UNIVERSITÉ TOULOUSE III PAUL SABATIER

Université de Toulouse

Travail d'étude et de recherche
licence informatique 3éme année

Soutenu le 8 juin 2015

Lacheray Benjamin

Pilotage d'un robot Lego Mindstorms NXT : Cartographie d'un labyrinthe et sortie rapide en deux temps

Sous la direction de :

Christine Rochange - Florence Bannay - Jérôme Mengin

2014 - 2015

Remerciements

Je remercie les enseignants qui nous ont encadré pour ce travail, d'avoir été là en cas de besoin, de la liberté qu'ils nous ont accordé pour le sujet, et de nous avoir permis d'utiliser ce matériel et ainsi initié à un domaine très passionnant.

Je remercie également Haluk OZAKTAS pour son aide et le temps consacré pour des installations très techniques.

Notre équipe

Ce travail d'étude de recherche dénombrait six étudiants. Rapidement, deux équipes se sont formées.

Ci-dessous, voici mon équipe.

ANTOINE Kevin, LACHERAY Benjamin (moi-même), Hydra (le robot) et MOUGEOT Matteo

Merci aussi à eux.

Table des matières

I. Introduction

Ce TER, travail d'étude et de recherche, s'inscrit dans le cadre de ma formation, une licence informatique. Il me permet de mettre en application mes connaissances, aussi bien celles acquises lors de mon cursus que celles acquises en auto-formation. Ce TER m'apporte également une nouvelle expérience, un projet en équipe avec une réelle autonomie, et sur un sujet très passionnant : la robotique.

Ce TER consistait à développer une application de pilotage pour un robot Lego Mindstorms NXT.

Le NXT est un robot conçu par Lego, il est constitué d'une « brique intelligente » programmable (pouvant communiquer en USB et en Bluetooth), de capteurs, de servomoteurs, et puis de simples éléments Lego. Le point fort de ce robot est sa totale modularité, comme un jeu de Lego classique, il peut prendre de nombreuses formes, ce qui est vraiment intéressant et même « amusant » en pratique. Il y a cependant quelques contraintes, la technologie du NXT n'étant pas toute récente (2006), le robot est limité en mémoire et en calcul[1]. Il faut donc faire attention à l'efficacité des programmes que nous développons, pour ne pas consommer trop de ressources matériels.

Au départ, quand le TER nous a été proposé, le sujet n'était pas clairement défini. Il était simplement question de « *Pilotage d'un robot LEGO en milieu hostile* », avec comme objectif d'amener le robot à réaliser certaines actions pour atteindre un but donné, en s'inspirant par exemple des jeux Sokoban ou Roborallye. Il aura fallu attendre deux semaines et trois réunions pour avoir une idée de sujet plus précise : la cartographie d'un labyrinthe, puis la sortie la plus rapide du robot d'un coté à l'autre de ce même labyrinthe.

Ce rapport retrace notre parcours, de la découverte des robots jusqu'à la réalisation final du projet. La première partie traite de nos travaux et de nos recherches avant que le sujet ait été fixé. Si c'est uniquement le projet en lui-même qui vous intéresse, rendez-vous page 12, partie « VIII. Dans le vif du sujet ».

1 48 MHz, 64 KB RAM, 256 KB Flash - http://en.wikipedia.org/wiki/Lego_Mindstorms_EV3#Overview

II. Découverte des robots

Lors de notre première réunion, nos tuteurs nous ont fournis les robots. Une boite a été donné à chacun d'entre nous comprenant :

- une « brique intelligente »
- 3 servomoteurs
- un capteur de distance (à ultrasons)
- un capteur de luminosité
- un capteur de son
- un capteur de pression
- divers éléments Lego utile pour la construction

Il nous a ensuite était expliqué que les robots avaient été flashés avec le firmware nxtOSEK.

À la base, les NXT sont équipés du firmware Lego officiel et sont programmables via le logiciel NXT-G. Ce logiciel permet une programmation « graphique », il est de ce fait un peu contraignant et limité. C'est donc, pour palier à ce problème, que nos robots étaient équipés de nxtOSEK.

NxtOSEK est une adaptation du système OSEK (« Systèmes ouverts et interfaces correspondantes pour l'électronique des véhicules automobiles ») pour le robot Lego. Ce système d'exploitation temps réel possède un système de tâches et de priorités bien élaboré. Grâce à ce firmware, il est possible de programmer les robots dans le langage C (ou C++).

Notre première mission donnée lors de cette réunion, fut la mise en place de notre environnement de travail. Nous avions une semaine pour accomplir cette tâche.

III. Premier pas avec nxtOSEK

1. Installation du cross-compilateur

Un cross-compilateur est un compilateur capable de compiler des programmes pour une autre plate-forme que celle sur laquelle il fonctionne. Dans notre cas nous compilions sur PC des programmes pour le NXT.

L'installation du cross-compilateur fut assez difficile à mettre en place. Tout d'abord, l'installation était possible seulement sur une distribution Linux (problématique pour mes collègues utilisant Mac et Windows). Ensuite, le script d'installation n'était pas vraiment à jour et cela posait également des problèmes sur de nombreuses distributions Linux trop récentes (dont la mienne). Afin de régler ces problèmes, nous avons tous installé une distribution dont on nous avait assuré la compatibilité avec le script d'installation : Scientific Linux (dans une machine virtuelle).

Il aura quand même fallu rechercher et installer de nombreux paquets logiciels manquants et même modifier légèrement le script pour que tout s'installe correctement.

2. Premier montage des robots

Après avoir mis en place notre environnement de travail, nous avons monté nos robots en suivant le modèle de base. Il s'agissait juste d'avoir un robot fonctionnel afin de pouvoir commencer à programmer, peu importe la forme du robot.

« *Golf-bot* » Modèle de base. Nous avons monté le même, le bras en moins, et le capteur de luminosité dirigé plus vers le bas.

3. Hello world

Une fois le cross-compilateur installé, il n'y avait plus de grandes difficultés pour lancer nos programmes sur le NXT. Après un premier hello world, de nombreux programmes ont suivis dans le but de nous familiariser avec le robot et l'environnement nxtOSEK. Nous avons ainsi tester les différents capteurs et les moteurs.

Notre première tâche était maintenant accomplie. Notre environnement de travail était prêt et nous nous nous étions familiarisé avec celui-ci.

Bien que nous nous soyons servis de nxtOSEK pendant plus de deux semaines, nous nous sommes ensuite séparé de se système par nécessité d'obtenir une communication plus simple avec le PC (j'y reviendrai par la suite).

IV. Recherche sur le possible

1. Seconde réunion

Lors de la seconde réunion avec les tuteurs, le sujet n'était toujours pas défini. Ils nous ont alors confiés 3 missions afin d'avoir une idée de ce qui était réalisable avec les robots :

- Recherche sur le capteur de distances afin de se situer
- Recherche sur le capteur de son afin de se situer
- Recherche sur la communication Bluetooth

Une mission a été confiée à chaque membre de l'équipe.

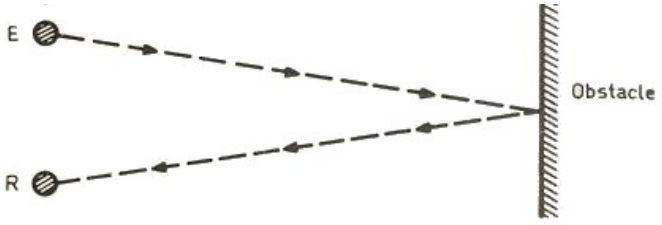
2. Résultats des recherches sur le capteur de distances

Le capteur de distances fonctionne avec des ultrasons et n'est pas parfait. Cela implique plusieurs choses :

- la forme des objets à détecter est importante
- il faut faire attention à la portée du capteur
- le capteur est imprécis

i. La forme des objets à détecter

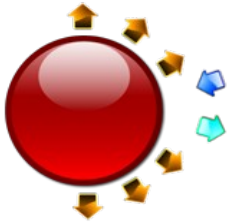
La forme des objets à détecter est importante. L'idéal est un objet bien plat, perpendiculaire à l'onde sonore émise :



Obstacle idéal pour le capteur de distance.

Dans le cas où l'objet n'est pas plat, une partie de l'onde sonore n'est pas renvoyée vers le récepteur et ainsi il peut arriver que l'objet ne soit pas détecté :

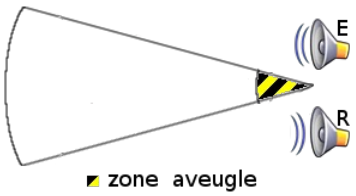
Renvoi des ondes sonores par un objet arrondi, une grande partie du signal est perdu.



Idem pour les objets plats mais avec une inclinaison non parallèle au capteur, le signal est largement perdu.

ii. La portée du capteur

La portée minimale du capteur n'est pas de 0. Quand l'onde sonore est émise, le récepteur est mis en pause le temps que l'onde s'éloigne afin que celle-ci ne soit pas détectée durant cette phase. Ceci implique qu'un objet trop près ne sera pas détectée puisque le récepteur sera en pause.



La zone « aveugle » dû à cette pause du récepteur.

En nous documentant, nous pouvons lire que la détection minimale est de 5cm. D'après nos tests, on peut parfois légèrement aller en dessous de cette limite, mais bien souvent le capteur ne détecte rien.

Concernant la portée maximale, les spécifications du capteur disent jusqu'à 255cm. En pratique, la valeur maximale s'approche plus des 170cm.

iii. La précision du capteur

Le capteur n'est pas très précis, il donne une bonne idée de la distance de l'objet mais son imprécision peut aller jusqu'à plusieurs centimètres. Par exemple, un objet situé à 15cm peut être détecté à 13cm… Ce qui nous a posé des problèmes par la suite, car nous aurions eu besoin de mesurer des distances de manière extrêmement précise.

iv. Conclusion

Le capteur de distance permet de détecter des objets bien plats, parallèles à celui-ci, et de donner une mesure de distance avec une précision de l'ordre du centimètre.

La conclusion pour notre problème initial, *se repérer dans un espace avec au moins 2 murs*, est que c'est possible, à condition de ne pas rechercher la précision absolue.

3. Résultats des recherches sur le capteur de son

L'idée de départ était de positionner un appareil immobile émettant un son d'une intensité constante. Le robot détectait ensuite ce son, et suivant son intensité il devait être capable de savoir s'il était proche ou non de la source sonore.

Trois problèmes se posent pour mettre au point cette solution de repère.

i. Du bruit parasite

Le robot s'entend lui même : en se déplaçant le robot produit du son, plus il va vite et plus il est bruyant. Ce bruit est en général de faible intensité mais il vient quand même parasiter le bruit de la source sonore et nuit aux résultats.

ii. Capteur imprécis

Le capteur de son n'est pas très précis : pour un même son il donne parfois des valeurs différentes. De plus, certains types de bruits sont moins bien détectés que d'autres. Par exemple, un claquement de doigts à 10cm du capteur ne se détecte pas beaucoup, tandis qu'un claquement de main assez éloigné se détecte mieux.

iii. Pas très agréable...

Un bruit constant est désagréable. Il n'est donc pas envisageable de placer une source sonore trop bruyante, pour le bien de tous…

iv. Conclusion

Cette solution n'est à retenir qu'en dernier recours. Nous pourrions grossièrement évaluer la distance de la source sonore… Mais vraiment pas de manière précise.

4. Résultats des recherches sur la communication Bluetooth

Le Bluetooth a dès le départ été annoncé comme une composante importante de notre sujet. L'idée était de combler le manque de performance du robot en le faisant communiquer avec un PC qui effectuerai les calculs à sa place. De cette manière, le processeur et la mémoire du robot pourraient être soulagés. Il était aussi envisagé de faire communiquer plusieurs robots entre eux.

Les recherches sur le Bluetooth devaient donc permettre de savoir si ces deux points étaient techniquement possible.

Après avoir acheté une clef Bluetooth, je me suis lancé dans de premiers essais de communication. La documentation[1] parlait en premier lieu d'un logiciel déjà existant : NXT Gamepad[2]. Ce logiciel permet de contrôler le NXT grâce à une manette branchée sur le PC, et d'y récupérer des données (valeurs en provenance des ports des capteurs, le niveau de batterie, le temps d'exécution du programme, etc.). Cependant, rien que pour utiliser cette application, ce fut très compliqué. Dans un premier temps, j'ai dû installer Windows dans une machine virtuelle pour la

1 http://lejos-osek.sourceforge.net/ecrobot_c_api_frame.htm
2 http://lejos-osek.sourceforge.net/nxtgamepad.htm

lancer. Après ceci, le « pairage » Bluetooth entre le PC et le NXT s'est révélé assez hasardeux : parfois le NXT était détecté, parfois non. Parfois le NXT demandé un mot de passe, parfois non. Parfois le pairage fonctionné, parfois non… Difficile de dire si le problème venait de mon PC, du NXT, de ma clef Bluetooth, du fait que mon Windows fonctionnait dans une machine virtuelle, ou encore si c'était tout simplement l'application ou le système nxtOSEK qui était difficile à utiliser.

Après de multiples essais, j'ai réussi à faire fonctionner NXT Gamepad. J'ai pu récupérer des données du NXT jusqu'au PC. Cependant, cette réussite n'était pas vraiment utile puisque nous avions besoin de transférer des données dans les deux sens (NXT → PC / PC → NXT).

Après avoir lu de nombreux articles traitant de la communication BT sur NXT, j'ai vite compris que cela serai compliqué avec le système nxtOSEK. Celle-ci est possible dans un sens et est compliquée de manière bilatérale. Pour faire communiquer plusieurs robots entre eux, il est indispensable de passer par un PC maître.

J'ai donc recherché une solution simple et déjà existante pour pouvoir communiquer facilement en Bluetooth, car ce problème pouvait être à lui seul un sujet de TER, et je ne voulais pas perdre trop de temps sur cette problématique.

C'est ainsi que j'ai découvert une API PC en Java[1] qui permet d'établir une communication Bluetooth simple avec le NXT, dans les deux sens. Mais je ne savais pas encore : cette API était uniquement compatible avec un firmware autre que nxtOSEK. Je venais en fait de découvrir NXJ LeJOS[2].

V. Fixer le sujet

1. Troisième réunion

Lors de la 3ème réunion avec les tuteurs, nous avons présenté tous ces résultats. Puis un sujet s'est dégagé : un labyrinthe dans lequel un ou plusieurs robots devraient sortir.

Notre mission suivante fut l'élaboration du cahier des charges, pour fixer complètement le sujet avec les étudiants sur ce TER.

2. Rédaction du cahier des charges

Le lendemain suivant cette réunion, nous nous sommes donc réunis sur Skype afin que chacun de nous présente ses idées sur le sujet.

Dés que le mot labyrinthe fut évoqué, j'ai tout de suite pensé à une compétition déjà existante qui m'a toujours impressionnée : la « micromouse contest »[3]. Une des épreuves de cette compétition se déroule en deux temps : d'abord la cartographie d'un labyrinthe par un robot, puis la sortie la plus rapide possible de ce même labyrinthe.

1 http://www.lejos.org/nxt/pc/api/index.html
2 http://www.lejos.org/nxt/nxj/tutorial/index.htm
3 https://www.youtube.com/watch?v=CLwICJKV4dw

J'ai donc proposé cette idée et elle fut adoptée à l'unanimité. Nous avons ensuite débattu sur de nombreux points pour élaborer précisément le cahier des charges.

Je vous le présente ci-dessous en version final.

3. Cahier des charges

i. Le labyrinthe

- Labyrinthe sous forme de **quadrillage**.

- Le sol du labyrinthe est **blanc** (ou juste une **couleur claire** pouvant être facilement différenciée du noir).

- Les cases du labyrinthe sont délimitées par des lignes **noires** (du Scotch noir par exemple).

- Les murs sont des **obstacles physiques** situé sur les extrémités des cases (carton, etc…).

- Le départ et l'arrivée sont positionnés à l'intérieur du labyrinthe, sur les côtés, et ses cellules ne sont pas connues du robot.

- L'arrivée est représentée par **deux bandes noires successives** (espacé d'environ 3cm), elle sera positionnée du coté opposé à celui de départ.

- Les cases mesureront environ **30 * 30 cm**, (35 * 35 = environ boite pizza)

- Le nombre de case du labyrinthe pourrait varier entre **4 * 4** et **10 * 10**.

- Le labyrinthe peut avoir des **boucles**.

ii. But

Deux équipes s'affrontent tour à tour. Le déroulement de la partie se fait en 2 phases :

1. **Première phase** : un robot parcourt le labyrinthe afin de le cartographier. Il peut y rester autant de temps que nécessaire.

2. **Deuxième phase** : un robot est introduit dans le labyrinthe et doit en sortir le plus rapidement possible à l'aide de la cartographie de son prédécesseur. Son temps dans le labyrinthe est chronométré.

- Lors de la 1ère et 2ème phase, le robot est introduit dans la même case de départ.

- L'équipe dont le robot sortira le plus rapidement du labyrinthe sera déclarée vainqueur.

- La forme des robots est libre (type de capteur utilisé, etc…).

VI. Passage sur NXJ leJOS

C'est après la rédaction du cahier des charges que j'ai entrepris de plus amples recherches sur la communication Bluetooth. En explorant plus en détails la piste que j'avais trouvé (l'API Java), j'ai finalement découvert le firmware NXJ leJOS.

NXJ leJOS est un firmware de remplacement pour le NXT. Il permet de le programmer en Java et présente de nombreux avantages[4]. Ce qui a fait que je me suis intéressé de plus prés à ce système, est qu'il supporte de manière complète le Bluetooth, et que la communication avec un PC est simple à établir grâce à l'API PC disponible.

Bien-sur, passer sur ce système impliquait beaucoup de changement pour nous tous, puisque nous allions devoir flasher tous nos robots, ainsi que réinstaller un environnement de travail. Quand j'ai été sûr que ceci était la meilleur solution pour nous, j'en ai parlé à mon équipe et aux autres étudiants du TER, et la plupart ont été d'accord pour essayer ce nouveau système.

1. Installation de NXJ leJOS

L'installation des outils pour utiliser le système NXJ leJOS s'est bien passée. Elle fut moins longue que l'installation pour nxtOSEK, et le très grand avantage est qu'ici nous n'avons pas besoin de machine virtuelle pour que tout fonctionne, et ceci est très agréable.

2. Flashage des robots

Nous avons ensuite dû flasher le firmware NXJ leJOS sur le NXT. Au début, nous avons eu quelques difficultés pour réussir la connexion USB entre le PC et le NXT, il s'agissait d'un problème de permissions sur Linux. Hormis cet obstacle, cette étape s'est déroulée sans difficultés.

3. Configuration d'Eclispe

Pour pouvoir compiler simplement des programmes pour le NXT, ainsi que des programmes PC pouvant facilement dialoguer avec le NXT, nous avons installé un plugin[5] pour l'IDE[6] Eclipse.

Cela n'a posé aucune difficulté pour moi qui utilise Linux. Cependant, mes collègues ont eu des problèmes sur Windows et Mac. Pour le résoudre, il a fallu utiliser la version 32 bits d'Eclispe sur Windows, mais le Bluetooth reste inutilisable sur Mac (problème connu d'Apple depuis la dernière mise à jour d'OS X).

4 http://www.lejos.org/nxt/nxj/tutorial/Preliminaries/Intro.htm
5 module d'extension
6 environnement de développement

VII. À vos marques, prêts, codez !

1. Organisation de l'équipe et premières tâches

A partir de ce moment là, nous entrions véritablement dans le vif du sujet. Celui-ci était clairement défini et nous avions tous nos outils en place.

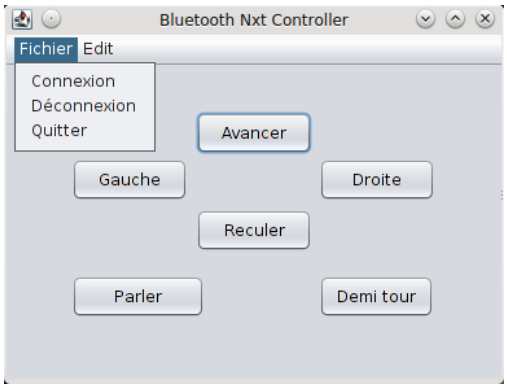
L'équipe était organisée de cette manière : tous les matins et après-midis, nous nous donnions rendez-vous dans une salle de TP (U3-202) pour travailler. Cela ne nous a pas empêché de travailler chacun de notre coté également.

En premier lieu, en plus de tester et de prendre en main les diverses fonctionnalités du nouveau firmware, nous nous sommes tous les trois lancés dans une tâche permettant d'avancer sur le sujet, tout en nous familiarisant avec notre nouvel environnement :

- Développement d'une application de contrôle du NXT par Bluetooth
- Tests de détection des lignes noires
- Premier programme pour se déplacer en fonction des murs alentours

2. Développement d'une application de contrôle du NXT par Bluetooth

Mon premier travail sur NXJ leJOS fut la réalisation d'un logiciel permettant de contrôler le NXT par Bluetooth. Ce travail avait deux buts : tester plus en profondeur la communication BT, et tester les déplacements sur le robot.



« Bluetooth Nxt Controller »

L'interface en elle-même a été réalisée avec le logiciel NetBeans et son outil de conception graphique « matisse » dont nous avons appris à nous servir cette année.

Le logiciel permet de se connecter à un NXT et de le diriger, à condition bien-sur que le programme de contrôle soit lancé sur le robot (programme que j'ai également développé).

La communication a été établie facilement. Cependant, pour utiliser plusieurs robots sur un même PC cela nous a posé beaucoup de problème au début. Une fois qu'un NXT a été « pairé » avec le PC, il est difficile d'en pairer un nouveau. Pour y arriver, il faut forcer l'ajout du nouveau robot dans le cache Bluetooth de NXJ.

Quant aux déplacements du robot, ils se sont révélés assez imprécis, même en « calibrant » le plus possible, tantôt le robot tourne un peu trop, tantôt il ne tourne pas assez. Ce manque de précision n'est pas énorme, mais en enchaînant les virages, le décalage du robot s'accumule et devient très gênant.

(cf code source : « Bluetooth NXT Controller V1 [PC] », « Bluetooth NXT Controller [NXT] »)

3. Tests de détection des lignes noires

Nous avions à notre disposition un capteur de luminosité. Celui-ci nous permet de percevoir des variations de couleurs puisque une couleur sombre ne renvoie pas autant de lumière qu'une couleur claire. Grâce à ceci, nous pouvons détecter les lignes noires séparant chaque case du labyrinthe au sol.

Nous avons effectué de nombreux tests pour établir un seuil de luminosité différenciant les bandes noires du sol, avec plusieurs éclairages différents pour que ce seuil soit le plus fiable possible.

(cf annexe 1)

Pour qu'ils soient plus précis, il nous aura également fallu placer les capteurs de luminosité les plus proche du sol possible (à 1mm).

4. Déplacements en fonction des murs alentours

i. Montage du nouveau robot : Hydra

À cette étape du projet, nous avions besoin d'un robot plus adapté à notre sujet que le modèle de base. Notre première idée fut de lui mettre les trois capteurs de distances dont nous disposions. De cette manière, à chaque entrée dans une nouvelle case du labyrinthe, le robot pouvait scanner directement tous les chemins ou murs alentours. (le NXT avait donc trois « têtes », d'où le nom de hydre)

C'est cette forme du robot qui a été gardé le plus longtemps, mais elle s'est affinée au fur à mesure de notre avancement, pour gagner en précision. D'abord toutes les têtes étaient à l'arrières, ensuite celles des cotés sont venus plus aux centre du robot, puis la tête arrière est venue devant, et enfin, nous avons baisser un peu la brique plus au niveau du sol.

La forme finale de « Hydra V2 »

Par la suite, afin de régler nos problèmes, le robot prendra une autre forme très différente.

ii. Le programme

Le premier programme que nous avons réalisé est simple, le robot avance, et, grossièrement :

- s'il y a des murs tout autour, le robot fait demi-tour
- s'il y a un mur en face et à droite, le robot tourne à gauche
- s'il y un mur en face et à gauche, le robot tourne à droite
- s'il y a un mur en face, le robot tourne à droite ou à gauche

C'est donc un robot qui s'adapte à son environnement, mais cela ne correspond pas à nos objectifs finaux. Hydra peut se déplacer dans un labyrinthe mais de manière très aléatoire, il se contente d'avancer n'importe où en tournant quand il y est obligé. De plus, les déplacements sont toujours très imprécis, même après avoir réécris et calibré toutes les méthodes de déplacements.

Un peu plus tard, nous avons également intégré la détection des lignes noires sur ce même programme. À chaque changement de case, un « beep » se fait entendre.

(cf code source : « Hydra V1 [NXT] »)

VIII. Dans le vif du sujet

Après avoir terminé ces premières tâches, nous avons réfléchi plus en détails à ce que nous devions faire. Dés le départ il était envisagé d'utiliser un PC pour effectuer les calculs à la place du robot. De plus, l'idée de créer une interface graphique dessinant le labyrinthe en temps réel me tenait à cœur. Il devenait donc certain que nous allions utiliser un PC.

Pour utiliser cette caractéristique au maximum, nous avons décider de faire en sorte que le robot soit « le plus stupide » possible. Le robot prendrai le moins de décision possible par lui-même. Il se contenterai de scanner la cellule dans laquelle il se trouve, d'envoyer les résultats, et d'attendre une instruction. Le PC serai vraiment le cerveau du robot, la mémoire et les calculs.

Le concept de base était posé.

Jusqu'ici, je vous ai présenté nos travaux et recherches dans un ordre chronologique. Maintenant, pour plus de clarté car il s'agit du projet final, je vous présenterai nos travaux réalisés par catégorie.

IX. Conception du programme coté robot

1. Les déplacements

i. Version 1

Les méthodes de déplacements avaient déjà été codé en partie lors du développement des programmes précédents. Nous les avons réécrites de manière plus propre, pour qu'elles soient utilisables simplement dans tous nos programmes.

Nous avons défini plusieurs types de déplacements possibles : avancer, s'arrêter, tourner de 90°

vers la droite ou la gauche, faire demi-tour. D'autres types de déplacements ont aussi été développés mais ils n'ont finalement pas été utilisés : 45° vers la gauche ou la droite, et reculer.

(cf code source : « Scanner Runner [NXT] » classe deplacement.java)

ii. Version 2

Par la suite, j'ai voulu développer une nouvelle méthode de déplacement pour faire avancer le robot en lui donnant des distances précises à parcourir. Pour cela, j'avais besoin du diamètre des roues du NXT. En effectuant une recherche sur le diamètre, j'ai découvert que NXJ leLOS proposait déjà une classe[7] permettant d'effectuer ce que je voulais : des déplacements précis, calculer en fonction des dimensions des roues et de leur écartement.

Nous avons donc réécris totalement nos méthodes de déplacements en utilisant cette classe. De ce fait, nous pouvons maintenant faire parcourir au robot des distances précises (données en cm). Malheureusement, la précisions n'a pas été améliorée. En effet, nous avions déjà tout fait pour être le plus précis possible, et les problèmes ne proviennent pas de nous mais des imperfections matériels du robot, ainsi que de la texture du sol.

(cf code source : « Scanner Runner [NXT] » classe deplacementOp.java)

C'est pour venir palier à ces imprécisions, que nous avons développé plusieurs solutions de redressements.

2. Les redressements

i. S'éloigner d'un mur

Le premier problème était que parfois, après des déplacements pas toujours précis, le robot se retrouvait trop près d'un mur. Il fallait donc l'en éloigner. Pour cela nous avons mis au point plusieurs méthodes. La première consistait à tourner dans la direction voulue en prenant un certain angle, puis à avancer, et ensuite à rétablir l'angle. Nous avons utilisé cette solution pendant un temps, puis nous en avons trouvé une plus élégante : simplement reculer, mais une roue à la fois.

Ci-dessous, la barre noire représente un mur, et les points oranges les roues du robot :

Position initiale *On a reculé la roue gauche* *Puis la roue droite*

Simple et efficace.

(cf code source : « Scanner Runner [NXT] » classe deplacementOp.java)

7 Classe Java, concept de la programmation orienté objet, voir le glossaire pour plus d'informations

ii. Rétablir l'angle

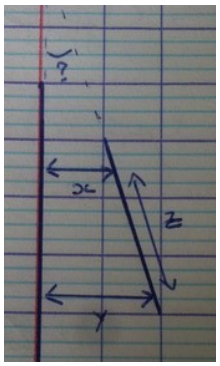
Première solution

Voilà une partie qui nous aura beaucoup préoccupée. J'ai longtemps cru que ce problème à lui seul nous empêcherai d'atteindre nos objectifs. Comme je l'ai dit de nombreuses fois déjà, les déplacements du robot ne sont pas assez précis. De ce fait, le robot perd vite son parallélisme avec les murs, et donc… il s'y cogne. Il nous a donc fallu réfléchir à des solutions pour rétablir ce parallélisme.

Notre première idée fut de simplement réorienter le robot quand il était trop proche d'un mur, car c'était sans doute que son angle était orienté vers ce même mur. Or, parfois, il s'est avéré que ce n'était pas le cas (le robot peut être trop proche d'un mur sans pour autant se diriger vers lui) et nous ne faisions donc qu'aggraver la situation. De plus, l'angle pour se redresser était loin de s'adapter à toutes les situations. À moins d'avoir énormément de chance, la solution n'était pas viable pour pouvoir parcourir tout un labyrinthe sans toucher de murs.

Solution exploitant les capteurs de distances

La 2ème solution est venue bien plus tard pendant le projet. Je ne pouvais pas me résoudre à avoir un robot qui ne sache pas se déplacer proprement. J'ai donc réfléchi et imaginé la solution suivante :



Le robot est représenté par la barre inclinée de longueur « z ». Le mur du labyrinthe est la barre verticale de gauche. En mesurant les distances « x » et « y », nous pouvons retrouver l'angle de déviation du robot par rapport au mur.

Il s'agit d'un simple problème de géométrie faisant intervenir le théorème de Thalès. (cf annexe 2)

En théorie cela fonctionne parfaitement. En pratique, beaucoup moins… Les capteurs de distances ne sont pas assez précis, et ici une erreur de 1cm ne pardonne pas. Dans de nombreux cas, l'angle est largement corrigé mais cela n'est pas suffisant. Quand l'angle à corriger est trop petit, cela dévie parfois encore plus le robot. De plus, le robot doit avancer de la longueur « z », et cela pose problème dans un environnement aussi étroit que notre labyrinthe (et prendre une longueur « z » trop petite fait que l'imprécision des capteurs se fait encore plus ressentir, car « x » et « y » sont très proche).

Solution exploitant le capteur de lumière

J'ai donc réfléchi à une nouvelle solution. Je me suis demandé quel repère nous pouvions utiliser pour réorienter Hydra… Et la solution m'a paru de suite évidente. Il y a les murs, et puis les lignes noires. Des robots suiveurs de lignes noires grâce au capteur de luminosité existent (ils sont donc parallèle à la ligne), donc nous pouvions forcément trouver une solution pour se réorienter

perpendiculairement aux lignes noires (et donc parallèle aux murs).

J'ai donc ensuite travaillé sur un système permettant d'établir cette solution. Cependant, cela m'a paru assez compliqué avec un unique capteur de luminosité, et surtout assez imprécis car nos bandes noires ne sont pas très larges (le système que j'avais imaginé avait besoin de la largeur des bandes). J'ai ainsi pensé à une solution beaucoup plus simple et précise : utiliser deux capteurs de luminosité. De cette manière, quand le robot arrive sur une ligne noire, si seulement un des capteurs détecte la ligne, c'est que le robot n'est pas perpendiculaire à celle-ci, et alors il suffit d'avancer l'autre roue du robot jusqu'à que l'autre capteur détecte la ligne.

C'était, je pense, la solution la plus précise pour résoudre notre problème (si on se contente du matériel fourni). J'en ai donc parlé à mes coéquipiers le plus rapidement possible. Et après avoir analysé les avantages et les inconvénients, nous avons décidé de changer la forme du robot pour pouvoir y placer un deuxième capteur de luminosité.

La forme du robot devait forcément changer car nous utilisions déjà tous les ports de capteurs disponibles. Nous avons donc dû placer un capteur de distance sur un moteur pour le faire pivoter, de cette manière, ce capteur de distance peut faire le travail des trois anciens et cela nous libère deux ports.

« Hydra V3 », la forme finale du robot.

Cette solution est arrivé tardivement et elle nous a donc forcé à adapter notre code. Il fallait maintenant tourner la « tête » dans la bonne direction avant de prendre des mesures et puis implémenter le système de redressement.

Autre solution

L'ultime solution aurait été d'utiliser un capteur de type boussole. Il en existe pour le NXT et cela aurait pu permettre au robot de toujours garder le cap, sans jamais avoir besoin de se redresser. Certes, pour notre projet cela n'était pas possible car nous devions nous contenter de notre matériel.

(cf code source : « Scanner Runner [NXT] » classe deplacementOp.java)

3. Fonctionnement général

Le robot doit être introduit dans la première cellule du labyrinthe : c'est à dire une cellule qui ne possède qu'une sortie possible située à l'avant du robot. Ceci est important car nous ne scannons pas cette première case.

Les différentes étapes (expliqué simplement) :

- Le robot avance jusqu'à trouver une ligne noire, s'il n'est pas perpendiculaire à cette ligne il se redresse.
- Ensuite il avance d'un nombre précis de centimètre de manière à se retrouver au milieu de la case suivante.
- Puis il regarde s'il n'est pas trop proche d'un mur, si c'est le cas, il s'en éloigne.
- Il scanne ensuite la cellule et envoie les résultats au PC.
- Il prend ensuite la direction que le PC lui indique.

Puis on répète ces étapes.

Si le robot se retrouve dans une impasse ou a détecté la ligne d'arrivée : le PC ne va plus lui indiquer une direction à prendre mais une suite d'instruction qui lui fera aller dans la cellule encore non visitée la plus proche. Si le labyrinthe est déjà entièrement connu, alors le PC indiquera au robot la séquence pour retrouver la sortie (s'il n'y est pas déjà, auquel cas le robot s'arrêtera).

Pendant que le robot effectue une suite d'instructions, il ne scanne pas les cellules puisque qu'elles sont déjà connues.

Note au cours du développement : pour tester nos travaux, nous avons été amené à développer une version du robot qui ne communique pas encore avec le PC, et où la séquence de déplacement est écrite « en dur » dans le programme.

(cf code source : « Scanner Runner [NXT] », classe Scanner.java)

4. Optimisation pour la phase deux

i. La forme

Lors de la phase où le robot doit sortir le plus rapidement possible, le robot prend une autre forme :

Dans cette phase, le robot ne doit scanner aucune cellule. On peut donc se passer du capteur de distance orienté vers l'avant et donc enlever le système de rotation. On conserve quand même un capteur de chaque coté pour pouvoir détecter si le robot est trop proche d'un mur.

Cette forme permet de gagner du temps car nous n'avons plus besoin de faire pivoter le capteur de distance.

De plus, le robot est plus stable et plus léger, ce qui est

toujours mieux puisque la vitesse va être au maximum.

ii. Le fond

Au niveau du programme, on réutilise en grande partie celui développé pour la phase une. Le robot doit être positionné dans la même cellule de départ que son prédécesseur, puis il reçoit par le PC toute la séquence d'instructions à suivre.

Les déplacements sont les même, mais ils ont été réécrit de manière à aller beaucoup plus vite. Idem pour les redressements, ils vont plus vite. Le robot perd en précision mais ce n'est pas un problème car, il a beaucoup moins de chemin à faire que son prédécesseur, et il ne scanne aucune cellule.

(cf code source : « Scanner Runner [NXT] » classe FastAndFurious.java)

X. Conception du programme coté PC

1. Fonctionnement général

Après avoir établie la connexion avec le robot, le logiciel attend un signal de celui-ci. Ce signal peut annoncer les résultats d'un scanne d'une cellule. Auquel cas, les résultats sont enregistrés puis l'ordinateur donne une instruction au NXT en fonction d'eux, c'est à dire, dans cet ordre de priorité :

- Tourner à droite
- Continuer tout droit
- Tourner à gauche

Si le robot est dans une impasse ou que c'est le signal de ligne d'arrivée qui a été reçu, on ne donne pas une instruction mais une suite d'instruction pour aller dans la case non visitée la plus proche, ou à la ligne d'arrivée (si il n'y a plus rien a visiter, et que le robot ne s'y trouve pas déjà (dans ce cas on envoie le signal d'arrêt)).

Après que tout le labyrinthe ait été parcouru, le logiciel se déconnecte du robot et passe en phase deux. Il faut ensuite se connecter à nouveau au NXT (quand le programme optimisé pour cette phase est lancé sur celui-ci). Dés que la connexion sera établie, la suite d'instructions pour aller à la case finale sera envoyée.

(cf code source : « Bluetooth Mapper [PC] » classe Main.java)

2. Représentation des données

i. Pour le labyrinthe

Dans notre programme, il fallait mémoriser chaque cellule du labyrinthe. Celui-ci étant une grille, nous avons décidé de le représenter par un tableau à deux dimensions.

Chaque case du tableau étant un type cellule défini par cinq attributs :

- north : indique si un mur se trouve au nord de cette cellule

- south : indique si un mur se trouve au sud de cette cellule
- east : indique si un mur se trouve à l'est de cette cellule
- west : indique si un mur se trouve à l'ouest de cette cellule
- visite : indique si la cellule a déjà été visité ou pas

Nous avons réussi à faire tout ce que nous voulions avec cette structure de données. Cependant, pour éviter quelques retraitements, je pense qu'il aurait été plus efficace d'utiliser une représentation plus de type graphe (notamment pour faciliter l'algorithme du plus court chemin et la gestion du référentiel).

(cf code source : « Bluetooth Mapper [PC] » classes CelluleLabyrinthe.java, LabyrintheEtAffichage)

ii. Pour les éléments communs au PC et au NXT

Pour plus de clarté et de simplicité, nous avons établie de nombreuses constantes, communes entre le programme du PC et du robot. Ces constantes sont par exemple la représentation des signaux (comme la détection de la ligne d'arrivée) et des instructions (comme « tourner à droite »).

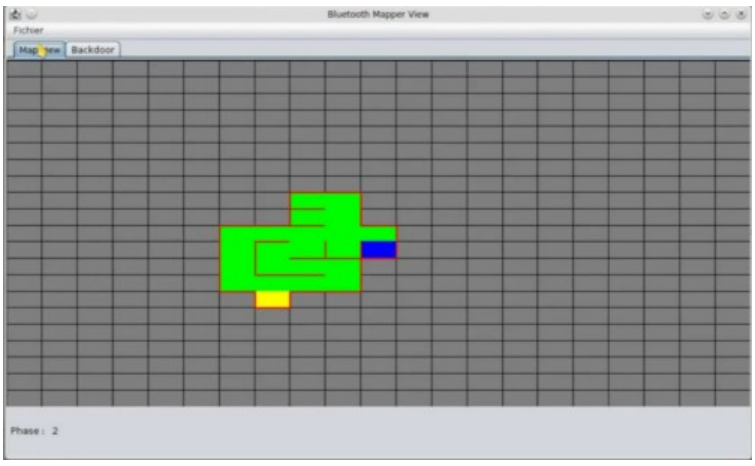
(cf code source : « Bluetooth Mapper [PC] » classe Constantes.java)

3. Interface et affichage du labyrinthe

L'interface a été réalisé avec NetBeans puis avec Eclipse. L'affichage du labyrinthe tout comme le reste de l'interface utilise les composants de la librairie Java Swing.

En occultant les détails trop technique, l'affichage du labyrinthe fonctionne simplement : on parcourt notre matrice qui représente le labyrinthe. Si la cellule a été visité on dessine la case en vert, et puis on dessine les murs l'entourant en rouge.

L'interface dispose aussi d'un onglet « Backdoor ». Celui-ci était prévu pour prendre le contrôle du robot si un problème se produisait. Nous devions simplement y implémenter l'application de contrôle développer auparavant. Cependant, ce n'étais pas notre priorité, et nous avons manqué de temps pour la mettre en place.



« *Bluetooth Mapper View* »

Ici la phase une vient d'être terminé. En bleu se trouve la case départ et en jaune la case d'arrivée.

Cet affichage, en plus d'être assez esthétique, nous a beaucoup servi pour le débogage.

(cf code source : « Bluetooth Mapper [PC] » classe Window.java, LabyrintheEtAffichage.java)

4. Algorithme du plus court chemin

i. Le plus court chemin

Afin d'aller le plus vite possible lors de la phase deux, il fallait passer par le plus court chemin de la case de départ à la case d'arrivée.

La solution qui m'a semblé être la plus efficace est un parcours en largeur du labyrinthe. Pour faciliter ce parcours, j'ai dû utiliser un arbre. Chaque nœud de l'arbre contient une coordonnée (la coordonnée d'une cellule), un parent, la direction de son parent (est, ouest, nord ou sud) et une liste de fils (de 0 à 4 fils).

L'algorithme fonctionne de cette manière (« file » et « newFile » sont des files) :

1. On place dans "file" le nœud de départ
2. Puis on cherche et place tous les fils de ce(s) nœud(s) dans "newFile", au fur à mesure on vide "file" (en enlevant les nœuds dont les fils ont été trouvés)
3. On place tous les nœuds de "newFile" dans "file" et on vide "newFile"
4. On reprend à l'étape deux
5. Quand on trouve le nœud correspondant au nœud d'arrivé, alors on arrête et on retrace son chemin en parcourant ses parents

→ Ce chemin est le plus court chemin entre le départ est l'arrivée

Note : dés qu'un fils est ajouté à un nœud, il ne peut plus être ajouté à aucun autre nœud. Ceci diminue la complexité de l'algorithme et surtout empêche les phénomènes de boucles.

L'utilisation de cet algorithme ne se fait pas uniquement dans la phase deux. Lorsque le robot est dans une impasse ou a détecté la ligne d'arrivée, il faut calculer le plus court chemin entre lui et la case non visitée la plus proche.

ii. Case non visitée la plus proche

Au départ, quand le robot devait aller dans une case non visitée, la case sélectionnée était la première trouvé. Cela n'était pas très performant, car la première case trouvée n'était pas forcément la plus proche du robot, et de ce fait, le robot pouvait se retrouver à faire des aller-retours inutile pour visiter le labyrinthe.

Lors de notre dernière réunion avec nos encadrants, ils nous ont vivement encouragé à mettre en place un voyageur de commerce pour explorer les cases non visitées, même si le temps passer pour faire la cartographie n'était pas notre objectif premier.

J'ai donc développé une solution intermédiaire, qui en pratique (parce que nos labyrinthes ne sont pas immenses) est, dans la majorité des cas, aussi performante qu'un voyageur de commerce. C'est maintenant la case non visitée la plus proche qui est sélectionnée et non pas une case aléatoire.

Pour ce faire, on fait la liste des cases non visitées, et on calcule leur distance au robot grâce à l'algorithme du plus court chemin. On sélectionne ensuite la case dont le chemin est le plus court.

(cf code source : « Bluetooth Mapper [PC] » classe LabyrintheEtAffichage.java)

5. Gestion du référentiel

Cette partie nous a joué beaucoup de mauvais tours… Le référentiel est l'orientation nord, sud, est, ouest. Ce référentiel est fixe et nous permet de gérer tout ce qui touche au labyrinthe, les relations entre les cases, l'affichage, etc. Par défaut, le nord est l'orientation que le robot a lors de sa position initiale (ce n'est pas le nord terrestre).

Comme le robot se déplace, sa droite ne correspond pas toujours à l'est par exemple. Ceci nous pose problème quand il faut donner une instruction au robot, ou quand il renvoie le résultat d'un scanne, car il faut faire le lien entre le référentiel fixe et le référentiel du robot (le robot connaît sa droite, sa gauche, etc. Mais pas le référentiel fixe).

Il nous a fallu écrire de nombreuses méthodes pour pouvoir passer d'un référentiel à un autre.

Par exemple, pour envoyer une séquence d'instruction au robot (c'est à dire par exemple « gauche », « droite », « avant », « droite »), nous utilisons l'algorithme du plus court chemin qui nous renvoie une suite d'orientations à prendre. Cette suite utilise le référentiel fixe (nord, sud, est et ouest). Elle doit donc être convertie, mais à chaque instruction effectué, le robot change d'orientation, donc la conversion n'est pas la même. Il faut donc faire très attention.

(cf code source : « Bluetooth Mapper [PC] » classe LabyrintheEtAffichage.java, ex : destToInstruction)

6. La communication Bluetooth

Au final, la communication n'aura pas été la plus grosse difficulté. Nous avons développer des méthodes de communication qui nous convenaient, en venant ajouter une surcouche aux méthodes proposé par NXJ. Nous sommes resté assez simple, le robot et le PC se contente de lire et d'envoyer des entiers un à un.

(cf code source : « Bluetooth Mapper [PC] » classe NxtConnexion.java)

7. Petit problème restant

Un problème très agaçant mais auquel on ne peut pas faire grand-chose, est que le capteur de distance est parfois capricieux. Cela n'arrive pas souvent, mais parfois il détecte un mur alors qu'il n'y en a pas, et plus rarement, il ne détecte pas de mur alors qu'il y en a.

Pour limiter ces problèmes, il faut faire très attention aux murs du labyrinthe, aucunes courbes, des angles bien nets, et pas de cases trop étroites. Il faut également faire très attention à la position du capteur. Mais malheureusement toutes ces précautions ne garantissent pas qu'il n'y est aucun problème.

XI. Autres...

1. Les pizzas

Notre labyrinthe a été réalisé avec des boites de pizzas. Nous aurions pu également utiliser d'autres cartons mais eux ont l'avantage de faire environ 35cm de coté (ce qui correspond à ce qui a été fixé dans le cahier des charges), et qu'ils sont fins tout en formant un angle, pratique pour faire

un labyrinthe rapidement.

2. Faire parler le robot

Pendant notre développement, nous avons également essayer de faire « parler » le robot. Nous avons réussi en lui faisant lire des fichiers audio dans un format très précis.

3. Communication entre plusieurs robots

Comme la question avait été abordé au tout début du TER, je m'y suis quand même intéressé de manière à savoir si c'était possible (si cela peut être utile aux étudiants des années suivantes).

D'après mes recherches et ce que nous avons réussi à faire, c'est possible. Je n'ai pas directement essayé mais je ne vois pas de problème en utilisant le firmware NXJ leJOS ainsi qu'un PC pour relier les robots.

XII. Bilan

Nos objectifs par rapport au cahier des charges ont été largement atteints. Ce ne fut pas sans difficultés, nous avons été confronté à de nombreux obstacles dont celui de l'imprécision des déplacements du robot. Ce problème était hors de notre contrôle et c'est pour cela qu'il m'a beaucoup inquiétait, je ne voulais pas qu'il nous empêche d'atteindre nos objectifs. Nous avons finalement su trouver une solution juste à temps, et cela nous a mis un peu la pression car elle est arrivée la même semaine que notre dernière réunion avec nos encadrants. Nos objectifs étant de finir le système de cartographie avant celle-ci. Et finalement, nous avons gagné notre pari en réussissant à faire une première cartographie complète la veille de la réunion !

Ce que je pense que nous aurions pu améliorer est la représentation des données. En voyant un labyrinthe en grille on pense forcément à une matrice, mais au vu des problèmes auxquels nous avons était confronté, je pense que nous aurions pu trouver quelque chose de mieux, ou d'au moins plus propre. En adoptant une structure plus « en graphe » par exemple, nous aurions certainement facilité l'algorithme du plus court chemin.

J'ai beaucoup aimé travailler sur ces robots. L'environnement nxtOSEK que nous utilisions au départ était intéressant, mais ce n'étais pas très agréable de travailler dessus. Nous devions passer par une machine virtuelle et ceci était vraiment contraignant. Au final je suis content que nous soyons passé sur NXJ leJOS, le développement fut beaucoup plus agréable. Même si je garde toujours une préférence pour le C plutôt que pour le Java, celui-ci nous a permis de rester sur nos machines respective. De plus, grâce au système d'auto-complétion et de la Java-doc, il a été très facile de prendre en main l'API de NXJ. Et un petit plus a été l'upload des programmes sur le NXJ directement par Bluetooth, très pratique, nous avons finalement utilisé les câbles USB qu'une fois pour flasher les robots.

Le fait que le projet ait pris la forme d'une petite compétition a aussi était motivant. Cela a rajouté un peu de challenge et favorisé la cohésion de notre équipe. En parlant d'équipe, le travail en groupe fut intéressant et motivateur. C'est toujours plus agréable de travailler à plusieurs, même si je pense que pendant ce projet, la répartition des tâches n'a pas toujours été bien faite.

J'ai trouvé ce projet très passionnant. La robotique est un domaine qui m'a toujours intéressé et sur un sujet ludique tel que celui-ci, cela ne pouvait pas me décevoir. J'ai été ravi de mettre en application ce que j'ai appris, et encore plus ce que je viens juste d'apprendre (programmation événementielle, Java).

Je ne regrette pas du tout d'avoir choisi ce TER, et j'en garderai un très bon souvenir.

XIII.Conclusion

Après avoir surmonté de nombreuses difficultés, les résultats des travaux de mon équipe satisfont le cahier des charges. Nous avons développé un robot capable de cartographier intégralement un labyrinthe de manière efficace, un autre capable d'en sortir rapidement, et une application PC qui dessine le labyrinthe en temps réel.

Tout ce travail a pu être réalisé en mettant en application bon nombres de connaissances et de savoirs acquis lors de notre cursus. Parmi eux : la programmation en C, en Java, la programmation événementielle, les notions sur les systèmes unix, la programmation multi-tâches... La rigueur enseigné tout au long de l'année nous a également été fort utile lors de ces travaux.

Ce TER m'a apporté une très bonne expérience d'un projet en équipe, qui a mis en jeu beaucoup de nos connaissances acquises. Il nous a fait découvrir un domaine fort intéressant dans un sujet très ludique.

À l'adresse suivante, vous pourrez trouver des photos ainsi que des vidéos de démonstration de notre projet : *http://benja135.free.fr/terlego.html*

XIV. Glossaire et acronymes

NXT : utilisé comme abréviation de robot Lego Mindstorms NXT

TER : travail d'étude et de recherche

Firmware : « logiciel embarqué », système d'exploitation du robot

Linux : désigne tout système d'exploitation basé sur le noyau Linux

Distribution Linux : système d'exploitation Linux spécifique

« Pairage » : fait d'associer deux périphéries Bluetooth entre eux

API : interface de programmation, logiciel offrant des services à un autre logiciel

Machine virtuelle : système d'exploitation (OS) fonctionnant grâce à un logiciel simulant la présence de ressources matérielles. De ce fait, un OS peut être lancé à partir d'un autre.

IDE : environnement de développement

C : langage de programmation

Java : langage de programmation (orienté objet)

Classe : « type d'objet », utilisé dans un langage de programmation orienté objet

BT : abréviation de Bluetooth

Hydra : nom de notre robot

XV. Bibliographie

Sites internet beaucoup utilisés

Documentation pour l'installation de nxtOSEK :

https://docs.google.com/document/d/1J5umh7rUo5SKlEKEV16xzfyVAqJxoCxqImCSNxe6m0g/edit

Documentation pour nxtOSEK :

http://lejos-osek.sourceforge.net/ecrobot_c_api_frame.htm

Documentation assez complète sur NXJ leJOS :

http://www.lejos.org/nxt/nxj/tutorial/index.htm

Documentation sur le capteur de distance, images partie IV.2 emprunté sur ce site :

http://www.sitedunxt.fr/articles/print.php?id=14

Divers articles traitant de la communication Bluetooth :

http://www.lejos.org/nxt/nxj/tutorial/Communications/Communications.htm

http://www.lejos.org/nxt/pc/api/index.html

http://lejos-osek.sourceforge.net/nxt2nxt.htm

http://disi.unitn.it/~palopoli/courses/ECL/Bluetooth_streaming_with_nxtosek.pdf

http://www.cs.ru.nl/lab/nxt/bluetooth.html

De nombreux autres sites ont dû être consulté, pour des documentations très diverses, cependant nous n'avons pas la possibilité de tous les retrouver.

Références aux enseignements de l'année :

- Programmation en C multi-tâches : pour le C et les notions sur les threads.
- Méthodologie de développement logiciel : pour nous avoir enseigné la rigueur et la propreté dans nos programmes.
- Construction et réutilisation de composants logiciels : pour nous avoir enseigné en détails les structures de données Java.
- Théorie des graphes : pour les notions utiles lors de la mise au point de l'algorithme du plus court chemin.
- Programmation événementielle : pour les interfaces graphique.
- Programmation en Java
- Anglais : pour toute la documentation.

XVI. Annexes

1. Étude sur le seuil

Éclairage normal

Valeurs capteur gauche / droite	Sol	Bande noire
Ombre des murs	570 / 530	326 / 300
Dégagé	616 / 573	375 / 334

Éclairage fort

Valeurs capteur gauche / droite	Sol	Bande noire
Ombre des murs	605 / 569	358 / 332
Dégagé	580 / 538	372 / 351

Seuil choisi : 450

2. Étude sur le rétablissement de l'angle