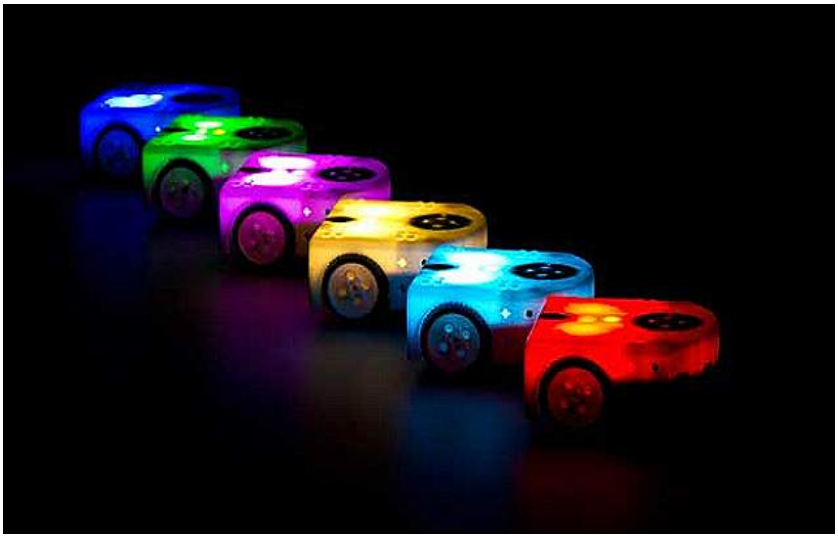
Architecture de contrôle pour un robot mobile

BILLOD Nicolas
LORTHIOIR Guillaume

Jan-May 2017



Encadrante : Aurélie Beynier

Sommaire

1 Introduction

Suite à l'acquisition de robots Thymio II par le LIP6, notre projet consiste à établir une architecture de contrôle pour ce type de robots, et de l'implémenter pour leur permettre de percevoir leur environnement, et d'effectuer des actions dans celui-ci. En effet, une telle architecture existe déjà mais elle est implémentée en Python, et pour des soucis d'efficacité une architecture logicielle développée en C nous a été demandée. Une architecture de contrôle est une manière d'organiser un ensemble de modules (regroupement de fonctions similaires), de plus ou moins haut niveau, qui vont interagir entre eux et permettre au robot d'effectuer un ensemble de tâches. L'implémentation de cette architecture de contrôle sera embarquée, avec tous ses modules, sur un Raspberry Pi 3, les robots ne disposant pas d'une mémoire suffisante pour la supporter.

Notre architecture doit être souple et bien documentée puisqu'elle est actuellement employée par des Thymios mais pourra par la suite être utilisée sur des drones ou d'autres type de robots. De plus, nous devons pouvoir y rajouter des modules complémentaires sans que cela perturbe le code initial (module de communication, caméra, etc), la documentation se doit donc d'être efficace pour permettre ces ajouts sans que cela ne pose trop de problèmes à l'utilisateur.

Nous allons dans un premier temps parler des contraintes de réalisation auxquelles nous avons du faire face au cours de ce projet, puis nous présenterons l'architecture implémentée ainsi que les modules qui la composent en y détaillant les principaux algorithmes. Nous nous intéresserons ensuite aux différents tests effectués sur les fonctions et leurs résultats. Nous nous sommes concentrés sur deux tâches qui nous semblaient les plus importantes pour les tests de haut niveau : la cartographie et le trajet vers une destination donnée. Pour finir, nous conclurons sur notre projet en évoquant ce qu'il nous a apporté et ses ouvertures pour l'avenir.

2 Contraintes de réalisation

Lors de la réalisation de ce projet nous avons du faire face à plusieurs contraintes aussi bien matérielles qu'informatiques. Certaines de ces contraintes de réalisation nous ont été imposées dès le début, et d'autres se sont révélées au cours de son avancement.

2.1 Contraintes matérielles

L'architecture de contrôle doit être développée sur un Raspberry Pi 3 qui est un nano-ordinateur d'une dizaine de centimètres, embarquant une version du système d'exploitation Debian prévue pour ce genre d'ordinateurs (distribution jessie-raspbian dans notre cas). L'utilisation d'un tel ordinateur pour contrôler le Thymio est nécessaire puisque même s'il est possible de flasher du code sur la mémoire du robot, celle-ci est très limitée. Le Raspberry Pi 3 est un bon compromis étant donné qu'il offre une puissance de calcul suffisante pour pouvoir faire tourner notre architecture, et grâce à sa petite taille on peut le placer sur le dos du Thymio avec une batterie pour l'alimenter, sans gêner le robot dans ses déplacements.

Une autre contrainte importante était que nous travaillions avec des robots Thymio-II [5]. Ce sont des petits robots rectangulaires d'une dizaine de centimètres de large pour une dizaine de centimètres de long (11cm x 11.2cm). Ils sont plats d'une épaisseur de quelques centimètres (5.3cm) et sont munis de deux roues : une à gauche et une à droite. Ces robots perçoivent leur environnement à l'aide de capteurs infrarouges qui sont situés sur les faces avants, arrières, ainsi qu'en dessous. Ils sont également munis d'autres capteurs, notamment d'un thermomètre, d'un accéléromètre ainsi que d'un microphone. Il nous a donc fallu nous contenter de ces différents outils pour représenter l'environnement perçu par les Thymios. La figure 1 est une photo d'un Thymio-II.

Figure 1 – Thymio-II

2.2 Contraintes informatiques

D'un point de vue informatique, les premières contraintes établies avec notre encadrante sont les suivantes : l'architecture doit être implémentée en langage C, le code devra être Open-Source et bien documenté. Le problème étant que les Thymios ont un langage de programmation qui leur est propre (langage Aseba), il nous a fallu trouver un moyen de faire communiquer ces robots avec notre architecture implémentée en C. La solution que nous avons trouvé est d'utiliser le D-Bus qui est un logiciel de communication inter-processus. Nous avons alors utilisé une librairie spécifique en C qui permet de manipuler le D-Bus, `GDBus` (`#include <gio/gio.h>`). La communication entre le D-Bus et le Thymio se fait via une interface fournie par le robot, c'est l'interface "Asebamedulla". Ce mécanisme nous permet donc de faire appel aux fonctions Aseba du Thymio à travers notre programme en C en utilisant les chaînes de caractères associées au nom des fonctions, que nous avons trouvé sur la documentation en ligne du code source Aseba [4], ou des variables Aseba dont les noms sont répertoriés sur le site officiel du Thymio [5].

Toutefois, à ce stade du projet, nous ne connaissions pas grand chose sur le D-Bus, nous avons donc commencé avec les premiers éléments sur lesquels nous sommes tombés, c'est-à-dire l'API en C pour DBUS. Après quelques essais, nous sommes parvenus à établir une faible connexion avec le robot mais étions encore très loin d'arriver à récupérer les variables (moteurs, capteurs, etc) ou à lui envoyer des valeurs, et nous ne trouvions que très peu de documentation sur le sujet. Nous avons pris la décision d'abandonner cette méthode et de s'orienter vers la librairie `dbus-glib` (de `GLib`) qui offrait des fonctions plus variées et plus adaptées.

Avec celle-ci nous avons pu facilement établir la connexion entre le code et le robot, mais après de nombreux tests infructueux nous n'arrivions qu'à récupérer les valeurs de très peu de variables (seulement les `string`), et toujours sans possibilité d'envoyer des valeurs (aux moteurs par exemple). A nouveau, changement de direction pour se tourner vers une autre librairie des applications `GLib` : `GIO` qui fournit une autre API haut-niveau pour D-Bus, `GDBus`.

Nouvelle période d'adaptation mais plus d'exemples à notre disposition et une documentation assez complète [6] nous ont permis d'arriver à un résultat plus concluant et donc une bonne connexion au D-Bus, et des envois/réceptions possibles des valeurs des variables.

La recherche pour parvenir à cette dernière méthode d'accès au robot a été assez fastidieuse puisque les Thymio II sont relativement récents et la documentation à ce sujet était réduite.

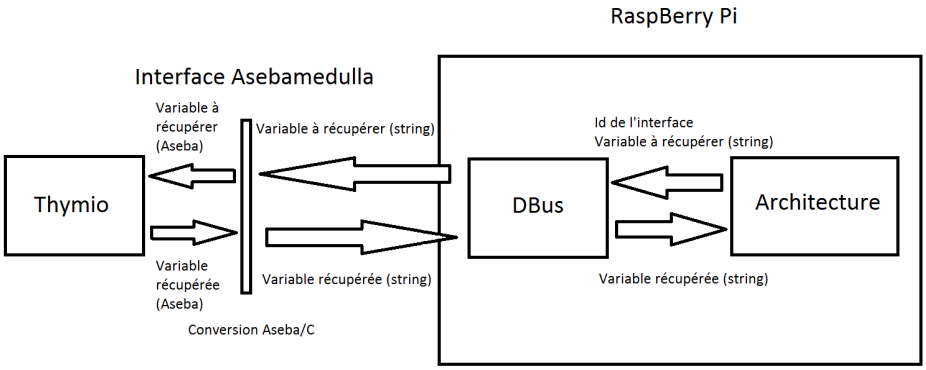


FIGURE 2 – Schéma de communication Raspberry/Thymio

La figure 2 détaille le procédé de communication entre le code de notre architecture, situé sur le Raspberry, et le robot Thymio.
Au préalable, nous relions l'interface Asebamedulla au robot Thymio via la ligne de commande `asebamedulla "ser:name=Thymio-II"` (les deux flèches les plus à gauche). Notre programme en C va ensuite établir une connexion de type `session` avec le D-Bus (deux flèches les plus à droite). Suite à cela, il va demander au D-Bus de se connecter à l'interface Asebamedulla (nom de la connexion : `ch.epfl.mobots.Aseba` et nom de l'interface : `ch.epfl.mobots.AsebaNetwork`). La connexion étant maintenant complètement établie (toutes les flèches de la figure 2), il est possible d'appeler les fonctions Aseba du robot.
Prenons l'exemple de la température : dans le code, nous demandons au D-Bus d'appeler la fonction "GetVariable" (string) du robot en lui donnant les entrées requises : "thymio-II" et "temperature" (strings). Il s'occupe ensuite de faire passer le message à l'interface Asebamedulla qui interagit alors avec le Thymio pour obtenir cette valeur, avant de la retourner au D-Bus. Finalement, ce dernier nous la retransmet.

3 L'architecture de contrôle

Suite à la prise en main de la communication entre le langage C et le robot, nous avons pu commencer à développer le code de façon organisée avec une architecture réfléchie.

3.1 Description de l'architecture déployée

Pour la conception de cette architecture nous avons effectué des recherches sur plusieurs documents tels que "Introduction to Autonomous Mobile Robots" de R. Siegwart et I. R. Nourbakhsh [7], le modèle de référence pour l'architecture de contrôle de telerobot [1] ou encore l'architecture mentionnée par Rodney A. Brooks dans son article intitulé "A Robust Layered Control System for a Mobile Robot" [2]. C'est principalement sur ce dernier que nous nous sommes appuyés pour cette partie. Il s'agit d'une architecture "à couches" composée de plusieurs niveaux, où les plus bas niveaux se contentent de récupérer des informations sur l'environnement ou sur l'état du robot à partir des données des différents capteurs, tandis que les niveaux supérieurs vont faire appel aux fonctions des modules des niveaux inférieurs pour exécuter des actions plus complexes (un déplacement, une cartographie, etc).

Notre architecture de contrôle est composée de six niveaux allant du numéro 0 au numéro 5. Le niveau 0 est le plus bas niveau et le 5 est le plus haut. On a alors le niveau 1 qui est relié au niveau 0 puisque le niveau 1 fait appel à des modules du niveau 0, puis le niveau 2 est relié au niveau 1 et au niveau 0 pour les mêmes raisons. Le niveau 3 fait appelle au niveau 0, 1 et 2. Le niveau 4 fait appelle au niveau 1 et 2, et enfin le niveau 5 qui fait appelle à tous les niveaux précédents.

Les explications suivantes décrivent dans les grandes lignes les fonctionnalités qu'offrent chaque niveau mais ne constituent pas une énumération exhaustive des fonctions de notre code. Celle-ci est fournie dans le manuel utilisateur que nous avons rédigé, et dans la documentation du code.
La figure 3 est un schéma permettant de mieux visualiser notre architecture :

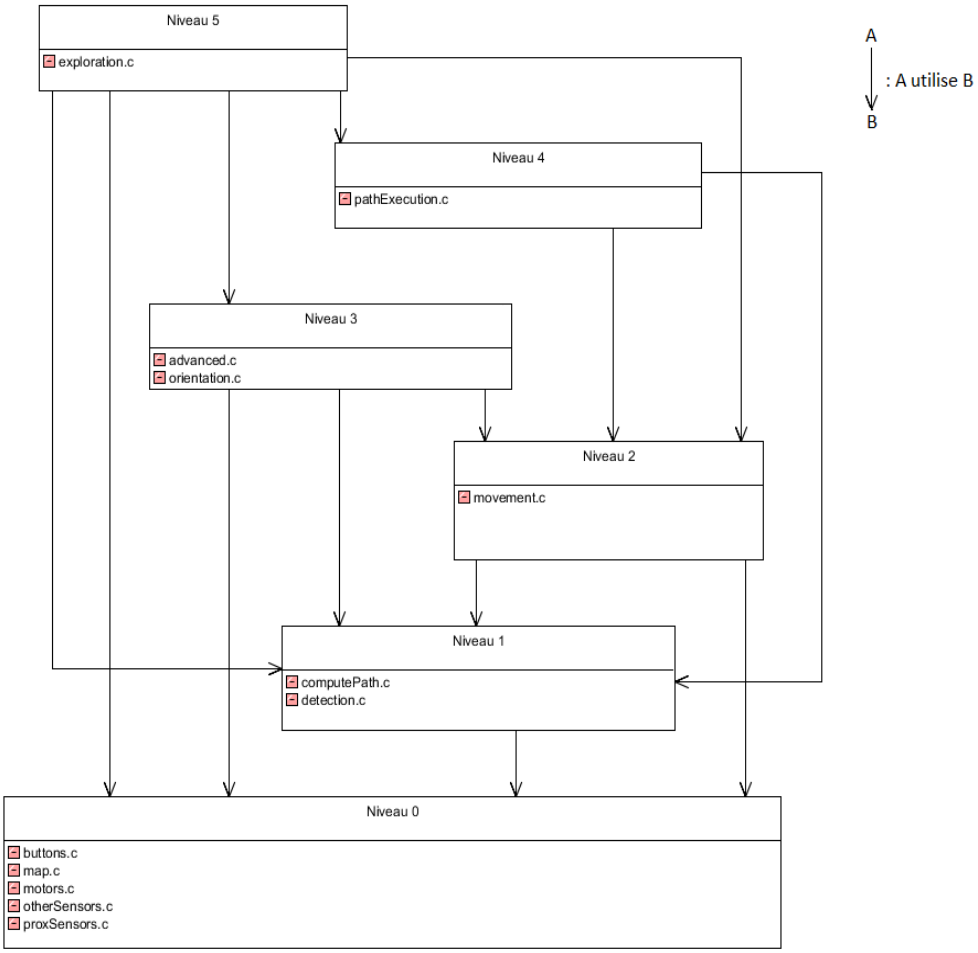


FIGURE 3 – Schéma de l'architecture de contrôle

3.2 Bas niveau

Le niveau 0 de notre architecture est le seul que nous puissions réellement considérer comme étant bas niveau. Il est essentiellement composé de modules permettant de recevoir les valeurs récupérées par les différents capteurs et d'envoyer des données vers le Thymio pour activer certaines fonctionnalités (notamment le fonctionnement des roues). Il permet aussi de récupérer les informations des fichiers textes "cartes" qui représentent l'environnement dans lequel se situe le Thymio. Ces fichiers doivent être écrits dans un format particulier défini dans

la partie 4 de modélisation de l'environnement (et dans le guide utilisateur). C'est le niveau qui servira d'interface entre notre programme informatique et le Thymio puisque c'est à travers lui que passent tous les échanges avec le robot.

3.3 Niveaux intermédiaires

Les niveaux intermédiaires sont les niveaux 1, 2 et 3. Ils sont qualifiés d'intermédiaires car ils ne permettent pas encore d'effectuer des comportements très complexes mais fournissent une bonne base pour le développement de ceux-ci. Ils sont principalement composés de modules de déplacements et de détection d'obstacles.

Le niveau 1 contient des fonctions qui vont faire appel aux fonctions du niveau 0, en particulier aux fonctions gérant les capteurs de proximité ainsi que celles lisant les entrées fichier. Ce niveau est principalement composé de fonctions de détection d'obstacles et d'une fonction de calcul de plus court chemin pour le robot Thymio. Cependant, il n'y a aucune fonction permettant le déplacement du robot, celles-ci sont implémentées dans les niveaux supérieurs.

Le niveau 2 est composé d'un unique module qui va permettre d'effectuer divers déplacements, des rotations, des déplacements linéaires avec ou sans détection d'obstacle (dans un cas si le robot rencontre un obstacle sur son chemin il s'arrête, dans l'autre cas il continue tout droit). Ce niveau a donc besoin d'accéder au module de détection d'obstacles ainsi qu'au module qui contrôle les moteurs du Thymio, c'est pour cela qu'il est connecté au niveau 0 et 1.

Le niveau 3 est à la frontière entre l'intermédiaire et le haut niveau puisqu'un de ses modules implémente un algorithme de déplacement aléatoire avec un algorithme d'évitement d'obstacle. Il contient également un module qui permet, à partir d'un angle donné et de l'orientation actuelle du robot, de déterminer dans quelle sens le robot va devoir pivoter pour effectuer sa rotation. Ce niveau nécessite donc d'être lié au niveau 2 pour les déplacements, au niveau 1 pour la détection d'obstacles et au niveau 0 pour certains réglages.

3.4 Haut niveau

Le haut niveau de notre architecture de contrôle est composé des niveaux 4 et 5. Ces niveaux implémentent des algorithmes complexes, notamment le parcours d'un point `A` à un point `B` dans un environnement connu ou encore l'exploration et la cartographie de lieu inconnu.

Le niveau 4 est constitué d'un unique module qui implémente un algorithme permettant d'exécuter un chemin, déjà déterminé, d'un point à un autre par le robot Thymio. Ce niveau nécessite donc d'être connecté au niveau 1 pour pouvoir déterminer ce plus court chemin à l'aide de la carte, ainsi qu'au niveau 2 pour les fonctions de déplacements.

Le niveau 5 est le plus haut niveau de notre architecture. Il est aussi muni d'un seul module, mais qui implémente notre algorithme le plus complexe : l'algorithme d'exploration avec cartographie. Ce niveau est donc forcément relié à tous les autres niveaux puisqu'il fait appel aux fonctions de déplacements, de

détections d'obstacles ainsi que celles d'orientation et de calcul de plus court chemin.

4 Modélisation de l'environnement

Jusqu'à maintenant, le robot ne faisait que des choix locaux en fonction des données de ses capteurs. Nous n'avions aucun moyen de lui donner une destination but. Pour pallier ce problème, nous avons choisi de discrétiser le monde réel en cases, et de le représenter avec un fichier texte (que nous appellerons "fichier carte"). Sur la première ligne nous trouvons le nombre de cases en largeur, puis en hauteur, ainsi que la taille des cases (en centimètres). La représentation de l'environnement commence ensuite sur la deuxième ligne. Un `x` représente un obstacle et un `o` pour informer que la case est libre. Le `b` correspond à la position initiale du robot et le `e` à la case objectif. La figure 4 est un exemple de fichier carte, et la figure 5 nous montre sa représentation réelle (vue de haut) où le point rouge correspond au robot et le point vert à sa destination but. Les bords noirs représentent les murs et le rectangle noir un obstacle.

```
8 8 10
xxxxxxxx
xoooeoox
xooooox
xooxxxox
xooxxxox
xooxxxox
xooboox
xxxxxxxx
```

FIGURE 4 – Fichier carte

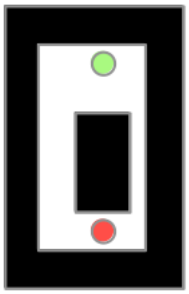


FIGURE 5 – Représentation réelle

C'est une représentation naturelle lorsqu'on discrétise le monde, et elle est suffisante pour nos applications. Toutefois, elle présente de nombreuses limitations (rigidité, taille, ...) par rapport à un modèle continu par exemple. Elle pourrait, par la suite, être remplacée par une grille d'occupation[3] qui n'est plus booléenne (obstacle ou non) mais avec des probabilités pour ainsi représenter l'incertitude liée aux capteurs.

5 Description des algorithmes

Dans la partie 3, nous avons brièvement évoqué les différents modules implémentés sans s'attarder sur la façon dont ils fonctionnent. Nous allons donc ici nous intéresser à certains algorithmes mis en place en commençant par celui

du vagabondage aléatoire (Random Walk), puis ceux du calcul et de l'exécution d'un chemin vers une destination, et finalement nous terminerons avec l'algorithme d'exploration d'un environnement inconnu.

5.1 Random Walk

Voici une description du vagabondage aléatoire, cet algorithme est assez simple à réaliser :

```
   input : Une vitesse de déplacement speed et une durée duration
 1 borneInf ← 15;
 2 borneSup ← 40;
 3 while temps écoulé < duration do
 4 |   distance ← entier aléatoire compris entre borneInf et borneSup;
 5 |   On avance de "distance" centimètres en s'arrêtant si on rencontre un
   |    obstacle;
 6 |   On affiche la distance parcourue;
 7 |   if (On détecte un obstacle) then
 8 |   |   On appelle la fonction d'évitement d'obstacle;
 9 |   end
10 |   Calcul d'un angle aléatoire compris entre 1 et 180;
11 |   Choix d'une direction aléatoire (droite ou gauche);
12 |   Tourner dans cette direction;
13 |   On affiche l'angle;
14 |   On met à jour le temps écoulé;
15 end
```

Algorithm 1: Algorithme du RandomWalk

L'algorithme est fait pour tourner une certaine période, donnée en entrée. Tant que cette durée n'est pas terminée, le robot sélectionne une distance aléatoire qu'il parcourt en s'arrêtant s'il rencontre un obstacle. La distance parcourue est volontairement bornée pour que le robot ne s'éloigne pas trop de sa zone de départ. Une fois qu'il a parcourue la distance aléatoire déterminée plus tôt, il va regarder s'il n'est pas cerné par des obstacles et s'éloigner de ceux-ci si c'est le cas pour éviter de se retrouver coincé. On choisit alors une direction aléatoire dans laquelle orienter le robot et on réitère la boucle tant que la durée d'exécution souhaitée n'est pas atteinte.

5.2 Chemin vers une destination

Grâce aux fichiers cartes définis dans la partie 4, nous avons donné au robot une représentation du monde avec la possibilité de lui indiquer une destination but. Pour calculer le chemin le plus court vers cet objectif, nous nous sommes tournés vers une implémentation de l'algorithme A*, représentée par l'algorithme 2. Cet algorithme prend en entrée un tableau représentant la carte (obtenue à partir d'un fichier carte par exemple) et retourne un tableau avec les

indices des cases à emprunter dans l'ordre. Pour ce faire, il initialise une copie du tableau carte en indiquant dans chacune des cases qu'elle est fermée (sauf "b"), qu'elle n'a pas de père (sauf "b", lui-même), qu'elle a un obstacle s'il y a un "x" et qu'elle n'en a pas sinon, et lui donne une valeur `h`, égale à la distance de Manhattan, et une valeur `g` nulle. Puis, tant que la case objectif est ouverte, on analyse les 4 cases autour de la case actuelle (au-dessus, à droite, en bas, et à gauche) et on actualise celles qui sont disponibles en mettant à jour leur père avec la case actuelle, leur valeur `g` avec celle de ce père + 1 et leur état à ouvert. On ferme alors la case actuelle et on en choisit une nouvelle parmi les ouvertes qui possèdent la plus petite valeur `h+g`. Finalement, lorsque la case objectif est fermée, il ne nous reste plus qu'à passer de père en père jusqu'à la case de départ pour trouver les indices des cases à suivre.

input : La largeur de la carte `width`, la hauteur `height` et `map`, un tableau de caractères de taille `height*width` représentant la carte

output: Tableau d'entiers contenant les indices des cases à emprunter dans l'ordre `pathToFollow` et la taille de celui-ci, `sizePath`

```
1  Déclaration de mapCell[height][width] qui est un tableau de structure
     Cell. Chaque Cell contient un entier i, un entier j, un entier h, en entier
     g, une case père, un boolean obstacle et un boolean open.;
2  Initialisation de mapCell en remplissant les indices i et j;
3  mapCell[i][j].open ← FALSE sauf pour la case "b";
4  mapCell[i][j].father ← NULL sauf pour la case "b" qui point sur
     elle-même;
5  mapCell[i][j].g ← 0;
6  mapCell[i][j].obstacle ← TRUE si "x", FALSE sinon;
7  mapCell[i][j].h ← distance de Manhattan entre cette case et la case "e";
8  iActuel ← i de "b";
9  jActuel ← j de "b";
10 while (mapCell[iEnd][jEnd].open) do
11 |   Analyse des 4 cases CaseX autour : au-dessus, à droite, en bas et à
   |     gauche;
12 |   mapCell[iCaseX][jCaseX].g ← mapCell[iActual][jActual].g + 1 //avec
   |     vérification que ça n'augmente pas g s'il est différent de 0;
13 |   mapCell[iCaseX][jCaseX].father ← &(mapCell[iActual][jActual]);
14 |   mapCell[iCaseX][jCaseX].open ← true;
15 |   mapCell[iActuel][jActuel].open ← false;
16 |   iActuel ← case ouverte avec le plus petit h+g;
17 |   jActuel ← case ouverte avec le plus petit h+g;
18 end
19 pathToFollow ← chemin construit en partant de la case "e" et en suivant
     les cases pères;
20 sizePath ← taille du chemin;
```

Algorithm 2: Calcul du chemin vers la destination finale

Suite à l'implémentation de l'algorithme précédent, nous pouvons calculer un chemin vers une destination donnée. Nous n'avons alors plus qu'à le faire exécuter par le robot. Pour cela, nous avons commencé par écrire une fonction d'exécution directe sans vérification d'obstacle. Lorsqu'il n'y a pas de `x` sur la case de la carte en entrée on la considère vraiment libre ; on n'utilise pas les capteurs, seule l'orientation du robot est actualisée entre chaque pas, et des mouvements linéaires d'une distance de la taille des cases sont exécutés.

Cependant, dans une optique d'exploration cela n'est pas très intéressant puisqu'il est nécessaire que la carte en entrée soit complète et à jour. Nous avons alors développé un algorithme plus dynamique qui met à jour la carte et recalcule le chemin le plus court à chaque obstacle inattendu rencontré, illustré par l'algorithme 3. En entrée, il faut lui fournir un tableau représentant la carte, l'orientation de départ, et le chemin à suivre (tableau indiquant les indices des cases à emprunter), et en sortie nous avons l'orientation après l'exécution du trajet. Tant qu'on n'est pas arrivé sur la case objectif, on essaie d'accéder à la prochaine case du chemin depuis la case actuelle. Si on rencontre un obstacle, on marque la case avec un "x" et on appelle l'algorithme précédent pour recalculer un chemin. Sinon, on avance d'une case et on met la position de "b" à jour.

input : La largeur de la carte `width`, la hauteur `height` et `map`, un tableau de caractères de taille `height*width` représentant la carte, la taille d'une cellule `sizeCell`, l'orientation de départ `orientation`, le chemin `pathToFollow` tableau d'entiers, la taille du chemin `sizePath`
output: Orientation à la fin `orientationOut`

```
1  iEnd ← i de "e";
2  jEnd ← j de "e";
3  for (i=0 ; i<sizePath-1 ; ++i) do
4      indexActuel ← pathToFollow[i];
5      indexEndPoint ← pathToFollow[sizePath-1];
6      On déplace le "b" sur la case actuelle ; On compare pathToFollow[i] et
         pathToFollow[i+1] pour connaître et effectuer la rotation par
         rapport à l'orientation actuelle;
7      On essaie d'avancer;
8      if (On croise un obstacle avant d'atteindre la case voulue) then
9          On recule de la distance parcourue avant détection de l'obstacle;
10         On marque la case pathToFollow[i+1] avec un "x";
11         On recalcule un chemin pathToFollow, on actualise sizePath et on
             recommence la boucle;
12     end
13 end
14 orientationOut ← orientation;
```

Algorithm 3: Exécution du chemin avec vérification des obstacles

5.3 Exploration

Le dernier module implémenté, et celui de plus haut niveau est l'exploration. Le robot est livré à lui-même dans un environnement inconnu et doit parvenir à produire un fichier carte représentant cet environnement, en étant assez proche de la réalité. Dans ce module, tous les autres niveaux de l'architecture sont utilisés. L'algorithme mis en place s'appuie principalement sur la recherche de la case non visitée la plus proche du robot avec un parcours en largeur, jusqu'à avoir tout exploré ou être piégé. Nous sommes partis sur cette méthode car les données récupérées par les capteurs sont assez limitées (capteurs infrarouges sur le devant du robot).

L'algorithme 4 permet d'avoir un aperçu de son implémentation. Nous avons en entrée la largeur du carré à explorer et en sortie l'écriture du fichier carte. Nous commençons par créer un tableau 2 dimensions pour représenter les cellules de la carte (ou plutôt deux, mais le second ne contient que les caractères pour produire le fichier). On suppose que le robot se trouve au centre (largeur / 2) et on initialise toutes les cases avec 'o', on indique qu'elles sont fermées et non explorées.

Tant qu'on n'a pas exploré toutes les cases accessibles (pour lesquelles il existe un chemin avec seulement des 'o'), on déplace le "b" sur la case actuelle, on indique qu'elle est explorée et on la ferme. Parmi les 4 cases autour (au-dessus, à droite, à gauche, et en-dessous), on s'oriente face à celles qui n'ont pas encore été explorées et on les ouvre. Si elles présentent un obstacle, on inscrit un 'x' et on les indique comme ayant été explorées. On réalise alors un parcours en largeur depuis la case actuelle jusqu'à trouver une case ouverte et non explorée. Si on en trouve une, on calcule et on exécute un chemin jusqu'à cette case (avec les algorithmes 2 et 3), puis on recommence. Sinon, cela signifie qu'on a fini de tout explorer ou qu'on est piégé (des murs ou des obstacles tout autour), on peut donc produire le fichier carte.

A noter qu'à tout instant le robot connaît son orientation, et sa position sur la carte.

input : Largeur du carré à explorer (le robot supposera qu'il est au centre) `width`, taille d'une case `sizeCell`, nom du fichier de sortie `filePath`
output: Ecriture du fichier carte

```
1  Déclaration de mapLilCell[height][width] qui est un tableau de structure
     LittleCell. Chaque LittleCell contient un boolean "explored" et un
     boolean "open";
2  Déclaration de map[height][width] qui est un tableau de caractères;
3  iActual ← (int)(width/2);
4  jActual ← iActual;
5  map[i][j] ← 'o';
6  mapLilCell[i][j].open ← FALSE;
7  mapLilCell[i][j].explored ← FALSE;
8  fin ← FALSE;
9  while (fin == FALSE) do
10 |   map[iActual*width+jActual] = 'b';
   |    mapLilCell[iActual*width+jActual].explored = TRUE;
   |    mapLilCell[iActual*width+jActual].open = FALSE;
11 |   Parmi les 4 cases "CaseX" autour, on s'oriente face à celles qui n'ont
   |    pas encore été explorées;
12 |   if (Obstacle en face) then
13 |   |   On place un "x" sur la case CaseX de map;
14 |   |   On passe explored de la case CaseX de mapLilCell à TRUE;
15 |   end
16 |   On passe open de la case CaseX de mapLilCell à TRUE;
17 |   On réalise un parcours en largeur depuis la case actuelle jusqu'à
   |    trouver une case de mapLilCell qui est TRUE pour open et FALSE
   |    pour explored;
18 |   if On trouve une case then
19 |   |   Calcul et exécution du chemin jusqu'à cette case;
20 |   end
21 |   if On ne trouve pas de case then
22 |   |   fin = TRUE;
23 |   end
24 end
25 Ecriture du fichier carte à l'aide du contenu de map;
```

Algorithm 4: Exploration

6 Tests de validation

Pour tester notre architecture de contrôle nous avons mis en place plusieurs méthodes de tests qui permettent d'évaluer les performances de celle-ci et ainsi de juger de son utilisabilité.

6.1 Tests des commandes de base

Pour tester nos fonctions bas niveau, nous avons utilisé le logiciel Aseba Studio qui est fourni sur le site web officiel des Thymios[5]. Ce logiciel permet, entre autre, de récupérer en temps réel toutes les informations liées aux différents capteurs du robot. Nous avons donc effectués plusieurs tests en plaçant divers obstacles autour du robot et en récupérant à travers les modules de notre architecture les valeurs que les capteurs du robot retournaient. Nous avons effectué une moyenne sur ces valeurs et nous les avons comparées aux valeurs retournées par le logiciel Aseba Studio en prenant soin de ne pas déplacer le Thymio de son environnement de test. Voici les résultats que nous avons obtenus :

Capteur pour le sol, robot déposé sur une surface beige

Sortie	Ambiant 0	Ambiant 1	Reflected 0	Reflected 1
Aseba Studio	21	28	807	726
Architecture	21	28	802	721

Delta 0	Delta 1
786	698
781	693

Température et vitesse des roues

Sortie	Température	V roue gauche	V roue droite
Aseba Studio	25.7	296	304
Architecture	25	300	300

Obstacle frontal situé à 1cm du robot

Sortie	Prox 0	Prox 1	Prox 2	Prox 3	Prox 4
Aseba Studio	2894	4540	4471	4553	1611
Architecture	2879	4534	4475	4540	1624

Obstacle situé à 1cm derrière le robot

Sortie	Prox 5	Prox 6
Aseba Studio	4592	4443
Architecture	4572	4428

Comme nous pouvons le constater, les résultats obtenus grâce à l'architecture de contrôle pour la réception des données des diverses capteurs sont très proches de ceux obtenus à l'aide du logiciel natif des robots Thymios. Nous pouvons donc en conclure que le fait de passer par le D-Bus et une interface Aseba pour convertir en C les valeurs récupérées depuis les capteurs du Thymio ne perturbe pas la validité de ces valeurs.

6.2 Tests avancés avec des scénarios

Nous avons principalement testé trois modules de haut niveau : les déplacements, le calcul et suivi d'un itinéraire d'un point `A` à un point `B`, et l'exploration avec la cartographie d'une zone. Nous avons filmé les tests effectués et avons placé les différentes vidéos dans un dossier "Tests" inclus dans le répertoire Git qui contient notre code. Les vidéos des tests pour les fonctions de déplacements basiques sont intitulées *squarre1* et *squarre2*, les vidéos pour les tests du calcul et du suivi d'itinéraire sont intitulées *goTo1* et *goTo2*, et enfin la vidéo pour le test de l'exploration est intitulée *explo*. Tous ces fichiers sont enregistrés au format mp4.

Notre méthode de test consiste à créer des scénarios dans lesquels le Thymio doit utiliser les modules que nous souhaitons tester. Nous comparons ensuite les résultats de ces scénarios obtenus aux résultats attendus lors du passage à la pratique, c'est à dire lorsqu'on les exécute réellement avec le robot.

Pour tester les fonctions de déplacements basiques nous avons donc conçu un scénario consistant à aller tout droit sur 30 cm, tourner ensuite à droite de 90 degrés, aller à nouveau tout droit sur 30 cm puis tourner de nouveau à droite de 90 degrés, et ainsi de suite jusqu'à obtenir un parcours prenant la forme d'un carré tout en retrouvant l'orientation initiale du robot à la fin à l'aide d'une dernière rotation. Ce test permet donc de vérifier que lorsqu'on demande au Thymio de parcourir une certaine distance ou d'effectuer une rotation d'un certain angle, celui-ci effectue bien l'action voulue. Les deux tests réalisés pour ce scénario ont donc été plutôt concluants puisque, comme nous pouvons le remarquer dans les deux vidéos *squarre1* et *squarre2*, le robot Thymio retrouve bien son emplacement et son orientation initiale à la fin de son parcours avec un très léger décalage de quelques centimètres.

Pour tester les fonctions de calcul et de suivi d'itinéraire, nous avons conçus un deuxième scénario dans lequel notre Thymio part d'un point `A` et veut atteindre un point `B` se situant à une cinquantaine de centimètres en face de lui mais un muret de 20 centimètres de long pour 10 centimètres de large se situe sur son chemin, il va donc devoir contourner ce muret d'une manière ou d'une autre pour atteindre sa destination. Comme on peut le constater sur les vidéos *goTo1* et *goTo2* le robot commence à aller tout droit puisque c'est théoriquement le chemin le plus court pour atteindre sa destination. Puis, il va détecter un obstacle sur sa trajectoire, il va donc calculer un nouvel itinéraire vers sa destination et faire le choix de contourner celui-ci par la gauche pour finalement

reprendre son chemin et atteindre la cible. A nouveau, nous obtenons le résultat voulu, ce qui témoigne du bon fonctionnement de nos fonctions de calcul et de suivi d'itinéraire.

Le dernier test que nous avons effectué nous a servi à évaluer le bon fonctionnement de notre algorithme d'exploration. Nous avons là encore conçu un scénario pour réaliser ce test. Dans ce scénario, nous plaçons le Thymio en haut à gauche d'une zone rectangulaire cernée d'obstacles, en bas à gauche de cette zone se trouve un espace. Si tout se passe bien le robot devrait commencer par explorer les cellules avoisinant la sienne pour cartographier la zone, il devrait répéter cela pour chaque cellule accessible, et lorsque qu'il n'existera plus de cellules voisines à la sienne non encore explorée il cherchera à atteindre des cellules plus éloignées qu'il n'a pas encore visité. Il devrait finir par s'arrêter lorsqu'il se rendra compte que la zone dans laquelle il se trouve est cernée par les obstacles. On peut observer dans le fichier vidéo *explo* que le Thymio explore bien la zone dans laquelle il se trouve en suivant le principe énoncé plus haut et que, de surcroît, il finit par s'arrêter après avoir fait le tour de la zone. Il ne rentre pas dans les obstacles et prend soin de s'en éloigner lorsqu'il les détecte. La figure 6 correspond à la carte qu'il a généré suite à l'exploration de cette zone :

```
9 9 10
OOOOOOOOO
OOOOOOOOO
OOOOOOOOO
OOOOXXXOO
OOOXOOOXO
OOOOXOOXO
OOOOXbXOO
OOOOOXOOO
OOOOOOOOO
```

FIGURE 6 – Contenu du fichier carte obtenu

Nous lui avions spécifié que la zone qu'il devait explorer était bien plus grande que la zone dans laquelle il se trouvait réellement, ce qui explique une telle dimension de la carte mais cela nous permet de constater que le robot arrête bien son exploration lorsqu'il se rend compte qu'il n'y a pas de chemin pouvant le mener aux cellules inexplorées de la carte. La partie qui nous intéresse est donc la partie située vers le centre, légèrement sur la droite de la carte. On identifie bien la zone rectangulaire cerné d'obstacles dans laquelle le Thymio se trouvait. De plus, la cellule où figure un "b" correspond bien à l'endroit où le robot a fini par se stopper.
Nous sommes donc plutôt satisfaits du résultat puisqu'il se rapproche beaucoup de ce qui était attendu lors de notre scénario. La zone cartographiée par le robot

se rapproche aussi beaucoup de la zone réelle dans laquelle se situait le Thymio.

7 Conclusion

Ce projet a été développé de façon continue sur l'ensemble du semestre avec des compte-rendus réguliers auprès de notre encadrante. Le cahier des charges, finalisé au début du mois de mars, a été globalement respecté et les objectifs atteints. En effet, les contraintes qui nous ont été imposées n'ont pas été violées, notre implémentation propose bien une architecture à niveaux dans lesquels les modules sont organisés de façon pertinente, et le plus haut niveau contient un module d'exploration pour le robot Thymio II.

Notre architecture est relativement souple et nous pourrions y intégrer de nombreuses extensions en prenant soin de bien les organiser par rapport au reste de l'architecture. Par exemple, l'ajout d'un module plus fiable de représentation du monde pourrait être mise en place. Il prendrait en compte des cartes basées sur le modèle "Occupancy grid map"[3] et ainsi nous n'aurions plus une représentation binaire de la carte (il y a un obstacle ou il n'y en a pas), puisque ce modèle est basé sur les probabilités. Nous aurions alors la possibilité de mieux gérer les cas d'environnements dynamiques (deux Thymios explorant la même carte par exemple), et de représenter l'incertitude des capteurs. Un module de communication pourrait aussi être ajouté, permettant ainsi d'implémenter des algorithmes d'exploration multi-agents et éviter de confondre un autre robot avec un obstacle divers. Enfin, comme l'ont montré les tests d'exploration et de parcours que nous avons effectué, la plus grande faiblesse de nos modules de hauts niveaux est liée à l'environnement et aux erreurs des capteurs : les divers frottements que va subir le Thymio au cours de ses déplacements vont petit à petit le dévier de sa trajectoire et avec les capteurs dont il est muni, le robot Thymio peut difficilement se rendre compte de cela. L'ajout d'un module manipulant de nouveaux capteurs (une caméra par exemple) serait donc l'idéal pour pallier ce problème.

Ce projet nous a aussi permis de découvrir un nouveau moyen de faire communiquer les processus entre eux, le D-Bus, et la réalisation d'une architecture logicielle était une chose relativement nouvelle pour nous. Cela nous apporte un bagage qui nous sera sûrement utile pour notre avenir dans le domaine informatique. De plus, au cours de ce projet nous avons pu nous mettre en condition réelle de développement d'un projet informatique par des professionnels (évaluation des besoins client et du système à fournir à l'aide du cahier des charges, autonomie pour la recherche de solution, compte-rendus de l'avancée du projet avec un planning à respecter, etc) et ainsi mieux cerner ce qui nous attend dans un futur proche lorsque nous seront confrontés à ce genre de problèmes.

8 Bibliographie

[1] James S. Albus, Harry G. McCain et Ronald Lumia. « NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) ». In : *NIST Technical Note* 1235 (1989).

[2] Rodney A. Brooks. « A robust layered control system for a mobile robot ». In : *Journal of Robotics and Automation* 2 (1985).

[3] Alberto Elfes et Carnegie Mellon University. « Using Occupancy Grids for Mobile Robot Perception and Navigation ». In : *Computer Society Press Los Alamitos* 22 (1989).

[4] Stéphane Magnenat. *AsebaNetworkInterface*. URL : `http://docs.ros.org/electric/api/aseba/html/classAseba_1_1AsebaNetworkInterface.html`.

[5] Stéphane Magnenat. *Thymio*. URL : `https://www.thymio.org/`.

[6] The GNOME Project. *High-level D-Bus Support*. URL : `https://developer.gnome.org/gio/stable/gdbus-convenience.html`.

[7] Roland Siegwart et Illah R. Nourbakhsh. *Introduction to autonomous mobile robots*. MIT Press, 2005.

9 Annexes

9.1 Cahier des charges

Cahier des charges

Lorthioir Guillaume, Billod Nicolas

Mars 2017

Contexte :

La robotique et l'intelligence artificielle étant deux domaines en plein essor, il est naturel pour les laboratoires de recherche en informatique de se doter des outils adéquats pour contribuer à l'avancée des ces deux domaines.

Dans ce contexte, le Laboratoire d'Informatique de Paris 6 (LIP6) a fait l'acquisition récente de robot Thymio 2. Il lui faut donc développer une architecture logicielle pouvant être supportée par ces robots et permettant de les contrôler. Cette architecture de contrôle devra permettre d'implémenter des comportements plus au moins complexes au sein des Thymios et être assez souple pour pouvoir s'adapter à d'autre types de robots.

L'espace mémoire fourni par les Thymios étant insuffisant, l'implémentation de l'architecture de contrôle se fera sur des nano-ordinateurs Raspberry Pi 3.

Besoins du client :

Objectif

L'objectif principal de ce projet consiste à développer une architecture de contrôle délibérative qui puisse permettre à des robots Thymio II couplés à des Raspberry Pi 3 de remplir des missions coopératives dans un environnement incertain et dynamique.

Architecture

Nous souhaitons que les robots agissent de façon autonome dans un environnement dynamique en utilisant une architecture de contrôle adaptée. Celle-ci comprendra au moins un module de perception et un module de délibération, exploitants les connaissances de l'environnement dans le but de déterminer les commandes à appliquer sur les effecteurs.

Nous souhaitons que l'architecture de contrôle implémentée soit une architecture "à couches", allant du plus bas-niveau (où les fonctions ne font que récupérer les informations des capteurs) au plus haut-niveau (où les fonctions réalisent des algorithmes plus complexes, comme la cartographie de l'environnement). Les fonctions de chaque niveau seront répertoriées dans un document annexe.

Perception

La perception de l'environnement par le robot s'effectuera à l'aide des capteurs infrarouges dont il est pourvu. Les valeurs réceptionnées par ces capteurs seront récupérées par l'ordinateur à travers le D-Bus et seront ensuite exploitées de manière à fournir une représentation du monde adéquate.

Déplacements

Les premières capacités de délibération mise en place seront des fonctions d'évitement d'obstacles et de navigations vers une destination donnée. L'incertitude liée aux déplacements peut être un problème, et ne doit pas être ignorée. Suite à cela, il pourra être envisagé de mettre en place des algorithmes déjà existants pour cartographier l'environnement au fur et à mesure des déplacements du robot.

Coopération

Les robots doivent aussi être en mesure de communiquer et de se coordonner pour réaliser des tâches coopératives, telles que l'exploration, la patrouille, etc. L'implémentation de ces fonctions ne fait pas partie de ce projet mais l'architecture mise en place devra prévoir l'intégration des informations reçues par message et l'exploitation, dans le module délibératif, de celles-ci.

Contraintes :

Le développement de ce projet est soumis à plusieurs contraintes :
- L'architecture de contrôle doit être programmé en langage C.
- Le code développé doit être open-source.
- Celui-ci sera disponible sur le compte gitLab du projet ThymSMA.
- La documentation technique et fonctionnelle du code fournie devra être écrite en anglais.
- L'architecture de contrôle développée est destinée à des robots Thymios 2, mais devra être compatible avec d'autre type de robots.
- Celle-ci devra être supportée par des nano-ordinateurs de type Raspberry Pi 3.
- La communication entre le robot Thymio et le Raspberry Pi 3 s'effectuera à l'aide du D-Bus et de l'interface asebamedulla.
- Un module de communication pour interagir avec un autre projet en développement est à prévoir.
- Des modules complémentaires doivent pourvoir être ajoutés sans perturber l'architecture initiale.

Solution envisagée :

Nous envisageons d'implémenter une architecture "à couches" en nous inspirant des écrits de Rodney A. Brooks, c'est-à-dire que les différentes couches de l'architecture suivront une certaine hiérarchie : si le robots souhaite exécuter un comportement complexe, la couche de plus haut niveau sera en charge d'appeler cette fonction qui, elle-même, appellera des fonctions appartenant à une couche de plus bas niveau, qui à leur tour appelleront des fonctions d'un niveau encore plus bas et ainsi de suite jusqu'à atteindre les fonctions de bases déjà implémentées dans le robot Thymio. Nous développerons cette architecture en C, et les différents niveaux de celle-ci seront répartis sur différents fichiers, c'est la manière dont les fichiers se référenceront les uns par rapport aux autres qui définira l'architecture en question.

La dernière transition (entre les fonctions du robot et les fonctions de notre code) se fera très probablement avec l'aide de la librairie fournie par GDBus.

La création d'un support amovible se fixant au robot Thymio nous permettra d'y installer le Raspberry Pi ainsi que son alimentation pour pouvoir permettre l'autonomie du robot. Ce support permettra aussi d'insérer d'autres périphériques, comme par exemple une caméra pour permettre une meilleure identification de l'environnement.

9.2 Carnet de Bord

Introduction

Notre sujet de recherche était axé sur les architectures de contrôle pour les robots mobiles. Ce sujet contient plusieurs dimensions : une première liée à l'architecture de notre code source, c'est à dire la manière d'agencer et de faire communiquer entre eux les différents modules qui le composent. Une seconde liée à la communication entre notre architecture de contrôle et le robot. Ainsi qu'une dernière dimension liée à la robotique elle-même puisqu'il faudra adapter notre architecture de contrôle en fonction du robot pour lequel celle-ci sera implémentée. Nous l'avons donc traité sous l'angle de jeune informaticien n'ayant pas encore beaucoup de pratique dans le domaine de la robotique mais ayant de bonnes connaissances théoriques.

Mots clés

Voici une liste de mots clés que nous avons fait ressortir de notre sujet :

— Robots mobiles, Architecture logiciel, Communication, DBus, Thymio-II, Raspberry-Pi, Aseba, Orientation d'un robot, Déplacement d'un robot, Modules, Cartographie, Exploration, Représentation du monde.

Descriptif de la recherche documentaire

La plupart des documents bibliographiques nous ayant été fournis au début du projet par notre encadrante, nous n'avons pas eu besoin d'utiliser les outils de recherche fournis par l'UPMC pour nos recherches bibliographiques. Cependant, un outils qui nous a été essentiel est le Web puisque notre sujet de recherche étant relativement récent (langage C pour les robots Thymio II), il n'existait pas de livres ou les revues pouvant nous renseigner sur la partie liée à la communication entre notre programme implémenté en langage C et notre robot Thymio-II. Nous avons donc du parcourir le Web à la recherche de forums ou de pages rédigées par des personnes qui, comme nous, souhaitaient faire communiquer un programme implémenté en C avec un périphérique auxiliaire.

Dans un premier temps, nous nous sommes inspirés d'une page web d'une personne suggérant d'utiliser une certaine API en C pour communiquer avec le robot. Toutefois, après plusieurs essais les résultats se sont trouvés être bien en dessous de nos espérances et nous avons donc du utiliser à nouveau la recherche Internet pour trouver une solution à notre problème. Nous avons fini pas trouver un site Internet expliquant le fonctionnement d'une librairie C qui nous permettait de communiquer de manière convenable avec notre robot.

Bibliographie

- **Livre :**

Roland Siegwart and Illah R. Nourbakhsh, *Introduction to autonomous mobile robots*. MIT Press, 2005.

- **Article :**

Rodney A. Brooks, "A robust layered control system for a mobile robot", *Journal of Robotics and Automation*, vol. 2, 1985.

James S. Albus, Harry G. McCain and Ronald Lumia, "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)", *NIST Technical Note*, vol. 1235, 1989.

- **Site Internet :**

Stéphane Magnenat, "Thymio", *Thymio*, 2017, https ://www.thymio.org/. [Accessed : May-2017]

The GNOME Project, "High-level D-Bus Support", *GNOME DEVELOPER*, 2014, https ://developer.gnome.org/gio/stable/gdbus-convenience.html. [Accessed : May-2017]

Stéphane Magnenat, "AsebaNetworkInterface", *Thymio*, 2017, http ://docs.ros.org /electric/api/aseba/html/classAseba_1_1AsebaNetworkInterface.html. [Accessed : May-2017]

Analyse des sources

Une des sources qui nous a été la plus utile était le premier des sites internet de la bibliographie, c'est-à-dire le site web officiel des Thymio. Nous l'avons découvert en essayant de nous documenter sur le robot Thymio-II avec lequel nous avons travaillé lors de notre projet. Pourquoi ce site en particulier ? Parce qu'il a été mis au point par le créateur des robots Thymio et nous pensons donc que son niveau de fiabilité est maximum. Ce site nous a été extrêmement utile pour comprendre comment fonctionnait notre robot, comprendre son langage de programmation et la manière dont on pouvait connecter le robot Thymio à notre Raspberry-Pi.

Un second site web indispensable est celui qui figure à la deuxième place de notre bibliographie, "High-level D-Bus Support". Nous l'avons découvert en effectuant une recherche sur le D-Bus et la manière de communiquer avec celui-ci. Nous avons effectué cette recherche suite à un premier échec sur la documentation du D-Bus. En effet, nous avions trouvé un site web qui fournissait des explications sur celui-ci ainsi qu'une API pour communiquer avec le D-bus mais ce n'était malheureusement pas suffisant. Niveau fiabilité, la page web "High-level D-Bus Support" est sans défaut puisque toutes ses indications ont donné de bons résultats et que cette page web a été développé par une équipe de développeurs fiables. Cette source nous a été très utile pour faire le lien entre notre programme informatique et le D-Bus.

Une dernière source qui nous a été précieuse était l'article rédigé par Rodney A. Brooks, "A robust layered control system for a mobile robot". Elle nous a été fourni par notre encadrante de projet dans le but de mieux comprendre ce

qu'était une architecture de contrôle. Cet article a donc un très bon niveau de fiabilité et nous a permis d'élaborer notre architecture de contrôle pour un robot mobile à l'aide des indications qui y étaient mentionnées.

9.3 Manuel utilisateur

User guide

Thymio-II architecture

Billod Nicolas
Lorthioir Guillaume

Jan-May 2017

Summary

1 Introduction

This document is a user guide for the code we wrote for a project during our first year of Master at the Pierre et Marie Curie university, France. The name of this project was "Control architecture for a fleet of mobile robots" and our supervisor was Aurélie Beynier from the multiagent group (SMA) of the LIP6 (Laboratoire d'Informatique de Paris 6).

First, we will explain how to install the necessary packages for running our code, and then we will talk about its architecture and describe the modules that it implements

2 Getting ready to use

The first thing to do before even trying to compile is to get the GDBus library brought by the GLib package. But, most likely, it has already been installed with your Linux distribution. Then, you should be able to compile the code using `make` but not execute it yet.

To do so, you will have to go to `https://www.thymio.org/en:linuxinstall` and follow the steps to install the Aseba package for your distribution. We will walk you through for the Raspbian one (jessie-raspbian distribution on our Raspberry Pi 3).

Raspberry Pi

Download `libdashel_1.1.0_armhf.deb`, `libenki_2.0_armhf.deb`, and `aseba_1.4.0_armhf.deb` packages from the thymio website. Then, use this command line to install them:

```
$ sudo dpkg -i packagename.deb
```

starting with `libdashel` and `libenki`, and then `aseba`. You may encounter some errors if your distribution is not up to date, to fix it you can try to execute the command lines:

```
$ sudo apt-get update
$ sudo apt-get -f install
```

and then try again.

Now that the packages are installed, you need to be connected in ssh with the -X for that part (or directly to the raspberry pi):

```
$ asebastudio "ser:device=/dev/ttyACM0"
```

It should launch the Aseba Studio with the Thymio II, assuming he is under `/dev/ttyACM0` (the `lsusb` command can be useful). Though, to run our code it is not the Studio we are interested in but the Asebamedulla program. Now, you have two possibilities: either launch a new terminal and execute

```
$ asebamedulla "ser:name=Thymio-II"
```

or just do it with a `&` but then you will have to kill its running processor. Once it is running, you can execute our code normally.

3 Code architecture

We organized the code using an architecture based on levels. In each level, there is at least one module (group of similar functions) which uses modules from sublevels; expect for Level 0 where modules do not need anything else. The `include` directory contains all the headers.

The `main.c` file contains the main function in which calls to modules are made. Most of them are already there in comments. To compile everything, use the command line `make`, and execute the generated `test` file.

3.1 Level 0

This level is used to collect data from the sensors, and for sending directives to the robot. Basically, getters and setters for motors, sensors and buttons, but also some functions for handling the maps.

A **map** file needs to have a precise form. It is a text file with on the first line the number of cells in width, in height, and the size of a cell in centimeters. Then, the mapping should start on the second line. An `x` means that there is an obstacle on the cell and an `o` means it should be free. The `b` is for the initial position of the robot and the `e` is for the objective (end point). Figure 1 shows an example of a map file. At the moment, those files are located with the `main.c` and are only needed by the `sizeMap()` and `toMap` functions from Level 0 (to get the width, the height, the size of cells and to turn it to a table of characters).

```
8 8 10
xxxxxxxx
xoooeoox
xoooooox
xooxxxox
xooxxxox
xooxxxox
xoooooox
xxxxxxxx
```

Figure 1: Content of a map file

The following files are located in this level.

- `proxSensor.c`: Getters for horizontal, ground ambiant, ground reflected, and ground delta sensors.

- `otherSensors.c`: Getters for temperature, accelorometer, and sound intensity values.
- `buttons.c`: Getters for forward, backward, left, right, and center button states (pushed or not).
- `motors.c`: Getters and setters for motors values. It goes from -500 to 500. A value of 500 approximately corresponds to a linear speed of 20 cm/s.[1]
- `map.c`: A getter from a map file for width, height, size values, and a table of characters (maximum of TAILLE_Max characters). Also a function to write a map file from those values.

3.2 Level 1

This level is used to detect obstacles, and also to compute a path from the `b` point to the `e` point. The first module calls a lot of functions from level 0, whereas the second one doesn't; it is only there because it requires a map (obtained using the map module from level 0) as an entry.

- `detection.c`: Module to dectect if there is an obstacle in front of the robot, on its back, its left, or its right. Also a function for the ground sensor and one to dectect sound.
- `computePath.c`: Compute a path to the objective from a map using the A* algorithm. It returns a table with the indexes of cells in the map to follow to reach the target.

3.3 Level 2

This level is used for all the robot's movements. The linear trajectory movement has been implemented three times, once with no paramaters except for the speed, once with a distance (centimeters), and once with distance and check for obstacles.

- `movement.c`: stop, linear movement with different paramaters (e.g. distance to travel, with a check for obstacles), and turn left or right.

3.4 Level 3

This level is used to orientate the robot. To be able to know which way the Thymio was facing in the world, we used the orientation circle shown in Figure 2. So, if its orientation is 0 it is facing the top side of the map file (cf. level 0), if it is 90 it is facing the left side, and so on.

We also used this level to implement a randomwalk module allowing the robot to wander around.

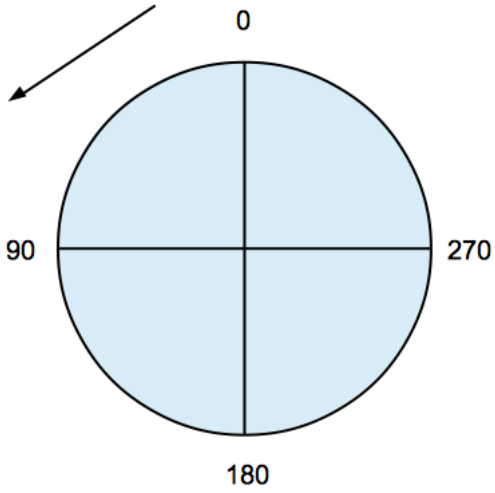


Figure 2: Orientation used by the robot

- `advanced.c`: A randomwalk with its own obstacle avoidance.
- `orientation.c`: Orientate the robot in the right direction (orientation) using Level 2 turn right and turn left functions.

3.5 Level 4

This level is used for executing a path computed in Level 1. It also needs the orientation and the movement modules from Levels 3 and 2.

- `pathExecution.c`: Either execute a path without checking for obstacles or with checks (doesn't update the map file yet but only the robot's internal map).

3.6 Level 5

This level is used for the exploration. It is the highest one so far and it needs the map module (Level 0), the detection module (Level 1), the path computing module (Level 1), the orientation module (Level 3) and the path execution one (Level 4) to work.

The robot will create a map file with a square grid with a given width. It will assume he is in the center cell and will start exploring from there, until there is no unexplored cell or the remaining ones are not accessible.

- `exploration.c`: Execution of an exploration ending with the generation of a map file.

4 Contact

If you have any questions about this user guide or the code itself, don't hesitate to contact us at gui.lorthioir@gmail.com and nicolasbillod@gmail.com. The written report can also be of help because it includes some detailed algorithms we used.