune de ces contraintes. (cf chapitre 2)
Donc comment simuler une vision "infinie" avec la plate-forme Kilobot ? Comment simuler un axe orthonormé ? Comment localiser les robots les uns par rapport aux autres ? Et plus important comment un robot peut se localiser dans l'espace ?

Implémentation

Nous pouvons résoudre le problème de la localisation dans l'espace (ainsi que la localisation par rapport aux autres) et de l'axe orthonormé en utilisant la méthode de la trilatération. Les robots Kilobot ne possèdent pas de caméra et ont une portée de communication limitée ce qui nous ne permet pas d'implémenter un vision infinie à proprement parler.

Première Approche

Nous pouvons, de manière naïve, résoudre le problème de localisation des robots en disposant des balises sur un cercle de diamètre 14cm (cf figure : 5.2), en effet la distance maximum d'émmission de messages est de 7cm donc le diamètre maximal du cercle est donc bien de 14cm.
Chaque point de la figure représente une balise, et les cercles permettent de visualiser les zones de communications.
Nous remarquons que l'ajout de cinq balises sur le cercle ne nous permet pas de recouvrir de manière optimale la zone (intérieur du cercle) donc nous doublons le nombre de balises.
Cela permet effectivement de couvrir toute la zone et de localiser de manière "précise".
Notre approche résout le problème de la localisation mais pose un autre problème. Comment augmenter

1. Giuseppe Prencipe. A new distributed model to control and coordinate a set of autonomous mobile robots : The corda model, 2000.

la zone de travail ?
En effet, la zone couverte permet de faire fonctionner correctement seulement deux kilobots.
Donc notre approche n'est pas viable pour simuler le modèle CORDA sur la plate-forme Kilobot.

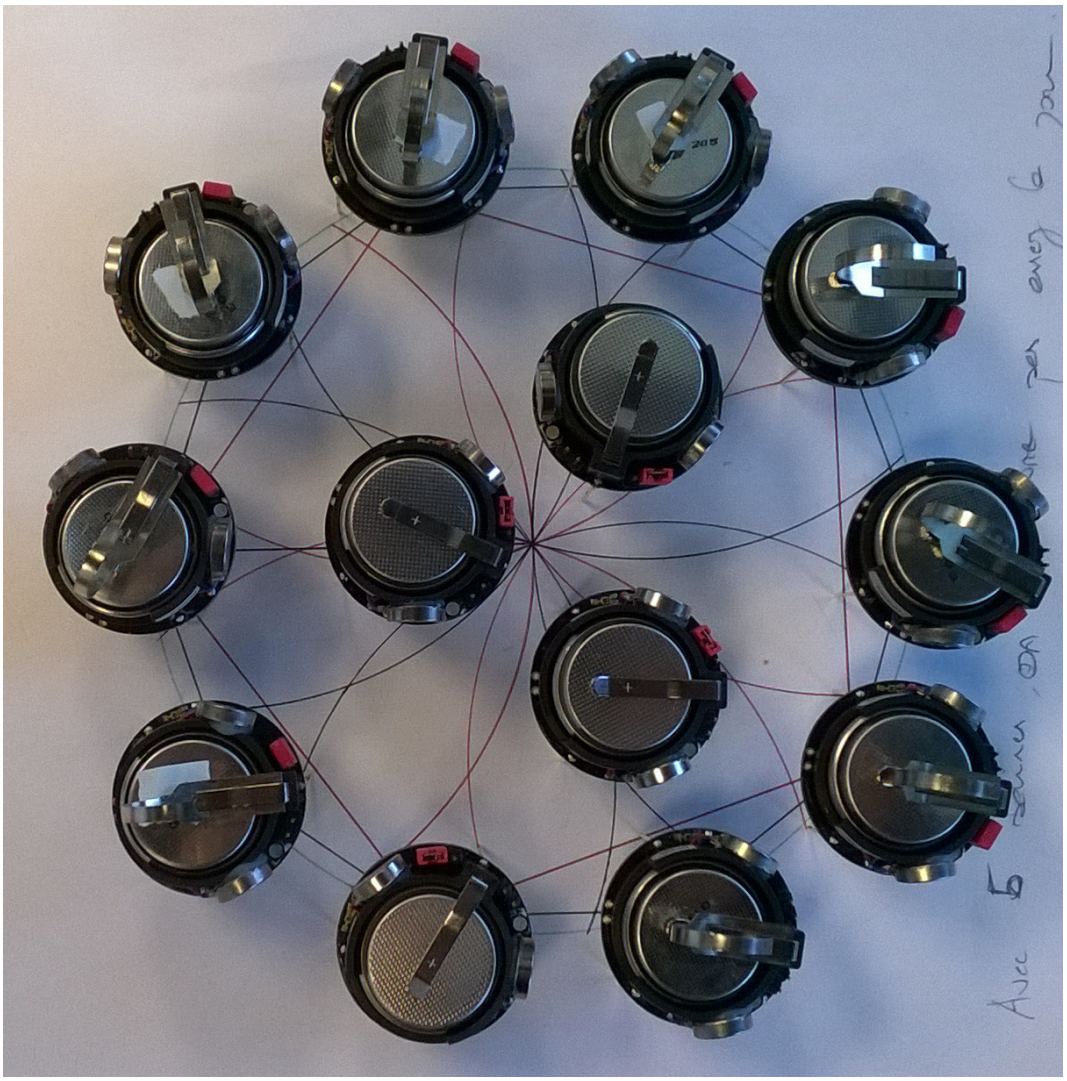

FIGURE 5.1 – Simulation de la première approche

Seconde Approche

La première approche résolvait le problème de localisation et de positionnement, mais ne permettait pas de faire fonctionner plus de deux robots de manière viable.
Donc notre deuxième approche devra, elle aussi, résoudre le problème de localisation et de positionnement mais devra, par contre, agrandir la zone de travail des robots (par exemple de permettre à 6 robots ou plus d'être dans la zone de travail sans se gêner).

Nous pouvons utiliser la méthode dite de "bounding box" mais cette méthode augmenterait le nombre de balises par rapport au nombre de robots utiles de manière trop importante (du faite de la faible distance de communication des Kilobots).
Nous pouvons utiliser la méthode de multilatération qui ne demande que trois balises et ne demande pas une connectivité directe avec les balises (la distance est estimée grâce à la propagation d'un gradient)

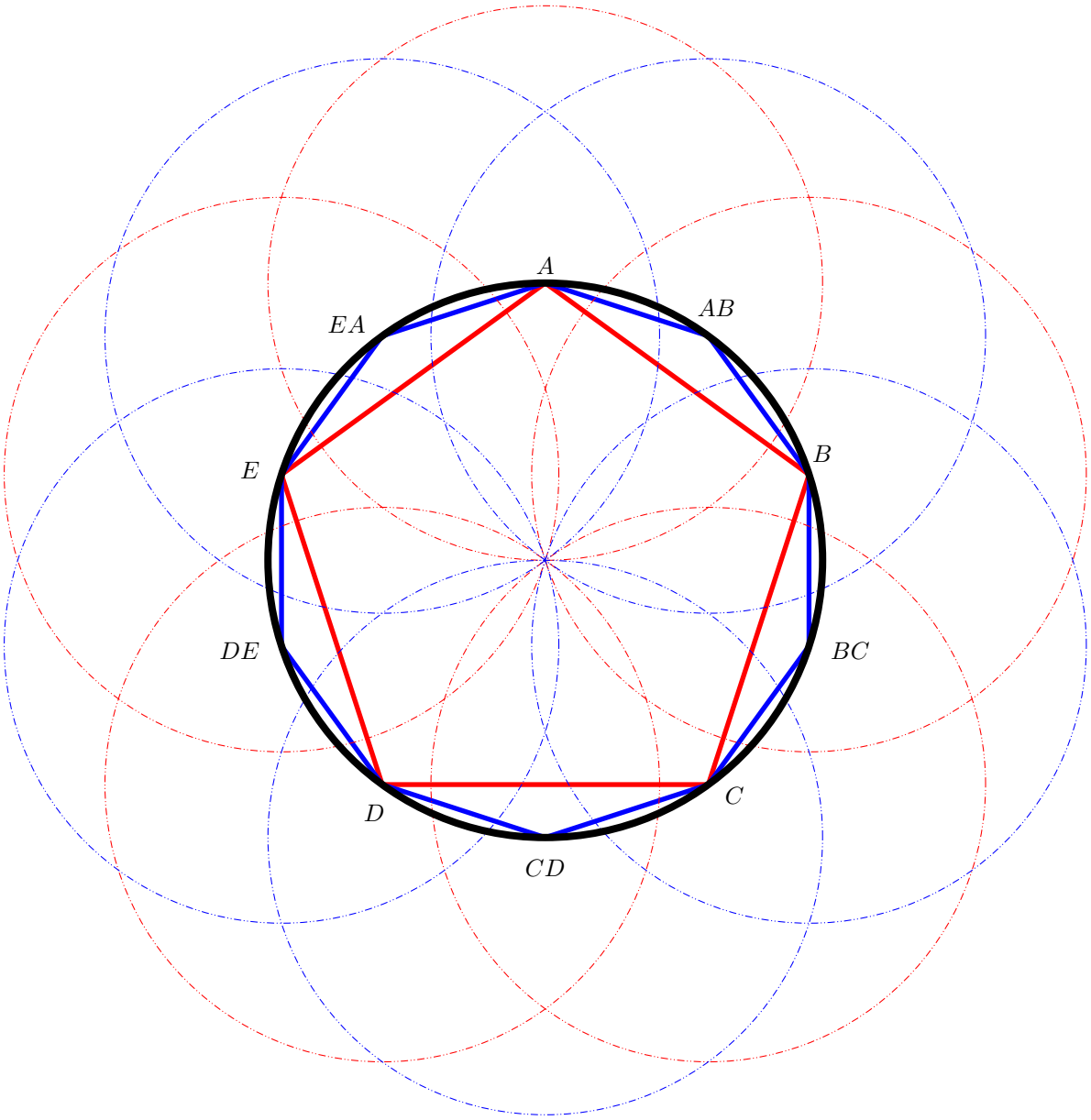

FIGURE 5.2 – Première Approche

mais cette méthode pose le problème de la puissance de calcul, en effet la plate-forme Kilobot offre une puissance de calcul limitée ne permettant pas d'implémenter cette méthode.

La solution serait de proposer une méthode hybride entre ces deux méthodes, soit la méthode de "bounding box" avec la distance estimée avec un gradient.
Cette solution permettra à terme d'agrandir la zone de travail mais aura comme conséquence une perte de précision.

Spécifications de l'API

Nous proposons l'implémentation d'une API permettant de simuler le modèle CORDA, elle devra comprendre les primitives suivantes :

getPosition qui permet à un robot de connaître sa position.

getVision qui permet à un robot de connaître la position des robots qui l'entourent.

toPosition qui permet à un robot de se rendre aux coordonnées en paramètre.

Cette API s'appuiera sur l'API existante de Kilobotics.

Chapitre 6

Annexe

Matériel Utilisé

Le matériel se compose des kilobots, de transmetteur et d'un PC quelqu'il soit.

Logiciel Utilisé

Nous utiliserons les logiciels fournis pour l'API Kilobotics pour le développement de notre API, notre gestionnaire de version sera Github.
Le projet sera hébergé sous l'organisation ***LSDev8***[1].
Nous utilisions (au début du projet) pour nos simulations le logiciel ***V-Rep***[2] mais celui-ci se révéla trop contraignant dans son utilisation donc nous optâmes pour le logiciel ***KbSim***[3] car celui-ci est un simulateur exclusivement pour kilobot (une copie de ce logiciel se trouve aussi sur l'organisation ***LSDev8***).

Tutoriels

Installation du contrôleur

Le contrôleur exploite un microcontrôleur Atmega m328p. Lors de la première connexion du contrôleur à un ordinateur Windows, Windows va automatiquement aller téléchargé les drivers correspondant via windows update et les installer. Il n'y a pas d'autre opérations à faire. Sous Linux, aucun driver n'est nécessaire.
Si vous utilisez le firmware original de la K-Team, il vous faudra alors installer pour Windows un driver supplémentaire afin de faire communiquer l'interface graphique avec le contrôleur. Pour cela, suivez les instructions fournies dans le manuel de la K-Team.

Si le contrôleur n'est pas reconnu par windows, assurez-vous qu'il apparaît bien dans le gestionnaire de périphérique.
Il doit normalement apparaître comme un Serial Port (FTDI) et un AVRisp mkII (Jungo).

Flashage du contrôleur et des kilobots

Si vous désirez passer au firmware kilobotics, il est nécessaire d'utiliser le logiciel AvrDude. Ce logiciel développé par Atmel permet de reprogrammer le microcontrôleur du contrôleur. Pour cela, connecter le contrôleur à un ordinateur via un cable usb, ouvrer un terminal et placez-vous dans le répertoire contenant le nouveau firmware (au format .hex) puis assurez-vous que le cavalier du contrôleur est bien

1. https ://github.com/LSDev8/
2. http ://www.coppeliarobotics.com
3. https ://github.com/ajhalme/kbsim

positionné en mode interne (c'est la position par défaut : c'est à dire sur la gauche).

FIGURE 6.1 – Cavalier en position interne

Enfin, exécuter la commande suivante :

avrdude -p m328 -P usb -c avrispmkII -U "flash :w :controller.hex :i"

Pour flasher les kilobots, l'opération est très similaire. Connecter le contrôleur, ouvrer un terminal et placez-vous dans le répertoire contenant le nouveau firmware (toujours au format .hex). Placez cette fois-ci le cavalier du contrôleur en mode externe (sur la droite) :

Puis connecter le câble au contrôleur et au kilobot via les 6 trous présents sur le côté du kilobot. Enfin, exécuter la commande suivante :

avrdude -p m328p -P usb -c avrispmkII -U "flash :w :bootloader.hex :i"

Si AvrDude ne reconnaît pas le contrôleur, assurez-vous d'avoir positionné le cavalier dans la bonne position. Sinon, la connexion entre le connecteur du câble et le kilobot est très sensible. Il faut que toutes les broches du connecteur soit en contact avec le kilobot. N'hésitez surtout pas à presser légèrement le connecteur contre le kilobot et à le placer légèrement en biais afin d'assurer la bonne connexion entre le connecteur et le robot.

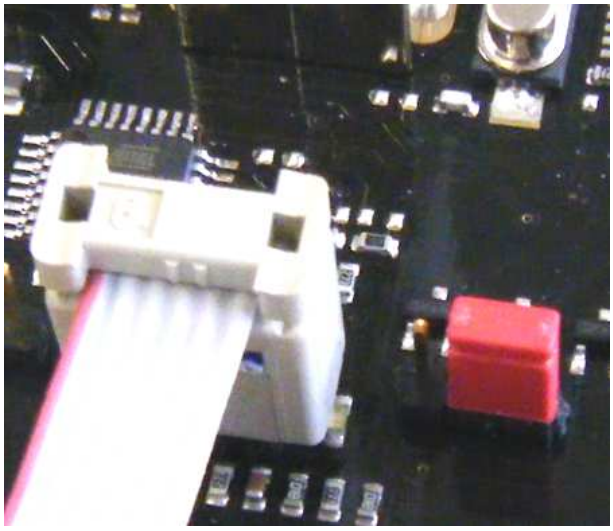

FIGURE 6.2 – Cavalier en position externe

Chargement des kilobots

Les kilobots peuvent être rechargés via le chargeur de la K-Team. Pour cela, passez les en mode charge grâce au contrôleur et à l'interface graphique kilogui, puis disposez les sur le chargeur. Ils clignoteront en rouge pendant la charge puis passeront au bleu une fois la charge terminée.

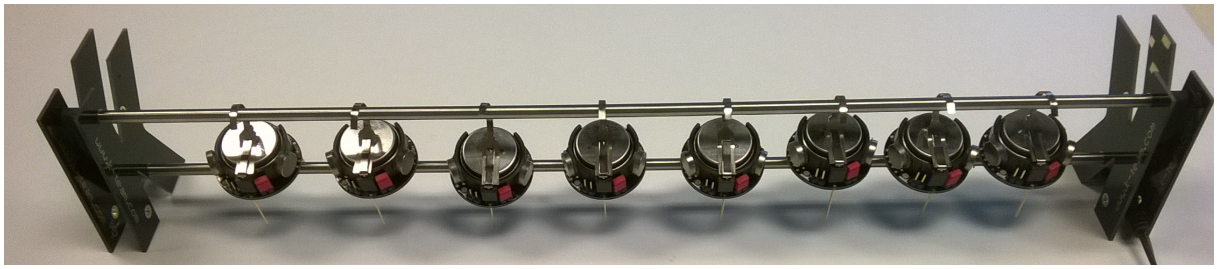

FIGURE 6.3 – Kilobots sur le chargeur

Attention, en fonction de la version de kilogui dont vous disposez, il se peut que vous ne disposiez pas du bouton charge. Il existe deux versions de kilogui : La version sous format .deb contient la calibration mais pas le mode charge. Il faut donc récupérer les sources de kilogui et les compiler vous-même. Attention, les sources de kilogui ne contiennent pas la calibration. Il faut donc utiliser les deux versions en fonction de l'action souhaitée (calibration ou chargement).

Compilation d'un programme pour les kilobots

Pour compiler un programme pour la plate-forme Kilobot, il suffit de changer dans le Makefile la valeur de la variable *EXEC* en début de fichier par le nom du fichier à compiler.

Attention, si la compilation échoue veuillez vérifier si les logiciels suivants sont installés : **avr-libc**, **gcc-avr** et **avrdude**.

Calibration des kilobots

Les kilobots se déplacant par vibration, il est primordiale de les calibrer. En effet, chaque exemplaire possède des pattes avec un angle différents. Il faut donc les calibrer individuellement. Pour calibrer un robot, allumer le puis positionner le en dessous du contrôleur. Vérifier que vous utilisé bien la version de kilogui disposant du module de calibration (la 64 bits).
Cliquer sur le bouton de calibration, vous obtenez la fenêtre suivante :

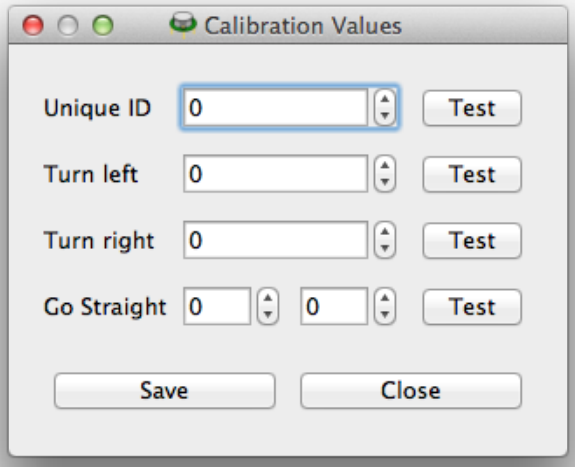

FIGURE 6.4 – Interface de calibration de KiloGUI

Elle comporte 5 champs :

Unique ID : permet d'associer à chaque kilobot un identifiant.

Turn Left : permet de définir la puissance de vibration utilisé par le kilobot pour tourner à gauche. Ce sera la valeur utilisé par la constante kilo_turn_left. Une puissance entre 60 et 90 est généralement suffisante pour obtenir une bonne rotation. Cliquer sur Test pour tester la nouvelle valeur.

Turn Right : idem que Turn Right. Attention : il a de grande chance qu'il faille utiliser une valeur différente que celle utilisé pour Turn Left.

Go Straight : Valeur utilisé par le kilobot lorsqu'il désire avancer en ligne droite. Chaque valeur correspond à un moteur de vibration. Une fois encore, il est fort probable que les valeurs requises soient différentes des valeurs précédentes.

Une fois la calibration terminée, cliquer sur Save pour enregistrer les nouvelles valeurs dans la mémoire du kilobot.

Codes Sources

Luciole/Firefly

```
1  /*##############################################*/
2  /*# Kilkobot Firefly - Synchronize kilobot led #*/
3  /*##############################################*/
4  #include <kilolib.h>
5
6  /* CONSTANTS */
7  #define PERIOD 32 /* Only 32 possible messages */
8
9  /* VARIABLES */
10 uint8_t Mclock;
11 message_t mesgs[PERIOD];
12 uint8_t offsets[PERIOD];
13 uint8_t total;
14 uint16_t avg;
15
16 /* Transmition Mclock message */
17 message_t *message_trans()
18 {
19   return &mesgs[Mclock];
20 }
21
22 /* Receiption Mclock message */
23 void message_receip(message_t *mess, distance_measurement_t *dist)
24 {
25   /* if robot ahead of neighbor... */
26   if (Mclock > mess->data[0])
27     {
28       /* ...by less than half a period */
29       if (Mclock - mess->data[0] < PERIOD/2)
30         offsets[Mclock - mess->data[0]]++;
31     }
32   else
33     {
34       /* ...by more than a half period */
35       if (mess->data[0] - Mclock > PERIOD/2)
36         offsets[Mclock + (PERIOD - mess->data[0])]++;
37     }
38 }
39
40 void setup()
41 {
42   for (int i=0; i<PERIOD; i++)
43     {
44       mesgs[i].data[0] = i;
45       mesgs[i].type = NORMAL;
46       mesgs[i].crc = message_crc(&mesgs[i]);
47       offsets[i] = 0;
48     }
49
50   kilo_ticks += (rand_hard())<<2;
51 }
52
53 void loop()
54 {
55   Mclock = (kilo_ticks>>2)%PERIOD;
56   if (!Mclock)
57     {
58       /* Blink in magenta */
59       set_color(RGB(1,0,1));
60       delay(100);
61       set_color(RGB(0,0,0));
62
63       total = 0;
64       avg = 0;
65       /* Compute average offset */
```

```
66        for(int i=0; i<PERIOD; i++)
67          {
68            avg += i*offsets[i]; /* the total offset amount */
69            total += offsets[i]; /* total number of neighbor offsets recorded */
70            offsets[i] = 0;      /* clear the array for next time */
71          }
72
73        /* Adjust clock by average offset */
74        if (total>0)
75          {
76            avg /= total;
77            kilo_ticks -= (avg<<2);
78          }
79      }
80  }
81
82  int main()
83  {
84    kilo_init();
85    kilo_message_tx = message_trans;
86    kilo_message_rx = message_receip;
87    kilo_start(setup, loop);
88  }
```

Orbit : maitre

```
1  //Kilobot master - Send messages to the slave.
2
3  #include <kilolib.h>
4
5  uint8_t message_sent = 0;
6  message_t msg;
7
8  void setup()
9  {
10   msg.type = NORMAL;
11   msg.crc = message_crc(&msg);
12 }
13
14 void loop()
15 {
16   //Blink when a message is sent.
17   if(message_sent)
18     {
19       message_sent = 0;
20       set_color(RGB(0,0,1));
21       delay(20);
22       set_color(RGB(0,0,0));
23     }
24 }
25
26 message_t *message_tx()
27 {
28   return &msg;
29 }
30
31 void message_tx_success()
32 {
33   message_sent = 1;
34 }
35
36 int main()
37 {
38   kilo_init();
39   kilo_message_tx = message_tx;
40   kilo_message_tx_success = message_tx_success;
41   kilo_start(setup, loop);
42
43   return 0;
44 }
```

Orbit : esclave

```
1  //Kilobot slave - Orbit around the master.
2
3  #include <kilolib.h>
4
5  static const uint8_t MIN_DISTANCE = 40;
6  static const uint8_t DISTANCE = 60;
7
8  uint8_t cur_dist = 0;
9  uint8_t new_msg = 0;
10 distance_measurement_t dist;
11
12 void message_rx(message_t *m, distance_measurement_t *d)
13 {
14   new_msg = 1;
15   dist = *d;
16 }
17
18 void loop()
19 {
20   //New message receive ?
21   if(new_msg)
22     {
23       new_msg = 0;
24       //Blink led.
25       set_color(RGB(0,1,0));
26       delay(20);
27       set_color(RGB(0,0,0));
28       cur_dist = estimate_distance(&dist);
29
30       //Too far.
31       if(cur_dist >= DISTANCE)
32         {
33           spinup_motors();
34           //Go right.
35           set_motors(0, kilo_turn_right);
36         }
37       //Too close.
38       else if(cur_dist <= MIN_DISTANCE)
39         {
40           spinup_motors();
41           //Go left.
42           set_motors(kilo_turn_left, 0);
43         }
44       //Distance ok.
45       else
46         {
47           spinup_motors();
48           //Go straight;
49           set_motors(kilo_straight_left, kilo_straight_right);
50         }
51     }
52 }
53
54 void setup() { }
55
56 int main()
57 {
58   kilo_init();
59   kilo_message_rx = message_rx;
60   kilo_start(setup, loop);
61
62   return 0;
63 }
```

Phototaxis

```
1  /*#################################################*/
2  /*# Kilkobot Phototaxis - Move kilobot to light #*/
3  /*#################################################*/
4  #include "kilolib.h"
5
6  #define LEFT 0
7  #define RIGHT 1
8
9  /* CONSTANTS */
10 static const uint16_t HI_BARRIER = 600; /* These constants are to avoid ambient light noise */
11 static const uint16_t LO_BARRIER = 300;
12
13 /* VARIABLES */
14 uint8_t currentDir;
15 int16_t currentLight = 0;
16
17 /* Track the light */
18 void lightTrack()
19 {
20   int16_t number = 0;
21   long avg = 0;
22
23   /* until we are under the noise barrier */
24   while (number < 300)
25     {
26       int16_t sample = get_ambientlight();
27       if (sample != -1)
28         {
29           avg += sample;
30           number++;
31         }
32     }
33
34   currentLight = avg / 300;
35 }
36
37 void turnRight()
38 {
39   spinup_motors();                /* kilobot's motor at full speed */
40   set_motors(0,kilo_turn_right);
41   set_color(RGB(0,1,0));          /* GREEN LIGHT */
42 }
43
44 void turnLeft()
45 {
46   spinup_motors();
47   set_motors(kilo_turn_left,0);
48   set_color(RGB(1,0,0));          /* RED LIGHT */
49 }
50
51 /* This setup is at your convenience */
52 void setup()
53 {
54   currentDir = LEFT;
55   turnLeft();
56 }
57
58 void loop()
59 {
60   lightTrack();
61
62   /* RIGHT */
63   if (currentLight > HI_BARRIER && currentDir == RIGHT)
64     {
65       currentDir = LEFT;
66       turnLeft();
67     }
68   /* LEFT */
```

```
69    else if (currentLight < LO_BARRIER && currentDir == LEFT)
70      {
71        currentDir = RIGHT;
72        turnRight();
73      }
74  }
75
76  int main()
77  {
78    kilo_init();
79    kilo_start(setup, loop);
80
81    return 0;
82  }
```

Gradient

```
1  #include "kilolib.h"
2
3  uint16_t timer = 0;
4  uint16_t gradient_value = UINT16_MAX;
5  uint16_t recvd_gradient = 0;
6  uint8_t new_message = 0;
7  message_t msg;
8
9  void message_rx(message_t *m, distance_measurement_t *d) {
10   new_message = 1;
11   // unpack two 8-bit integers into one 16-bit integer
12   recvd_gradient = m->data[0]  | (m->data[1]<<8);
13 }
14
15 message_t *message_tx() {
16   if (gradient_value != UINT16_MAX)
17     return &msg;
18   else
19     return '\0';
20 }
21
22 void update_message() {
23   // pack one 16-bit integer into two 8-bit integers
24   msg.data[0] = gradient_value&0xFF;
25   msg.data[1] = (gradient_value>>8)&0xFF;
26   msg.crc = message_crc(&msg);
27 }
28
29 void setup() {
30   if (kilo_uid == 0)
31     gradient_value = 0;
32   update_message();
33 }
34
35 void loop() {
36   delay(1);
37   timer ++;
38   if (timer >= 5000)
39     {
40       if(kilo_uid == 0)
41         {
42           gradient_value = 0;
43         }
44       else
45         {
46           gradient_value = UINT16_MAX;
47         }
48       timer = 0;
49     }
50   if (new_message) {
51     if (gradient_value > recvd_gradient+1) {
52       gradient_value = recvd_gradient+1;
53       update_message();
54       timer = 0;
55     }
56     new_message = 0;
57     switch(gradient_value%3) {
58     case 0:
59       set_color(RGB(1,0,0));
60       break;
61     case 1:
62       set_color(RGB(0,1,0));
63       break;
64     case 2:
65       set_color(RGB(0,0,1));
66       break;
67     }
68   }
```

```
69 }
70
71 int main() {
72   // initialize hardware
73   kilo_init();
74   // register message callbacks
75   kilo_message_rx = message_rx;
76   kilo_message_tx = message_tx;
77   // register your program
78   kilo_start(setup, loop);
79
80   return 0;
81 }
```

Bibliographie

[1] Alejandro Cornejo and Radhika Nagpal. Long-lived distributed relative localization of robot swarms. *CoRR*, abs/1312.1915, 2013.

[2] Yoann Dieudonné, Ouiddad Labbani-Igbida, and Franck Petit. Circle formation of weak mobile robots. In AjoyK. Datta and Maria Gradinariu, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 4280 of *Lecture Notes in Computer Science*, pages 262–275. Springer Berlin Heidelberg, 2006.

[3] YuriK Lopes, AndréB Leal, TonyJ Dodd, and Roderich Groß. Application of supervisory control theory to swarms of e-puck and kilobot robots. In Marco Dorigo, Mauro Birattari, Simon Garnier, Heiko Hamann, Marco Montes de Oca, Christine Solnon, and Thomas Stützle, editors, *Swarm Intelligence*, volume 8667 of *Lecture Notes in Computer Science*, pages 62–73. Springer International Publishing, 2014.

[4] Thomas Moinel. Coordination d'un essaim de robots mobiles (30 kilobots) par des mécanismes inspirés de l'intelligence collective observée chez les animaux sociaux. 2012.

[5] Giuseppe Prencipe. A new distributed model to control and coordinate a set of autonomous mobile robots : The corda model, 2000.

[6] Michael Rubenstein, Christian Ahler, Nick Hoff, Adrian Cabrera, and Radhika Nagpal. Kilobot : A low cost robot with scalable operations designed for collective behaviors. *Robotics and Autonomous Systems*, 62(7) :966–975, 2014.

[7] A Ramezan Shirazi, H Oh, and Y Jin. Morphogen diffusion algorithms for tracking and herding using a swarm of kilobots. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8717 L :139 – 150, 2014. The original publication is available at http ://www.springerlink.com.

[8] Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots : Formation of geometric patterns. *SIAM Journal on Computing*, 28 :1347–1363, 1999.