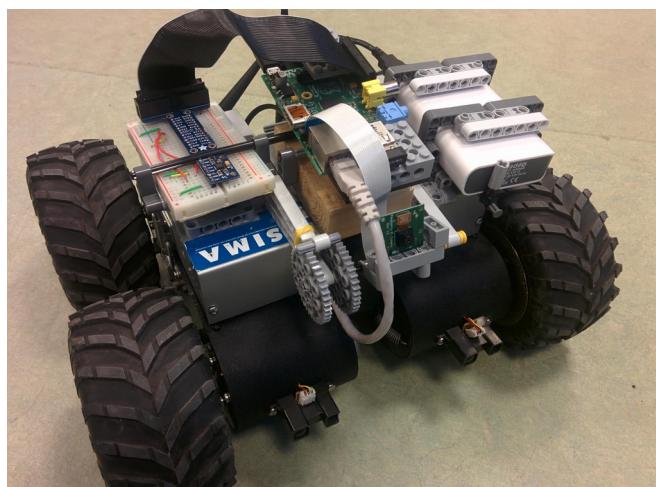




Institut Supérieur d'Informatique de
Modélisation et de leur Applications
Campus des Cézeaux
24 avenue des Landais
BP 10125
63173 AUBIERRE Cédex

Rapport d'ingénieur
Projet de 2ème année
Filière Informatique des systèmes embarqués

AMÉLIORATION DE L'ORIENTATION D'UN WIFIBOT



Présenté par :
PIERRE Cyrille
IMPERY Thomas

Tuteurs : Michel CHEMINAT, Laurent DELOBEL

Responsable : Romuald AUFRERE

Durée : 60 heures

Soutenance de projet : **Mardi 17 Mars 2015**

Remerciements

Nous remercions M. Cheminat et M. Delobel pour nous avoir conseillé tout au long de ce projet. Leur aide nous a permis de mieux avancer lorsque nous avions des interrogations.

Nous remercions M. Aufrère pour nous avoir fourni une application permettant de diriger le robot, et expliqué comment celle-ci fonctionnait. Nous le remercions également pour nous avoir fourni un volant ainsi que des pédales pour mieux diriger le robot.

Résumé

Le but de ce projet de deuxième année est d'ajouter un capteur à un robot wifibot afin de pouvoir détecter et évaluer les potentielles déviations qu'il pourrait avoir lorsque celui-ci se déplace

Afin de réaliser ce projet, il faut en premier lieu effectuer les quelques branchements nécessaires entre la carte **ARDUINO** et le **magnétomètre**. Il faut comprendre comment communiquer avec ce capteur afin de pouvoir recevoir ses données, et savoir comment les interpréter. Une fois ceci fait, la mise en place d'une application lisant sur le port série est nécessaire afin de recevoir les données du capteur sur l'ordinateur. Comme toute application envoyant des données sur le réseau, un **protocole** d'envoi a été mis en place. Ces données n'étant pas très parlantes, il est donc plus judicieux de créer une **boussole** prenant ces données en compte, afin de mieux voir dans quel sens est orienté le capteur. Pour finir, le magnétomètre étant très sensible aux **perturbations magnétiques** proches, nous avons mis en place une application permettant de le **calibrer** automatiquement.

Beaucoup de modifications ont été faites au robot tout au long du projet, afin de pouvoir mieux travailler. Ces modifications seront détaillées plus tard dans le rapport.

Mots-clés : ARDUINO, magnétomètre, protocole, boussole, perturbations magnétiques, calibrer.

Abstract

The goal of this project of second year was to add a sensor to a wifibot robot in order to be able to detect and evaluate potential deviations which it can have when moving.

To realize that project, it was necessary in the first place to do some connections between the **ARDUINO** card and the **magnetometer**. We had to understand how the sensor communicated to be able to receive his datas, and to interpretate it. Once this made, the introduction of a software application was necessary to read datas coming from serial port to receive it on the computer. Like any network applications, sending **protocol** had to be made. These datas weren't very relevant, so we decided to create a **compass**. With that compass we were able to see in what direction the sensor was. In the end, as the magnetometer was very sensitive to any **magnetics interferences**, we created an application which was able to **calibrate** it automatically .

Many modifications were done on the robot all along the project. These modifications helped us to work better. We will explain those ones later on the report.

Key words : ARDUINO, magnetometer, protocol, compass, magnetic interferences, calibrating.

TABLE DES MATIÈRES

<u>1 -</u> Déroulement du projet.....	2
1.1 - Diagramme de Gantt prévisionnel.....	2
1.2 - Diagramme de Gantt réel.....	2
2 - Prise en main du matériel.....	3
2.1 - Le wifibot.....	3
2.1.1 - Architecture du robot.....	3
2.1.2 - Communication avec le robot.....	4
2.1.3 - Configuration du Wifi.....	5
2.2 - Le magnétomètre.....	7
2.2.1 - Technologie du capteur.....	7
2.2.2 - Le capteur MPU9150.....	7
2.3 - Modification du robot.....	8
2.3.1 - La caméra.....	8
2.3.2 - Installation des nouveaux modules.....	8
2.3.3 - L'application de pilotage du robot.....	10
3 - Acquisition des données du magnétomètre.....	11
3.1 - Analyse du fonctionnement du capteur.....	11
3.1.1 - Le protocole I ² C.....	11
3.1.2 - Les registres du capteur.....	12
3.2 - Communication avec le capteur sur Arduino.....	13
3.2.1 - Montage des composants sur Arduino.....	13
3.2.2 - Analyse de l'application.....	15
3.2.3 - Conception de l'application.....	16
3.3 - Portage de l'application sur Raspberry Pi.....	18
3.3.1 - Communication avec le MPU9150.....	18
3.3.2 - Description de l'application.....	19
4 - Interfaçage et contrôle du robot.....	21
4.1 - Boussole.....	21
4.1.1 - Théorie.....	21
4.1.2 - Pratique.....	22
4.2 - Calibrage.....	24
4.2.1 - Théorie.....	24
4.2.2 - Pratique.....	26

Introduction

Pouvoir diriger un robot à distance parfaitement est très difficile. Il existe de nombreuses contraintes liées au terrain, ou même à des temps de latence. Nous ne traiterons pas le problème de la latence dans ce rapport, mais plutôt la prise en compte des événements aléatoires qui peuvent faire dévier le robot de sa destination initiale. Ces événements peuvent être liés au robot lui-même, lorsque celui-ci n'aura pas avancé comme prévu. Ou lié au terrain, si une des roues glisse par exemple.

Ces changements de direction non voulu sont problématiques puisqu'ils empêchent le robot d'arriver à une destination voulue. Le but de ce projet est donc de pouvoir les capter, afin qu'une prise en compte automatique de celles-ci ait lieu du côté du contrôleur.

Actuellement, lorsque nous demandons au robot d'effectuer une ligne droite et qu'une déviation a lieu, le robot continuera dans la direction de cette déviation. Il faut donc être capable de mesurer cette déviation grâce à des capteurs pour l'annuler sans intervention humaine.

Il existe différents types de capteurs : les magnétomètres permettant de mesurer le champ magnétique autour d'un dispositif. Les accéléromètres pour détecter les accélérations d'un système. Et enfin les gyroscopes pour mesurer la vitesse angulaire d'un système. La solution choisie a été la première, le magnétomètre. Il s'agit de la solution la plus simple à mettre en place puisque le capteur pourra nous donner un angle par rapport au pôle nord, et donc à posteriori un angle de déviation par rapport à la destination voulue.

1 - Déroulement du projet

1.1 - Diagramme de Gantt prévisionnel

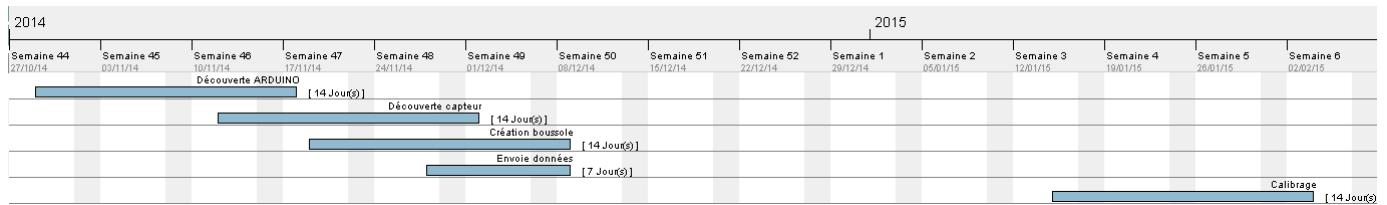


Fig. 1.1 : Diagramme de Gantt prévisionnel

Dès le début du projet, nous nous sommes demandés comment arriver à 60 heures de projet, celui-ci nous semblait peut consistant et relativement simple. On peut voir sur le diagramme prévisionnel que nous pensions finir le projet début janvier.

La dernière tâche n'était pas réellement prévue au départ, mais nous avons trouvé judicieux de la rajouter pour montrer la différence entre le temps que nous pensions devoir mettre pour la faire, et le temps réel que nous avons consacré.

1.2 - Diagramme de Gantt réel

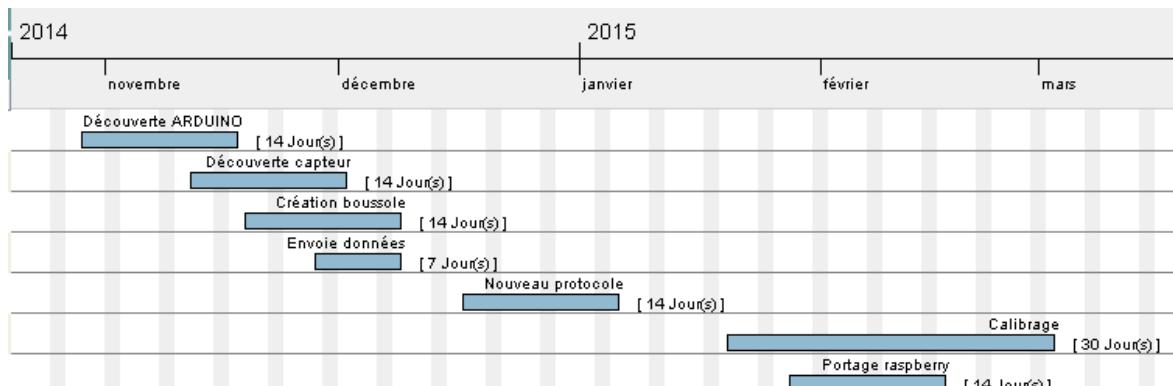


Fig. 1.2 : Diagramme de Gantt réel

On peut voir qu'il y beaucoup plus de tâches sur le diagramme réel, que sur le prévisionnel. Cela vient du fait qu'il y a beaucoup de choses que nous n'avions pas penser à faire au départ.

2 - Prise en main du matériel

Avant de commencer à expliquer en détail le fonctionnement des différentes applications réalisées, on va d'abord s'intéresser au matériel que l'on a utilisé tout au long de ce projet. Le but de ce dernier étant l'intégration d'une boussole dans un robot, il est nécessaire d'en avoir un sous la main pour travailler. C'est pourquoi nous avons utilisé un Wifibot. Concernant le capteur utilisé pour la boussole, nous avons utilisé le MPU9150. Après avoir expliqué le fonctionnement de ces différents outils, on parlera de quelques modifications que l'on a apportées sur le robot pour pouvoir travailler convenablement.

2.1 - Le wifibot



Fig. 2.1: Un exemple de Wifibot

Le wifibot est un robot mobile créé par une société française : Nexter Robotics. Il est doté de 4 roues motrices, d'une caméra et d'un module Wifi permettant de le contrôler à distance. La grande taille de ses roues lui permet de rouler sans aucun problème en extérieur et la portée de son émetteur Wifi est suffisante pour le contrôler d'assez loin.

2.1.1 - Architecture du robot

Ce robot a la particularité d'être équipé d'un routeur permettant de communiquer avec les différents éléments qui compose sa structure. Chacun de ces éléments sont indépendants et accessibles à distance par l'intermédiaire d'un point d'accès Wifi. Ils sont tous reliés entre eux dans un même réseau local grâce à un routeur équipé de 4 prises Ethernet. Ce robot a donc l'avantage d'être facilement modifiable puisqu'il suffit de brancher un nouveau module sur le routeur pour intégrer de nouvelles fonctionnalités. Voici les modules mis en place sur les robots que nous avons à notre disposition :

Une caméra IP

Cette caméra est une simple caméra IP que l'on peut acheter n'importe où dans le commerce. Les images de la caméra sont accessibles à partir d'une interface web. On peut visualiser ce que voit le robot directement depuis un navigateur. Mais il est également possible de récupérer les données pour pouvoir les traiter dans une application. L'interface web permet de contrôler la vitesse de rafraîchissement des images ainsi que leurs résolutions. Il est également possible de contrôler l'orientation de la caméra sur certain modèle de caméra IP qui ont été intégrés sur les 7 robots disponibles.

Un piloteur

Le wifibot dispose également d'un système embarqué dédié au pilotage du robot. Comme la caméra IP, le

piloteur possède une interface web permettant de réaliser quelques mouvements simples. Si l'on souhaite contrôler de façon plus poussée le robot, il est évidemment possible d'écrire une application qui le pilote directement avec une connexion TCP. On peut alors le piloter de deux façons suivantes :

- en boucle ouverte : Il n'y a alors aucun asservissement des moteurs, la rotation des moteurs n'est donc pas contrôlée.
- en boucle fermée : Le robot mesure la vitesse de rotation des moteurs grâce à des capteurs angulaires qui mesurent directement la vitesse de rotation des roues pour s'assurer que la consigne envoyée est bien respectée.

Le robot comporte également deux capteurs de distance infrarouge à l'avant permettant de faire des détections de collision.

Un routeur

Le routeur sert à relier tous les autres modules du robot ensemble. Grâce à son hub Ethernet, il met à disposition des prises Ethernet pour l'ajout de nouveaux modules. Il met surtout à disposition le point d'accès Wifi que l'on peut configurer comme on le souhaite à partir de l'interface web.

2.1.2 - Communication avec le robot

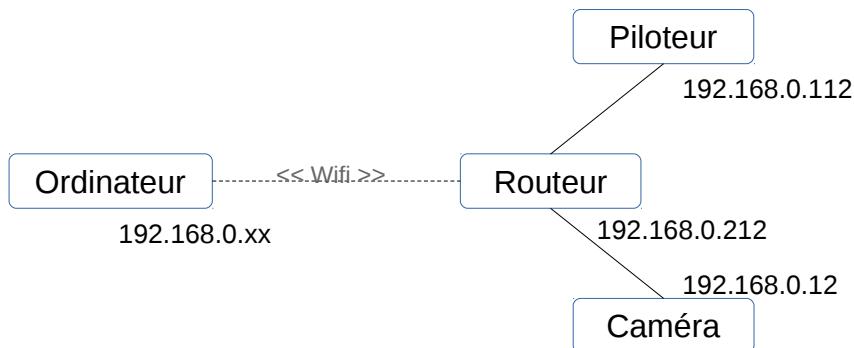


Fig. 2.2 : Représentation du réseau interne au robot

Comme tous les modules du Wifibot sont connectés sur un même réseau local, chacun possède une adresse IP. Il est donc possible d'établir un schéma du réseau interne au robot :

Comme on peut le voir sur le schéma ci-dessus, pour pouvoir communiquer avec le robot, il est nécessaire de se connecter au réseau établi par le routeur. Cependant nous pouvons nous connecter de deux façons différentes :

- en utilisant un port Ethernet libre. On profite alors d'un très bon débit avec peu de latence mais on est lié physiquement au robot.
- en utilisant le point d'accès du routeur. La connexion est alors de moins bonne qualité mais il n'y a plus de contrainte physique.

Dans tous les cas, il est nécessaire de s'attribuer manuellement une adresse IP puisque ce réseau ne dispose pas de serveur DHCP. Une fois la connexion établie, on peut tester le contrôle en utilisant soit les interfaces web des différents modules du robot, soit en utilisant une application dédiée au pilotage du robot. M. Aufrère nous a mis à disposition une application Qt sur les machines Windows de la salle de robotique qui permet de piloter simplement le robot à l'aide de la souris ou du clavier.

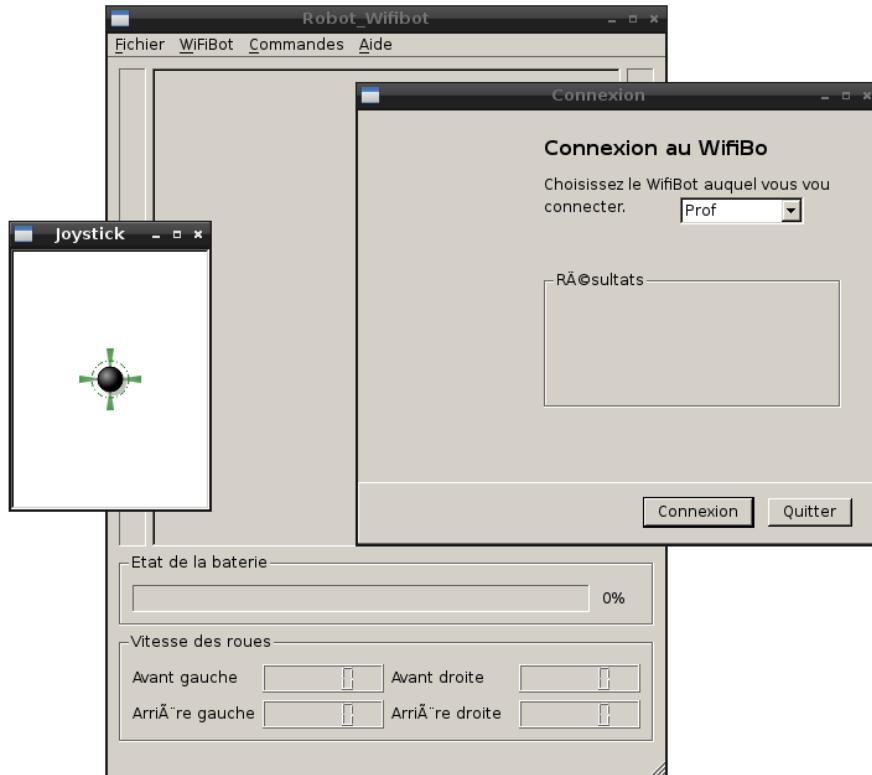


Fig. 2.3 : Application de pilotage des Wifibots

Cette application permet de prendre en main efficacement la plupart des fonctionnalités du robot en restant très simple d'utilisation. Il suffit de choisir le robot que l'on souhaite contrôler dans la liste des robots disponibles puis d'établir la connexion. L'interface de l'application montre ensuite l'image de la caméra et la valeur des capteurs de distance. Cette application communique donc avec le serveur gérant la caméra et le serveur gérant le pilotage du robot.

2.1.3 - Configuration du Wifi

Concernant le routeur, il existe deux façons de mettre en place un réseau en utilisant la technologie Wifi. La méthode de connexion la plus connue est sans doute le mode *infrastructure* mais il existe également le mode *ad hoc*.

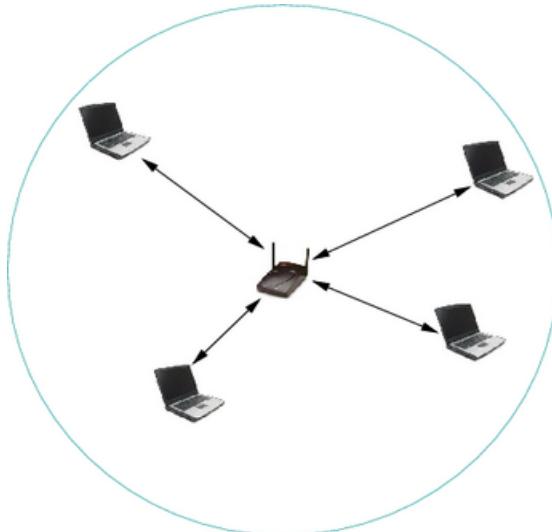


Fig. 2.4 : Réseau Wifi en mode infrastructure

Dans le mode *infrastructure*, une machine fait office de serveur et toutes les autres machines qui souhaitent se connecter au réseau sans fil communiqueront uniquement en passant par le serveur. Ce type de réseau a l'avantage d'avoir une bande passante raisonnable (25 Mbit/s réels en norme 802.11g et 100 Mbit/s réels en norme 802.11n) mais il a l'inconvénient d'avoir une assez faible portée. En effet, puisque toute les communications passent par le serveur, la portée du réseau se limite uniquement à celui du routeur.

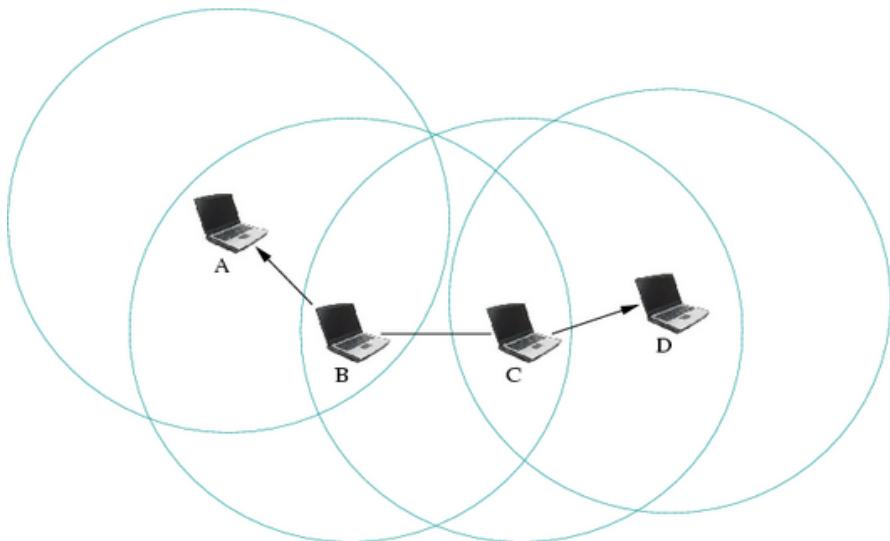


Fig. 2.5 : Réseau Wifi en mode ad-hoc

Dans une architecture *ad hoc*, il n'y a pas de point d'accès. Les machines communiquent directement avec leurs voisins. Pour pouvoir se connecter dans ce réseau, il suffit juste de configurer son interface Wifi en mode *ad-hoc* et de spécifier le nom du réseau. L'avantage de cette architecture est que la portée du réseau n'est cette fois-ci plus limitée uniquement à une machine. Comme l'illustre la figure ci-dessus, l'ordinateur A est capable de communiquer avec l'ordinateur D alors qu'ils sont pourtant hors de portée (leurs cercles ne se coupent pas). Le réseau passe en réalité par les ordinateurs B et C.

Ce type d'architecture est très pratique pour les Wifibots car il permet de contrôler le robot avec une plus grande portée qu'en mode *architecture*. Cependant le mode *ad hoc* est beaucoup plus limité en bande passante et possède également une certaine latence qui dépend du nombre de machine relais entre l'émetteur

et le récepteur. Cette contrainte est très importante pour la vidéo car il faut éviter au maximum les décalages entre la commande du robot et le retour visuel.

2.2 - Le magnétomètre

Bien qu'il soit nécessaire d'utiliser un robot dans ce projet, le sujet porte avant tout sur un capteur capable de mesurer un angle absolu dans l'espace. Pour ce faire, il faut utiliser un capteur capable de mesurer le champ magnétique terrestre. Ce type de capteur s'appelle un magnétomètre.

2.2.1 - Technologie du capteur

Pour bien comprendre comment utiliser un magnétomètre, il peut être utile d'avoir quelques connaissances sur son fonctionnement. Le phénomène physique utilisé pour mesurer un champ magnétique s'appelle l'effet Hall.

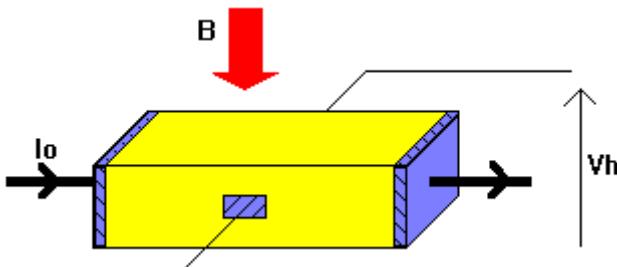


Fig. 2.6 : Modélisation de l'effet Hall

Lorsqu'un courant électrique (I_o sur le schéma ci-dessus) traverse un semi-conducteur et que ce dernier est plongé dans un champ magnétique (B), on mesure alors une tension (V_h) appelée tension Hall. Cette tension a la particularité d'être proportionnelle à la valeur du champ magnétique et du courant circulant dans le semi-conducteur. Comme nous pouvons contrôler la valeur du courant, il est possible de calculer la valeur du champ magnétique simplement en mesurant la tension Hall.

Le magnétomètre étant soumis en permanence au champ magnétique terrestre, il est possible de connaître son orientation en fonction de la valeur du champ traversant le capteur. Cependant ce capteur ne permet de mesurer le champ que sur un seul axe. Si l'on souhaite s'orienter par rapport Nord, il est nécessaire d'avoir une mesure du champ magnétique sur au moins deux axes. C'est pourquoi les magnétomètres intègrent souvent trois capteurs à effet Hall pour mesurer le champ sur chacun des axes géométriques de l'espace.

2.2.2 - Le capteur MPU9150

Dans ce projet, nous avons utilisé un capteur nommé MPU9150 qui intègre les 3 magnétomètres mais également 3 accéléromètres et 3 gyromètres. On a donc une centrale inertie 9 axes puisqu'elle est constituée de 9 capteurs. Le MPU9150 intègre également un thermomètre qui peut être utilisé pour corriger la valeur de certains capteurs sensible à la chaleur.



Fig. 2.7 : Le MPU9150

L'accéléromètre permet de mesurer une accélération et le gyromètre mesure une vitesse angulaire mais dans le cas de notre projet, nous sommes limités uniquement à l'utilisation des 3 axes du magnétomètre. Chacun de ces capteurs ont une résolution de 13 bits signés. Ils peuvent donc prendre des valeurs allant de -4096 à 4095. La grandeur physique associée à ces capteurs étant un champ magnétique, ces valeurs peuvent être converties en Tesla. D'après la documentation la résolution physique du magnétomètre est de plus ou moins 1200 µT. Cette résolution semble bien faible mais est pourtant largement suffisante pour mesurer le champ magnétique terrestre car sa valeur avoisine les 20 µT.

2.3 - Modification du robot

Afin de pouvoir travailler de façon plus confortable avec le robot, nous avons effectué quelques modifications. Ces modifications proviennent surtout d'une autre idée de projet que nous avons développé en parallèle. Cette idée était de piloter un Wifibot à partir d'un volant et de pédales ainsi qu'un système de vision par casque.

2.3.1 - La caméra

La caméra qui est disponible par défaut sur les Wifibots n'est pas très performante. Sa résolution est très faible et son temps de latence est assez important. De plus, elle envoie la vidéo sous la forme d'une séquence d'image JPG. On l'a donc remplacée par une caméra capable d'envoyer un vrai flux vidéo. Ayant à disposition une carte Raspberry Pi équipée de la caméra Pi, nous avons alors décidé de changer la caméra.

Concernant la carte Raspberry Pi, il s'agit d'un nano-ordinateur de la taille d'une carte de crédit. Bien que très petite, cette machine est capable de faire tourner des systèmes d'exploitation Linux. Le système actuellement installé est *Raspbian* qui est une version de *Debian* spécialement conçue pour la Raspberry. Cette carte est donc assez puissante pour faire fonctionner une caméra. Elle est aussi équipée d'un port Ethernet. On peut donc la connecter au routeur de la même façon que l'ancienne caméra IP du robot.

Avec cette caméra, on peut travailler avec un vrai flux vidéo que l'on peut faire transiter par le réseau. Il est possible d'obtenir des vidéos de très bonne qualité comme par exemple : une résolution de 1920 pixels de large pour 1080 pixels de haut pour un débit de 30 images par secondes (fps), ou encore 1280 x 720 pixels en 60 fps. Cependant ces qualités d'images nécessitent un débit très important. Or, sur un réseau Wifi, le débit est limité à environ 12 Mo/s maximum (en norme 802.11g). Il est par conséquent nécessaire de compresser la vidéo avant de l'envoyer sur le réseau. La compression va alors diminuer la qualité de la vidéo mais cela restera toujours plus fluide que la vidéo obtenu avec la caméra IP.

2.3.2 - Installation des nouveaux modules

Pour installer la caméra ainsi que la carte Raspberry Pi sur le Wifibot, nous avons mis en place une structure en LEGO. Cette structure n'est peut-être pas la meilleure des solutions mais elle a l'avantage d'être rapide à

mettre en place et facilement modifiable. Voici une image du robot modifié :

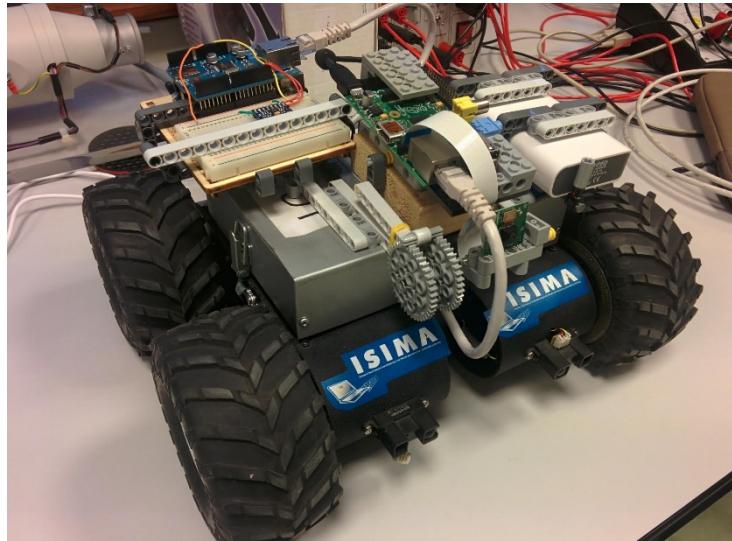


Fig. 2.8: Le Wifibot modifié avec la carte Arduino

Sur cette image, on peut voir la carte Raspberry située au centre du robot. La caméra se trouve à l'avant du robot, à droite du câble Ethernet. La Raspberry est alimentée par une batterie 5V que l'on peut voir en blanc, sur la droite du robot. On peut également voir sur la gauche, la carte Arduino qui a été utilisé pour écrire l'application de communication avec le magnétomètre. Elle est accompagnée d'une carte de prototypage sur laquelle on peut apercevoir le magnétomètre.

Actuellement, nous avons passé ce capteur directement sur la Raspberry. La carte Arduino n'est alors plus utile. Voilà à quoi ressemble le robot dans cette nouvelle configuration :

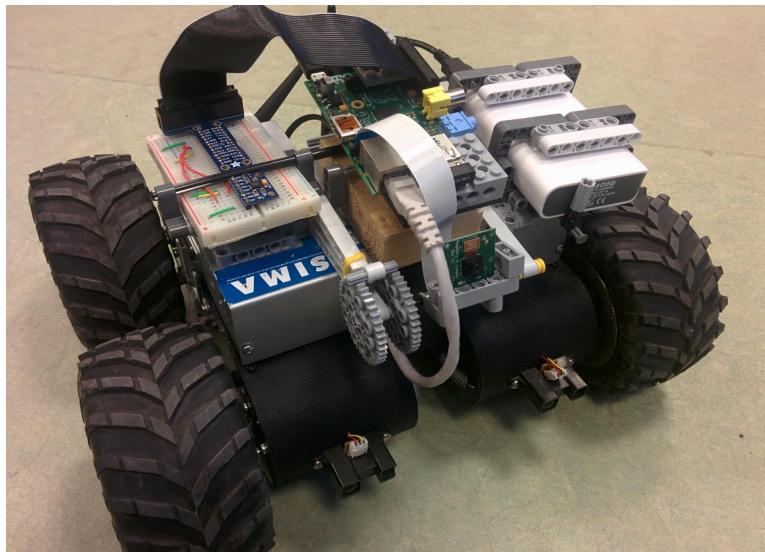


Fig. 2.9: Le Wifibot avec le magnétomètre branché à la Raspberry Pi

Sur cette image, la carte Arduino qui était sur la gauche a été remplacée par une carte de prototypage. On peut voir sur cette carte, le magnétomètre ainsi que la nappe noire connectée à la Raspberry.

2.3.3 - L'application de pilotage du robot

Nous avons également créé une application permettant de contrôler le Wifibot à partir d'un volant et de pédales. Cette application a surtout été créée pour montrer quelque chose de visuelle et attractif pour la Journée Porte Ouverte de l'école. Mais cette application offre également l'avantage d'être beaucoup plus maniable que l'application Qt.

Elle a été réalisée en C++ en utilisant la bibliothèque SDL2 pour contrôler le volant et les pédales. Son fonctionnement est très simple : lorsqu'un événement se produit sur le volant où l'une des pédales, on effectue un calcul simple pour déterminer quelles valeurs doivent être envoyées aux moteurs du robot. Ce programme nécessite donc de communiquer par réseau avec le serveur de contrôle du robot.

3 - Acquisition des données du magnétomètre

Maintenant que l'on sait comment fonctionne le matériel, on va pouvoir s'intéresser à la récupération des valeurs du magnétomètre. Pour se faire, il va donc falloir utiliser une carte électronique capable de communiquer avec le capteur. Dans cette partie du rapport, on étudiera le fonctionnement de deux applications d'acquisition de données écrite sur deux plateformes différentes : une carte Arduino Uno et une carte Raspberry Pi.

3.1 - Analyse du fonctionnement du capteur

Avant de commencer l'analyse des applications, il est nécessaire de comprendre comment communiquer avec le capteur. L'application doit être capable d'initialiser, de démarrer ou même interrompre l'acquisition des données du capteur.

3.1.1 - Le protocole I²C

La communication avec le capteur MPU9150 se fait par l'utilisation d'un bus de données nommé I²C (Inter Integrated Circuit) qui se compose de 2 lignes :

- SDA (Serial Data Line) : ligne de données bidirectionnelle,
- SCL (Serial Clock Line) : ligne d'horloge de synchronisation bidirectionnelle.

Ce bus est capable de communiquer avec différents périphériques branchés en parallèle sur ces deux lignes. Lors de l'envoi de données il faudra cependant préciser l'adresse du périphérique qui doit recevoir ces données. On a donc une architecture maître/esclave car un des périphériques (souvent le microcontrôleur) envoie des commandes et les autres ne font que répondent. Cela signifie qu'un périphérique esclave ne prendra la parole sans qu'on lui ait demandé.

Sur le MPU9150 le bus I²C s'utilise avec des registres. Le capteur possède une mémoire interne permettant de stocker les valeurs des différents capteurs ainsi que ses paramètres de configuration [1]. Cette mémoire se compose de bloc d'une taille de 8 bits dont l'adresse est également 8 bits. La mémoire possède donc une capacité de 256 blocs. Lorsque l'on souhaite configurer le capteur, il suffit alors d'envoyer la nouvelle valeur du bloc que l'on souhaite modifier. La trame envoyée sur le bus I²C sera de la forme suivante :

Maitre :	S	Adresse + W	RA	Donnée	P
Esclave :		ACK	ACK	ACK	ACK

Tableau 1: Trame I²C d'une écriture dans un registre

- S : envoi du bit de start
- Adresse + W : l'adresse du périphérique sur 7 bits + un bit indiquant que la commande est une écriture
- ACK : envoi d'un acquittement de la part du périphérique esclave (le capteur)
- RA : l'adresse du registre sur 8 bits dans la mémoire interne du capteur
- Donnée : la donnée sur 8 bits que l'on souhaite écrire
- P : envoi du bit de stop pour signaler la fin de transmission I²C

Lorsque l'on souhaite récupérer la valeur d'un registre, la commande envoyée sur le bus I²C sera légèrement différente :

Maitre :	S	Adresse + W		AR		S	Adresse + R			NACK	P
Esclave :			ACK		ACK			ACK	Donnée		

Tableau 2: Trame I²C d'une lecture de registre

- Adresse + R : l'adresse du périphérique sur 7 bits + un bit indiquant que la commande est une lecture
- Donnée : cette fois ci la donnée est envoyée par l'esclave puisque l'on souhaite lire le capteur
- NACK : envoi du bit Not-Acknowledgement pour indiquer que l'on a reçu la donnée

3.1.2 - Les registres du capteur

On va maintenant s'intéresser aux registres qui vont nous être utile pour pouvoir récupérer les données du magnétomètre. Toutes ces informations sont disponibles dans la documentation du composant. D'après cette documentation, on apprend qu'en réalité le MPU9150 n'est en fait qu'une fusion de deux autres composants dans un même boîtier :

- Le MPU6050 : une centrale inertuelle 6 axes (accéléromètre + gyromètre).
- L'AK8975C : un magnétomètre 3 axes.

Pour notre application, la partie qui nous intéresse est l' AK8975C car on souhaite récupérer uniquement les données du magnétomètre. On peut alors se demander comment communiquer avec capteur ?

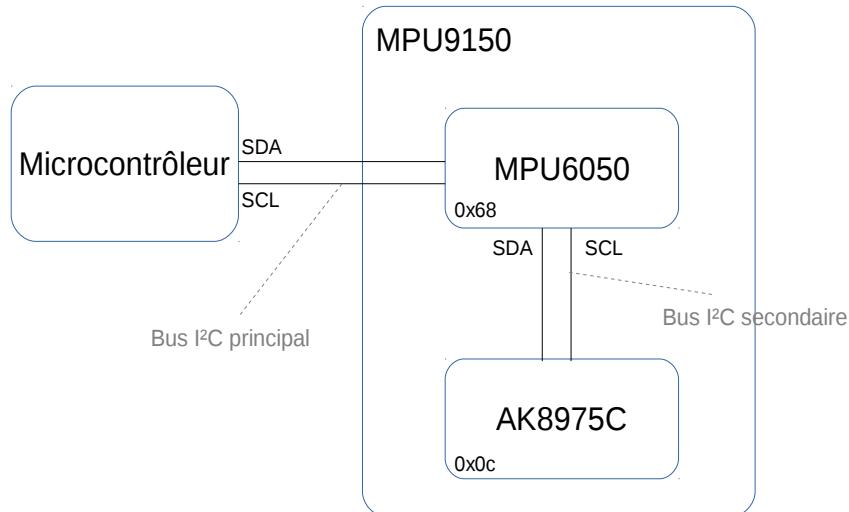


Fig. 3.1 : Bus de communication interne et externe au MPU9150

L' AK8975C est relié au MPU6050 par l'intermédiaire d'un second bus I²C sur lequel le MPU6050 est le maître. Il n'est donc pas accessible directement depuis notre microcontrôleur qui est branché sur le bus I²C principal. Il existe cependant un registre dans le MPU6050 qui permet de relier le bus I²C secondaire directement sur le bus principal. Ainsi, il devient plus simple de communiquer avec l' AK8975C car on aura plus besoin de passer par le MPU6050. Il suffira pour le microcontrôleur de communiquer avec le périphérique à l'adresse 0x0c à la place de l'adresse 0x68.

Une fois que l'AK8975C est accessible depuis le microcontrôleur, on peut récupérer les données des 3 axes du magnétomètre. Mais si actuellement on essaye de lire ces valeurs, on obtiendra uniquement des valeurs nulles. En effet, lorsqu'il est alimenté, le magnétomètre ne mesure pas en permanence le champ magnétique. Il est donc nécessaire de démarrer une mesure manuellement.

Le magnétomètre fonctionne en mode *single measurement*. Cela signifie que le composant ne mesurera

qu'une seule fois le champ magnétique avant de basculer dans le mode *power-down*. Si l'on veut faire plusieurs mesures, il sera donc nécessaire de remettre le capteur en mode *single measurement* avant chacune de ces mesures.

Pour la lecture des valeurs des capteurs, on a vu précédemment qu'ils étaient codés sur 13 bits or les registres du composant ne font que 8 octets. On a donc 2 registres par valeur. Voici la liste des registres à lire pour récupérer la valeur des 3 axes du magnétomètre :

Adresse	Nom du registre	Description
0x03	HXL	8 bits de poids faible du capteur de l'axe X
0x04	HXH	8 bits de poids fort du capteur de l'axe X
0x05	HYL	8 bits de poids faible du capteur de l'axe Y
0x06	HYH	8 bits de poids fort du capteur de l'axe Y
0x07	HZL	8 bits de poids faible du capteur de l'axe Z
0x08	HZH	8 bits de poids fort du capteur de l'axe Z

Tableau 3: Registre des mesures du magnétomètre

Comme on peut le voir dans ce tableau, les données sont codées dans le format *Little Endian*. Cela signifie que les 8 bits de poids faible sont placés avant les 8 bits de fort dans l'ordre croissant des adresses mémoires. On remarque également que l'on utilise 16 bits pour coder un nombre de 13 bits signés. Les valeurs ne varieront que de -4096 à 4095. Les 5 bits de poids fort ne pourront donc prendre pour valeur que 00000 ou 11111 en binaire si le nombre est respectivement positif ou négatif.

3.2 - Communication avec le capteur sur Arduino

Comme on vient de le voir, il est nécessaire d'utiliser un bus I²C pour pouvoir communiquer avec le magnétomètre. Il faut donc le relié à un microcontrôleur équipé de ce bus. Le Wifibot s'utilise essentiellement en ajoutant des modules branchés en Ethernet au routeur du robot. Il va donc falloir passer par une carte électronique équipé d'un bus I²C et d'un port Ethernet. Ce montage est tout à fait possible à l'aide d'une carte Arduino équipé d'un module Ethernet.

3.2.1 - Montage des composants sur Arduino

Une carte Arduino est un circuit imprimé qui intègre un microcontrôleur de la marque Atmel AVR ainsi que divers composants permettant de contrôler l'alimentation ou de communiquer par port série avec un ordinateur. Les cartes Arduino sont très populaires et ont la particularité d'être open-source (pour la bibliothèque Arduino) et open-hardware. Dans le cadre de ce projet nous avons utilisé la carte Arduino Uno équipée d'un microcontrôleur Atmega328P travaillant à 16 MHz.



Fig. 3.2 : Une carte Arduino Uno

Pour pouvoir le connecter en Ethernet au robot, il faut lui ajouter un module Ethernet. Celui dont on dispose vient se brancher par-dessus la carte Arduino. Les différents connecteurs de l'Arduino sont cependant toujours accessibles. On peut donc relier le MPU9150 à la carte Arduino à partir des broches SDA et SCL du bus I²C.

Avant de relier le capteur à la carte Arduino, il faut vérifier que les bus des deux composants sont compatibles. La carte Arduino fournit une tension de 5V sur le bus I²C alors que le MPU9150 ne peut recevoir que du 3.3V. Relier ces composants risquerait de griller le capteur. Il existe cependant plusieurs solutions à ce problème. On peut par exemple utiliser un composant dédié à la conversion 5V en 3.3V pour les bus I²C. Ce type de composant serait indispensable si l'on mélangeait des périphériques 5V et 3.3V sur le même bus. Mais dans notre cas, nous n'avons qu'un seul composant en 3.3V. S'il n'y a qu'un seul composant sur le bus, il existe alors un montage beaucoup plus simple permettant de faire communiquer le maître et l'esclave. Sur la carte Arduino, le bus I²C est *collecteur ouvert*. Cela signifie que les broches du bus n'ont aucune force électromotrice. Avec une simple résistance de tirage on peut modifier la tension du bus.

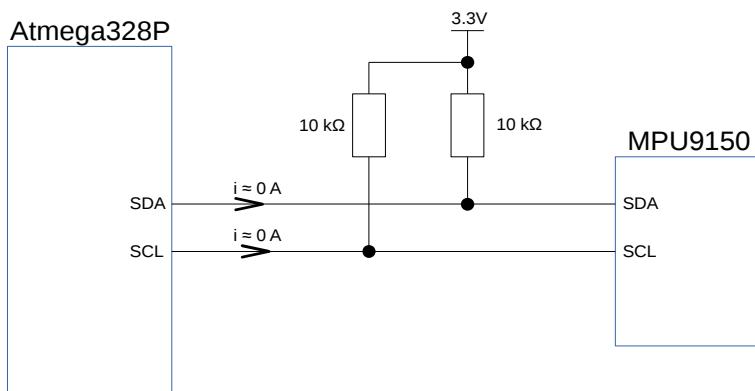


Fig. 3.3 : Résistance de tirage (pull up) sur le bus I²C

Étant donné qu'il ne circule pratiquement pas de courant sur le bus, il ne circule donc pas de courant dans la résistance. En utilisant la loi d'Ohm on en déduit que la tension dans la résistance est nulle donc le bus est bien tiré à 3.3V. Ainsi, le MPU9150 est capable de comprendre le microcontrôleur. Par contre, dans l'autre sens c'est différent. Le microcontrôleur est sensé recevoir une tension de 5V mais le capteur n'envoie que du 3.3V. Heureusement cette tension est supérieure à la tension minimale requise du niveau logique haut. Le microcontrôleur est donc lui aussi capable de comprendre le MPU9150.

En réalité, il n'est même pas nécessaire de mettre des résistances de tirage dans le montage puisqu'elles sont

déjà intégrées dans le capteur. Par conséquent, le montage ne se résume qu'au branchement des broches SDA et SCL du microcontrôleur sur les broches SDA et SCL du MPU9150. Il faut également relier la masse et le +3.3V pour l'alimentation du composant.

3.2.2 - Analyse de l'application

On va maintenant pouvoir détailler le programme à mettre en place pour pouvoir piloter le capteur par le réseau. La première étape va être de déterminer qu'est-ce que l'on souhaite obtenir ? L'objectif de l'utilisation de ce capteur est d'améliorer la localisation d'un Wifibot grâce à l'obtention de son orientation absolue dans un plan. Le robot se déplacera sur un sol plat ; il n'est donc pas indispensable d'utiliser l'axe du magnétomètre qui est orthogonale au sol (l'axe Z).

On a donc un vecteur 2D formé par les composantes X et Y du magnétomètre. Si l'on travaille dans les coordonnées polaires, on obtient alors l'angle par rapport à l'axe X ainsi que la norme du vecteur. L'application pourra donc envoyer sur le réseau cet angle mais il serait peut-être plus judicieux de choisir une référence. En effet, il serait plus intéressant que lorsque l'on démarre le robot, on parte avec un angle nul. Or le robot ne sera pas toujours placé dans la même orientation au démarrage. Il faut donc être capable de réinitialiser l'angle à zéro quand on le souhaite par le réseau.

Un dernier point important concernant le capteur, celui-ci utilise l'effet Hall pour mesurer le champ magnétique. Ce type de capteur nécessite donc un calibrage avant de pouvoir être utilisé correctement. Les différentes valeurs que peuvent prendre les axes du magnétomètre possèdent un biais qui fausserait complètement les mesures de l'angle si l'on ne calibrerait pas le magnétomètre. Et comme ce biais peut varier à cause de l'environnement du capteur, il est nécessaire de pouvoir le modifier par le réseau.

Pour résumer, l'application doit :

- envoyer sur le réseau l'angle représentant l'orientation du robot par rapport à un angle de référence.
- être capable de changer l'angle de référence depuis le réseau
- calibrer les capteurs depuis le réseau

Concernant l'envoi de l'angle sur le réseau, il peut être judicieux d'envoyer également les valeurs brutes du capteur pour faire un traitement du côté de l'ordinateur. L'angle ainsi que les données des capteurs seront envoyé 50 fois par secondes afin d'avoir une application de contrôle du robot très réactive.

Maintenant que l'on sait ce qui doit transiter sur le réseau, il est important de définir un format de trame pour chacune de ces commandes. La communication Ethernet se faisant par des sockets TCP, nous pourrions utiliser des chaînes de caractères contenant les caractéristiques des capteurs mais cela impliquerait de convertir des nombres en chaînes de caractères et inversement. Une autre possibilité serait de réduire au maximum la taille des trames en travaillant non pas en octets mais en bit. C'est à dire stocker chacune des valeurs des capteurs sur 13 bits et de compléter les bits restant par la valeur de l'angle. Cette solution est intéressante trop poussée pour le gain minime qu'elle procurerait. La meilleure des solutions serait d'arrondir à l'octet supérieur pour chaque caractéristique que l'on souhaite envoyer. Par exemple utiliser 2 octets pour stocker la valeur sur 13 bits d'un des capteurs.

Il est également nécessaire d'envoyer un identifiant pour que l'on puisse facilement savoir de quel type de commande il s'agit. Voici un tableau décrivant le format des trames que l'on a mis en place :

Commande	Taille (bits)	Type	Valeur	Description
Envoi des données du magnétomètre	8	entier non signé	0	Identifiant
	16	entier signé	-4096 à 4095	Valeur du magnétomètre X
	16	entier signé	-4096 à 4095	Valeur du magnétomètre Y
	16	entier signé	-4096 à 4095	Valeur du magnétomètre Z
	16	virgule fixe	-180.0 à 180.0	Angle (7 bits après la virgule)
Réinitialisation de l'angle	8	entier non signé	1	Identifiant
Calibrage	8	entier non signé	2	Identifiant
	16	entier signé	-16536 à 16535	Bais du magnétomètre X
	16	entier signé	-16536 à 16535	Bais du magnétomètre Y
	16	entier signé	-16536 à 16535	Bais du magnétomètre Z

Tableau 4: Format des trames réseau de l'application d'écoute du magnétomètre

La taille des trames des différentes commandes fait respectivement 9 octets, 1 octet et 7 octets. N'ayant pas de restriction au niveau du réseau, ces tailles sont correctes. Concernant l'envoi de l'angle, nous aurions pu nous restreindre à l'envoi de nombre entier de -180 à 180 mais la plage de valeurs nécessitait dans tous les cas au moins 2 octets. Étant donné que l'angle est à la base un flottant, autant utiliser les bits restants pour la partie décimale du nombre.

3.2.3 - Conception de l'application

Les cartes Arduino n'ont pas uniquement l'avantage d'être facile d'utilisation. Du fait de leur popularité, elles ont aussi une grande quantité de bibliothèques de composants déjà écrites. De plus, l'utilisation du langage C++ rend leur implémentation très claire et facile à comprendre. C'est pourquoi il existe déjà une classe contenant les méthodes de base pour communiquer avec un MPU9150. Il en est de même pour l'utilisation du module Ethernet. L'écriture de cette application est donc assez facile. Il ne reste plus qu'à mettre en place les liens entre le capteur et le réseau.

Bien que cette application reste relativement simple, il est quand même nécessaire d'écrire quelques classes qui n'existent pas de base comme par exemple une surcouche au MPU9150 pour convertir les données brutes en angle. Voici un diagramme de classe représentant la composition du programme :

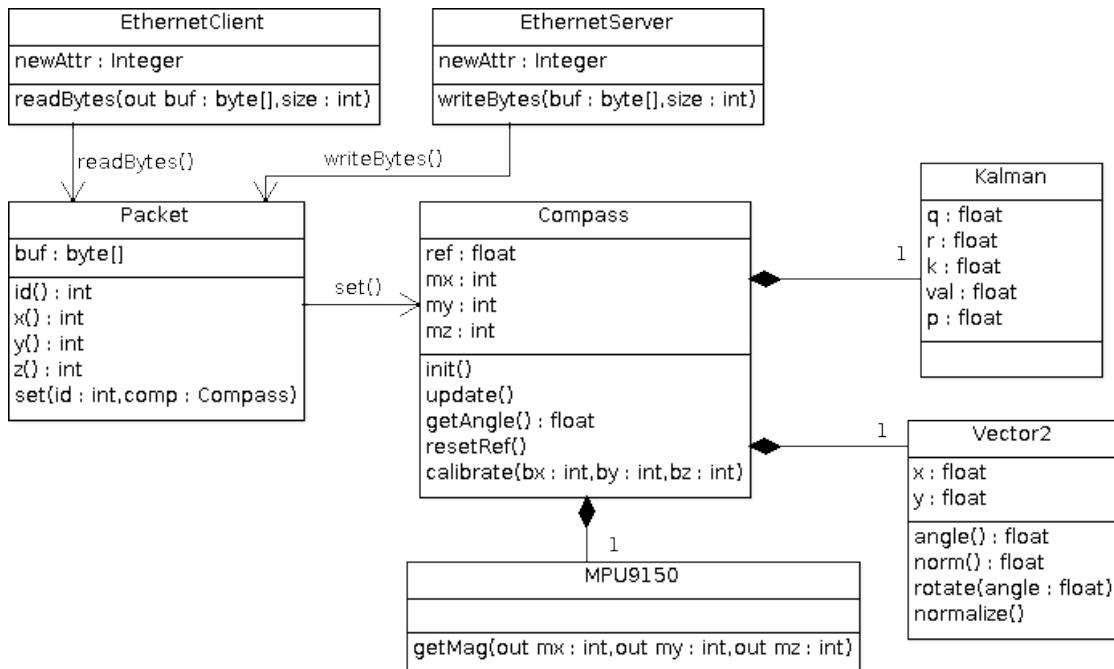


Fig. 3.4 : Diagramme de classe de l'application sur l'Arduino Uno

La classe MPU9150 est une classe que l'on peut trouver parmi les bibliothèques additionnelles d'Arduino. Elle possède un très grand nombre de méthodes permettant d'effectuer toutes les opérations possibles sur le capteur. Dans notre cas, une seule méthode nous est utile : la récupération des valeurs du magnétomètre. Ces 3 valeurs sont récupérées à partir de la classe Compass. Elle calcule l'angle à partir des valeurs brutes auquel elle applique un filtre.

La classe Vector2 est une classe très simple qui permet de manipuler les vecteurs. On peut par exemple calculer leur norme, les faire tourner, obtenir l'angle par rapport à l'axe X, les normaliser, etc. Cette classe est utilisée par Compass pour effectuer le calcul de l'angle à partir des données du magnétomètre et de l'angle de référence.

Les magnétomètres étant très sensibles au bruit, On peut ajouter un filtre passe bas pour adoucir les perturbations de l'angle. Ce calcul est géré par la classe nommée Kalman implémentant un filtre de Kalman simplifié. En effet, le vrai filtre de Kalman effectue des calculs matriciels mais dans celui-ci, les matrices sont de taille 1x1. Il est utilisé pour son côté prédictif qui offre l'avantage d'avoir une valeur plus stable tout en restant très réactif aux changements brutaux de valeur.

Les classes EthernetServer et EthernetClient sont des classes disponibles dans la bibliothèque d'Arduino. Elles sont utilisées avec le module Ethernet pour pouvoir envoyer des données sur le réseau. Sur ce diagramme, ne sont écrites que les deux seules méthodes utilisées dans ces classes. Lorsque l'on souhaite envoyer des données à tous les clients connectés, il suffit d'appeler la méthode `writeBytes()` en précisant en paramètre le buffer contenant les données à envoyer.

La classe Packet sert justement à créer le buffer utilisé pour l'envoi et la réception de commande depuis le réseau. Elle possède une méthode permettant de remplir le buffer à partir des données du magnétomètre pour ensuite envoyer le buffer sur le réseau. Et réciproquement, il est possible de créer une instance de Packet contenant par exemple les données de calibrage lorsqu'un client souhaite configurer le capteur.

On peut résumer le fonctionnement de l'application de la manière suivante : Lorsqu'un client se connecte, l'application va envoyer en boucle les données du magnétomètre ainsi que l'angle. L'acquisition de nouvelles

données se fait par la méthode *update()* de la classe Compass. Cet envoi s'effectue à une fréquence d'environ 50 fois par secondes. Si jamais le client envoie une commande de réinitialisation du référentielle, on appelle la méthode *resetRef()* de la classe Compass. Cette méthode permet de prendre l'angle actuel comme nouvel angle de référence. Et pour finir, si jamais le client envoie la commande de calibration, on appelle la méthode *calibrate()* de Compass en précisant en paramètre les nouvelles valeurs de biais pour chacun des capteurs. Le biais est cependant incrémental ; cela signifie qu'il viendra s'ajouter à l'ancienne valeur de biais.

3.3 - Portage de l'application sur Raspberry Pi

Comme on a pu le voir dans la partie 1.3 de ce rapport, la caméra du Wifibot a été remplacée par une Raspberry Pi équipée de la caméra Pi. Il peut donc être intéressant de n'utiliser qu'une seule carte et de faire fonctionner le magnétomètre sur la carte Raspberry Pi.

3.3.1 - Communication avec le MPU9150

La carte Raspberry Pi peut en effet être utilisée pour faire de l'électronique car elle possède des ports GPIO. Ces ports sont des broches sur lesquels on peut brancher des composants électroniques. A partir de ces broches, la Raspberry dispose d'un bus SPI, d'un port série et également d'un bus I²C. Il est donc possible de communiquer avec le magnétomètre.

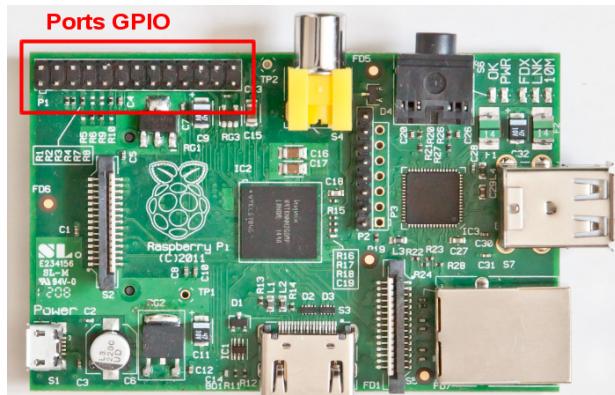


Fig. 3.5 : Position des GPIOs sur une carte Raspberry Pi

L'avantage du bus I²C de la Raspberry est qu'il fonctionne déjà en 3.3V, il n'y aura donc aucun problème de compatibilité avec le capteur. Le montage est très simple : il suffit de brancher les broches SDA et SCL sur celles du Raspberry ainsi que l'alimentation et la masse. Il faut également relier la broche de l'adresse du magnétomètre à la masse pour utiliser son adresse par défaut sur le bus I²C.

Du côté logiciel, il existe différentes bibliothèques pour utiliser les GPIOs. Il y a par exemple Rpi.GPIO pour le langage Python ou encore Pi4J en Java. Mais étant plus à l'aise avec le langage C++, nous avons opté pour la bibliothèque WiringPi écrite en C. Elle offre l'avantage d'avoir les mêmes noms de fonctions que la bibliothèque standard d'Arduino.

Par contre, contrairement à Arduino, il n'existe pas de classe déjà écrite pour le MPU9150. Il est donc nécessaire de réécrire ce driver. Il n'est cependant pas indispensable de l'écrire en entier. On a besoin uniquement de récupérer les valeurs du magnétomètre. Il faut donc seulement initialiser le capteur et lire les registres de ces valeurs. La fonction qui s'occupe de l'initialisation du capteur se résume à ces quelques étapes :

- Connexion au MPU6050 en I²C

- Redirection du bus I²C esclave sur le bus I²C principal
- Connexion à l'AK8975C en I²C
- Déconnexion du MPU6050

Après ces étapes, on dispose d'une liaison directe avec le magnétomètre (AK8975C). La connexion avec le MPU6050 n'a servi qu'à activer le déroutage du bus I²C esclave qui est interne au MPU9150. Concernant la lecture des données des capteurs, les étapes doivent se dérouler de la manière suivante :

- Passage du capteur en mode *single measurement*
- Attente de 10 ms
- Lecture des mesures des 3 capteurs (6 octets)

Après avoir passé le capteur en mode *single measurement*, il est nécessaire d'attendre 10 millisecondes avant de commencer la lecture des registres puisqu'une mesure prend un certain temps. Si l'on essaye de réduire ce temps, les valeurs seront erronées. Il est possible de savoir précisément quand la mesure se termine à partir d'un autre registre de l'AK8975C. On peut même déclencher une interruption en utilisant la broche prévue à cet effet sur le composant. Mais pour l'application que l'on souhaite faire, il n'est pas utile d'optimiser le temps d'acquisition de ces valeurs. En effet, l'application récupère une nouvelle valeur toutes les 20 millisecondes. Cela signifie qu'il faudra endormir le programme le temps restant.

3.3.2 - Description de l'application

Une fois le driver opérationnel, le reste de l'application reste très proche de l'application écrite pour l'Arduino Uno. Les classes comme Kalman, Vector2 ou encore Packet reste inchangées, ce qui n'est pas le cas de la partie réseau.

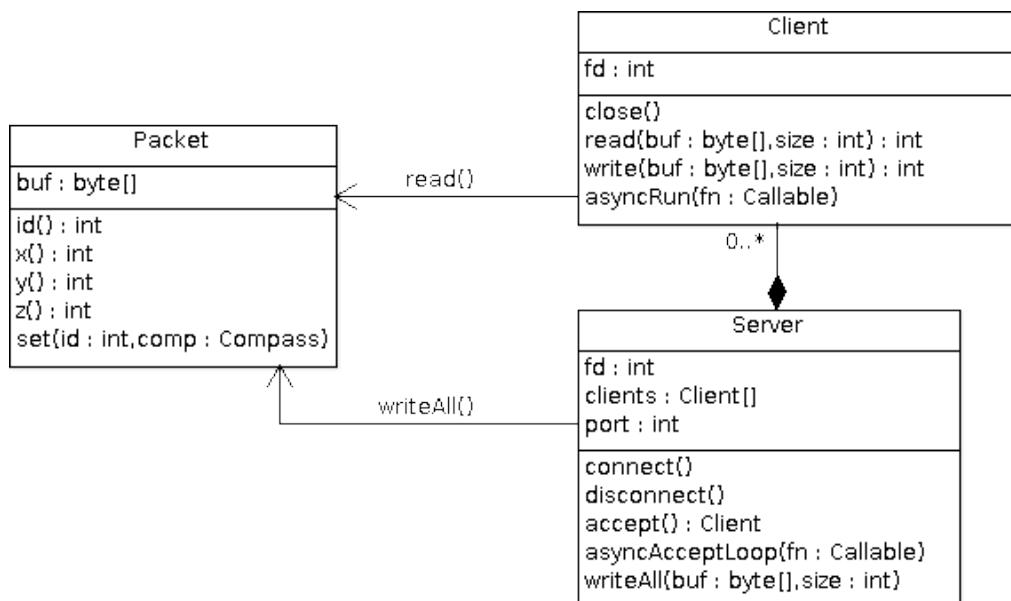


Fig. 3.6 : Diagramme partiel de l'application d'écoute du magnétomètre sur Raspberry Pi

Cette application ajoute deux nouvelles classes (*Server* et *Client*) qui viennent remplacer les classes EthernetServer et EthernetClient de l'application sur Arduino. Dans cette version, il est possible d'avoir plusieurs clients connectés au serveur de l'application. On peut donc par exemple lancer le programme de calibration du capteur et la boussole en même temps. Voici comment se déroule la connexion de plusieurs clients :

La première étape est de connecter le serveur sur le réseau en utilisant la méthode *connect()*. Cela a pour conséquence d'ouvrir un port sur l'interface réseau du Raspberry Pi. Les applications clients peuvent alors se connecter à ce port. Pour que le serveur puisse attendre la connexion d'un client, il suffit d'appeler la méthode *accept()*. Cette fonction est bloquante tant qu'un client ne se connecte pas. Si jamais un nouveau client se connecte, la fonction *accept()* retournera alors une nouvelle instance de la classe *Client*. On pourra alors lui envoyer ou recevoir des données en utilisant les méthodes *read()* et *write()* de cette classe. A noté qu'il est également possible d'envoyer des données à tous les clients connectés en utilisant la méthode *writeAll()* de la classe *Server*.

Ces deux classes sont donc très efficaces pour mettre en place un serveur multi-clients pour l'application, mais il y a cependant un problème. Pour que l'application soit utilisable par plusieurs clients simultanément, il faut être capable d'envoyer les données du magnétomètre tout en étant à l'écoute des clients par la méthode *read()* et en attente de nouveaux clients par la méthode *accept()*, or ces deux méthodes sont bloquantes. Il existe au moins deux solutions à ce problème. Une des solutions serait d'utiliser la fonction C nommée *select()* disponible dans la bibliothèque standard d'UNIX. Elle permet d'attendre qu'un événement se produise chez un des clients avant de rendre la main. L'autre solution serait de diviser le programme en plusieurs threads. Nous avons choisi de mettre en place la deuxième solution.

L'utilisation de threads dans une application est souvent très bénéfique pour les performances de celle-ci. En effet, si l'on parallélise au maximum le travail, celui-ci sera alors fait en beaucoup moins de temps. C'est pourquoi il ne faut pas hésiter à créer des threads lorsque l'on doit effectuer des tâches qui peuvent prendre du temps. Dans le cas de notre application, on peut compter 2 threads plus 1 thread pour chaque client.

Le thread principal du programme va se charger de lancer en boucle l'acquisition d'une nouvelle valeur en appelant la méthode *update()* de la classe *Compass*. Ensuite, il enverra ensuite les données à tous les clients connectés grâce à la méthode *writeAll()* de la classe *Server*. Un autre thread s'occupera uniquement d'attendre la connexion de nouveaux clients. Si un client se connecte, on lui affectera un nouveau thread qui sera chargé d'écouter les commandes qu'il envoie avant d'agir en conséquence.

Pour conclure sur cette application, on peut dire que sa gestion du réseau et son utilisation de threads la rendent plus efficace que sa version sur Arduino. Mais il est toutefois possible d'obtenir des performances comparables en utilisant sur Arduino des interruptions ainsi que des timers pour optimiser son exécution. La différence va donc surtout se jouer au niveau de la facilité de créer une application lorsque l'on utilise des outils de haut niveaux comme la bibliothèque standard de C++11 ou encore les sockets TCP de Linux.

4 - Interfaçage et contrôle du robot

4.1 - Boussole

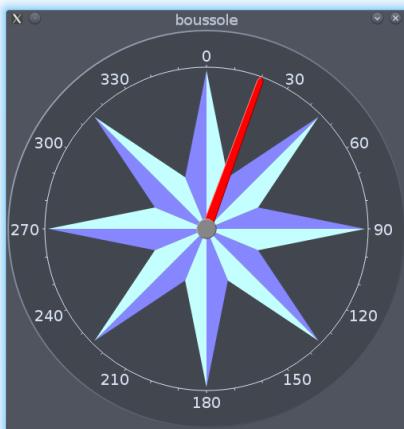


Fig. 4.1: Aperçu de l'application Boussole

Les données brutes envoyées par le capteur ne permettent pas de pouvoir travailler convenablement. La conception d'un élément graphique permet de mieux suivre l'évolution des données. En effet grâce à celui-ci, il est facile de voir le vecteur du champ magnétique. La boussole permet donc de voir dans quel sens le capteur est orienté, et par conséquent comment sera orienté le robot, une fois le capteur posé dessus.

4.1.1 - Théorie

Fonctionnement d'une boussole

Avant de voir comment fonctionne l'application, il faut d'abord s'intéresser au fonctionnement une réelle boussole.

La Terre possède un gigantesque champ magnétique. Celui-ci, comme tous les aimants, possède 2 pôles, un pôle sud et un pôle nord. Ainsi n'importe quel objet métallique magnétisé s'alignera sur la direction formée par ces 2 pôles. Comme vous aurez pu le deviner, l'élément principal de la boussole, l'aiguille, est magnétisé. Le côté de cet objet constitué d'une aiguille, indique donc le pôle nord magnétique terrestre, qui est par ailleurs un pôle de magnétisme sud. Mais comme ce pôle est proche du pôle nord géographique, il a été choisi par convention de le définir comme pôle nord.

4.1.2 - Pratique

Avec l'analyse que l'on a faite auparavant du capteur, et maintenant que l'on en sait un peu plus sur le fonctionnement d'une boussole, nous allons pouvoir détailler la réalisation de l'application Boussole.

Diagramme de classes

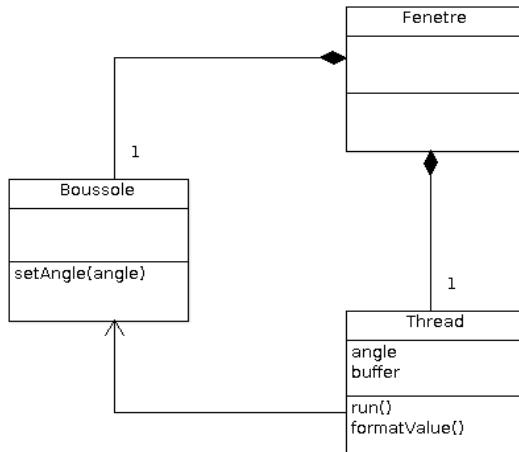


Fig. 4.2: Diagramme de classes de l'application Boussole

Cette application contient seulement 3 classes. La *Fenetre* est le composant graphique principal qui va contenir la *Boussole*. La *Fenetre* possède un *Thread* qui sert à recevoir les données.

Une fois ce *Thread* créé, il est démarré grâce à la méthode *run*. Elle est composée d'une boucle infinie en attente active pour la réception des données. Cela explique pourquoi il faut utiliser un autre *Thread* que le thread principale pour la réception des données, sinon l'application graphique ne fonctionnerait pas le temps que la boucle se finisse, c'est à dire jamais. En premier lieu, ce *Thread* lit **l'entrée standard** pour recevoir les données, puis il stocke une trame en entier dans **buffer**, qu'il formate ensuite dans la méthode *formatValue*.

Ce formatage est réalisée car l'angle reçu se trouve sur un ensemble de 16 bits, et par conséquent à cheval sur 2 octets. Il faut donc réaliser un décalage et un masquage afin de récupérer l'angle, que l'on stocke ensuite dans *value*.

Enfin ce *Thread* émet un signal, afin de notifier à la boussole qu'une nouvelle valeur vient d'arriver. C'est alors que la *Boussole* change la valeur de son aiguille vers cet angle.

Fonctionnement

Le fonctionnement de cette application est très simple. Il suffit de démarrer l'application sur la carte qui est en charge d'envoyer les données du capteur du côté du serveur. Du côté client, il faut simplement démarrer un **client TCP**. Ces données étant affichées sur l'entrée standard, il suffit de rediriger la sortie du client sur l'entrée de la boussole, ce qui permet de voir en temps réel l'orientation du capteur.

Découverte d'une subtilité du codage informatique

En programmant des applications réseaux, on peut tomber sur une erreur difficile à corriger. En effet, les données récupérées par le réseau et affichées dans l'application peuvent ne pas être identiques aux données envoyées. Voici un exemple pour illustrer le problème en rapport avec le codage des nombres.

Prenons le nombre 254 en exemple et regardons la différence dans les étages de codage entre un *char* et un *unsigned char* :

```
char : (11111110)2 = (254)10  
unsigned char : (11111110)2 = (254)10
```

Si on souhaite les **caster** dans un short, on obtient ceci :

```
char : (1111 1111 1111 1110)2  
unsigned char : (0000 0000 1111 1110)2
```

C'est là que la différence apparaît. Mais pourquoi une différence, alors que l'on a voulu stocker le même nombre ? Il faut comprendre comment le compilateur fait pour caster un (unsigned) char dans un short. En premier lieu il va regarder le type, pour un unsigned char, le nombre étant positif, il va remplir les 8 bits en plus par des 0. Si c'est un char, il va d'abord regarder le signe, grâce au bit de poids fort, et compléter les 8 bits avec la même valeur que celui-ci. Cela explique la différence entre les 2 valeurs finales, et le fait qu'il faut absolument utiliser un unsigned char pour réaliser un buffer.

L'autre manière de régler ce problème, serait d'effectuer un masquage pour récupérer seulement les bits qui nous intéressent (les 8 bits de poids faible).

Voici les deux manières de faire avec un code C++:

```
int16_t v1 = buffer[8]           | (buffer[7] << 8); // Si buffer est un unsigned char  
int16_t v1 = buffer[8] & 0x00ff | (buffer[7] << 8); // Sinon
```

Code 1: Deux versions de copie de bits

Mise en évidence de certains problèmes liés au capteur

Si l'on teste cette application dans un endroit possédant des perturbations magnétiques (c'est à dire partout), on s'aperçoit rapidement que l'aiguille n'indique plus forcément le nord magnétique, mais un objet situé dans la pièce.

C'est pourquoi il faut réaliser une application capable de calibrer le magnétomètre avec le champ magnétique local au robot.

4.2 - Calibrage

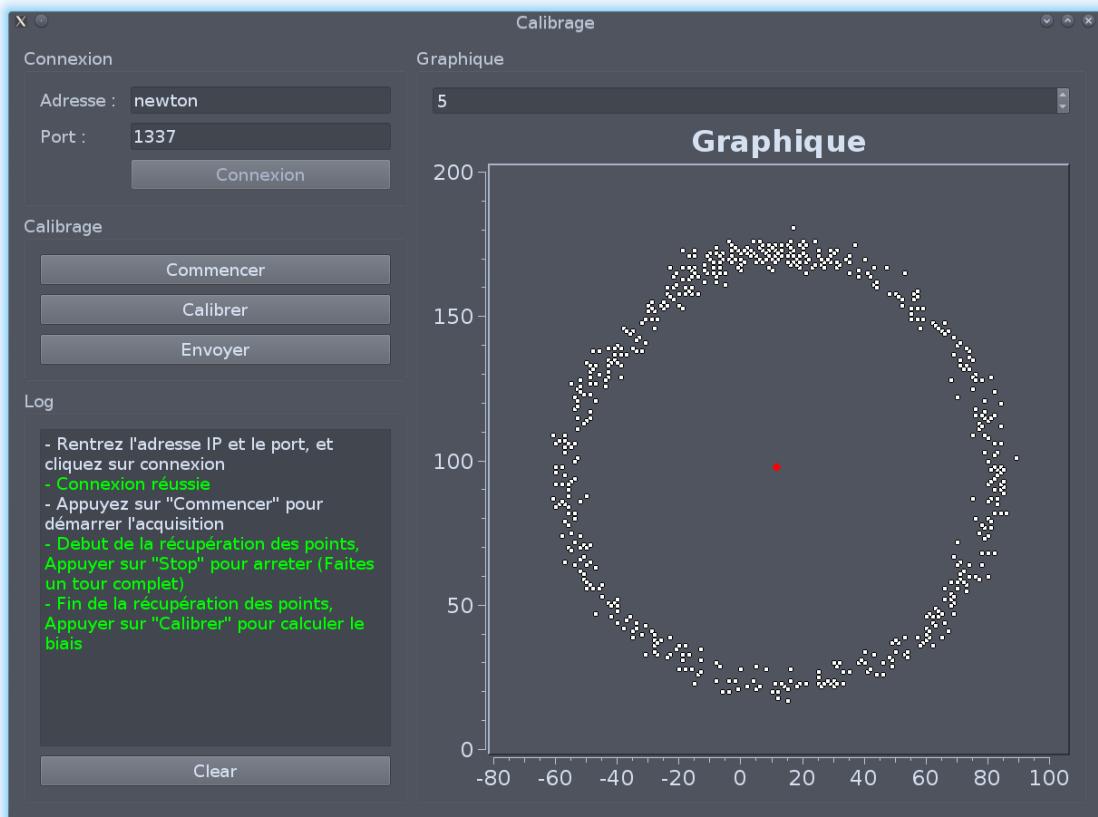


Fig. 4.3: Aperçu de l'application de calibrage

L'application se divise en 4 parties [3] :

- Une partie permettant la gestion de la connexion
- Une partie permettant de gérer le commencement et l'arrêt de la capture des points
- Une partie pour afficher le nuage de points en temps réel, ainsi que son biais
- Une partie permettant d'afficher des messages et donner du **feedback** à l'utilisateur

4.2.1 - Théorie

Le capteur étant troublé par les perturbations magnétiques, on peut supposer qu'un calibrage les prenant en compte dès le départ pourrait y pallier.

Le but de l'application serait donc de passer d'un nuage de points tel que N1 à nuages de points tel que N2

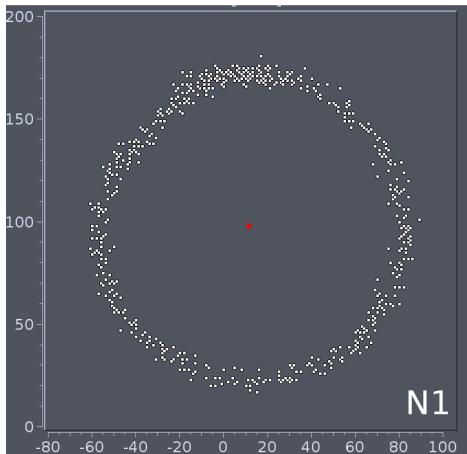


Fig. 4.4: Avant calibration

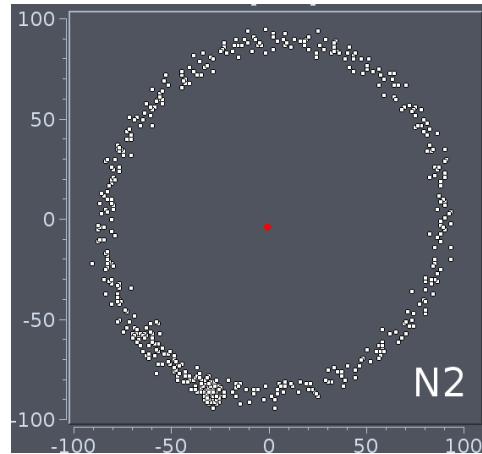


Fig. 4.5: Après calibration

On peut voir que N1 n'est pas centré en l'origine, son centre est égal à (10, 100), alors que le centre de N2 est en (0, 0). On appellera cette déviation : le biais, sa détermination passe par plusieurs étapes.

En premier lieu, il faut calculer le pseudo-diamètre du pseudo-cercle, formé par le nuage de points (si la capture est correcte). Pour cela on récupère les N premiers points en x, puis on calcule une médiane. Idem pour les N derniers points. Puis on calcule la distance entre ces 2 médianes, ce qui donne une bonne approximation du diamètre.

Ensuite, on récupère un couple de points au hasard qui est séparé par un certain pourcentage de ce diamètre (par exemple $\frac{1}{3}$), afin qu'il soit relativement bien éloigné l'un de l'autre. On calcule le segment qui est définie par ces 2 points, ainsi que sa médiatrice. On refait cette opération avec un autre couple de points. Puis on calcule l'intersection de ces 2 médiatrices.

Pour calculer la médiatrice, il faut deux informations caractéristiques à la définition de n'importe quelle droite : un point quelconque sur la droite et un vecteur directeur. Pour le point on récupère le centre du segment. Pour le vecteur directeur, il suffit de récupérer le vecteur défini par les 2 points, et d'effectuer une rotation de 90° (ou une rotation de i en complexe).

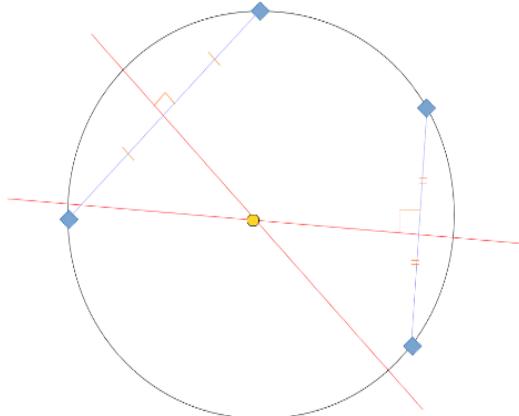


Fig. 4.6: Représentation géométrique de l'intersection des médiatrices

On obtient alors le vecteur directeur, et une description de la droite.

On calcule N intersections (N doit être très inférieur au nombre de points constituant le nuage), puis on calcule la valeur médiane de ces intersections. On calcule la médiane et non pas la moyenne, car les valeurs incohérentes fausseraient le résultat final.

Comme le montre l'image, le nuage de points est représenté par le cercle noir. On récupère 2 couples de points (les diamants bleus), avec leur segment respectif en bleu. On trace leur médiatrice en rouge, et on récupère l'intersection représentée par un octogone jaune.

4.2.2 - Pratique

Diagramme de classes

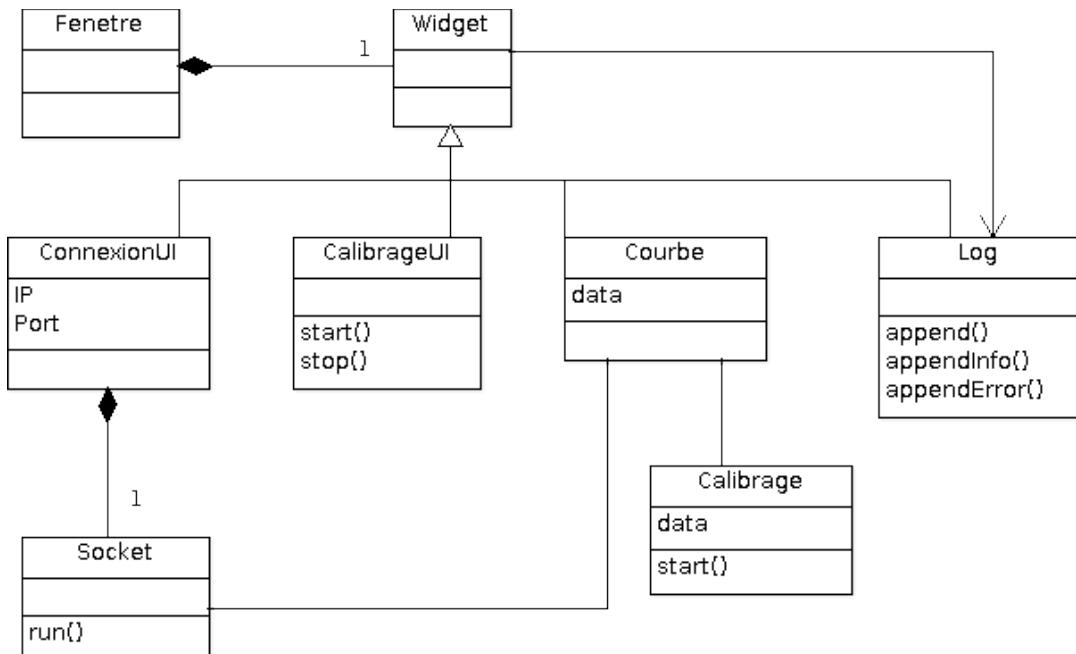


Fig. 4.7 : Diagramme de classes de l'application de calibrage

Comme expliqué plus tôt, l'interface est divisée en différents **widgets**, nous allons nous attarder un peu plus sur le fonctionnement propre à chacun.

ConnexionUI

Ce widget est le plus simple de l'application. Il est constitué d'un champ de texte pour rentrer **l'adresse IP**, d'une **spinBox** pour rentrer le **port** et d'un bouton.

Des valeurs par défauts sont déjà rentrées, mais elles peuvent être modifiées si besoin. L'appui sur *Connexion* récupère les valeurs de l'IP et du port afin de créer une **socket** TCP.

CalibrageUI

Si la connexion s'est bien déroulée, un nouveau widget apparaît, le widget de calibrage. L'appui sur *Commencer* permet de dire à la socket d'envoyer les données à la courbe, afin que celle-ci les sauvegarde et les affiche. Le texte du bouton change alors pour devenir *Stop*. L'appui sur ce bouton, notifie aussi la socket de l'arrêt de l'envoi des données.

L'appui sur *Calibrer* appelle la méthode *calibrate* de Courbe. Nous reviendrons plus tard sur le fonctionnement de ce calibrage.

Enfin le bouton *Envoyer* permet d'appeler la méthode *write* de la Socket, afin d'envoyer la valeur du biais sur le réseau

Courbe [4]

Cet élément est composé d'un graphique, et d'une courbe. Cette courbe est mise à jour grâce à sa méthode *setCurve*. Cette méthode est connectée à l'envoi des données de la socket, ce qui permet de modifier la courbe à chaque envoi de données.

La méthode *calibrate* permet d'appeler la fonction *start* de Calibrage. Elle calcule le biais, crée un nouveau point avec celui-ci, puis l'affiche sur le graphique.

Log

Cette classe est un singleton. Cela signifie qu'il ne peut y avoir qu'une seule instance de l'objet durant l'exécution du programme. Cela permet d'accéder à cette instance à partir de n'importe quelle classe, sans devoir la passer en argument à chaque fois. Pour accéder à l'instance il suffit d'utiliser sa fonction *get*. Nous pouvons alors utiliser l'instance comme une classe normale, et envoyer des messages à n'importe quel moment pour notifier l'utilisateur.

Un code couleur différent est utilisé pour chaque type de message pour être plus visuel (Un évènement s'est bien déroulé : vert, un événement s'est mal déroulé : rouge, un message d'information : couleur par défaut (ici blanc)).

Socket

La classe Socket permet de créer une socket TCP grâce aux informations récupérées dans *ConnexionUI*. La création de cette socket se fait grâce à du code C standard. Cette classé hérite d'un thread, ce qui permet via la méthode *run* d'effectuer un *read* pour recevoir les données. Tout comme pour la réalisation de l'application de la boussole, il faut formater les données afin de pouvoir les lire.

Après avoir reçu et formaté les données, un signal est envoyé pour changer la courbe. La boucle infinie qui est lancé dans la méthode *run* démarre par un test pour savoir si la récupération des points a commencé, ce qui permet d'éviter de lire les **trames** s'il ne faut pas les traiter. La boucle se finie sur l'envoie des données au graphique pour qu'il les affiche.

Calibrage

C'est la classe principale du programme, c'est elle qui se charge de calculer le biais. Afin de pouvoir réaliser ce calcul, il faut le décomposer en plusieurs étapes

On commence par calculer le « diamètre »

```

/** @brief Fonction pour comparer 2 QPoints en x */
class CompareXPoint
{
public:
    bool operator()(const QPointF &p, const QPointF &p2)
    {
        return p.x() < p2.x();
    }
};

float Calibrage::getDiameter()
{
    // On récupère le minimum et le maximum en x
    auto minmax = std::minmax_element(points->begin(), points->end(), CompareXPoint());

    // On renvoie la différence entre les 2
    return minmax.second->x() - minmax.first->x();
}

```

Code 2: Calcul du pseudo-diamètre [2]

Ensuite, il faut prendre deux points au hasard, séparé d'au moins 1/3 de ce « diamètre ». Pour calculer la médiatrice du segment formé par ces 2 points, il faut procéder ainsi :

```

QLineF Calibrage::getBisection(QPointF p, QPointF p2)
{
    // On récupère un point sur la médiatrice
    QPointF center((p2.x() + p.x()) / 2, (p2.y() + p.y()) / 2);

    // On calcule le vecteur directeur
    QVector2D vectDirecteur(-p2.y() + p.y(), p2.x() - p.x());

    // On le normalise
    vectDirecteur.normalize();

    // On définit la médiatrice
    QLineF mediatrice(center - (vectDirecteur.toPointF() * diameter),
                      center + (vectDirecteur.toPointF() * diameter));

    return mediatrice;
}

```

Code 3: Calcul de la médiatrice

On réitère cette opération pour avoir une autre médiatrice, puis on calcule l'intersection de ces 2 droites. Le biais sera calculé par la valeur médiane des composantes x des intersections, et de la valeur médiane des composantes y.

```

QPointF Calibrage::calculMediane(QVector<QPointF> &intersects)
{
    float x, y;
    int size = intersects.size(); // On récupère le nombre d'intersection

    // On récupère la valeur médiane en x
    std::sort(intersects.begin(), intersects.end(), CompareXPoint());
    x = intersects.at(size / 2).x();

    // On récupère la valeur médiane en y
    std::sort(intersects.begin(), intersects.end(), CompareYPoint());
    y = intersects.at(size / 2).y();

    return QPointF(x, y);
}

```

Code 4: Calcul du point médian

Fonctionnement

Le fonctionnement de l'application est très simple. Avant de la démarrer, il faut d'abord s'être connecté au même réseau local que la carte, et avoir lancé le programme permettant l'envoi des données sur la socket TCP, sur la carte.

Une fois ceci fait, on rentre la bonne adresse IP (où nom de domaine), et le bon port. On se connecte, et on démarre l'acquisition. Il faut alors réaliser une rotation de 360° avec le robot pour construire un cercle avec le nuage de points. Ensuite on arrête l'acquisition, on calcule le biais, et on l'envoie sur le réseau. La prochaine acquisition se fera avec ce biais, et donc sera en théorie centré en l'origine.

Confirmation des problèmes

Nous avons pu voir auparavant, que le capteur était vraiment perturbé par les champs magnétiques. Ce calibrage permet d'en tenir compte, et donc de s'en abstraire pour fonctionner normalement. Le problème est qu'une fois que le robot bougera, le capteur étant très sensible, le moindre objet métallique à proximité reviendra le perturber de nouveau.

Ce projet nous permet de conclure sur une chose importante, c'est qu'il est vraisemblablement impossible de calculer la déviation d'un objet seulement avec magnétomètre. Cela pourrait être utilisable avec aucun objet métallique à proximité, ce qui est dur de nos jours.

Perspectives

Il reste plusieurs choses à faire dans le cadre de ce projet pour pouvoir corriger la rotation du robot.

Intégration d'une centrale inertielle

Comme nous avons pu le voir dans la partie IV, le magnétomètre n'a pas pu répondre correctement à notre problématique. Celui-ci, outre son biais que l'on peut désormais enlever, n'est pas capable de fonctionner dans tous les cas. L'ajout d'autres capteurs permettraient de palier à ce problème. Les gyroscopes et accéléromètres possèdent eux-aussi quelques problèmes, et ne donnent pas toujours les résultats que nous voudrions. Lorsque ces 3 capteurs sont couplés, ils permettent d'avoir une rotation absolue dans l'espace, et ce dans n'importe quel cas.

Asservissement du robot

Pour l'instant, le robot peut connaître sa rotation par rapport à un référentiel de départ. Mais aucun travail n'a encore été fait pour savoir si le robot dévie de la rotation voulue lors d'un déplacement.

A partir de cette détection, il faudrait alors opérer un asservissement sur celui-ci. Autrement dit, à partir du moment où l'on détecte une déviation, il faudrait que la future commande du contrôleur la prenne en compte. Cela permettrait alors de corriger cette déviation.

Amélioration logiciel

Actuellement le programme de calibrage du magnétomètre est parfaitement fonctionnel, mais quelques améliorations de celui-ci pourraient être faites. Si la connexion au robot se déroule mal, l'application se fige un certain temps. Il faudrait aussi pouvoir passer le pointeur sur le biais afin de l'afficher. Cela permettrait de savoir si le calcul s'est bien déroulé, et donc de savoir si le biais tend vers (0, 0).

Conclusion

Le but de ce projet était d'intégrer un capteur afin de connaitre l'orientation d'un robot Wifibot en temps réel. Cela dans le futur but de pouvoir détecter une quelconque déviation de celui-ci par rapport à une position voulue.

Cet objectif a été atteint, puisque le Wifibot possède sa rotation par rapport à un angle de référence. Cet angle est correct tant que le robot ne s'approche pas d'autres objets métalliques.

Ce projet nous a apporté beaucoup de choses, puisque nous avons pu travailler avec plusieurs cartes électroniques (Arduino, Raspberry pi), tout en utilisant des connaissances apprises cette année dans différentes matières. Nous avons utilisé des connaissances en réseau pour pouvoir communiquer entre plusieurs machines dans un même réseau local. Nous avons également utilisé les connaissances en systèmes embarqués et en électronique, afin de réaliser le montage des différents modules, ainsi que leur code respectif.

Nous avons appris de nombreuses choses sur les différents capteurs disponibles sur le marché. Nous savons désormais qu'il est compliqué d'avoir la rotation d'un objet dans l'espace en temps réel. En effet il existe beaucoup de phénomènes physiques qui iront perturber la valeur de chacun de ces capteurs.

Bibliographie

- [1] Documentation du MPU9150 [document PDF] [09/18/2013]
http://www.invensense.com/mems/gyro/documents/RM-MPU-9150A-00v4_2.pdf
- [2] Documentation C++ (Standard Template Library) [en ligne] [11/03/2015]
<http://cplusplus.com>
- [3] Documentation de Qt5 [en ligne] [11/03/2015]
<http://doc.qt.io/qt-5/>
- [4] Documentation de QWT [en ligne] [11/03/2015]
<http://qwt.sourceforge.net/annotated.html>

Index des figures

Fig. 1.1 : Diagramme de Gantt prévisionnel.....	2
Fig. 1.2 : Diagramme de Gantt réel.....	2
Fig. 2.1: Un exemple de Wifibot.....	3
Fig. 2.2 : Représentation du réseau interne au robot.....	4
Fig. 2.3 : Application de pilotage des Wifibots.....	5
Fig. 2.4 : Réseau Wifi en mode infrastructure.....	6
Fig. 2.5 : Réseau Wifi en mode ad-hoc.....	6
Fig. 2.6 : Modélisation de l'effet Hall.....	7
Fig. 2.7 : Le MPU9150.....	8
Fig. 2.8: Le Wifibot modifié avec la carte Arduino.....	9
Fig. 2.9: Le Wifibot avec le magnétomètre branché à la Raspberry Pi.....	9
Fig. 3.1 : Bus de communication interne et externe au MPU9150.....	12
Fig. 3.2 : Une carte Arduino Uno.....	14
Fig. 3.3 : Résistance de tirage (pull up) sur le bus I ² C.....	14
Fig. 3.4 : Diagramme de classe de l'application sur l'Arduino Uno.....	17
Fig. 3.5 : Position des GPIOs sur une carte Raspberry Pi.....	18
Fig. 3.6 : Diagramme partiel de l'application d'écoute du magnétomètre sur Raspberry Pi.....	19
Fig. 4.1: Aperçu de l'application Boussole.....	21
Fig. 4.2: Diagramme de classes de l'application Boussole.....	22
Fig. 4.3: Aperçu de l'application de calibration.....	24
Fig. 4.4: Avant calibration.....	25
Fig. 4.5: Après calibration.....	25
Fig. 4.6: Représentation géométrique de l'intersection des médiatrices.....	25
Fig. 4.7 : Diagramme de classes de l'application de calibrage.....	26

Index des tableaux

Tableau 1: Trame I ² C d'une écriture dans un registre.....	11
Tableau 2: Trame I ² C d'une lecture de registre.....	12
Tableau 3: Registre des mesures du magnétomètre.....	13
Tableau 4: Format des trames réseau de l'application d'écoute du magnétomètre.....	16

Index des codes

Code 1: Deux versions de copie de bits.....	23
Code 2: Calcul du pseudo-diamètre [2].....	28
Code 3: Calcul de la médiatrice.....	28
Code 4: Calcul du point médian.....	29

Lexique

Adresse IP	Adresse unique permettant d'identifier un ordinateur sur un réseau
Buffer	Structure permettant de stocker temporairement des données.
Caméra IP	Une caméra IP est une caméra que l'on peut connecter à un réseau puisqu'elle dispose d'un port Ethernet. On peut alors accéder aux images de la caméra par l'intermédiaire d'une interface Web.
Cast	Dans beaucoup de langages informatiques, toutes les variables sont définies par un type bien précis. L'action de « cast » permet de passer d'un type à un autre. Cette opération peut être dangereuse si le type d'arrivé n'est pas fait pour stocker le type de départ.
Client TCP	Application permettant de lire toutes les données reçus venant d'une adresse IP et d'un port donné.
Collecteur ouvert	Une broche en collecteur ouvert est une broche dont le potentiel n'est pas fixé.
Driver	Un driver est une application de très bas niveau servant d'interface entre un périphérique et une application de plus haut niveau.
Entrée standard	Autre nom donné pour toutes les entrées relatives au clavier.
Ethernet	Ethernet est un protocole de communication réseau basé sur l'envoi d'informations sous forme de paquets.
Feedback	Retour d'expérience donné à l'utilisateur une action donnée.
GPIO	Les ports GPIO (General Purpose Input/output) sont des ports d'entrée/sortie dédié au branchement de composants électroniques. Ils sont très utilisés sur les microcontrôleurs ainsi que les nano-ordinateurs comme la Raspberry Pi.
Hub	C'est un appareil permettant d'interconnecter d'autres machines sur un même réseau.
Latence	Lors d'une communication, la latence correspond au temps que met le message à arriver au destinataire.
LittleEndian	L'endianness correspond à l'ordre dans lesquels les octets sont stockés en mémoire. En LittleEndian, les octets de poids faible sont stockés avant les octets de poids fort.
Log	Ensemble de messages d'informations.
Port	Numéro permettant de distinguer différents interlocuteurs sur une même adresse IP.

TCP	Transmission Control Protocol est un protocole de transport d'informations sur le réseau qui a la particularité d'être en mode connecté. Cela signifie que les machines se répondent en envoyant des acquittements pour contrôler l'envoi des données.
Thread	Aussi appelé fil d'exécution, un thread est la plus petite séquence d'instruction qu'un processeur peut diriger indépendamment des autres. L'utilisation de plusieurs threads permet donc la réalisation de tâches en parallèle du point de vue de l'utilisateur.
Trame	Paquet d'informations envoyé sur le réseau.
Socket	Sous linux, une socket est un type de fichier permettant la communication sur le réseau.
Spinbox	Widget permettant de rentrer une valeur entière.