



# Adaptation et optimisation d'un Turtlebot

---

PROJET DE MASTER 2 ESME SUDRIA – DEPARTEMENT  
ÉNERGIE, SYSTEMES ET ENVIRONNEMENT - SOCIETE EXTIA

**ENCADRANT DE PROJET :**  
**M. GEORGE Laurent**

LABEYRIE Julie – CALMELS Julien - MERCIER Quentin  
ESME SUDRIA | MEC 3

## REMERCIEMENTS :

Ce projet n'aurait pu aboutir sans l'aide de nombreuses personnes, notamment du département Systèmes et Énergie de l'ESME Sudria.

Ainsi, nous tenons tout particulièrement à remercier M. GEORGE, notre encadrant de projet pour sa disponibilité, sa réactivité et sa patience qui nous ont permis de mener à bien ce projet et d'enrichir considérablement nos connaissances.

Nous adressons également nos remerciements à M. AIT ABDERRAHIM directeur du département Systèmes et Énergie pour son aide ponctuelle et efficace.

Enfin nous remercions M. PAGIS de l'entreprise EXTIA pour nous avoir proposé ce projet et apporté ses conseils et son expérience.

## SOMMAIRE :

INTRODUCTION .....	5
CAHIER DES CHARGES .....	6
1. EXTIA .....	6
2. Objectifs.....	6
3. Besoins et contraintes.....	6
4. Application .....	7
PARTIE 1 : AUTONOMIE DU ROBOT .....	8
I - GÉNÉRALITÉS SUR LE TURTLEBOT .....	8
1. Description.....	8
2. Prise en main.....	9
3. Étude technique du robot .....	12
a. Asservissement et capteurs.....	12
b. Caméra de profondeur.....	13
II – CHOIX DE LA CARTE ÉLECTRONIQUE .....	17
1. Veille concurrentielle .....	17
2. Tableau comparatif des cartes .....	17
3. Critères de choix et interprétation .....	19
4. Choix de la carte .....	21
III – MISE EN PLACE DE LA CARTE BEAGLEBONE BLACK .....	22
1. Caractéristiques .....	22
2. Installation.....	23
a. Tutoriel .....	23
b. Configurations.....	24
c. Communication avec la carte BeagleBone Black .....	25
d. Alimentation.....	27
3. Performances.....	28
a. Test performances vidéo .....	28
b. Test de fonctionnement du robot sans la caméra.....	29
c. Test de la navigation .....	30
d. Limitations de la carte .....	31
4. Conclusion .....	35
IV - MISE EN PLACE DE LA CARTE ODROID .....	36
1. Caractéristiques .....	36

2. Installation .....	37
3. Performances .....	38
a. Test performances vidéos.....	38
c. Performances de la carte .....	39
4. Conclusion .....	40
 <b>PARTIE 2 : APPLICATION .....</b>	 41
<b>I- SCENARIO COMPLET .....</b>	<b>41</b>
1. Navigation aléatoire.....	41
2. Évitement sans contact.....	41
3. Interaction et détection de personne.....	41
<b>II - NAVIGATION ALÉATOIRE ET DÉTECTION DE PROFONDEUR .....</b>	<b>42</b>
1. Conversion des données.....	42
2. Traitement des données.....	43
3. Problèmes rencontrés et améliorations .....	47
<b>III – CAMÉRA ORIENTABLE.....</b>	<b>49</b>
1. Caractéristiques .....	49
2. Programmation de la carte .....	51
3. Mécanique .....	53
<b>IV – DÉTECTION DE PERSONNE.....</b>	<b>56</b>
1. Cob People Detection .....	56
2. Facedetector de Vicoslab .....	56
<b>V – JOUER DE LA MUSIQUE SUR LE TURTLEBOT .....</b>	<b>57</b>
1. Introduction .....	57
2. Jouer une note via sa fréquence .....	58
3. Jouer une note via son appellation .....	59
4. Jouer un morceau de musique.....	60
https://www.youtube.com/watch?v=8gA3vHog_MI .....	61
5. Conclusion .....	61
<b>VI – APPLICATION COMPLÈTE .....</b>	<b>62</b>
 <b>PARTIE 3 : GÉNÉRALITÉS SUR LE PROJET.....</b>	 63
<b>I - GESTION DU STOCKAGE .....</b>	<b>63</b>
<b>II – PROBLEMES RENCONTRÉS ET SOLUTIONS.....</b>	<b>64</b>
<b>III - ÉVOLUTIONS POSSIBLES : .....</b>	<b>66</b>
<b>CONCLUSION : .....</b>	<b>67</b>

ANNEXES .....	68
BIBLIOGRAPHIE .....	68
TABLE DES FIGURES.....	70
DIAGRAMMES DE GANTT.....	72
SCHEMA BLOC DE LA ODROID .....	74
TUTORIELS :.....	75
TUTORIEL 1 : Installation d'Ubuntu sur une BeagleBone : .....	75
TUTORIEL 2 : Installation Ubuntu Odroid .....	77
TUTORIEL 3 : Installation de ROS .....	81
TUTORIEL 4 : Utilisation de ROS déporté : .....	83
TUTORIEL 5 : Installation d'un package ROS via Catkin .....	85
TUTORIEL 6 : Création d'un package ROS .....	87
TUTORIEL 7 : Utilisation du package sound_publisher.....	99
TUTORIEL 8 : Utilisation de dynamic_reconfigure.....	105
TUTORIEL 9 : Manuel d'utilisation de notre Turtlebot .....	112
TUTORIEL 10 : Commandes de base ROS.....	114

## INTRODUCTION

Cette réalisation a été conduite dans le cadre du projet de troisième année du cycle Ingénieur à l'ESME Sudria en partenariat avec la société EXTIA. L'objectif, dans un premier temps, était de remplacer l'ordinateur portable contrôlant un robot mobile déjà existant ; le Turtlebot par une carte électronique. Puis dans une seconde partie, de développer des applications ludiques permettant au robot d'interagir avec son environnement au sein de la société EXTIA.

Le Turtlebot, créé en 2011, est composé d'une base mobile et d'une caméra 3D. Il est contrôlé via un ordinateur portable qui lui est relié en permanence. L'idée première de ce projet était de remplacer ce PC pour obtenir un système moins encombrant tout en offrant les mêmes performances que le système original. L'intérêt étant d'optimiser le Turtlebot en diminuant sa masse, son coût, sa consommation électrique et en libérant la place occupée par le PC.

Nous avons commencé par réaliser l'étude technique détaillée du Turtlebot, cela nous a donné les lignes directrices du projet. A la suite de cela, nous avons mené une étude comparative des cartes électroniques pour identifier la plus adaptée à notre besoin. Le laboratoire d'énergie de l'école ayant mis à notre disposition une BeagleBone Black, nous avons commencé par installer cette carte électronique sur le Turtlebot. Les résultats n'ayant pas été satisfaisants, nous avons, grâce aux conclusions tirées, sélectionné et mis en place une carte Odroid-XU4.

Dans un second temps, une application a été mise en place afin de permettre à ce robot de se déplacer en autonomie au sein de la société EXTIA et d'interagir avec son environnement, notamment via de la détection faciale.

Pour cela nous avons apporté des modifications mécaniques au robot pour qu'il puisse orienter sa caméra, nous avons mis en place un module de détection de visage, et enfin nous avons implémenté un module de détection d'obstacles.

# CAHIER DES CHARGES

## 1. EXTIA

La société EXTIA, société de conseil en ingénierie est venue à notre rencontre avec un projet précis ; adapter un robot mobile Turtlebot pour qu'il puisse se déplacer de manière autonome au sein de leur entreprise et y effectuer diverses tâches. Le projet étant organisé en deux parties, notre rôle premier était de remplacer l'ordinateur contrôlant le Turtlebot par une carte avec un Linux embarqué, puis ensuite de développer une application.

## 2. Objectifs

- État de l'art, étude de l'existant, prise en main du robot à l'aide du PC.
- Étude des différents capteurs et de leurs gammes de mesures.
- Mise en place d'une BeagleBone Black à la place du PC pour étude des performances
- Choix et mise en place de la carte électronique la plus adaptée
- Applications liées à l'entreprise

## 3. Besoins et contraintes

Après concertation avec l'entreprise EXTIA, la liste des besoins et contraintes concernant la carte électronique a pu être décrite de la manière suivante :

### INDISPENSABLES :

- **Encombrement minimum** : afin de l'intégrer au Turtlebot
- **Intégration facile** : minimiser les actions requises pour mettre en place notre solution sur un Turtlebot
- **Coût faible** : inférieur 100€
- **Consommation minimale** : éviter de réduire l'autonomie du robot de manière significative
- **3 ports USB** (ou 2 port USB et 1 UART) : afin de connecter la caméra et le câble USB de la base mobile
- **Wifi intégré** : pour se connecter en ssh
- **CPU** : avoir une puissance de calcul suffisante pour traiter les images localement
- **Alimentation** : compatible avec les sorties présentes sur la base mobile
- **Stockage** : espace suffisant pour installer Linux et le système d'exploitation du robot

## OPTIONNELS POUR LES ÉVOLUTIONS DU PROJET :

- **Des GPIO** : pour connecter des actionneurs supplémentaires
- **Des PWM** : pour contrôler des moteurs
- **Carte SD** : pour récupérer des données enregistrées

## 4. Application

Concernant l'application à développer pour EXTIA, une grande liberté nous a été donnée. Le seul objectif étant de développer une application ludique liée au Turtlebot et lui permettant d'interagir au sein d'un environnement type Open-space.

## PARTIE 1 : AUTONOMIE DU ROBOT

Cette première partie sera consacrée au remplacement du PC contrôlant le robot par une carte électronique.

### I - GÉNÉRALITÉS SUR LE TURTLEBOT

#### 1. Description

##### **Le robot :**

Le Turtlebot est un robot d'intérieur open source en kit créé en 2011. Il est composé d'une base Kobuki, plus connue comme étant une base de robot aspirateur. C'est d'ailleurs de là que vient son nom de "Turtlebot", Kobuki signifiant "tortue" en coréen. Cette base dispose de deux roues motrices pour se déplacer et de trois bumpers à l'avant pour détecter la présence d'objets. Elle est prévue pour accueillir des capteurs infrarouges avant et dessous. Dans l'application originelle les capteurs devant servaient à identifier des obstacles au ras du sol, tandis que les capteurs dessous servaient à identifier une absence de sol (une marche d'escalier par exemple).

La base Kobuki est surmontée d'une tablette ainsi que d'une caméra vidéo 3D Asus Xtion Pro Live.

L'ensemble est commandé via un ordinateur portable avec un processeur Atom dual-core, de 4Go de RAM et d'une batterie de 2200 mAh.

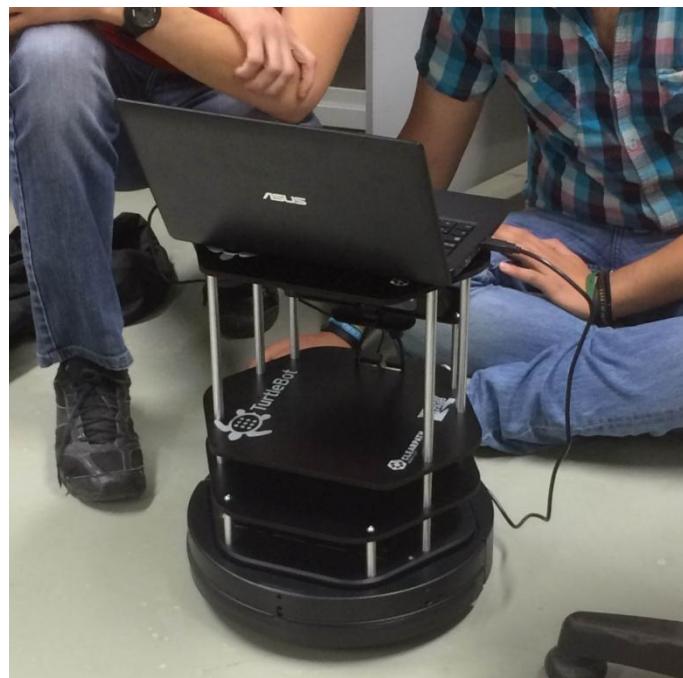


Figure 1 : Photo du Turtlebot

### **Le système d'exploitation :**

**ROS (Robot Operating System)** est une surcouche logicielle appliquée à la robotique. Elle permet de faire abstraction de la partie hardware. Ainsi une application développée pour un robot pourra être utilisée sur un autre robot ne disposant pas des mêmes caractéristiques. La contrainte étant de développer l'application en s'adaptant à ROS. Ce système fonctionne en utilisant le principe d'un maître et de plusieurs esclaves, ce qui permet de faire fonctionner un système robotique complet en utilisant plusieurs ordinateurs différents.

### **Il utilise un principe de nœuds et de topics :**

Chaque processus crée un nœud. Puis chaque nœud peut soit créer des topics soit s'abonner à un topic. Un topic correspondant à une information apportée par un capteur, une caméra, une image traitée numériquement, etc.

Ainsi un nœud peut accéder aux informations transmises par d'autres nœuds. De cette manière en imaginant un système composé de 3 PCs, si le premier PC dispose d'une caméra vidéo, le deuxième de capteurs ultrasons, le troisième pourrait alors utiliser ces deux informations combinées pour une autre application.

Étant donné que le Turtlebot est un robot open source prévu pour être utilisé avec ROS, tous les packages nécessaires à son utilisation existent déjà.

## **2. Prise en main**

Dans un premier temps nous avons testé les applications déjà présentes sur le robot afin de nous familiariser avec le Turtlebot 2 et le système ROS.

### **Commandes de bases :**

Avant de pouvoir commencer à contrôler le robot il faut l'initialiser avec un roslaunch  
`roslaunch turtlebot_bringup minimal.launch`

Pour lancer la caméra il faut lancer les packages ROS relatifs à la caméra :

`roslaunch turtlebot_bringup 3dsensor.launch`

Nous avons également pu nous connecter à l'ordinateur du Turtlebot à distance grâce à un ssh et ainsi contrôler plus aisément le robot

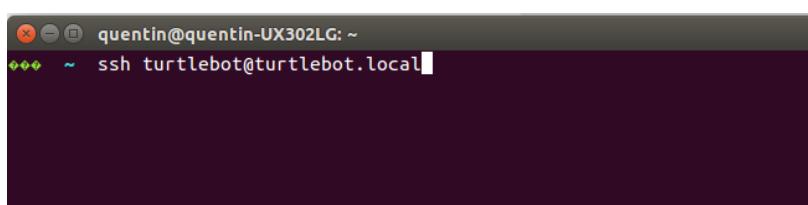


Figure 2 : ssh

Le contrôle du robot avec le clavier a été testé avec la commande teleop  
`roslaunch turtlebot_teleop keyboard_teleop.launch`

```
/opt/ros/indigo/share/turtlebot_teleop/launch/keyboard_teleop.launch http://localhost:11311
turtlebot_teleop_keyboard (turtlebot_teleop/turtlebot_teleop_key)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[turtlebot_teleop_keyboard-1]: started with pid [15788]

Control Your Turtlebot!
-----
Moving around:
  u    i    o
  j    k    l
  m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit

currently:      speed 0.2      turn 1
```

Figure 3 : téléop

### Mode suiveur :

Nous avons pu tester le mode suiveur du robot. Dans ce mode le robot suit la personne qui se trouve devant lui.

`roslaunch turtlebot_follower follower.launch`

Une vidéo est disponible avec le lien suivant : <https://www.youtube.com/watch?v=F-jFqmlIKEU>



Figure 4 : mode suiveur

## Vision 3D :

Nous avons pu constater la qualité de la vision 3D du robot grâce à sa caméra.

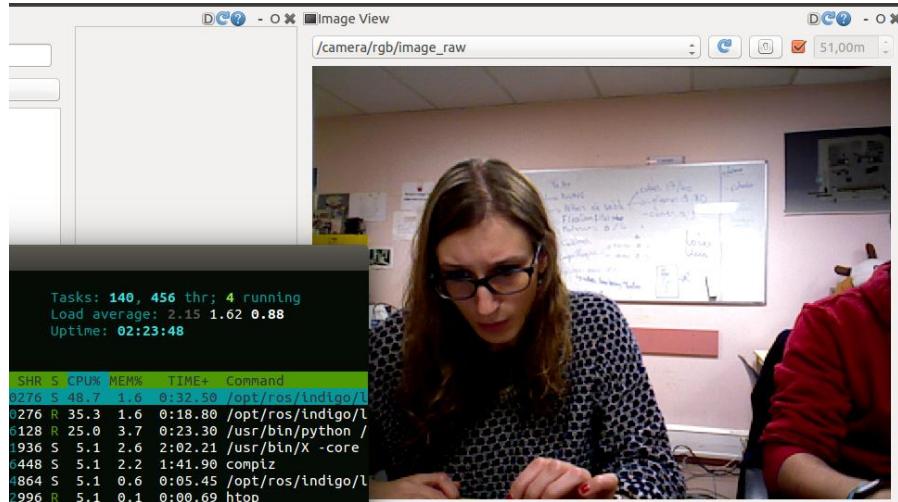


Figure 5 : caméra 3D

## Cartographie :

Nous avons également testé la fonctionnalité cartographie du Turtlebot. Pour cela, nous avons connecté l'ordinateur du robot via ssh. Nous avons lancé les configurations minimales (roslaunch) et la téléopération par clavier à distance. Il faut ensuite utiliser les commandes suivantes :

```
roslaunch turtlebot_navigation gmapping_demo.launch
//pour lancer l'application de cartographie
roslaunch turtlebot_rviz_launchers view_navigation.launch
//pour lancer la visualisation de la carte
```

Pour visualiser la carte en cours de création, nous avons utilisé Rviz. Rviz est un outil de visualisation pour ROS (figure ci-dessous). Nous avons dû lancer cette application directement sur l'ordinateur du robot car nous n'avions pas le package ROS nécessaire installé pour le lancer sur l'ordinateur à distance.

Nous avons ensuite fait se déplacer le robot dans la salle pour qu'il la cartographie. Une fois la carte réalisée, nous avons utilisé la fonctionnalité de déplacement autonome du robot. Il suffit pour cela d'indiquer au robot où il se situe sur la carte et de lui donner un point et une orientation d'arrivée. Si cet objectif est atteignable le robot va s'y diriger de manière autonome. On notera qu'il est impossible pour le robot d'effectuer des déplacements de manière autonome si la téléopération par le clavier est en cours.

Une vidéo de notre robot se déplaçant de manière autonome sur une carte est disponible ici : <https://www.youtube.com/watch?v=l2tr6gyuJY0>

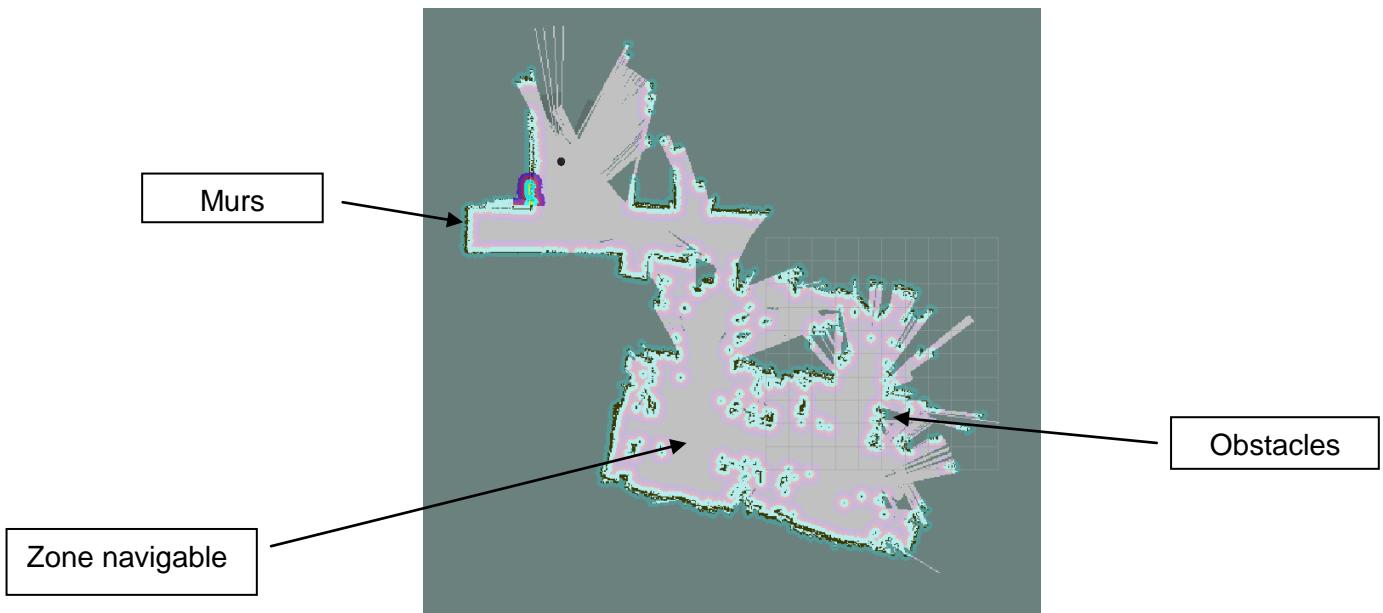


Figure 6 : cartographie

### 3. Étude technique du robot

#### a. Asservissement et capteurs

Dans un second temps, nous avons procédé au démontage complet du robot afin de réfléchir à la possibilité d'implantation de nouveaux capteurs mais également d'avoir un aperçu de la carte contrôlant les moteurs. L'objectif étant de comprendre comment celui-ci était fait afin de savoir si l'ajout de capteurs ainsi que la modification de l'asservissement étaient réalisables.

Suite à cette opération, nous nous sommes rendu compte que l'asservissement était réalisé dans la carte de la base Kobuki du robot aspirateur qui n'est pas open source. Cela signifiait donc qu'il nous était difficile de modifier l'asservissement. Cependant, il existe des protocoles UART pour communiquer avec cette carte qui permettent d'avoir accès aux positions des codeurs et d'envoyer des commandes moteurs. Bien que difficile, l'asservissement est envisageable. Nous nous sommes également rendus compte que des capteurs infrarouge étaient déjà installés dans la base et utilisables par ROS.

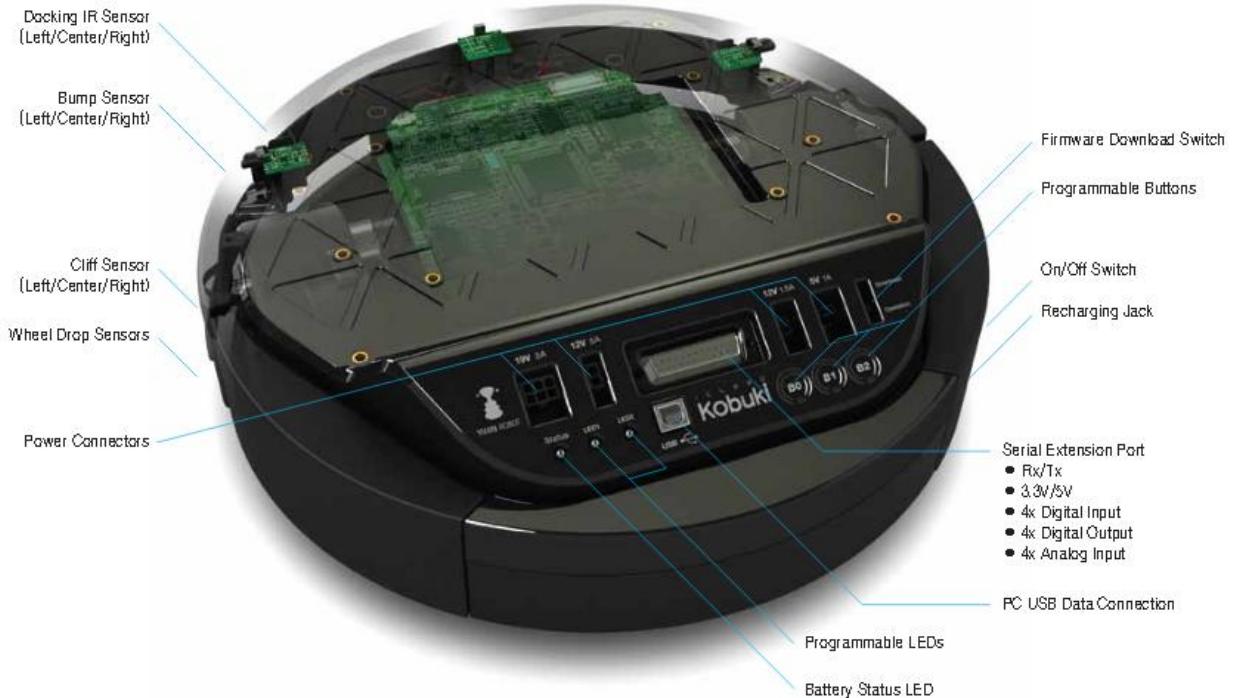


Figure 7 : base Kobuki

### b. Caméra de profondeur

La caméra embarquée, une Asus Xtion Pro live, permet de détecter la distance des éléments devant elle entre 0.8m et 3.5m. L'image de profondeur a une résolution de 640x480 et est renvoyée à une fréquence de 30 images par seconde. La résolution de l'image RGB est de 640x480. La caméra est adaptée à une utilisation en USB 2.0 et USB 3.0. Sa consommation est censée être inférieure à 2.5 W.

La caméra a un champ de vison de 58° par 45°, ce qui permet au robot d'appréhender ce qui se passe devant lui mais pas sur les cotés.

Cette caméra fonctionne très bien pour une utilisation intérieure, malheureusement son utilisation n'est pas appropriée en extérieur, la présence de la lumière du soleil perturbe le fonctionnement du capteur de profondeur.

Il est également possible de modifier la résolution de la caméra ainsi que sa fréquence d'acquisition. Les différents tests que nous avons faits par la suite concernent les résolutions VGA et QVGA.

Format d'affichage vidéo	Définition		Pixels (en millions)	Ratio largeur/hauteur
	X (largeur)	Y (hauteur)		
QVGA	320	240	0,08	1,33
VGA	640	480	0,31	1,33

Tableau 1 : QVGA/VGA

Dans la suite nous nous intéressons au débit nécessaire au bon fonctionnement de ce périphérique d'acquisition.

### ➤ Étude pratique :

Nous avons mesuré la vitesse de transmission des images provenant de la caméra du Turtlebot. Pour ce faire nous nous sommes servis de l'application rqt et plus particulièrement du topic monitor. Nous avons ensuite relevé le débit en fonction des flux que l'on souhaitait visualiser.

▶ <input type="checkbox"/> /camera/depth_registered/image_raw/theora/parameter_updates	dynamic_reconfigure/Config	not
▶ <input type="checkbox"/> /camera/depth_registered/points	sensor_msgs/PointCloud2	not
▶ <input type="checkbox"/> /camera/depth/camera_info	sensor_msgs/CameraInfo	not
▶ <input type="checkbox"/> /camera/depth/disparity	stereo_msgs/DisparityImage	not
▶ <input type="checkbox"/> /camera/depth/image	sensor_msgs/Image	not
▶ <input checked="" type="checkbox"/> /camera/depth/image_raw	sensor_msgs/Image	18.45MB/s 29.85
▶ <input type="checkbox"/> /camera/depth/image_raw/compressed	sensor_msgs/CompressedImage	not
▶ <input type="checkbox"/> /camera/depth/image_raw/compressed/parameter_descriptions	dynamic_reconfigure/ConfigDescription	not
▶ <input type="checkbox"/> /camera/depth/image_raw/compressed/parameter_updates	dynamic_reconfigure/Config	not
▶ <input type="checkbox"/> /camera/depth/image_raw/compressedDepth	sensor_msgs/CompressedImage	not

Figure 8 : débits sur rqt

### Relevés :

	PC Turtlebot - VGA 30 Hz		PC Turtlebot - QVGA 30 Hz	
	Débit	Fréquence	Débit	Fréquence
Caméra infrarouge	18,49M/s	29,83Hz	4,57M/s	29,62Hz
Caméra profondeur	18,45M/s	29,85Hz	4,65M/s	29,81Hz
Caméra RGB	27,56M/s	29,72Hz	6,91M/s	29,74Hz

Tableau 2 : débits du PC

## ➤ Étude théorique pour des images VGA 30Hz:

### Pour l'image RGB:

On calcule le débit théorique que l'on est censé obtenir. C'est une image RGB, donc 3 canaux codés chacun sur 1 octet, soit 3 octets par pixel. L'image fait 640x480 pixels, ce qui représente 921 Ko. L'image est rafraîchie à un taux de 30 fps (images par seconde), ce qui nous donne un débit théorique de **27,6 Mo/s**.

### Pour l'image de profondeur :

On a un seul canal codé sur 2 octets. On a au total 640x480 pixels avec un taux de rafraîchissement de 30 fps, ce qui donne :

$$2 \cdot 30 \cdot 640 \cdot 480 = \mathbf{18.4 \text{ Mo/s}}$$

### Pour l'image infrarouge:

On a un seul canal codé sur 2 octets. On a au total 640x480 pixels avec un taux de rafraîchissement de 30 fps, ce qui donne :

$$2 \cdot 30 \cdot 640 \cdot 480 = \mathbf{18.4 \text{ Mo/s}}$$

On a donc un total :  $27.6 + 18.4 + 18.4 = \mathbf{64.4 \text{ Mo/sec}}$

## ➤ Étude théorique pour des images QVGA 30Hz:

### Pour l'image RGB:

On calcule le débit théorique que l'on est censé obtenir. C'est une image RGB, donc 3 canaux codés chacun sur 1 octet, soit 3 octets par pixel. L'image fait 320x240 pixels, ce qui représente 230.4 Ko. L'image est rafraîchie à un taux de 30 fps (images par seconde), ce qui nous donne un débit de **6.9 Mo/s**.

### Pour l'image de profondeur :

On a un seul canal codé sur 2 octets. On a au total 320x240 pixels à un taux de rafraîchissement de 30 fps, ce qui donne :

$$2 \cdot 30 \cdot 320 \cdot 240 = \mathbf{4.6 \text{ Mo/s}}$$

### Pour l'image infrarouge:

On a un seul canal codé sur 2 octets. On a au total 320x240 pixels à un taux de rafraîchissement de 30 fps, ce qui donne :

$$2 \cdot 30 \cdot 320 \cdot 240 = \mathbf{4.6 \text{ Mo/s}}$$

On a donc un total  $6.9 + 4.6 + 4.6 = \mathbf{16.1 \text{ Mo/s}}$

➤ Conclusion :

Un port USB 2.0 a une vitesse de **60 Mo/sec** maximum en high speed et un USB 3 a une vitesse de **625 Mo/sec**.

En VGA 30HZ, on atteint les limites du débit de l'USB 2.0. Il serait donc préférable d'opter pour une carte possédant un port USB 3.0.

Remarquons néanmoins que dans la pratique, seules les images de profondeur et RGB sont utilisées, soit  $27.6+18.4 = \mathbf{46 \text{ Mo/sec}}$  en VGA à 30Hz, auxquelles viendront s'additionner les débits nécessaires aux autres périphériques USB (carte wifi par exemple).

## II – CHOIX DE LA CARTE ÉLECTRONIQUE

### 1. Veille concurrentielle

Afin de nous guider dans le choix des composants les plus adaptés à notre projet nous avons effectué une veille technologique afin de déterminer si le fait de remplacer l'ordinateur du Turtlebot par un Linux embarqué avait déjà été réalisé.

Après étude, de nombreuses solutions ont déjà été mises en place pour remplacer l'ordinateur. Parmi elles nous avons plus précisément étudié les suivantes :

**BeagleBone Black** : Mise en place dans plusieurs projets. Il en ressort que son intégration au robot est assez simple, mais que la carte n'est pas assez puissante pour pouvoir gérer correctement les applications vidéos et que la bande passante USB est trop limitée.

**BeagleBoard XM** : L'installation de ROS est compliquée et la caméra de profondeur nécessite des packages supplémentaires pour fonctionner. La gestion des applications vidéo est lente.

**RaspberryPI** : Il est possible de faire marcher le Turtlebot grâce à la Raspberry, mais les applications vidéo sont impossibles à utiliser.

**PandaBoard** : Cette carte permet de faire fonctionner correctement le Turtlebot ainsi que la caméra. Cependant elle n'est pas facile d'utilisation.

Les liens de ces différentes analyses sont disponibles en annexe -> Bibliographie -> Veille concurrentielle.

Ces études nous ont permis d'appréhender les difficultés rencontrées. Au vu de ces résultats, nous avons décidé d'élargir nos recherches à d'autres cartes disponibles sur le marché et d'approfondir notre étude afin de choisir la plus adaptée. Cela nous a amené à faire un comparatif des cartes en se basant sur leurs caractéristiques.

### 2. Tableau comparatif des cartes

L'étude technique nous a permis d'établir des critères de choix critiques :

- La puissance CPU
- La RAM
- Les contrôleurs USB

Carte	Nb de cœurs	Type Cœur	Fréquence	RAM	Flash	Micro SD	USB 2.0	USB 3.0	Contrôleurs indépendants	Ethernet indépendant USB	Ethernet	Ethernet	Wifi	Prix
PC Turtlebot	2	Intel Celeron	2.16 GHz	4 Go			2	1	2	oui	1	oui	205€	
BeagleBone Black	1	Cortex A8	1 GHz	512 Mo DDR3	4 GB	oui	1	0	1	non	1	non	55 \$	
Raspberry pi 2	4	Cortex A7	900 Mhz	1 Go	/	oui	4	0	1	oui	1	non	35 \$	
Intel edison	2	2 cœurs Atom Silvermont	500 MHz	1 Go DDR3	4GO	oui	1	0	1	N/C	0	oui	53 euros	
Odroid c1	4	ARM® Cortex®-A5(ARMv7)	1.5 GHz	1 Go	/	oui	4	0	1	oui	1	non	37\$	
Nvidia Jetson TK1	4	ARM Cortex A15 + GPU Nvidia	2.5 GHz	8 Go	16GB	oui	0	1	1	oui	1	non	240€	
Panda Board	2	Cortex A9	1 GHz	1 Go	/	oui	2	0	1	non	1	non	185€	
Nitrogen6_MAX	4	Cortex A9	1 GHz	4 Go DDR3	4 Go	oui	3	0	1	oui	1	oui	249 \$	
Nitrogen Nit6Q_W	4	Cortex A9	1 GHz	1 Go DDR3	4 Go	oui	3	0	1	oui	1	oui	225 \$	
Wandboard Dual	2	Cortex A9	1 GHz	1 Go DDR3	/	oui	1	0	1	oui	1	oui	99 \$	
Wandboard Quad	4	Cortex A9	1 GHz	2 Go DDR3	/	oui	1	0	1	oui	1	oui	129 \$	
Udoo Quad	4	Cortex A9	1 GHz	1 Go DDR3	/	oui	3	0	1	oui	1	oui	130€	
Odroid XU4	8	Cortex A15 + Cortex A7	2 GHz + 1.4 GHz	2 Go	à acheter	oui	1	2	2	oui	1	non	74 \$	
Odroid u3	4	Cortex A9	1.7 GHz	2 Go	à acheter	oui	2	0	1	oui	1	non	69 \$	
Banana Pi pro	2	Cortex A7	1 GHz	1 Go		oui	2	0	1	oui	1	non	62 \$	
Mips Creator Ci20	2	Ingenic JZ4780	1.2 GHz	1 Go	4 Go	oui	2	0	1	oui	1	oui	65 \$	
Orange Pi plus	4	Cortex A7	1.6 GHz	1 Go	8 Go	oui	4	0	1	oui	1	oui	39 \$	

Tableau 3 : comparatif des cartes

### 3. Critères de choix et interprétation

En termes de puissance CPU, il n'était pas facile d'évaluer les besoins réels afin de faire fonctionner ROS correctement. Cependant nous nous sommes fixés comme critère d'être le plus possible équivalent au PC Turtlebot. Le robot fonctionnant correctement sur celui-ci il est logique de vouloir s'en approcher.

Beaucoup de cartes présentent un nombre important de ports USB, cependant il est important de différentier le nombre de ports USB disponibles avec le nombre de contrôleurs USB implémentés sur la carte. En effet, beaucoup de cartes électroniques se vantant d'avoir 4 ports par exemple, intègrent finalement un hub directement sur la carte. Auquel cas les 60 Mo/s théoriques disponibles seraient répartis entre les 4 ports. Nous nous intéresserons donc au nombre de contrôleurs plutôt qu'au nombre de ports physiques présents sur la carte.

Notre étude nous a permis d'établir un classement des cartes étudiées en termes de performances et de faisabilité pour le projet :

Classement	Carte	Prix	Conclusion
1	Odroid XU4	74 \$	Fortement conseillé
2	Nvidia Jetson TK1	240 €	Fortement conseillé
3	Udoo Quad	130 €	Envisageable
4	Nitrogen Nit6_MAX	249 €	Envisageable
5	Nitrogen Nit6Q_W	225 \$	Envisageable
6	Wandboard Quad	129 \$	Envisageable
7	Orange Pi Plus	39 \$	Envisageable
8	Wandboard Dual	99 \$	Envisageable
9	Mips Creator Ci20	65 \$	Limité
10	Odroid U3	69 \$	Limité
11	Odroid C1	37 \$	Très limité
12	Raspberry Pi 2	35 \$	Très limité
13	Banana Pi Pro	62 \$	Très limité
14	Intel Edison	53 €	Très limité
15	Panda Board	185 €	Très limité
16	BeagleBone Black	55 \$	Très limité

Tableau 4 : comparatif carte résumé

1. Le choix qui nous apparaissait le plus adapté était la **Odroid XU4**, en effet elle dispose de deux processeurs quad core, un cortex A15 ainsi qu'un cortex A7, la rendant, au moins équivalente, en termes de puissance de calcul au PC fourni avec le robot. Elle a également un contrôleur USB 3.0 et un contrôleur USB 2.0 ce qui nous permettrait d'avoir un débit suffisant pour acquérir les informations vidéos et d'avoir du wifi. Son seul bémol est sa consommation relativement élevée et sa mémoire à ajouter séparément.
2. La **Nvidia Jetson TK1** était une autre bonne alternative puisqu'elle est quasiment aussi puissante que la Odroid UX4. Elle dispose également d'un gpu Nvidia qui pourrait être un réel plus pour le traitement vidéo. Cependant son nombre limité de GPIO et le fait qu'il n'y ait qu'un seul port USB 3.0 la rendent compliquée d'utilisation.
3. La carte **Udoo Quad** intègre une bonne puissance de calcul ainsi que du wifi. Cependant les ports USB étant des ports 2.0, leur bande passante, comme démontré précédemment, risquait de se montrer un peu limitée.
4. Les cartes **Nitrogen6\_MAX et 6\_QW** ressemblent quelque peu à la Udoo Quad,
5. mais présentent l'inconvénient de ne pas avoir une très grande communauté sur internet.
6. La **Wandboard Quad** dispose uniquement d'USB 2.0 impliquant les mêmes limitations que précédemment. De plus son wifi est contrôlé par un UART du microprocesseur, ce qui pourrait éventuellement la ralentir.
7. **L'orange Pi plus** est une carte bon marché qui présente à peu près les mêmes performances que la Raspberry Pi 2 avec l'avantage de proposer du wifi.
8. La **Wandboard Dual** est une Wandboard Quad deux fois moins puissante. Elle présente donc les mêmes limitations. De plus son processeur risquerait d'être handicapant pour traiter la vidéo.
9. La **Mips Creator Ci20** dispose uniquement de ports USB 2.0. Son processeur semble aussi être un facteur limitant à la réalisation du projet sur cette carte.
10. **L'odroid U3** dispose d'un bon processeur, cependant il n'y a pas de wifi et son unique contrôleur USB 2.0 ne se prêtait pas vraiment à l'utilisation de la caméra sous ROS.
11. **L'Odroid C1, la Raspberry Pi 2 et la Banana Pi Pro** sont à peu près équivalentes en terme de performances, leur processeur quad core d'ancienne génération est limité,
12. mais reste néanmoins potentiellement suffisant pour le projet. Cependant elles ne disposent que d'un seul contrôleur USB 2.0 pour faire fonctionner à la fois l'acquisition vidéo et le wifi, ce qui d'après la théorie ne suffit pas.
13. **L'intel Edison** dispose directement du wifi, malheureusement son processeur paraît insuffisant pour faire fonctionner le robot Turtlebot et sa caméra. Elle présente également le désavantage d'avoir une communauté assez restreinte.
14. La **Pandaboard** dispose d'un processeur correct, cependant elle n'a pas de wifi et contrairement aux autres cartes son contrôleur USB 2.0 est également lié au contrôleur

Ethernet. Ce qui signifie que l'on ne peut pas décharger le bus USB en reliant la carte au réseau via Ethernet si nécessaire. Le contrôleur USB ne permet pas d'acquérir suffisamment d'images et de les renvoyer en wifi.

16. La **BeagleBone Black** arrive en dernier dans le classement. En effet, elle présentait les mêmes contraintes que la Pandaboard. C'est aussi la carte qui disposait du processeur le moins rapide parmi celles étudiées.

Notre classement a été uniquement réalisé par rapport aux performances et ne tient pas compte du prix, c'est pourquoi notre choix ne s'est pas forcément porté sur la première carte du classement.

#### 4. Choix de la carte

Le laboratoire Énergie et Systèmes ayant mis à notre disposition dès le début du projet une BeagleBone Black, et dans un souci de coûts nous avons décidé, malgré les résultats de l'étude comparative, d'essayer d'utiliser cette carte pour piloter le Turtlebot.

L'étude du fonctionnement de ROS sur cette carte nous a permis de confirmer les hypothèses émises pour les critères de choix ainsi que de prendre en main ROS tout en se familiarisant avec les distributions Linux.

## III – MISE EN PLACE DE LA CARTE BEAGLEBONE BLACK

### 1. Caractéristiques

Les caractéristiques détaillées de la BeagleBone Black sont les suivantes :

- Alimentation 5V DC ou par câble USB en 500 mA maximum
- Microprocesseur TI Sitara AM3358 ARM Cortex A8, cadencé à 1GHz, sur 32 bits
- 1 mémoire RAM de 512 MB à 606 Mhz
- 1 mémoire flash de 2GB
- 1 mémoire EEPROM, pour pouvoir stocker jusqu'à 32 KB à laquelle on accède via le bus I2C.
- Un système d'exploitation installé par défaut, Angstrom Linux
- 4 leds que l'on peut contrôler grâce aux ports d'entrée/sortie GPIO
- 65 ports entrée/sortie GPIO
- 2 connecteurs d'extension de 46 pins chacun et qui supportent 3.3 V maximum
- 1 bouton reset pour remettre à 0 le microprocesseur
- 1 bouton boot pour forcer la BeagleBone à démarrer sur la carte SD ou sur le port USB
- 3 ports I2C,
- 5 ports UART
- 8 sorties PWM
- Un emplacement pour carte SD
- 1 port USB,
- 1 port Ethernet,
- 1 Bus SPI
- ADC intégré
- 7 entrées analogiques

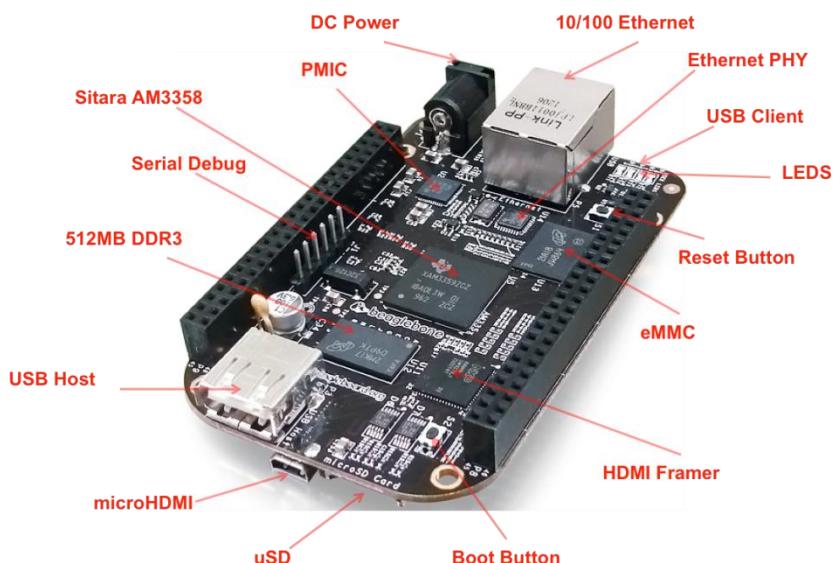


Figure 9: caractéristiques BeagleBone

## 2. Installation

### a. Tutoriel

#### ➤ Ubuntu

La première étape à la réception de notre BeagleBone Black a été d'installer Ubuntu. Pour cela nous avons suivi un tutoriel sur Internet :



Figure 10 : Tutoriel Ubuntu

Celui-ci décrivant l'installation d'Ubuntu sur plusieurs cartes et de plusieurs manières différentes, il était relativement facile de s'y perdre. C'est pourquoi nous avons créé un tutoriel spécialement adapté à la BeagleBone Black disponible en annexe->Tutoriels->Tutoriel 1 : installation d'Ubuntu sur la BeagleBone Black.

Étant donné que la majorité de l'espace de stockage de la carte était occupé par l'installation d'Ubuntu, nous avons ajouté une carte SD que nous avons montée dans l'arborescence afin de pouvoir installer ROS.

#### ➤ Montage de la carte SD :

```
df //Affiche l'espace disque  
cfdisk /dev/mmcblk0 //pour créer une partition  
mkfs.ext4 /dev/mmcblk0p2 //pour créer un système de fichiers sur la carte SD  
mkdir /opt/ros //pour créer le dossier d'installation
```

```

nano /etc/fstab
//on édite le fichier /etc/fstab pour ajouter le point de
montage de la carte. Pour cela, on rajoute la ligne :
/dev/mmcblk0p2 /opt/ros ext4 noatime,errors=remount-ro 0 1
mount -a
//monte le système de fichiers dans l'arborescence

```

## ➤ ROS

Enfin, nous avons installé ROS sur notre carte. Nous avons également suivi un tutoriel pour le réaliser. Les étapes ainsi que les packages utilisés pour cette application sont disponibles en annexe->Tutoriels->Tutoriel 3 : installation de ROS.

## Ubuntu ARM install of ROS Indigo

There are currently builds of ROS for Ubuntu Trusty armhf. These builds include most but not all packages, and save a considerable amount of time compared to doing a full source-based installation.

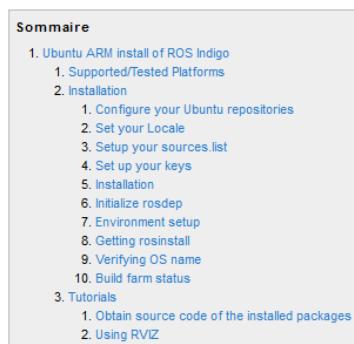


Figure 11 : tutoriel ROS

### b. Configurations

La configuration utilisée pour Ubuntu sur notre carte est la suivante :

- BeagleBone Black/Green: v4.1.10-ti-r21 kernel
- ubuntu-14.04.3

### c. Communication avec la carte BeagleBone Black

#### ➤ SSH :

##### Mode d'utilisation :

Pour ouvrir des terminaux sur la carte via le PC, il faut s'y connecter via un protocole ssh. Pour cela il faut que la carte et le PC soient tous les deux sur un même réseau internet.

Dans notre cas, n'ayant pas de wifi sur la carte nous avions deux possibilités :

- Utiliser un câble Ethernet
- Utiliser un dongle Wifi

Un routeur wifi nous permet de partager un réseau local entre notre carte et d'autres PC. A partir de là, il suffit d'ouvrir un terminal et de lancer la commande suivante :

```
ssh utilisateur@adresse_ip
```

Si l'on connaît le nom de la carte ainsi que le nom de l'ordinateur on peut aussi utiliser la commande ssh de la manière suivante :

```
ssh utilisateur@nom_de_la_carte.local
```

Ce qui a l'avantage de ne pas dépendre de l'adresse ip qui n'est pas forcément la même à chaque connexion.

Dans notre cas on a :

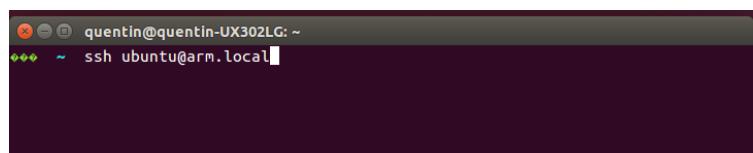


Figure 12 : ssh carte

##### Paramètres :

Pour obtenir l'adresse ip sous linux, il faut taper la commande `ifconfig` dans un terminal. On obtient une sortie similaire à celle-ci :

```
eth0      Link encap:Ethernet HWaddr 00:07:E9:D5:E0:5D
inet addr:192.168.14.1 Bcast:192.168.14.255 Mask:255.255.255.0
          inet6 addr: 2001:6a8:204::1/48 Scope:Global
          inet6 addr: fe80::207:e9ff:fed5:e05d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:346293248 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1089423722 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
```

```

RX bytes:1501808809 (1.3 GiB) TX bytes:4184566400 (3.8 GiB)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:16436 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

```

Ici on peut voir qu'il y a deux interfaces réseaux ; eth0 et lo. La première correspond à notre connexion en Ethernet à la carte, la connexion lo quant à elle est une interface virtuelle de "loopback" qui ne nous concerne pas. Si une interface wifi est connectée à la carte ou au PC, alors il y aura également un champ wlanX. L'adresse ip se trouve sur la deuxième ligne "**inet  
addr:192.168.14.1**".

Le nom de l'utilisateur peut être obtenu dans un terminal en tapant la commande suivante :  
`echo $USERNAME`

Celui de la carte avec la commande :

`hostname`

**Remarque :** On peut également trouver ces informations à tout moment en haut du terminal si celui-ci est ouvert sur la carte :

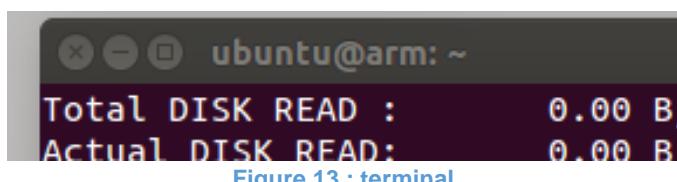


Figure 13 : terminal

### ➤ PuTTY :

L'application PuTTY est un émulateur de terminal qui nous permet de voir en temps réel ce qu'il se passe sur la BeagleBone. Il permet également de lui envoyer des commandes via le port série de debug. L'application permet ainsi d'établir des connexions directes par liaison série RS-232. En revanche, elle ne permet pas d'envoyer des programmes à la carte.

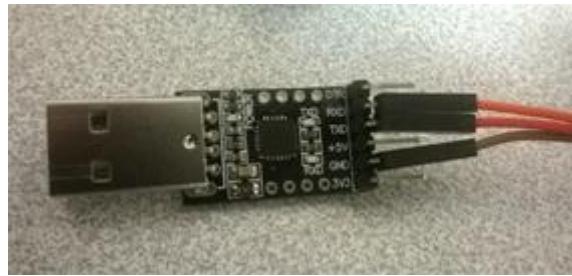
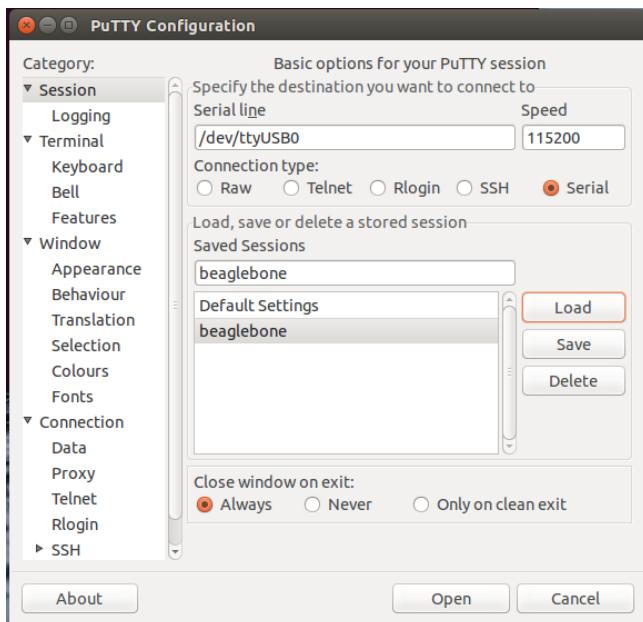


Figure 14 : port série debug

Pour lancer l'application il suffit d'écrire la ligne de commande suivante :

```
sudo putty
```

Il faut ensuite sélectionner la BeagleBone préalablement paramétrée sous PuTTY. De bons tutoriels étant disponible facilement, les étapes de paramétrage ne seront pas détaillées ici.



 A screenshot of a PuTTY terminal window titled '/dev/ttyUSB0 - PuTTY'. The window displays the boot logs of a BeagleBone. The logs include messages from the loader, kernel, and various drivers. It shows the IP address being assigned as 192.168.7.2 and the root user logging in successfully.

```
[ 0.758366] bone-capemgr bone_capemgr.9: loader: failed to load slot-6 BB-BON
[ 0.774875] ELT-HDMIN:00A0 (prio 2)
[ 0.837464] omap_hsmmc mmc.5: of_parse_phandle_with_args of 'reset' failed
[ 0.849179] pinctrl-single 44e10800.pimux: pin 44e10854 already requested by
44e10800.pimux; cannot claim for gpio-leds.8
[ 0.856470] pinctrl-single 44e10800.pimux: pin-21 (gpio-leds.8) status -22
[ 0.856470] pinctrl-single 44e10800.pimux: could not request pin 21 on device
pinctrl-single
Loading, please wait...
Scanning for Btrfs filesystems
systemd-fsck[200]: rootfs: clean, 77488/230144 files, 417183/919296 blocks
Debian GNU/Linux 7 beaglebone tty00
default username:password is [debian:temppwd]
Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian
The IP Address for usb0 is: 192.168.7.2
beaglebone login: [ 25.168433] libphy: PHY 4a101000,mdio:01 not found
[ 25.173653] net eth0: phy 4a101000,mdio:01 not found on slave 1
Debian GNU/Linux 7 beaglebone tty00
default username:password is [debian:temppwd]
Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian
The IP Address for usb0 is: 192.168.7.2
beaglebone login:
```

Figure 15 : lancement de Putty

#### d. Alimentation

A l'origine, la carte tirait son énergie d'une alimentation de laboratoire. Nous avons par la suite commandé des connecteurs afin de réaliser un câble permettant de la connecter sur la sortie 5V/1A du Turtlebot.

### 3. Performances

#### a. Test performances vidéo

Afin de tester les performances vidéo de la BeagleBone, nous avons décidé de faire un comparatif des débits et fréquences des différentes caméras et de les comparer à celles du PC.

Dans un premier temps, et pour valider la cohérence des relevés faits sur le PC durant l'étude technique (Partie I-3-b), nous avons décidé d'effectuer les mêmes tests sur un autre ordinateur.

**Relevés sur un second PC :**

	PC Quentin- VGA 30 Hz		PC Quentin- QVGA 30 Hz	
	Débit	Fréquence	Débit	Fréquence
Caméra infrarouge	18,47M/s	29,98Hz	4,56M/s	29,88Hz
Caméra profondeur	18,45M/s	29,97Hz	4,61M/s	29,63Hz
Caméra RGB	27,31M/s	29,52Hz	6,93M/s	29,72Hz

Tableau 5 : débits PC Quentin

On remarque que l'on obtient les mêmes valeurs qu'avec le PC du Turtlebot, ce qui nous permet de valider la cohérence des mesures effectuées précédemment en fonctionnement normal.

Pour vérifier ensuite que les performances de la BeagleBone Black soient suffisantes en comparaison du PC, nous nous sommes connectés en ssh dessus. Nous avons ensuite fait les tests en choisissant des paramètres d'images inférieurs à ceux utilisés pour les tests des PC (QVGA) car la carte est incapable de traiter dans un temps correct les images VGA.

**Nous avons mesuré les performances sur la carte de deux manières différentes :**

1. Dans un premier temps tout le flux vidéo est transmis au PC, et les mesures sont réalisées sur ce dernier (carte via Ethernet).
2. Dans un second temps le flux vidéos reste sur la carte et les mesures y sont réalisées directement (carte local).

On remarque que les performances mesurées localement sont meilleures que celles à distance, mais que la carte n'arrive tout de même pas à atteindre la consigne de 30 images par seconde, malgré le fait que la résolution soit plus faible que celle du PC.

	Carte via ethernet- QVGA 30 Hz		Carte local- QVGA 30 Hz	
	Débit	Fréquence	Débit	Fréquence
Camera infrarouge	1.98M/s	12,8hz	2,8 M/s	21,7 Hz
Camera profondeur	2.40M/s	15,40 Hz	2,30M/s	19Hz
Camera RGB	1,92M/s	8,22Hz	2,88M/s	13Hz

Tableau 6 : débits BeagleBone

Même en ayant une qualité vidéo réduite, la BeagleBone n'arrive pas à atteindre les performances de l'ordinateur.

### Tableau récapitulatif :

	PC Turtlebot - VGA 30 Hz		PC Turtlebot - QVGA 30 Hz		Carte local- QVGA 30 Hz	
	Débit	Fréquence	Débit	Fréquence	Débit	Fréquence
Camera infrarouge	18,49M/s	29,83Hz	4,57M/s	29,62Hz	2,8 M/s	21,7 Hz
Camera profondeur	18,45M/s	29,85Hz	4,65M/s	29,81Hz	2,30M/s	19Hz
Camera RGB	27,56M/s	29,72Hz	6,91M/s	29,74Hz	2,88M/s	13Hz

### b. Test de fonctionnement du robot sans la caméra

Nous avons connecté la base Kobuki du robot à la BeagleBone et lancé un téléop au clavier. Le robot fonctionne correctement et répond aux commandes envoyées par le clavier.

Une vidéo du test est disponible : <https://www.youtube.com/watch?v=mub8I7ctiO0>



Figure 16 : photo du Turtlebot avec la BeagleBone

### c. Test de la navigation

Les précédents tests nous ont montré que l'on atteignait les limites de la carte électronique. Cependant, nous avons souhaité approfondir notre étude en utilisant le logiciel de cartographie ROS ainsi que le "Follower" et comparer les résultats avec ceux du PC.

Malheureusement le test pratique a démontré que leur fonctionnement n'était pas possible. Pour le follower, nous avons bien réussi à lancer les packages nécessaires, la caméra s'est bien allumée mais le robot était dans l'impossibilité de nous suivre.

La cartographie ne fonctionnait pas non plus, lors de l'utilisation de cette dernière, le logiciel n'acquérirait aucune image et ne pouvait donc pas cartographier.

#### d. Limitations de la carte

Les performances vidéo n'étant pas celles escomptées, nous avons réalisé différents tests pour comprendre d'où venait ce ralentissement.

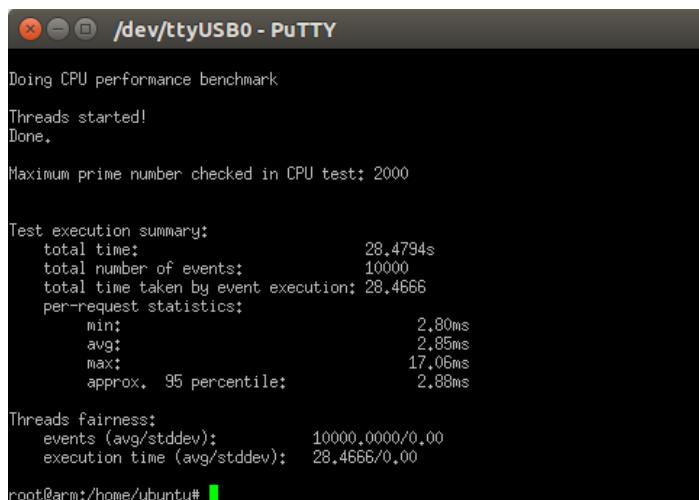
##### ➤ Benchmark du cpu

Pour comparer les performances du cpu de la carte avec celui du PC Turtlebot nous avons installé le logiciel sysbench.

```
sudo apt-get install sysbench
```

Pour lancer un test permettant d'évaluer les performances du CPU il faut utiliser la commande suivante (le test peut prendre une trentaine de secondes) :

```
sysbench --test=cpu --cpu-max-prime=2000 run
```



```
Doing CPU performance benchmark
Threads started!
Done.

Maximum prime number checked in CPU test: 2000

Test execution summary:
    total time:          28.4794s
    total number of events:      10000
    total time taken by event execution: 28.4666
    per-request statistics:
        min:                  2.80ms
        avg:                  2.95ms
        max:                  17.06ms
        approx. 95 percentile: 2.88ms

Threads fairness:
    events (avg/stddev):   10000.0000/0.00
    execution time (avg/stddev): 28.4666/0.00
root@arm:/home/ubuntu#
```

Figure 17 : Benchmark BeagleBone

L'information importante à relever est le "total time : 28.4794s"

Résultat pour le PC Turtlebot : 1.9521s

Résultat pour la Beaglebone black : 28.4794s

Résultat pour une Raspberry Pi2 (à titre comparatif) : 31s

On se rend ainsi compte des limites du processeur de la BeagleBone Black.

##### ➤ Visualisation de l'utilisation du CPU et de la RAM

Dans un second temps nous avons voulu mesurer le taux d'utilisation du CPU de la carte lors d'une utilisation normale du Turtlebot.

### Observation :

La commande “top” permet de visualiser tous les processus fonctionnant sur l’ordinateur et d’évaluer l’utilisation des ressources qui en résulte. Il est présent par défaut sur Linux mais son utilisation peu intuitive donne un nombre d’indications limité. C’est pourquoi nous avons installé et utilisé sa version améliorée : htop.

Pour installer htop il faut exécuter l’instruction suivante :

```
apt-get install htop
```

Et pour l’utiliser :

```
htop
```

Htop nous a alors montré que le processeur de la carte n’était pas assez performant, en effet lors d’acquisition vidéo en QVGA 30Hz ce dernier était à 100% d’utilisation. De plus, les processus utilisant le maximum de ressources étaient tous liés à ROS.

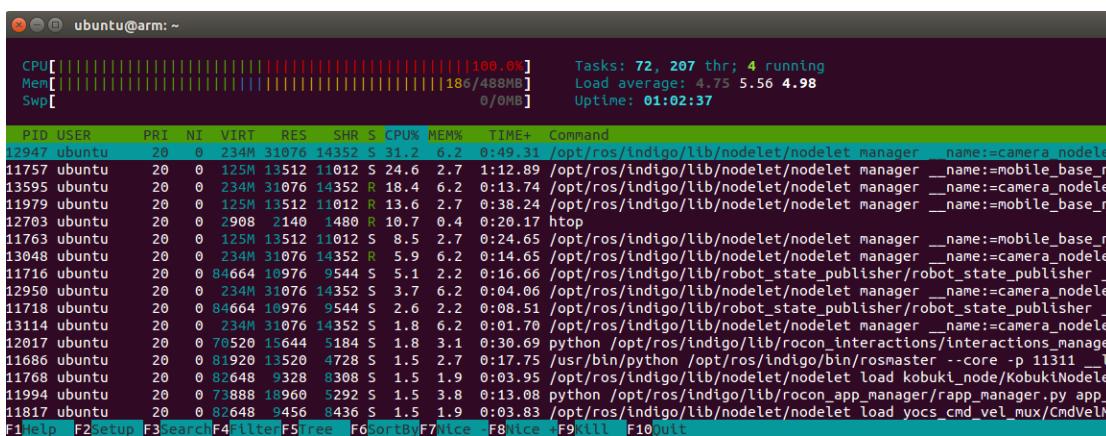


Figure 18 : htop Beaglebone

En comparaison, voici l’utilisation du processeur du PC du Turtlebot pour une vidéo en QVGA 30Hz :

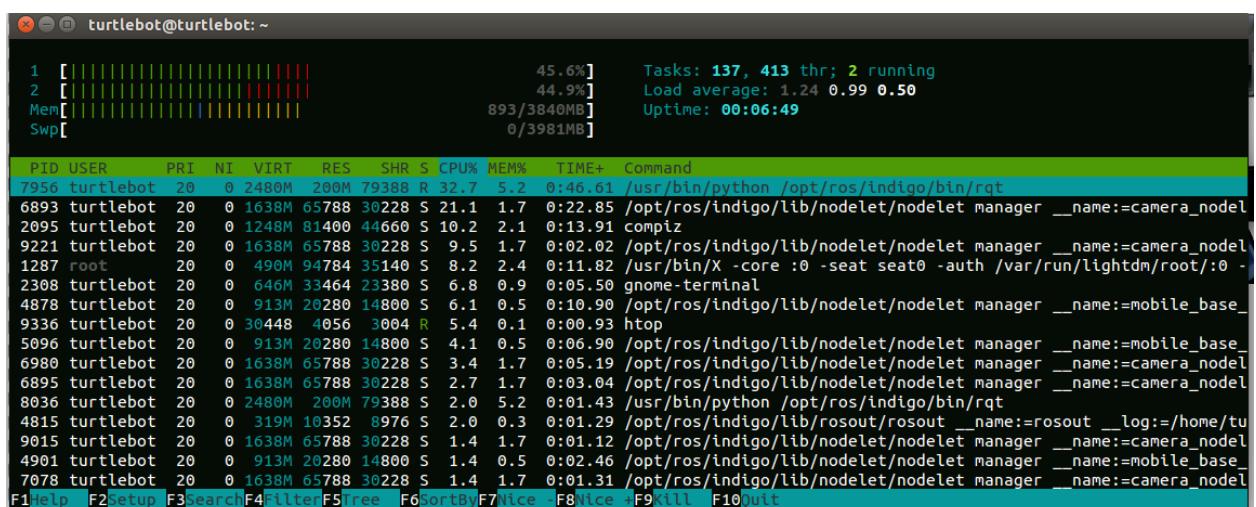


Figure 19 : htop PC

On voit que le processeur n'utilise que 50% de ses performances.

### Tentative d'amélioration des résultats :

Le cpu de la carte étant au maximum de ses capacités, on a utilisé la commande :  
[cpufreq-info](#)

Cette commande permet de vérifier la fréquence de fonctionnement du CPU, et également ses fréquences d'utilisation. Dans notre cas on obtient les statistiques suivantes, nous prouvant que la fréquence du CPU monte bien à son maximum.

```
cpufreq stats:  
300 MHz: 79.93%  
600 MHz: 0.99%  
720 MHz: 0.16%  
800 MHz: 0.56%  
1000 MHz: 18,36% (fréquence maximale pendant 18,36% du temps)
```

Afin d'essayer d'améliorer les performances du CPU, on lui impose une fréquence constante de fonctionnement à 1 GHz :

```
cpufreq-set -f 1000MHz
```

Malgré ce changement, l'utilisation du CPU reste à 100% en fonctionnement normal du robot, la limitation n'est donc pas liée à la fréquence.

### ➤ Recherche limitation contrôleur micro SD

Nous avons également voulu vérifier si la carte SD n'était pas à l'origine de la limitation des performances, liée éventuellement à une bande passante trop faible. En effet la pratique nous a montré que la carte SD pouvait ralentir notre système, notamment lors de l'utilisation de la complétion de commande dans le terminal.

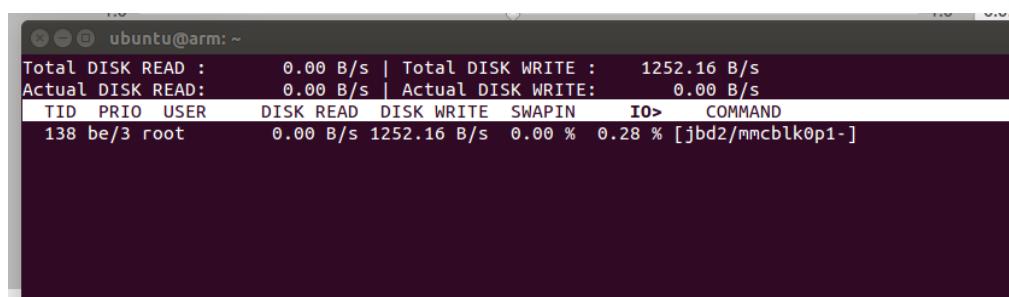
Pour cela nous avons utilisé la commande iotop pour observer les écritures sur le disque. Pour installer iotop il faut exécuter l'instruction suivante :

```
apt-get install iotop
```

Et pour l'utiliser :

```
iotop
```

Finalement iotop n'a pas révélé de problème, ce qui est cohérent car tous les programmes et données sont exécutés dans la mémoire RAM et non pas dans la carte SD. Cependant un contrôleur SD plus rapide donnerait plus de confort à l'utilisation de la carte.

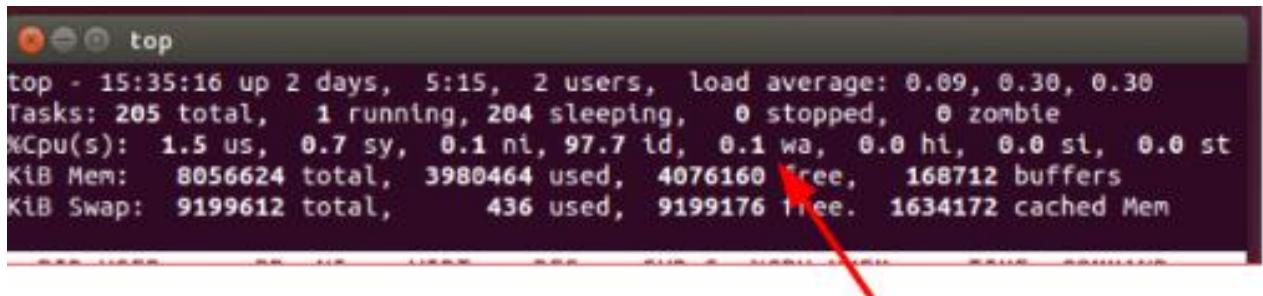


The screenshot shows the iotop command running in a terminal window. The output provides disk I/O statistics. At the top, it shows total and actual disk read/write rates. Below that is a header for the process list, which includes columns for TID, PRIO, USER, DISK READ, DISK WRITE, SWAPIN, IO%, and COMMAND. A single process entry is listed: TID 138, PRIO be/3, USER root, DISK READ 0.00 B/s, DISK WRITE 1252.16 B/s, SWAPIN 0.00 %, IO% 0.28 %, and COMMAND [jbd2/mmcblk0p1-].

Figure 20 : iotop BeagleBone

### ➤ Recherche limitation USB

Pour identifier si la bande passante de l'USB pouvait limiter l'accès aux données et donc engendrer l'insuffisance du taux de rafraîchissement, nous avons utilisé la commande top. En effet, même si son utilisation n'est pas très intuitive, le champ "wa" permet de savoir si le système est ralenti par des périphériques.



```
top - 15:35:16 up 2 days, 5:15, 2 users, load average: 0.09, 0.30, 0.30
Tasks: 205 total, 1 running, 204 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.5 us, 0.7 sy, 0.1 ni, 97.7 id, 0.1 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 8056624 total, 3980464 used, 4076160 free, 168712 buffers
KiB Swap: 9199612 total, 436 used, 9199176 free. 1634172 cached Mem
```

Figure 21 : top BeagleBone

On observe que le champ est la plupart du temps égal à zéro. Ce qui permet de penser que l'USB n'est pas à l'origine du problème. En pratique, la vidéo n'étant qu'en QVGA, soit une faible résolution, cela est relativement rassurant.

## 4. Conclusion

En définitive, la BeagleBone Black permet de faire fonctionner la base Kobuki du robot correctement, mais dès que l'on ajoute les fonctionnalités de la caméra, elle atteint ses limites. L'issue de ce test n'est pas surprenante, en effet notre comparatif avait montré que cette carte possédait des limitations évidentes. Nos précédents tests ont permis de déterminer qu'elles étaient liées au processeur de la carte.

Tout le travail réalisé n'est pas pour autant perdu. Il nous a permis de déterminer précisément les performances requises pour faire fonctionner un tel système. De plus ces manipulations nous ont permis de nous familiariser avec l'environnement Linux et avec ROS.

Pour assurer les performances requises par le cahier des charges, et compte tenu de son coût relativement faible, nous avons choisi de remplacer la BeagleBone par une carte Odroid-XU4.

## IV - MISE EN PLACE DE LA CARTE ODROID

### 1. Caractéristiques

Les caractéristiques détaillées de la Odroid-XU4 sont les suivantes :

CPU	Samsung Exynos-5422 : Cortex™-A15 and Cortex™-A7 big.LITTLE processor with 2GByte LPDDR3 RAM
eMMC 5.0 module(Option)	16GB/32GB : Sandisk iNAND Extreme 64GB : Toshiba eMMC
PMIC	Samsung S2MPS11 9 high-efficiency Buck, 1 Buck-Boost regulators, RTC and 38 LDOs. Contact Samsung for more information
Ethernet controller	Realtek RTL8153 The Realtek RTL8153-CG 10/100/1000M Ethernet controller combines an IEEE 802.3u compliant Media Access Controller (MAC), USB 3.0 bus controller. QFN-48 Package
USB 3.0 Hub	Genesys GL3521 The Genesys GL3521 is a 2-port, low-power, and configurable USB 3.0 SuperSpeed hub controller. QFN-48 Package
USB Load Switch	NCP380 Protection IC for USB power supply from OnSemi.
Input Power protector	NCP372 Over-voltage, Over-current, Reverse-voltage protection IC from OnSemi.
LED indicator	Blue LED to display the status of operating system
HDMI connector	Standard Type-A HDMI, supports up to 1920 x 1080 resolution
IO Ports	USB 3.0 Host x 2, USB 2.0 Host x 1, PWM for Cooler Fan, UART for serial console Ethernet RJ-45, 30Pin : GPIO/IRQ/SPI/ADC, 12Pin : GPIO/I2S/I2C
Storage slot	Micro-SD slot, eMMC 5.0 module connector
DC Input	5V / 4A input, Plug specification is inner diameter 2.1mm and outer diameter 5.5mm

Tableau 7 : caractéristiques Odroid

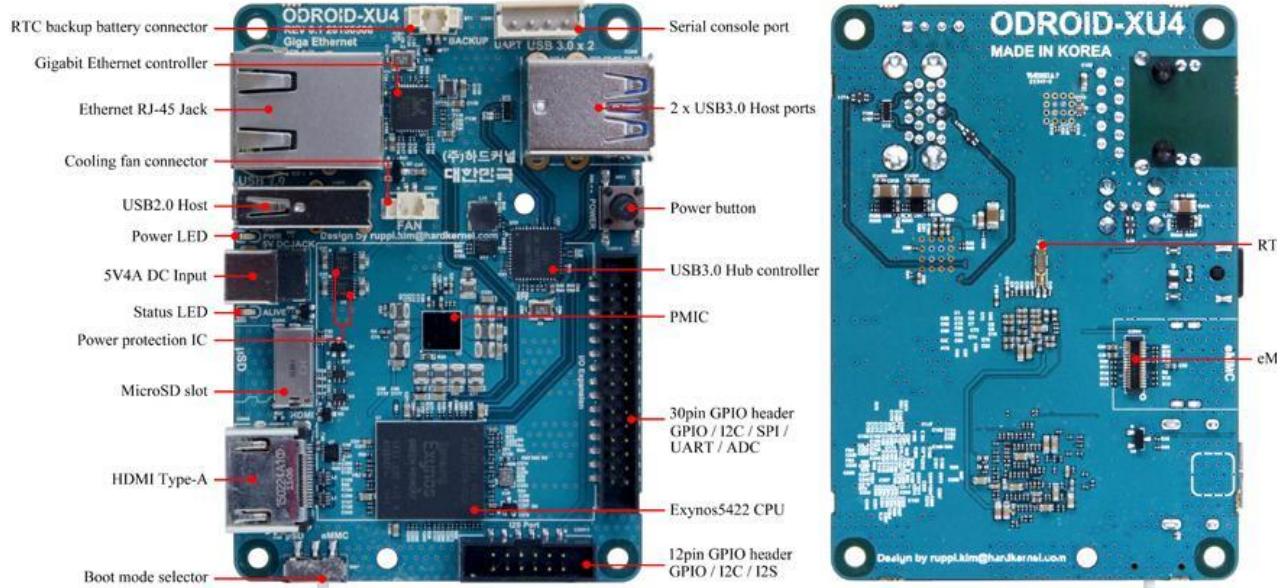


Figure 22 : vue de la Odroid

Le schéma bloc complet de la carte se trouve en annexe.

## 2. Installation

Comme pour la BeagleBone nous avons installé Ubuntu sur la carte Odroid. Pour cela nous avons choisi une version compilée spécialement pour la carte avec les caractéristiques suivantes :

- - Ubuntu 14.04.1 LXDE
- - Linux Kernel 3.10.51

Les étapes de l'installation se trouvent en annexe -> Tutoriel 2.

De plus nous souhaitions que la carte soit alimentée par la sortie 12V/5A du robot. Afin d'adapter cette sortie à l'entrée d'alimentation de la Odroid soit 5V / 4A, nous avons mis en place un Buck (abaisseur de tension) dont les caractéristiques sont les suivantes :

Parameters:	
Input voltage	DC 12/24V (9V-32V)
Output voltage	5V DC
Output power	50W
Minimum dropout	0.5V
Min voltage difference	2V
Conversion efficiency	90%
Soft start time	about 500mS
Output ripple	50mV(Max) 20M band-width
Operating temperature	-40°C to +85°C
Temp rise in full load	50 Celsius
Adjust ratio in load	+/- 1%
Volt adjust ratio	+/- 2%
Dynamic response speed	5% 200µS
Short-circuit protection	sustainable and self-healing
Input reverse connection protection	none
Small size	65x58x22mm



Figure 23 - Caractéristiques du Buck

### 3. Performances

#### a. Test performances vidéos

On effectue des mesures de débit à l'aide du topic monitor. Les résultats obtenus sont les suivants :

	PC Turtlebot - VGA 30 Hz		Carte Odroid via Ethernet VGA 30Hz		Carte Odroid local VGA 30Hz	
	Débit	Fréquence	Débit	Fréquence	Débit	Fréquence
Camera infrarouge	18,49M/s	29,83Hz	6,8M/s	11 Hz	18,6M/s	30Hz
Camera profondeur	18,45M/s	29,85Hz	6,6M/s	10Hz	18,48M/s	30Hz
Camera RGB	27,56M/s	29,72Hz	6,2M/s	6,8Hz	27,78M/s	29.8Hz

Tableau 8 : débits de la Odroid

Ces débits correspondent avec ceux calculés de manière théorique. On remarque néanmoins que l'on obtient un débit plus faible en passant par le réseau, la vitesse étant sûrement limitée par le contrôleur du bus Ethernet.

Du point de vue de la qualité vidéo, les résultats obtenus sur la carte Odroid en local sont satisfaisants et identiques à ceux obtenus sur le PC d'origine Turtlebot.

### b. Test de la navigation

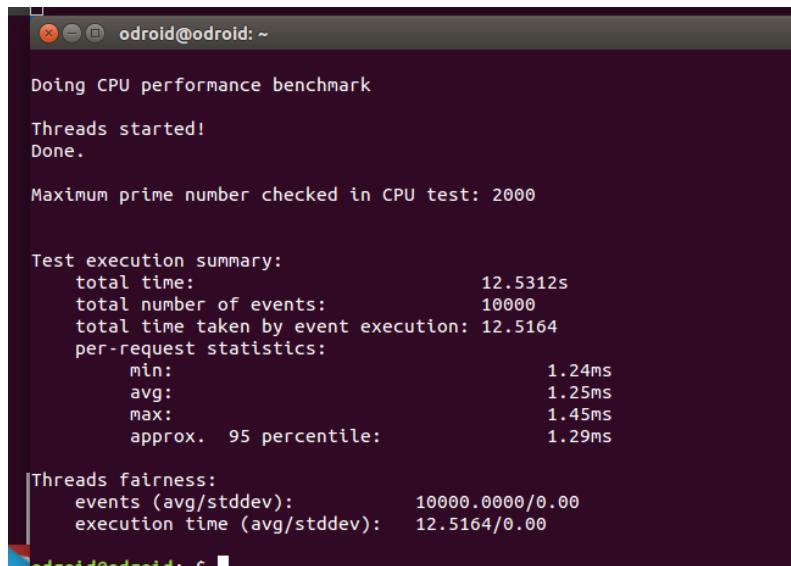
Nous avons effectué des tests sur le mode follower, ainsi que la cartographie et le déplacement autonome. Tous les programmes fonctionnent correctement sans ralentissement.

Une démonstration du mode follower avec la Odroid est disponible au lien suivant :

<https://www.youtube.com/watch?v=XBQZJg21d-o>

### c. Performances de la carte

Afin d'évaluer les performances de la carte nous avons réalisé un Sysbench :



```
odroid@odroid: ~
Doing CPU performance benchmark
Threads started!
Done.

Maximum prime number checked in CPU test: 2000

Test execution summary:
    total time:          12.5312s
    total number of events: 10000
    total time taken by event execution: 12.5164
    per-request statistics:
        min:                1.24ms
        avg:                1.25ms
        max:                1.45ms
        approx. 95 percentile: 1.29ms

Threads fairness:
    events (avg/stddev):   10000.0000/0.00
    execution time (avg/stddev): 12.5164/0.00
```

Figure 24 : benchmark Odroid

Résultat pour le PC turtlebot : 1.9521s

Résultat pour la BeagleBone black : 28.4794s

Résultat pour la Odroid XU4 : 12.5312s

On voit que le système est plus rapide que la BeagleBone Black, mais reste toutefois un peu plus lent que l'ordinateur. Cette lenteur est néanmoins compensée par le nombre de coeurs processeur présent sur la carte Odroid. Elle en possède 8 contre 2 pour le PC.

Nous avons ensuite mesuré l'utilisation des processus avec un Htop en ayant lancé le teleop et la caméra :

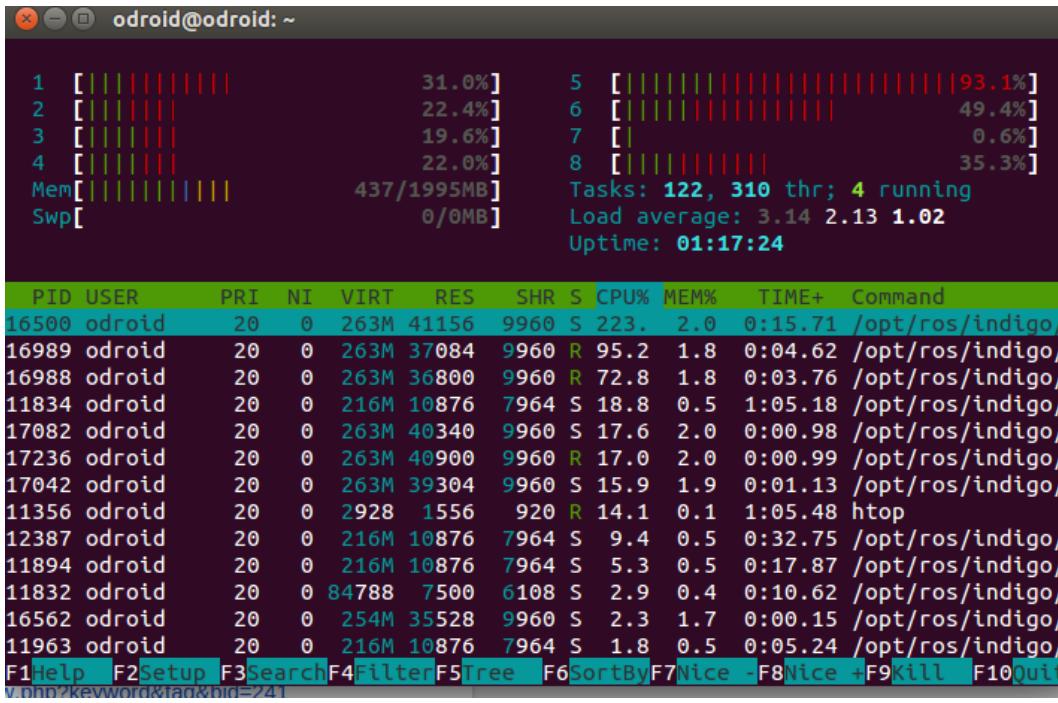


Figure 25 : htop Odroid

Il en résulte qu'à l'exception d'un, les cores ne sont pas utilisés à leur maximum. Il reste donc de la puissance de calcul en réserve.

#### 4. Conclusion

Ainsi nous avons pu observer que les performances de la Odroid nous permettent de nous passer du PC Turtlebot avec efficacité. Toutes les fonctionnalités du Turtlebot, que se soit en terme de navigation ou de vision sont assurées. Le seul bémol constaté est la consommation assez importante de la carte.

## PARTIE 2 : APPLICATION

### I- SCENARIO COMPLET

Afin de répondre à la demande de l'entreprise EXTIA, à savoir avoir un robot ludique capable d'évoluer dans un environnement de type Open-space, nous avons mis en place un scénario complet. L'objectif de ce dernier est d'offrir une application visuelle comprenant plusieurs blocs techniques que nous avions préalablement choisi : une navigation aléatoire, des interactions suite à de la détection de personne et enfin de l'évitement sans contact.

#### 1. Navigation aléatoire

Nous avons tout d'abord créé notre propre navigation autonome. Le robot se déplace dans son environnement jusqu'à ce qu'un de ses bumpers entre en contact avec un obstacle

#### 2. Évitement sans contact

Nous avons mis en place de la détection d'obstacles via la caméra de profondeur, ce qui nous a permis d'améliorer la navigation. Le fonctionnement est similaire à celui avec les bumpers. Les deux méthodes d'évitement (bumpers et caméra) sont actives en même temps. Le bumper sert à détecter les objets bas, la caméra sert pour les obstacles hauts.

#### 3. Interaction et détection de personne

Le robot ainsi autonome, nous voulons le faire interagir avec des personnes. Nous avons décidé que, quand quelqu'un tape dans le bumper central, le robot lève sa caméra pour lancer la détection de visage. Si un visage est détecté, le robot tourne sur lui-même puis s'en va. Sinon il tourne d'un quart de tour dans un sens puis un demi-tour dans l'autre, pour imiter quelqu'un qui ferait "non" de la tête.

De plus, nous avons permis à la base Turtlebot de jouer aléatoirement des morceaux de musique pendant ses phases de navigation autonome.

Ces différents blocs techniques sont décrits plus en détails dans les parties suivantes.

## II - NAVIGATION ALÉATOIRE ET DÉTECTION DE PROFONDEUR

La navigation aléatoire du Turtlebot a été faite très simplement à l'aide de ses bumpers :

- Si le bumper gauche tape un objet, le robot tourne à droite
- Si le bumper droit tape, le robot tourne à gauche
- Si c'est le bumper central, le robot recule un peu et tourne sur un côté.

Le problème est que cette méthode un peu intrusive, le robot devant entrer en contact avec les obstacles pour les éviter. Cela se traduit par des déplacements qui semblent peu "intelligents". De plus, il arrive que les obstacles soient au-dessus des bumpers et ne soient, par conséquent, pas détectés. Le robot fait alors du sur-place et reste coincé sans possibilité de se dégager.

Afin de pallier ce défaut nous nous proposons d'utiliser la caméra de profondeur, pour permettre au robot d'éviter les obstacles sans entrer en contact avec eux.

### 1. Conversion des données

La caméra de profondeur publie des images dans le format image de ROS. Pour la profondeur, nous utiliserons l'image brute, qui est publiée sur 32 bits, avec 1 channel de donnée, sur le topic /camera/depth/image\_raw. Cette donnée représentant la distance de chaque pixel de l'image par rapport à la caméra. Il nous faut donc convertir ces images pour pouvoir les traiter.

Pour ce faire nous utilisons le module CV\_bridge. Cela nous permet de convertir les images ROS au format compatible avec OpenCV. Afin de réaliser l'opération, nous avons créé un nœud qui s'abonne au flux vidéo de profondeur et qui renverra la direction qu'il faut prendre.

Pour faire la conversion proprement dite, on utilise la commande :

```
image_temp = image.imgmsg_to_cv2(data, "passthrough")
```

On voit ici que l'on convertit la "data", c'est à dire l'image provenant de la caméra, en un format compatible avec OpenCV, et le "passthrough" signifie que l'on utilise le même encodage que l'image de départ.

Ensuite on convertit les images pour qu'elles soient exploitable avec le module numpy grâce à l'instruction :

```
imageCV = np.array(image_temp, dtype=np.float32)
```

Une fois que la donnée est convertie, on encadre les valeurs contenues dans les images entre 0 (distance minimale) et 10 m (la distance maximale de la caméra). Cela permet d'éliminer les valeurs erronées qui pourraient ne pas être comprises dans la plage de valeur.

```
imageCV=np.clip(imageCV,0,dist_max)
```

On vient de convertir les images reçues par la caméra de profondeur, on va maintenant s'attacher à traiter cette information.

## 2. Traitement des données

Pour donner des consignes au robot, il faut analyser l'image de profondeur que l'on vient de traiter.



Figure 26 - Caméra de profondeur

On remarquera qu'il n'est pas utile de traiter toute l'image. En effet, on ne traitera pas les bords noirs, qui ne représentent pas une donnée exploitable. On note aussi que notre Turtlebot est plus petit qu'un tabouret (encadré en rouge sur la photo), on ne va donc traiter que la partie basse de l'image, vu que tout ce qui se situe dans la partie supérieure ne constitue pas un obstacle, le robot pouvant passer dessous.

On découpe donc l'image en 3 zones :

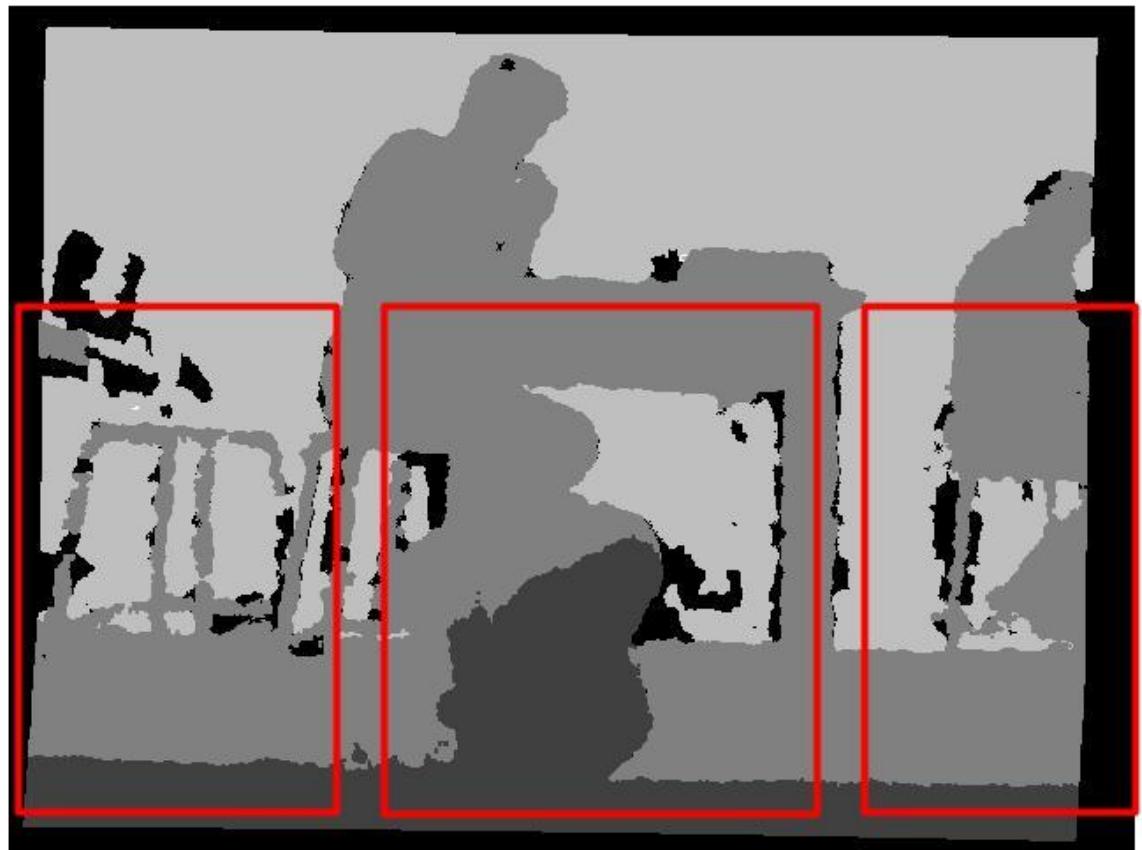


Figure 27 - Découpage de l'image

Les instructions pour le découpage permettent d'obtenir les images suivantes :

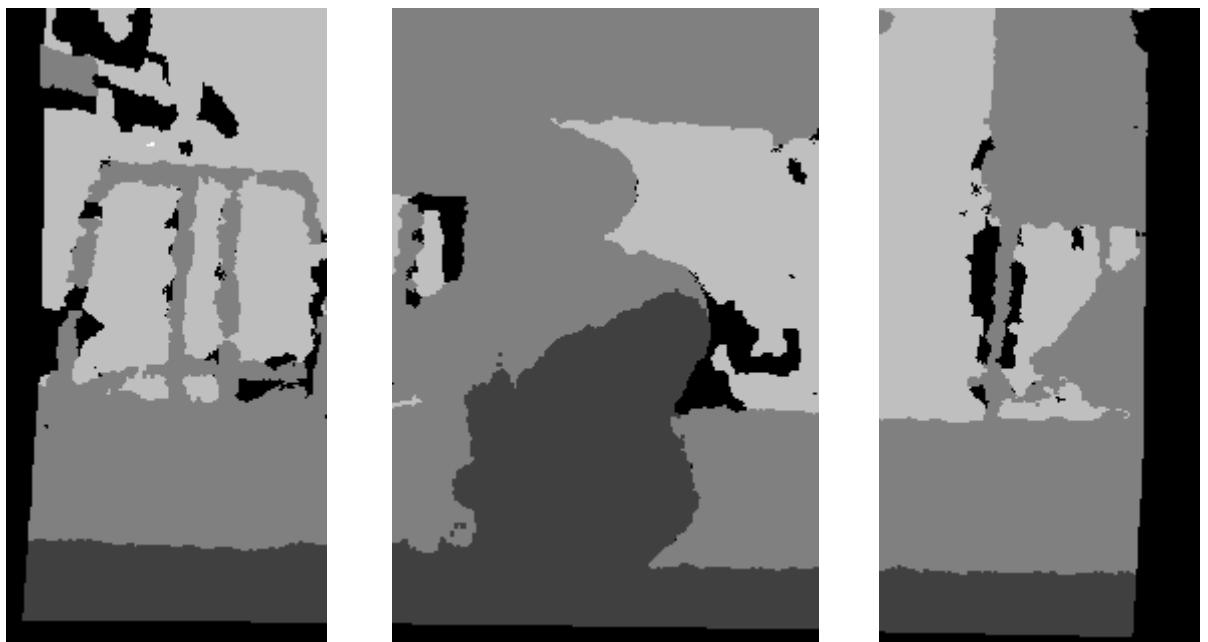


Figure 28 - Image découpée

Une fois que l'on a nos 3 images, on fait la comparaison de chaque pixel par rapport à un seuil (50 cm dans notre cas). Tous les points qui seront à 50 cm ou moins auront la valeur “True”, les autres auront la valeur “False”.

Ensuite on regarde la moyenne de “True” et si plus de 25% du masque est à “True” on considère qu'il y a un obstacle sur ce masque.

On va alors comparer les valeurs renvoyées par chaque masque pour donner une direction au robot.

Quand on détecte un objet à droite et à gauche, il vaut mieux faire demi-tour. On signale donc qu'il y a un obstacle devant.

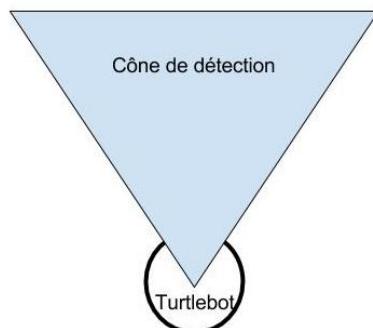
```
if average_left > 0.25 and average_right > 0.25:      # No possibility -> turn back
    position.position = position.FRONT
```

Sinon, l'ordre des “elif” permet de donner des priorités aux actions à effectuer. En effet, le premier cas détecté sera effectué et les suivants ne seront pas pris en compte. Dans notre cas, on a voulu donner plus de priorité à la détection des obstacles dans la zone centrale.

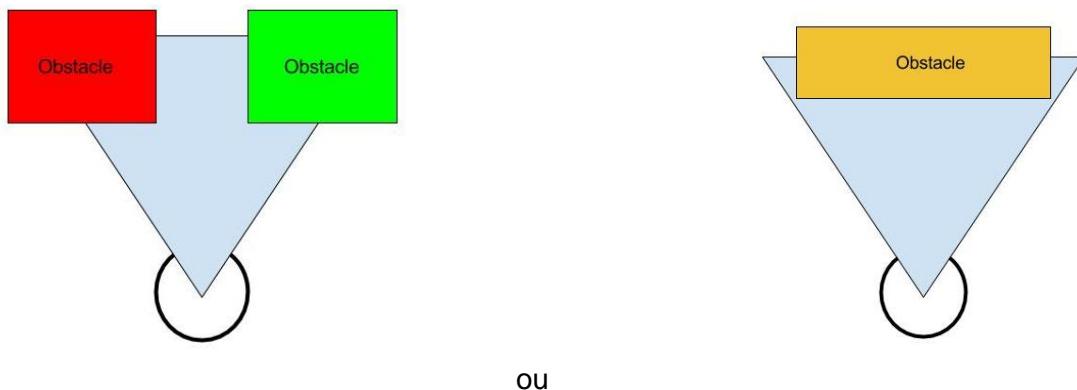
```
elif average_center > 0.25:                      # Priority to the CENTER detection
    position.position = position.CENTER
elif average_left > 0.25:
    position.position = position.LEFT
elif average_right > 0.25:
    position.position = position.RIGHT
else:
    position.position = position.NONE
```

### Schémas explicatifs des directions en fonction des obstacles :

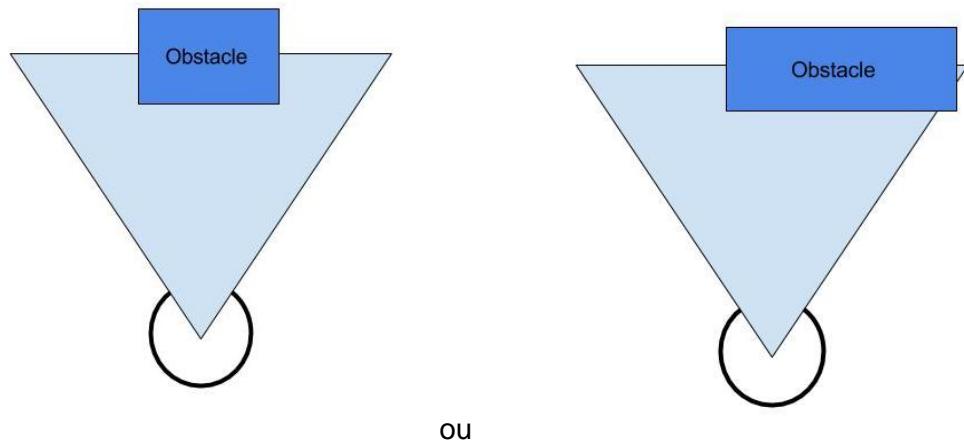
**NONE** : aucun obstacle n'est détecté, aucune action particulière.



**FRONT** : obstacle au centre, notre robot fait demi-tour



**CENTER** : obstacle au centre, notre robot recule et tourne d'un coté



**LEFT** : obstacle sur un côté, notre robot tourne du côté opposé

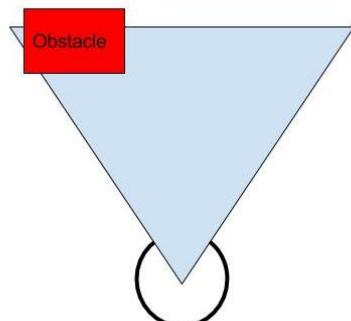


Figure 29 - Schémas de position des obstacles

Les informations relatives à la position de l'obstacle sont publiées sur le nœud et donc si on s'abonne à notre topic, on peut faire de la détection d'obstacle à distance.

### 3. Problèmes rencontrés et améliorations

Nous nous sommes rendus compte que, quand l'image couleur (pour la détection des visages) et l'image de profondeur sont publiées, on ne peut récupérer que l'image couleur. Pensant que cela venait du temps de traitement des images, nous avons réduit la résolution des images.

Pour faire cette opération nous utilisons le client de reconfiguration dynamique. D'abord on donne le chemin du nœud à reconfigurer :

```
client=dynamic_reconfigure.client.Client("/camera/driver")
```

Puis on met à jour la valeur de la variable qui nous intéresse. Dans notre cas, c'est depth\_mode que l'on met à 8 pour avoir le format QVGA.

```
client.update_configuration({"depth_mode":8, "data_skip":0})
```

Après cette opération nous n'observons plus de ralentissements sur les flux vidéo.

Étant donné que l'on détecte les obstacles à partir d'un certain seuil, nous nous proposons également de pouvoir reconfigurer ce seuil de manière dynamique.

Pour ce faire nous avons ajouté un fichier .cfg pour prendre en compte les modifications de réglage du seuil de détection d'obstacles. Pour mettre en place des dynamics reconfigure pour des nœuds, vous pouvez consulter le tutoriel dédié à cette tâche (TUTORIEL 8 : Utilisation de dynamic\_reconfigure).

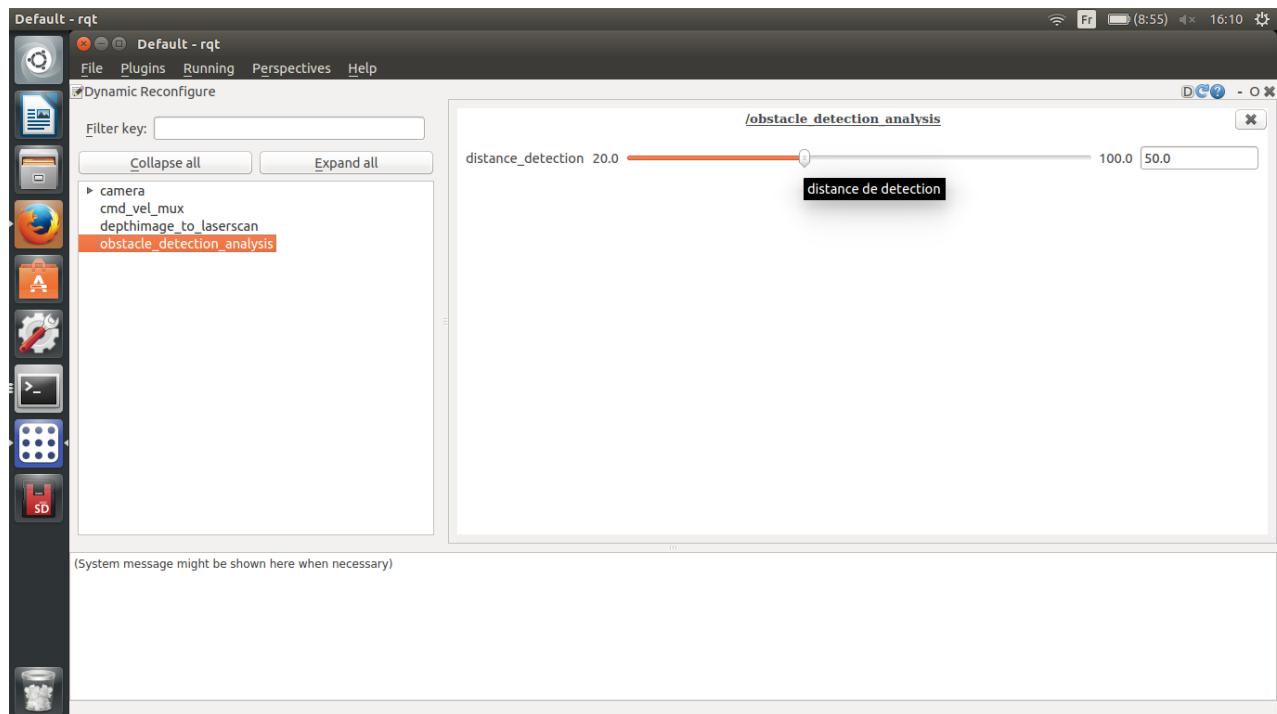


Figure 30 - Reconfiguration dynamique du seuil de détection

Le diagramme complet des nœuds pour la détection de profondeur est le suivant :

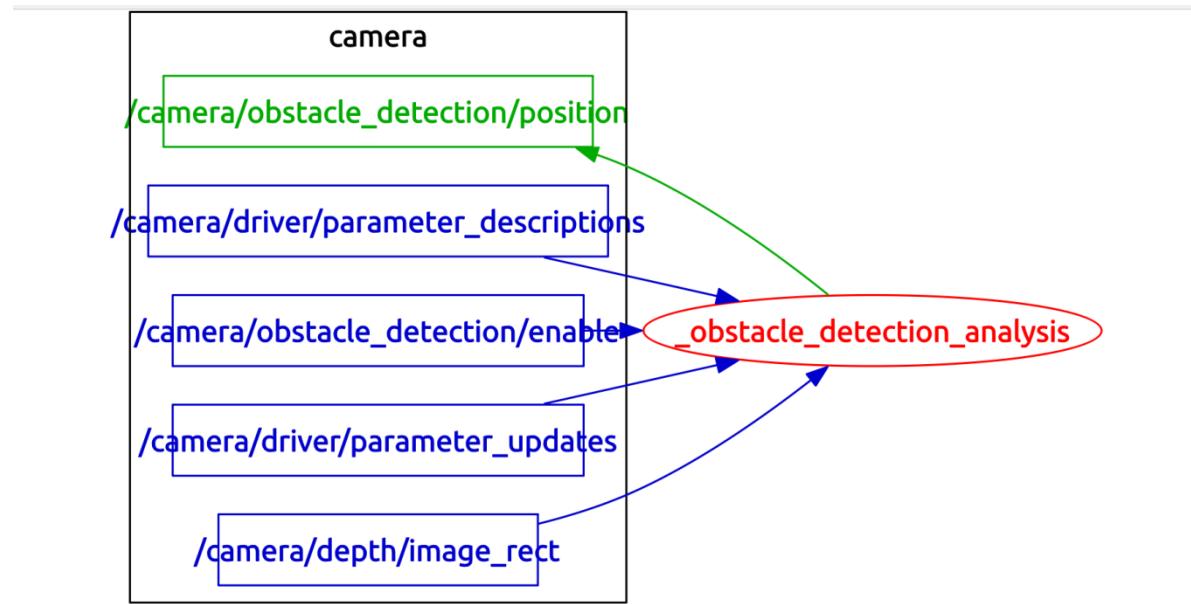


Figure 31 - Node graph detection

### III – CAMÉRA ORIENTABLE

#### 1. Caractéristiques

Nous avons constaté lors de nos différentes expérimentations que pour détecter des visages, le placement de la caméra n'était pas idéal. En effet, pour ce faire, il fallait que la personne se trouve à plus de 2m50 de la caméra.



Figure 32 - Position initiale de la caméra

Nous avons alors envisagé deux solutions : la surélever ou bien l'orienter avec un angle qui nous permettrait de détecter des visages à une distance plus courte. Cependant, cette deuxième solution aurait eu pour conséquence de rendre la caméra inutilisable pour de la cartographie.

C'est pourquoi nous avons décidé de mettre en place une caméra pouvant être orientée de manière logicielle en fonction de nos besoins. Nous avons donc opté pour un servomoteur et avons modélisé sous Solidworks les quelques pièces nécessaires à cette modification.

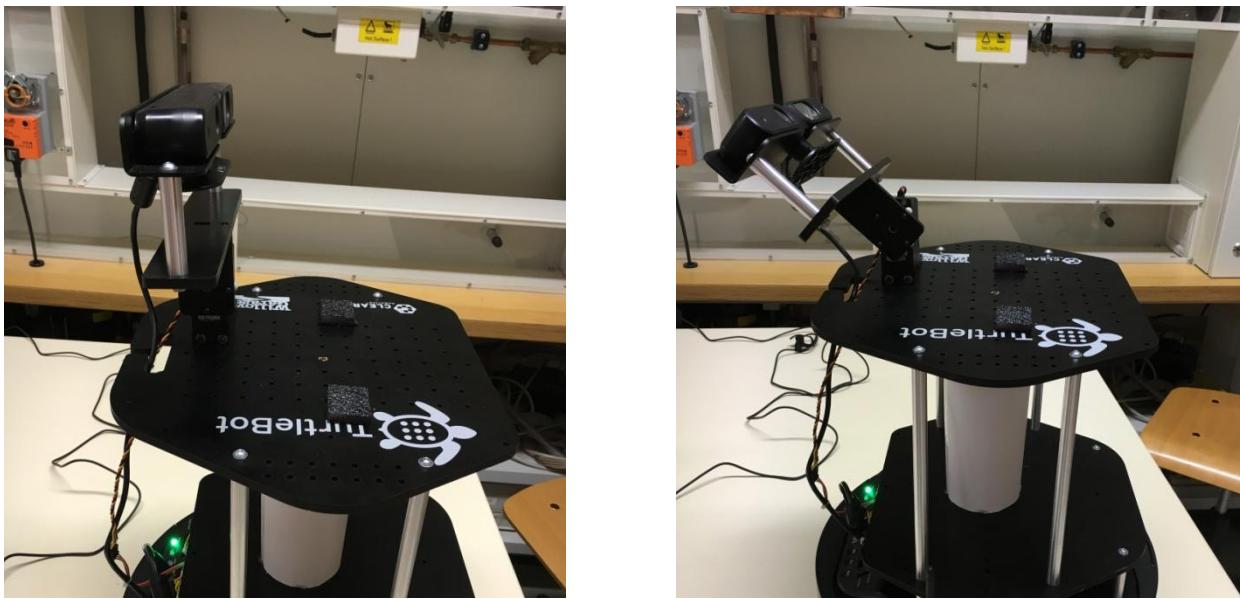


Figure 33 - Caméra sur servomoteur orientable

Un servomoteur se contrôle par une PWM de fréquence 50 Hz (période de 20 ms). En théorie la valeur de la PWM à l'état haut doit varier entre 1 à 2 ms, en pratique cela peut varier un peu plus selon les servomoteurs. Un état haut de 1.5 ms correspond à la position milieu du servomoteur. 1 et 2 ms correspondent respectivement à +/- 90° par rapport à ce point.

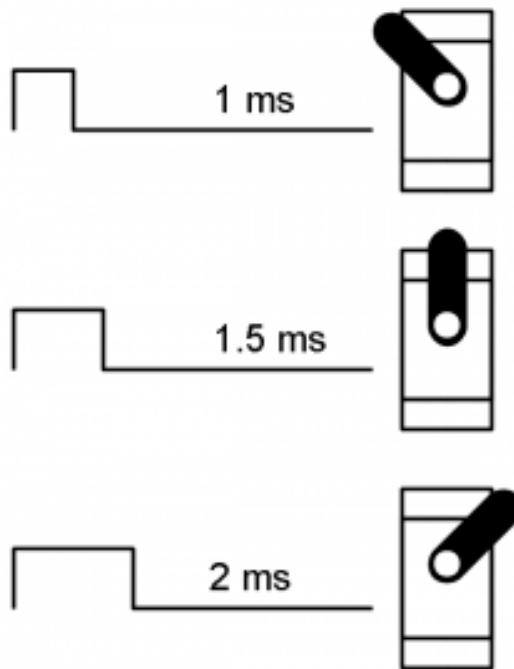


Figure 34 - Positions possibles du servo moteur

**Un servomoteur a besoin de 3 fils pour fonctionner :**

- V+ : +5V
- V- : masse
- PWM de commande

En voulant mettre en place le servomoteur, nous nous sommes heurtés à un problème considérable ; la seule sortie PWM disponible sur notre carte Odroid est utilisée pour faire fonctionner le ventilateur. La base Kobuki quant à elle n'en dispose pas.

Nous avons alors envisagé d'en simuler une logiciellement sur la Odroid. Mais après quelques recherches sur le forum de cette dernière, le fabricant ne recommande pas cette pratique et soutient même qu'elle ne pourra pas fonctionner à cause de la manière dont leur DMA gère les GPIO de la carte.

Finalement, nous avons décidé d'utiliser une petite carte électronique avec un microcontrôleur pour générer une PWM. Notre caméra n'ayant besoin que de deux positions (mode déplacement et mode détection de visage), nous avons utilisé un câble de commande pour dire à notre carte laquelle des positions nous souhaitons.

## 2. Programmation de la carte

### Mise en place de la carte :

Afin de ne pas perdre de temps à créer notre propre carte électronique, et parce qu'en faire une à la main aurait été tout aussi encombrant voire plus, nous avons décidé de réutiliser une carte électronique à notre disposition. Nous avons soudé uniquement les composants nécessaires à notre utilisation et apporté les modifications indispensables.

Notre carte embarque un DSPIC 33FJ128MC804 de chez Microchip. Le programme quant à lui est plutôt simple : nous avons une entrée connectée à une sortie digitale de type drain ouvert à la base Kobuki. Nous scrutons en permanence la valeur de cette sortie. Si celle-ci vaut 1, nous envoyons le rapport cyclique nécessaire variant entre 0 et 5V pour incliner la caméra de 45° sinon nous laissons la caméra à sa position initiale.

Nous avons optimisé la consommation de la carte en désactivant tous les modules que nous n'utilisions pas. Sa consommation est ainsi très négligeable puisqu'elle est de 20 mA.

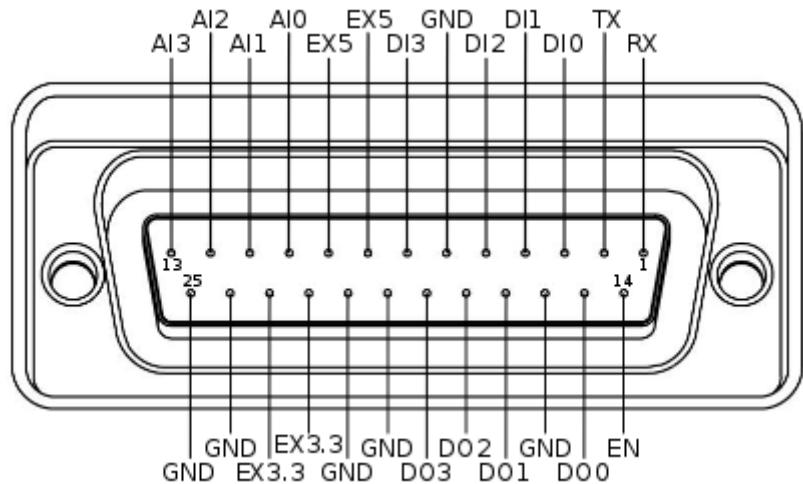


Figure 35 - Sorties de la base Kobuki

Nous récupérons la sortie D00 du connecteur d'expansion de la base Kobuki puis nous le connectons à un microcontrôleur de la manière suivante :

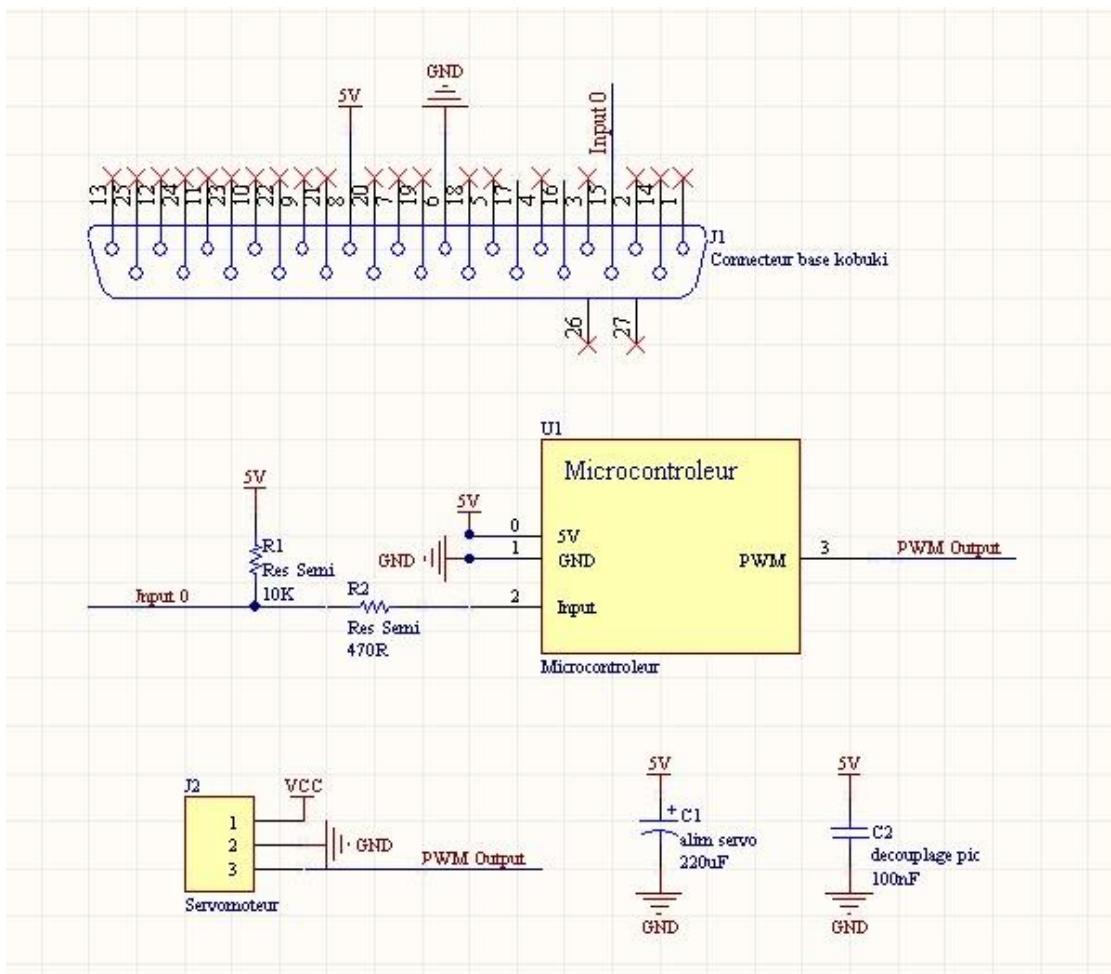


Figure 36 - Schéma de la carte

La sortie étant en drain ouvert, elle est soit forcée à l'état bas, soit en état de haute impédance. Il nous faut donc une résistance de pull-up afin de différencier les deux états sur notre microcontrôleur.

### Nœuds et topics :

Toutes les sorties de la base Kobuki sont déjà implémentées sous ROS, il suffit donc de publier sur un topic existant pour demander au servomoteur de s'orienter.

Pour rappel :

ROS fonctionne sur un principe de nœuds et de topics. Un nœud est un processus qui s'exécute, il peut également publier ou s'abonner à des topics ROS. Un topic ROS permet d'échanger des données entre plusieurs processus, ces derniers s'y abonnant, et si une nouvelle valeur y est publiée alors le processus (ou nœud) la recevra instantanément. Les processus peuvent également décider d'y publier des informations. Le type d'informations échangées sur les topics est défini par le programmeur lors de la création d'un package ROS.

Il faut publier sur le topic `/mobile_base/commands/digital_output` pour commander les GPIO de la base. Ce topic prend des messages qui contiennent deux vecteurs : le premier, "mask" permet à l'aide de booléens de définir quels sont les pins dont on veut modifier l'état. Le second, "values" définit les états des pins souhaités.

Cependant il nous semblait important que tous les nœuds ROS puissent avoir accès à l'information de l'orientation de la caméra, afin de traiter les données de manière cohérente. Nous avons donc créé un nouveau nœud avec des topics associés.

Notre nœud souscrit à un topic : `/camera/orientation/request`. Lorsqu'un autre nœud publie une valeur sur ce topic alors il publie lui-même sur deux topics :

- Le premier pour envoyer la commande à la carte qui génère la PWM
- Le deuxième avec la valeur de l'orientation de la caméra pour que tous les nœuds puissent la prendre en compte

Pour demander au servo moteur d'orienter la caméra il faut utiliser l'instruction :

```
rostopic pub /camera/orientation/request  
turtlebot_scenario/OrientationRequest "value: true"
```

## 3. Mécanique

Les pièces pour surélever la caméra et l'orienter ont été modélisées sous Solidworks, puis réalisées avec un CNC (Computer Numerical Control).

La caméra et le servomoteur sont représentés en noir. Les barres métalliques grises sont celles d'origine du Turtlebot. Les plaques créées via Solidworks sont représentées en bleu.

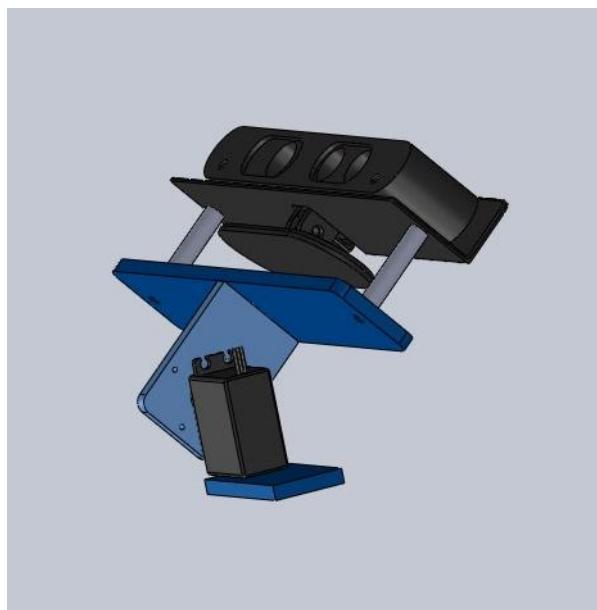


Figure 37 - Pièces Solidworks en mouvement

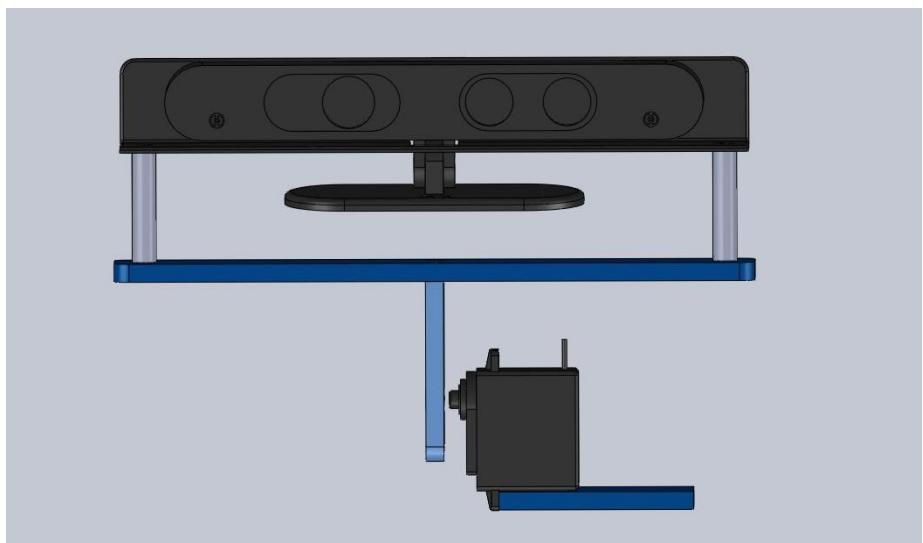


Figure 38 - Pièces Solidworks de face

## IV – DÉTECTION DE PERSONNE

### 1. Cob People Detection

Dans un premier temps, nous avons cherché des packages préétablis permettant de détecter, puis de reconnaître des personnes. Une veille nous a permis de trouver le package suivant plutôt prometteur : [http://wiki.ros.org/cob\\_people\\_detection](http://wiki.ros.org/cob_people_detection)

Complètement intégré à ROS, il utilise à la fois la caméra RGB pour détecter les visages et la caméra de profondeur afin d'éliminer les faux positifs et ainsi détecter un visage même s'il n'est pas complètement face à la caméra.

Nous avons utilisé ce package sur nos ordinateurs, cependant, au moment de l'intégrer sur la Odroid, nous nous sommes rendus compte qu'il utilisait des bibliothèques openni2 non compatibles et non distribuées pour les core ARM. Nous avons alors été dans l'impossibilité de l'utiliser sur la Odroid et avons dû trouver une autre solution.

### 2. Facedetector de Vicoslab

Facedetector est un second package de détection de visage adapté à ROS. Il fonctionne grâce à opencv2, il utilise la caméra RGB et un filtre Haar pour détecter un visage.

Il est disponible à ce lien : [https://github.com/vicoslab/vicos\\_ros](https://github.com/vicoslab/vicos_ros)

Les caractéristiques pseudo-Haar sont très utilisées en vision par ordinateur notamment pour détecter des visages sur un flux vidéo en temps réel. Un tel filtre permet d'indiquer les frontières entre les zones sombres et les zones claires et ainsi de détecter un visage. Les calculs sont très rapides ce qui en fait une caractéristique idéale pour un tel type d'application.

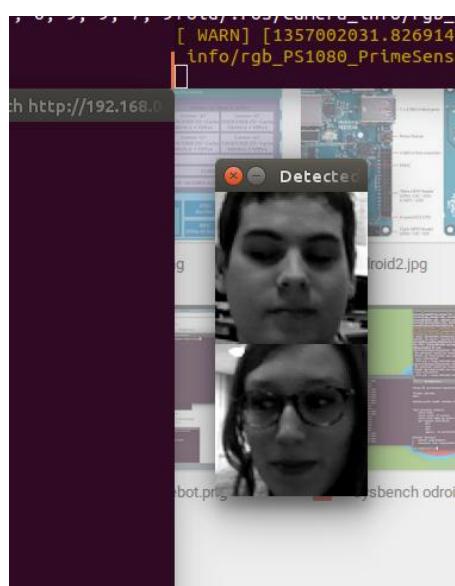


Figure 39 - Détection de visages

Lorsqu'un visage est détecté il publie l'image du visage sur un topic ainsi que les coordonnées de celui-ci. Nous pouvons alors nous abonner à ce topic et réceptionner les données pour savoir à tout moment si un visage a été détecté.

## V – JOUER DE LA MUSIQUE SUR LE TURTLEBOT

### 1. Introduction

La base Kobuki peut jouer des sons prédéfinis à savoir :

- ON : son lors de la mise en marche
- OFF : son lors de l'arrêt
- RECHARGE : son quand on branche ou débranche le chargeur
- BUTTON : son quand on appuie sur les boutons
- ERROR
- CLEANINGSTART
- CLEANINGEND

Il nous est alors apparu logique qu'elle puisse jouer d'autres sons que ceux déjà existants, nous avons alors recherché ce qu'il était possible de faire.

Le Github de Yujin, la société commercialisant la base Kobuki du Turtlebot nous a apporté la réponse. Dans leur documentation Doxygen ils expliquent le fonctionnement du protocole de la liaison série permettant d'interagir avec la base.

<https://yujinrobot.github.io/kobuki/doxygen/enAppendixProtocolSpecification.html>

#### Structure of Bytestream

A bytestream can be divided into 4 fields; Headers, Length, Payload and Checksum.

Name	Header 0	Header 1	Length	Payload	Checksum
Size	1 Byte	1 Byte	1 Byte	N Bytes	1 Byte
Description	0xAA (Fixed)	0x55 (Fixed)	Size of payload in bytes	Described below	XOR'ed value of every bytes of bytesream except headers

## Structure Of Sub-Payloads

Sub-payload can be divided into three parts; Header, Length and Data.

Name	Header	Length	Data
Size	1 Byte	1 Byte	N Byte(s)
Description	Predefined Identifier	Size of data in byte(s)	Described below

### Sound

Play custom sounds with note and duration.

	Name	Size	Value	Value in Hex	Description
Header	Identifier	1	3	0x03	Fixed
Length	Size of data field	1	3	0x03	Fixed
Data	Note	2			$1/(f \cdot a)$ , where $f$ is frequency of sound in Hz, and $a$ is 0.00000275
	Duration	1			Duration of playing note in milli-seconds

#### Warning:

This command is implemented on the kobuki with firmware, but not implemented yet on driver software(kobuki\_driver).

Figure 40 - Protocole de la liaison série

Nous avons alors compris qu'il existait une fonctionnalité permettant de jouer une note définie par sa fréquence et sa durée. Seulement cette fonctionnalité n'a pas été implémentée dans le driver, ce qui signifie qu'elle n'est donc pas disponible dans ROS. Il nous faut donc utiliser notre propre liaison série pour envoyer nous même les messages à la base Kobuki.

Dès le début, ce procédé montre une faiblesse importante ; en effet si nous envoyons des données sur la liaison série en même temps que ROS il y aura alors "collision" des données et les messages ne seront pas reçus correctement.

Aucune solution concluante n'ayant pu être trouvée, nous nous contenterons donc pour la suite de jouer des sons personnalisés uniquement lorsque la base sera à l'arrêt. Les fonctionnalités décrites dans les prochaines parties sont accessibles dans notre package ros sound\_publisher.

Pour bien comprendre le fonctionnement d'un package et en créer un, vous pouvez vous rendre dans notre tutoriel 6 : Création d'un package ROS qui explique pas à pas les étapes de la création d'un package.

## 2. Jouer une note via sa fréquence

Comme nous le montre l'étude des trames séries à envoyer, nous devons, dans un premier temps, créer un fichier qui nous permettra d'envoyer les trames séries suivantes:

- 0xAA
- 0x55
- 0x05
- 0x03
- 0x03
- LSB note
- MSB note
- Duration
- Checksum

Nous avons pour cela créé notre nœud ROS dans le fichier `sound_pub.py`, à l'intérieur duquel nous avons mis en place l'envoi des trames série. Ce nœud s'appelle `sound_pub`. De plus, il souscrit à un topic `sound/tones` qui envoie des variables que nous avons créées de type `TonesArray`. Un `TonesArray` est en fait un tableau de `Tones`, des variables elles-mêmes définies par une fréquence sur un float de 32 bits et une durée sur un entier de 16 bits.

La publication d'un tableau de notes sur ce topic entraîne l'appel d'une fonction `callback` qui appelle à son tour et de manière répétée la fonction du driver pour publier une note. Cette répétition est nécessaire pour que notre note soit jouée suffisamment longtemps. En effet la durée maximum d'une note jouée par la base est d'environ une seconde. Le paramètre durée de la variable `Tone` peut lui, durer plusieurs dizaines de secondes et s'exprime en ms.

Entre chaque changement de note nous attendons quelque millisecondes à l'aide d'un `rospy.sleep()` pour que les notes se détachent bien les unes des autres comme nous le ferions en jouant d'un instrument de musique.

Afin de rendre ce temps d'attente modulable facilement pour s'adapter à toutes les mélodies, nous l'avons intégré dans une variable dynamique comme dans le package précédent. Cette variable est accessible sous le nœud `/sound_publisher` avec le nom `waiting time` et contient le temps à attendre entre chaque note en ms.

Pour jouer une série de notes, il suffit donc de les publier sur le topic `/sound/tones` et qu'elles soient de la forme `TonesArray` c'est à dire `[Tones(f1, d1), Tones(f2, d2), Tones(f3, d3), ...]`

### 3. Jouer une note via son appellation

Pour jouer une note via son appellation (c'est à dire : do, ré, mi, fa, sol, ...) nous avons créé un second nœud ROS "musical\_node\_publisher" dans le fichier `musical_note_pub.py`.

Ce nœud souscrit au topic `/sound/musical_note` qui prend comme paramètre un `MusicalTonesArray`. De la même manière que précédemment, un `MusicalTonesArray` est un

tableau de MusicalTones qui sont définis dans le fichier MusicalTones.msg du dossier msg du package sound\_publisher.

Ces MusicalTones sont définis par une chaîne de caractères composée du nom de la note en minuscule sans accent (do, ré, mi, fa, ..) suivi s'il y a lieu d'un '#' ou d'un b pour bémol (do#, reb, re#, ...). Nous avons également ajouté l'appellation anglaise des notes c'est à dire (A, B, Bb, C#, ..) ainsi que la note 'silence' pour correspondre aux partitions. Le deuxième paramètre correspond au numéro de l'octave de la note et enfin le dernier à la durée de la note.

Une fois la chaîne de notes reçue, elle est convertie note par note pour être de la forme (fréquence, durée), puis le tableau de notes en fréquence est publiée sur le topic /sound/tones afin qu'il n'y ait qu'un seul nœud ROS qui envoie des messages sur la liaison série.

En conclusion, pour jouer des notes définies par leur nom il faut publier sur le topic /sound/musical\_note un tableau de MusicalTones c'est à dire [MusicalTones(note1, octave1, durée1), MusicalTones(n2, o2, d2), ...]

#### 4. Jouer un morceau de musique

Enfin, nous avons voulu simplifier le code en évitant d'écrire à chaque fois toutes les notes des morceaux joués dans notre scénario. Nous avons donc pris l'initiative de créer un troisième nœud ROS qui se charge de publier les chansons pour nous. Ce nœud s'appelle song\_publisher et se trouve dans le fichier **song\_pub.py**.

Il écoute sur le topic /sound/play\_song et attend un message de type SongTitle. Ce message est défini comme d'habitude dans le dossier msg du package sous le nom SongTitle.msg.

Il est défini de la manière suivante :

```
uint8 Star_Wars          = 0
uint8 Indiana_Jones       = 1
uint8 Au_Clair_De_La_Lune = 2
uint8 La_Marseillaise     = 3
uint8 Lavender_Town       = 4

uint8 song
```

Les premiers uint8 sont en fait des defines pour stocker le nom des chansons et le uint8 song est la variable qui contiendra le son à jouer.

Une fois un message de type SongTitle envoyé sur le topic /sound/play\_song, le nœud envoie le tableau de notes associés dans le script sur les autres topics. Il peut au choix publier des partitions écrites en (fréquence, durée) ou en (note, octave, durée) sur les topics associés, c'est à dire soit /sound/tones ou /sound/musical\_note

La méthode détaillée pour ajouter de nouveaux morceaux de musiques est décrite dans notre tutoriel 7 : utilisation du package sound\_publisher.

La vidéo suivante montre le Turtlebot jouant un morceau que nous lui avons implémenté :

[https://www.youtube.com/watch?v=8qA3vHog\\_MI](https://www.youtube.com/watch?v=8qA3vHog_MI)

## 5. Conclusion

En définitive, avec le package sound\_publisher il est possible de jouer une courte série de notes définies en (fréquence, durée) en publiant sur /sound/tones, ou bien une courte série de notes définies en (nom, octave, durée) en publiant sur /sound/musical\_note, ou encore de publier un long morceau de musique en publiant sur /sound/play\_song.

Dans tous les cas, on ne peut le faire que lorsque le robot est à l'arrêt. Il faut également attendre la fin du morceau pour le faire redémarrer sans quoi, des collisions sur le bus série vont apparaître, ce qui se soldera par des déplacements du robot très saccadés et une musique qui ne fonctionnera pas.

Le diagramme complet des nœuds pour le son est le suivant :

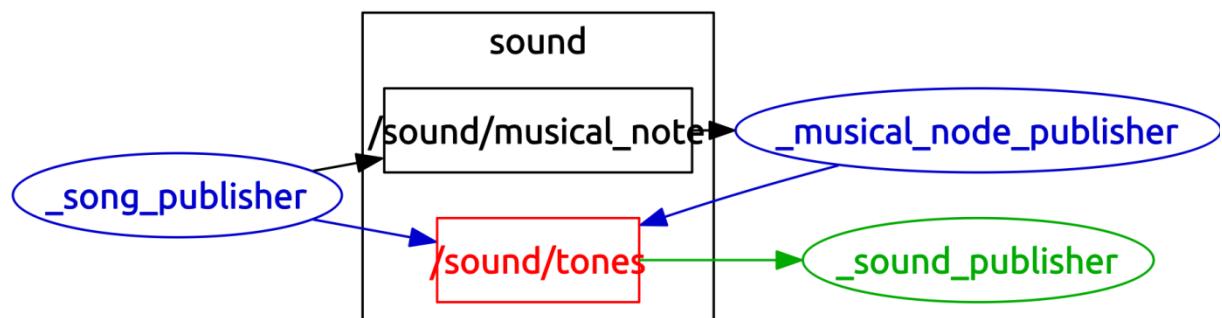


Figure 41 - Node graph son

On peut y voir les différentes interactions entre nos nœuds. Le nœud publiant les morceaux de musique peut soit publier sur le topic /sound/musical\_note, soit sur le topic /sound/tones.

Le nœud musical\_node\_publisher lui écoute le topic /sound/musical\_note et republie les notes converties sur le topic /sound/tones.

Enfin le nœud sound\_publisher écoute sur le topic /sound/tones afin d'envoyer les commandes via la liaison série.

## VI – APPLICATION COMPLÈTE

Ainsi, ces différents blocs techniques permettent de mettre en place une application complète pour le Turtlebot. Il est maintenant capable de se déplacer de manière autonome et évitant les obstacles, mais également d'interagir avec les gens qu'il rencontre et de jouer de la musique.

Le graphe des nœuds pour le scénario complet est le suivant :

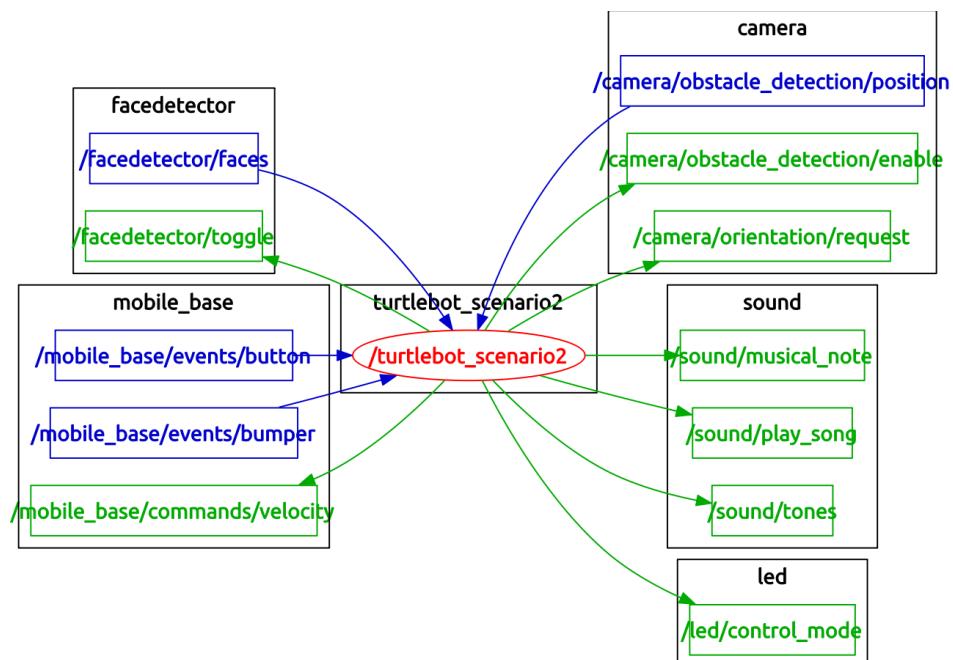


Figure 42 - Node graph scenario

## PARTIE 3 : GÉNÉRALITÉS SUR LE PROJET

### I - GESTION DU STOCKAGE

Afin de faciliter la gestion du stockage au sein de notre trinôme, nous avons mis en place plusieurs solutions.

La première a été de créer une adresse email commune afin de regrouper tous nos échanges concernant le projet. De plus nous avons créé un drive Google nous permettant de partager des fichiers et d'y avoir accès en ligne à tout moment. Cela nous a également permis de créer des documents Word en ligne qui offrent la possibilité de travailler à plusieurs sur le même fichier, synonyme de gain de temps pour la rédaction du rapport. Cela a également permis à notre encadrant de projet de pouvoir suivre l'avancement de nos rapports au jour le jour et de nous faire des remarques en temps réel.

De plus, un compte Youtube a été créé pour pouvoir partager les vidéos d'avancement du projet. Le lien de notre chaîne Youtube est le suivant :

[https://www.youtube.com/channel/UC6WHISUiSzchMfl\\_r759TMw](https://www.youtube.com/channel/UC6WHISUiSzchMfl_r759TMw)

Enfin, un compte Github a été mis en place. Il permet de conserver notre code et également de le mettre à disposition de toute personne souhaitant y accéder. Le lien de notre Github est le suivant: <https://github.com/TurtlebotMec3>

**Vous y trouverez notamment deux repository :**

- package\_ros
- example\_package\_ros

Le premier contient tous les packages que nous avons créés et permet ainsi à toute personne d'utiliser les mêmes fonctionnalités que nous sur son Turtlebot.

Le second permet d'appuyer notre tutoriel sur la création d'un package ROS.

## II – PROBLEMES RENCONTRÉS ET SOLUTIONS

Nous avons rencontré quelques problèmes au cours de notre projet. La majorité était des problèmes mineurs qui se sont solutionnés rapidement par nous-mêmes ou grâce aux réunions hebdomadaires avec M. GEORGE.

Cependant quelques éléments nous ont posés plus de difficultés :

- **Les limitations de la BeagleBone.**

En effet, comme vu précédemment, la BeagleBone Black présente de nombreuses limites et ne permettait pas d'utiliser le Turtlebot de manière optimale.

Deux solutions se présentaient à nous ; limiter les applications réalisées dans le cadre de notre projet ou bien échanger notre carte contre une plus performante. La seconde solution a été retenue et après de nombreuses recherches et d'études comparatives, nous nous sommes tournés vers une Odroid-XU4.

- **La détection et la reconnaissance de personnes**

Lors de la seconde partie du projet nous avons mis en place une librairie de détection et reconnaissance faciale : Cob People Detection. Cette dernière fonctionnait parfaitement sur un ordinateur via le Turtlebot. Cependant en essayant de l'intégrer à la Odroid, nous nous sommes rendus compte qu'elle n'était pas compatible avec les cores ARM. Face à ce contretemps nous avons dû trouver une librairie compatible.

- **Utilisation des caméras simultanément**

Le scénario complet de notre projet nécessite l'utilisation simultanée des caméras de profondeur et RGB. Cette utilisation nous a posé problème. En effet nous avons observé que nous n'arrivions pas toujours à obtenir les images des deux caméras simultanément. Afin d'y pallier, nous avons dû faire des concessions sur la qualité de l'image et nous avons modifié les paramètres de la caméra de profondeur pour qu'elle soit en QVGA.

- **Jouer de la musique sur le Turtlebot**

Un des points que nous souhaitions mettre en place sur notre scénario était de jouer de la musique tout en se déplaçant. Aucune solution n'ayant été trouvée, nous avons envoyé un email à Yujin Robot, créateur de la base Kobuki pour savoir si une solution était envisageable. Ce mail est cependant resté sans réponse.

- **Lumière et caméra**

Un problème important décelé au cours de ce projet est l'effet de la lumière du soleil sur la caméra de profondeur. En effet, trop de lumière infrarouge peut fausser l'estimation de la caméra puisque celle-ci est basée sur cette technologie et donc laisser supposer qu'un objet est proche alors que ce n'est pas le cas. Cela pose de nombreux problèmes lors de l'évitement d'obstacles ; le robot pense qu'un obstacle se trouve devant lui alors que ce n'est pas le cas. La seule solution est de travailler en intérieur et assez loin des fenêtres.

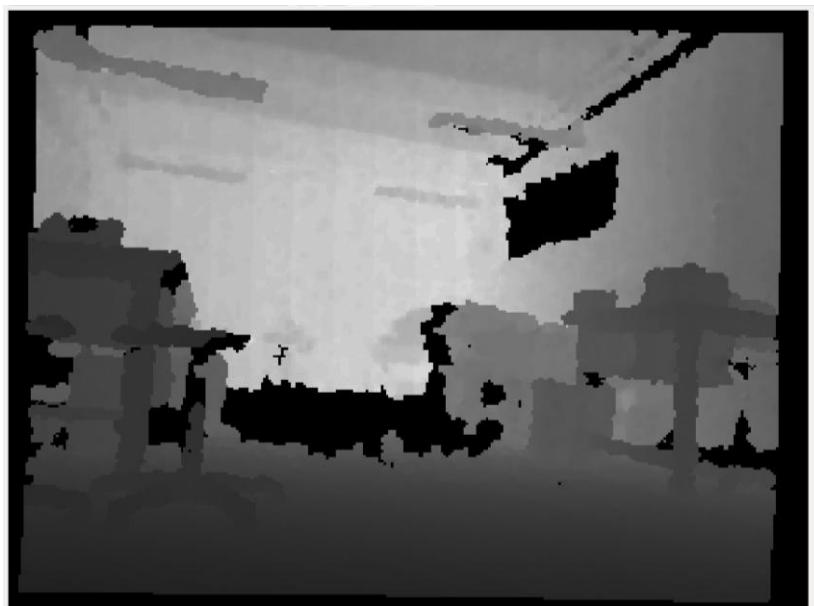


Figure 43 - Image de la caméra de profondeur dans le laboratoire d'énergie

On peut ici voir l'effet de la lumière du soleil sur la caméra, la grosse tache noire en haut de l'image est en fait une fenêtre qui se trouve à plusieurs mètres du robot. Elle devrait se trouver dans les mêmes tons gris que le reste de l'image.

### III - ÉVOLUTIONS POSSIBLES :

Notre projet est terminé et fonctionnel, cependant de nombreuses évolutions sont envisageables. Nous avons listé plusieurs points qui pourraient être considérés :

- **Entreprise :**

Bien que le scénario créé pour le Turtlebot soit déjà adapté à une utilisation en entreprise, de nombreuses améliorations sont possibles. Le Turtlebot pourrait proposer d'autres applications telles que de la maintenance avec notamment la vérification visuelle (ampoules, état du sol, ...), de la mesure du bruit ambiant, une poubelle interactive...

- **Sécurité :**

Le Turtlebot pourrait prendre le rôle d'un robot veilleur de nuit, c'est à dire se déplacer en autonomie dans des locaux, avoir un rôle dissuasif (bruit, lumière), être capable de détecter des situations inhabituelles, être commandé à distance le cas échéant. Le robot possède une caméra infrarouge, un avantage important pour ce type d'utilisation.

- **Santé :**

De multiples applications liées à la santé pourraient être développées sur le Turtlebot comme par exemple un système d'alerte en cas de chute, un pilulier intelligent...

## CONCLUSION :

Ainsi l'objectif du projet a été atteint. Après un essai infructueux avec la BeagleBone Black nous avons réussi à remplacer le PC originel commandant le Turtlebot par une carte électronique : la Odroid-XU4. Les résultats obtenus grâce à cette carte sont satisfaisants et nous permettent de nous passer du PC Turtlebot avec efficacité. Toutes les fonctionnalités du Turtlebot sont assurées, autant en terme de navigation que de vision.

De plus, nous avons mis en place un scénario permettant au Turtlebot d'interagir au sein d'un environnement de type Open-space. En effet, ce dernier est capable de se déplacer en autonomie tout en évitant les obstacles, de jouer de la musique et de détecter les êtres humains qu'il rencontre.

La finalité du projet a été atteinte ; la mise en place de la carte a permis de diminuer la masse et le coût du Turtlebot tout en libérant la place occupée par le PC. Enfin, le scénario complet permet une application ludique du robot.

D'un point de vue plus personnel, ce projet nous a permis de mettre en application de nombreuses connaissances apprises en cours, mais également d'en développer beaucoup de nouvelles, notamment sur les systèmes embarqués, Ubuntu et ROS. Nous avons été confrontés à de nombreux aspects de l'ingénierie, que ce soit du côté matériel ou logiciel. Un tel projet nous permet également d'apprendre à travailler en groupe sur une mission à long terme.

## ANNEXES

### BIBLIOGRAPHIE

#### INTERNET

##### ASUS XTION PRO LIVE :

[https://www.asus.com/3D-Sensor/Xtion\\_PRO\\_LIVE](https://www.asus.com/3D-Sensor/Xtion_PRO_LIVE)

##### VEILLE TECHNOLOGIQUE :

[https://www.youtube.com/channel/UC3SGW\\_kv6Z0ZRexrVlm3ukw/videos/](https://www.youtube.com/channel/UC3SGW_kv6Z0ZRexrVlm3ukw/videos/)  
<http://answers.ros.org/question/146926/beaglebone-and-turtlebot/>  
<http://answers.ros.org/question/10387/how-to-run-ros-on-beagleboard-effectively/>  
<http://answers.ros.org/question/54761/raspberry-pi-and-turtlebot/>  
<http://answers.ros.org/question/120816/running-a-kinect-pandaboard-and-getting-rviz-on-a-remote/>

##### COMPARATIFS CARTES :

[http://www.hardkernel.com/main/products/prdt\\_info.php?g\\_code=G141578608433&tab\\_idx=1](http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141578608433&tab_idx=1)  
(comparatif Odroid/raspberry pi2)

##### TURTLEBOT ET ROS :

<http://www.ros.org/>  
<http://learn.turtlebot.com/2015/02/10/99/>

##### KOBUKI :

<https://github.com/yujinrobot/kobuki>  
<http://kobuki.yujinrobot.com/home-en/about/>  
<http://yujinrobot.github.io/kobuki/doxygen/enAppendixProtocolSpecification.html>  
<http://kobuki.yujinrobot.com/wiki/connectors/>

##### TUTORIELS :

<http://elinux.org/BeagleBoardUbuntu>  
<http://wiki.ros.org/indigo/Installation/UbuntuARM>  
[http://com.odroid.com/sigong/nf\\_file\\_board/nfile\\_board\\_view.php?keyword&tag&bid=241](http://com.odroid.com/sigong/nf_file_board/nfile_board_view.php?keyword&tag&bid=241)  
<http://robotic-controls.com/learn/beaglebone/beaglebone-internet-over-usb-only>  
<https://trac.vicos.si/ros/wiki/Teaching/Exercises/Speech>

##### PACKAGES DE ROS :

<http://packages.ros.org/ros/ubuntu/>  
[http://otamachan.github.io/sphinxros/indigo/packages/kobuki\\_msgs.html](http://otamachan.github.io/sphinxros/indigo/packages/kobuki_msgs.html)  
[http://wiki.ros.org/cob\\_people\\_detection](http://wiki.ros.org/cob_people_detection)  
[https://github.com/vicoslab/vicos\\_ros](https://github.com/vicoslab/vicos_ros)

## **LOGICIELS UTILISÉS :**

PUTTY  
SUITE OFFICE  
GPARTED  
MS PROJECT  
SOLIDWORKS  
PYCHARM

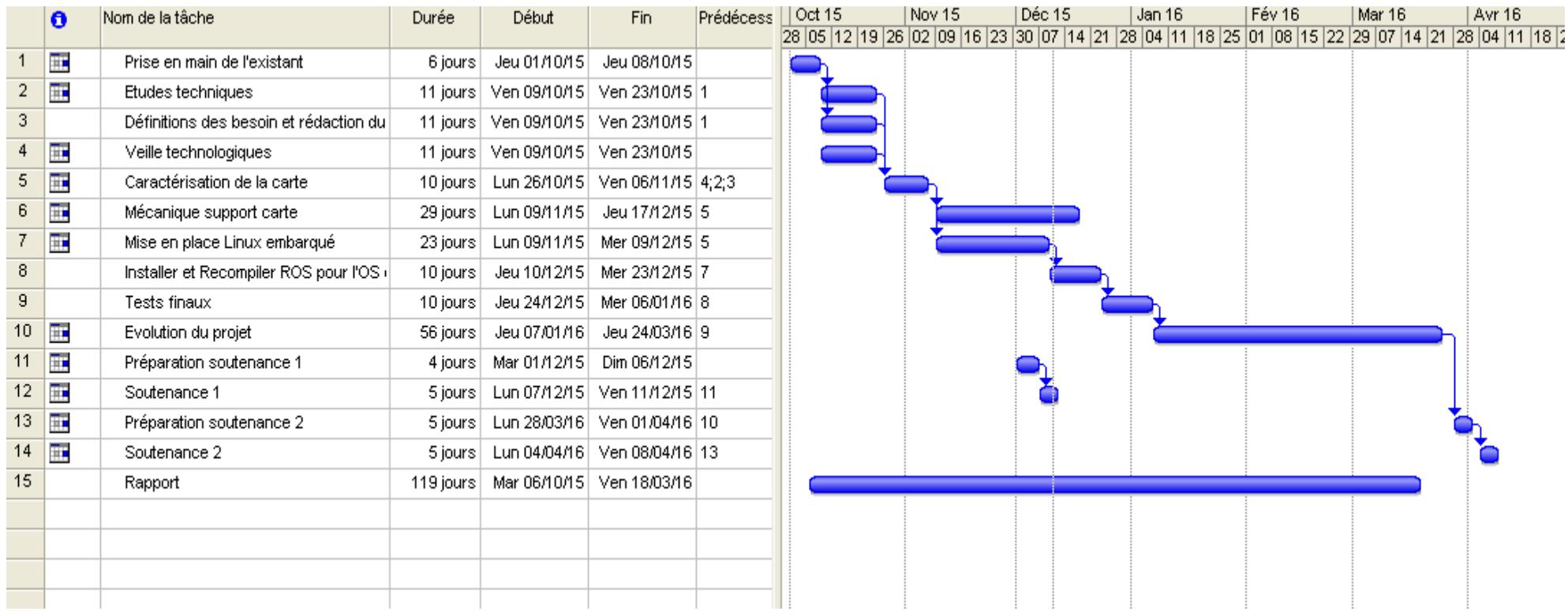
## TABLE DES FIGURES

Figure 1 : Photo du Turtlebot .....	8
Figure 2 : ssh.....	9
Figure 3 : téléop .....	10
Figure 4 : mode suiveur .....	10
Figure 5 : caméra 3D .....	11
Figure 6 : cartographie .....	12
Figure 7 : base Kobuki.....	13
Figure 8 : débits sur rqt.....	14
Figure 9: caractéristiques BeagleBone .....	22
Figure 10 : Tutoriel Ubuntu .....	23
Figure 11 : tutoriel ROS .....	24
Figure 12 : ssh carte .....	25
Figure 13 : terminal.....	26
Figure 14 : port série debug.....	27
Figure 15 : lancement de Putty.....	27
Figure 16 : photo du Turtlebot avec la BeagleBone .....	30
Figure 17 : Benchmark BeagleBone .....	31
Figure 18 : htop Beaglebone .....	32
Figure 19 : htop PC .....	32
Figure 20 : iotop BeagleBone .....	34
Figure 21 : top BeagleBone .....	35
Figure 22 : vue de la Odroid .....	37
Figure 23 - Caractéristiques du Buck.....	38
Figure 24 : benchmark Odroid .....	39
Figure 25 : htop Odroid.....	40
Figure 26 - Caméra de profondeur .....	43
Figure 27 - Découpage de l'image.....	44
Figure 28 - Image découpée.....	44
Figure 29 - Schémas directionnels .....	46
Figure 30 - Reconfiguration dynamique du seuil de détection.....	47
Figure 31 - Node graph detection .....	48
Figure 32 - Position initiale de la caméra .....	49
Figure 33 - Camera sur servomoteur orientable .....	50
Figure 34 - Positions possibles du servo moteur .....	50
Figure 35 - Sorties de la base Kobuki.....	52
Figure 36 - Schéma de la carte.....	52
Figure 37 - Pièces Solidworks en mouvement.....	54
Figure 38 - Pièces Solidworks de face.....	55
Figure 39 - Détection de visages .....	56
Figure 40 - Protocole de la liaison série.....	58
Figure 41 - Node graph son.....	61
Figure 42 - Node graph scenario .....	62
Figure 43 - Image de la caméra de profondeur dans le laboratoire d'énergie .....	65
Figure 44 - Diagramme de GANTT prévisionnel .....	72

Figure 45 - Diagramme de GANTT réel .....	73
Figure 46 - Schéma bloc Odroid .....	74

Tableau 1 : QVGA/VGA.....	14
Tableau 2 : débits du PC .....	14
Tableau 3 : comparatif des cartes.....	18
Tableau 4 : comparatif carte résumé .....	19
Tableau 5 : débits PC Quentin.....	28
Tableau 6 : débits BeagleBone.....	29
Tableau 7 : caractéristiques Odroid .....	36
Tableau 8 : débits de la Odroid.....	38

## DIAGRAMMES DE GANTT



**Figure 44 - Diagramme de GANTT prévisionnel**

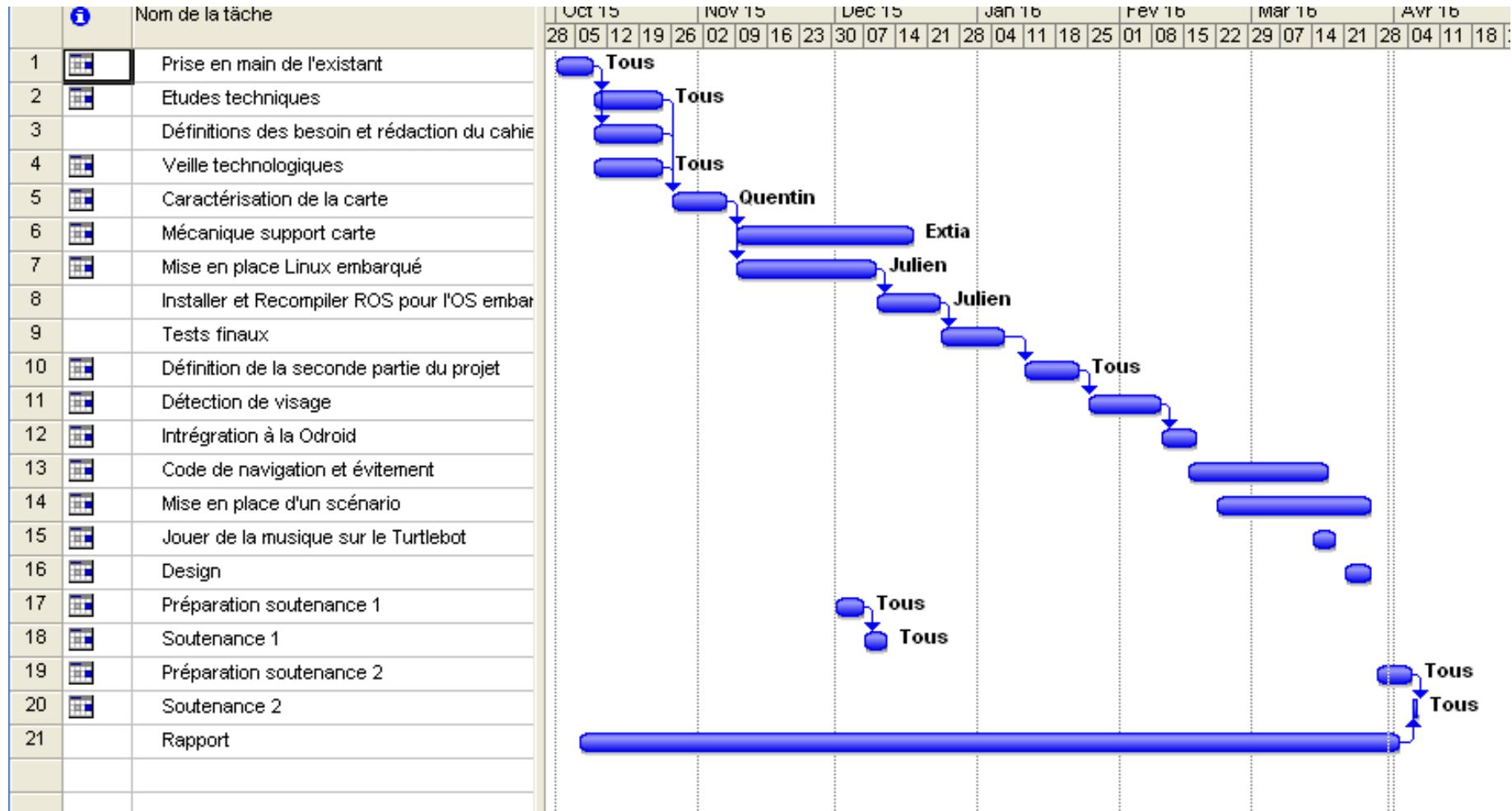


Figure 45 - Diagramme de GANTT réel

## SCHEMA BLOC DE LA ODROID

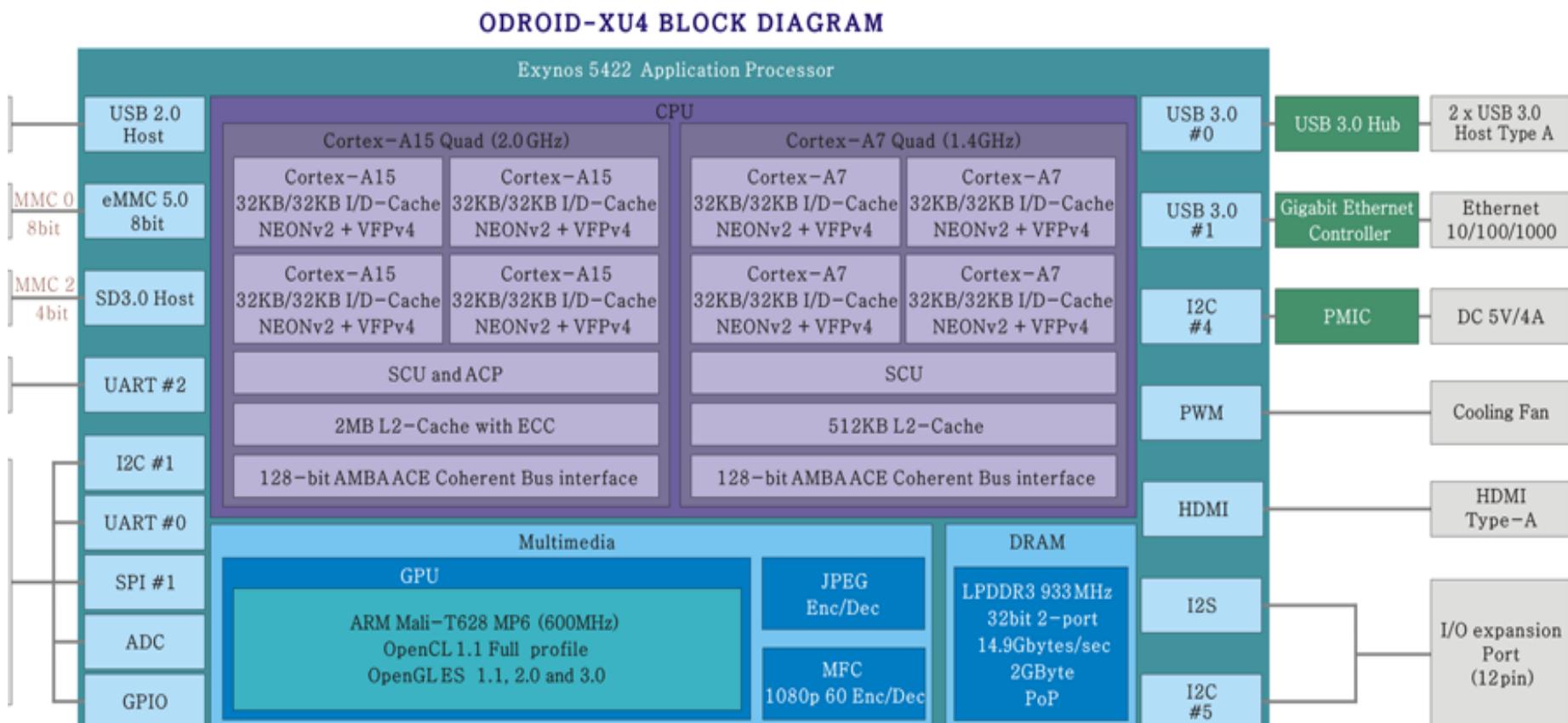


Figure 46 - Schéma bloc Odroid

## TUTORIELS :

### TUTORIEL 1 : Installation d'Ubuntu sur une BeagleBone :

#### PRÉ-REQUIS :

- Vous devez disposer d'une carte BeagleBone Black
- Une carte micro SD d'au moins 8 Go
- Un ordinateur avec Linux installé et une connexion internet
- Un lecteur de carte SD pour votre ordinateur
- Un adaptateur SD microSD

Sur le PC avec Linux dans un terminal exécutez les commandes suivantes :

Tout d'abord on va télécharger une image Pré-buildée d'Ubuntu :

```
wget https://rcn-ee.com/rootfs/2015-11-13/elixir/ubuntu-14.04.3-console-armhf-2015-11-13.tar.xz
```

Dézipper l'archive :

```
tar xf ubuntu-14.04.3-console-armhf-2015-11-13.tar.xz
```

Se placer dans le dossier ainsi créé :

```
cd ubuntu-14.04.3-console-armhf-2015-11-13
```

Exécuter le script et identifier le nom de la carte SD :

```
sudo ./setup_sdcard.sh --probe mmc
```

```
turtlebot@turtlebot: ~/ti-linux-kernel-dev/ubuntu-14.04.3-console-armhf-2015-10-09
system.sh.sample
tools
ubuntu-14.04.3-console-armhf-2015-10-09
ubuntu-14.04.3-console-armhf-2015-10-09.tar.xz
version.sh
turtlebot@turtlebot:~/ti-linux-kernel-dev$ cd ubuntu-14.04.3-console-armhf-2015-10-09/
turtlebot@turtlebot:~/ti-linux-kernel-dev/ubuntu-14.04.3-console-armhf-2015-10-09$ sudo ./setup_sdcard.sh --probe mmc
running:

Are you sure? I Don't see [/dev/idontknow], here is what I do see...

lsblk:
NAME      MAJ:MIN RM    SIZE RO TYPE MOUNTPOINT
sda        8:0    0 465,8G  0 disk 
└─sda1     8:1    0   512M  0 part /boot/efi
└─sda2     8:2    0 461,4G  0 part /
└─sda3     8:3    0   3,9G  0 part [SWAP]
mmcblk0   179:8   0    7,4G  0 disk 
└─mmcblk0p1 179:9  0   1,7G  0 part
turtlebot@turtlebot:~/ti-linux-kernel-dev/ubuntu-14.04.3-console-armhf-2015-10-09$
```

Ici la carte SD est le périphérique mmcblk0.

Réexécuter le script avec les paramètres suivants (sdX étant le nom de la carte SD):

```
sudo ./setup_sdcard.sh --mmc /dev/sdX --dtb beaglebone
```

Ici on utilise donc :

```
sudo ./setup_sdcard.sh --mmc /dev/mmcblk0 --dtb board
```

Ensuite il faut télécharger le Flasher :

```
wget https://rcn-ee.com/rootfs/2015-11-13/flasher/BBB-eMMC-flasher-ubuntu-14.04.3-console-armhf-2015-11-13-2gb.img.xz
```

Dézipper l'archive :

```
unxz BBB-eMMC-flasher-ubuntu-14.04.3-console-armhf-2015-11-13-2gb.img.xz
```

Enfin l'installer sur la carte SD (remplacer sdX par le nom de votre carte SD comme précédemment):

```
sudo dd if=./BBB-eMMC-flasher-ubuntu-14.04.3-console-armhf-2015-11-13-2gb.img of=/dev/sdX
```

Une fois l'opération finie, il faut mettre la carte SD sur la BeagleBone puis la redémarrer. Un chenillard devrait alors apparaître sur les leds Bleues. Une fois le procédé fini, toutes les leds deviendront bleues.

Enlever la carte SD puis redémarrer la BeagleBone. Vous serez alors sur Ubuntu. Le nom d'utilisateur est ubuntu et le mot de passe temppwd

## TUTORIEL 2 : Installation Ubuntu Odroid

### PRÉ-REQUIS :

- Vous devez disposer d'une carte Odroid XU4
- Une carte micro SD d'au moins 8 Go
- Un ordinateur avec Linux installé et une connexion internet
- Un lecteur de carte SD pour votre ordinateur
- Un adaptateur SD microSD

Ce tutoriel va vous guider pour installer Ubuntu sur une carte SD.

Dans un premier temps il faut télécharger une image de la version d'Ubuntu à installer pour Odroid. Elle peut être trouvée à cette adresse :

[http://odroid.in/ubuntu\\_14.04lts/](http://odroid.in/ubuntu_14.04lts/)

Dans notre cas nous optons pour la version suivante : `ubuntu-14.04.1lts-lubuntu-odroid-xu3-20150212.img.xz`

[http://odroid.in/ubuntu\\_14.04lts/ubuntu-14.04.1lts-lubuntu-odroid-xu3-20150212.img.xz](http://odroid.in/ubuntu_14.04lts/ubuntu-14.04.1lts-lubuntu-odroid-xu3-20150212.img.xz)

Bien que notre carte soit une Odroid XU4, nous utilisons une version d'Ubuntu pour une Odroid XU3. Cela ne pose pas de soucis car les deux cartes sont entièrement compatibles logiciellement.

Il faut ensuite télécharger le fichier md5sum associé :

[ubuntu-14.04.1lts-lubuntu-odroid-xu3-20150212.img.xz.md5sum](http://odroid.in/ubuntu_14.04.1lts-lubuntu-odroid-xu3-20150212.img.xz.md5sum)

Une fois l'archive téléchargée, à partir d'un terminal utilisez l'instruction

`md5sum Nom_de_l'archive.xz`

La chaîne de caractères affichée doit être identique à celle présente dans le fichier .md5sum téléchargé précédemment. Ce procédé permet de vérifier l'intégrité du fichier. Si les valeurs étaient différentes, il faudrait retélécharger l'archive à nouveau.

Dézippez ensuite l'archive à l'aide la commande `unxz Nom_de_l'archive.xz`

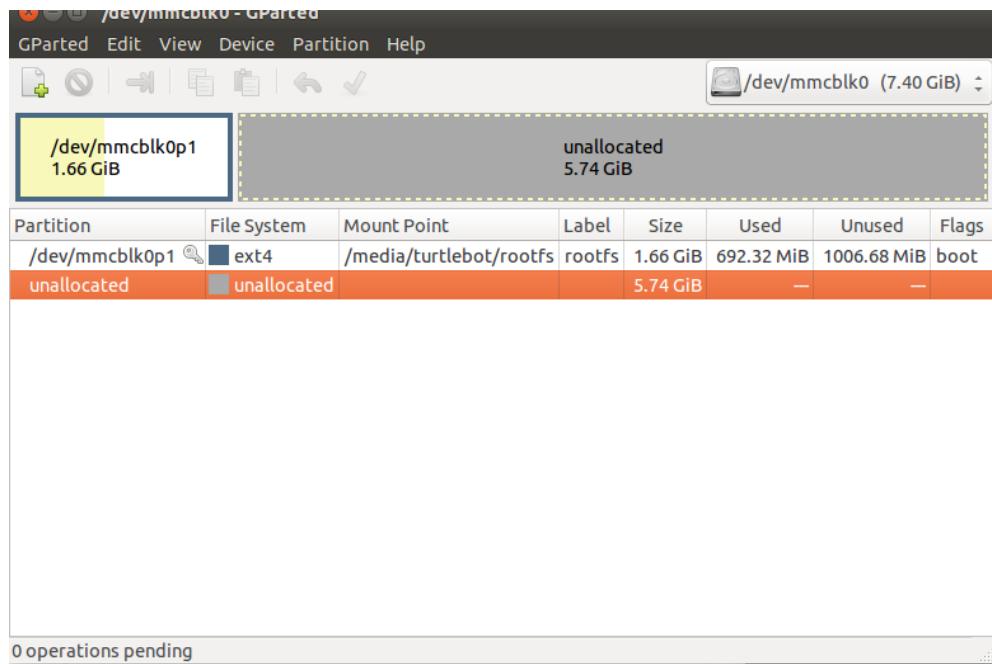
Vous devriez après un long moment obtenir un fichier .img. Il contient l'image à flasher sur la carte SD.

Il est préférable de d'abord réinitialiser votre carte SD afin d'être sûr que celle-ci soit vierge. Il faut aussi également identifier votre point de montage. Pour cela le plus simple est d'utiliser un logiciel appelé gparted.

Pour l'installer : `sudo apt-get install gparted`

Pour l'utiliser : `sudo gparted`

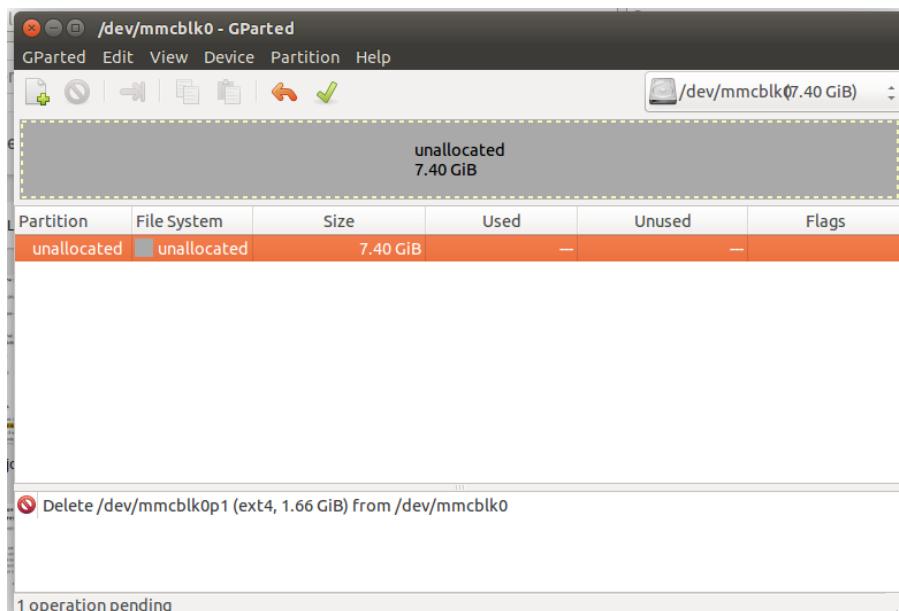
Une fois celui-ci ouvert vous devriez avoir une interface ressemblant à celle-ci :



Ici on voit que le point de montage de notre carte SD est /dev/mmcblk0p1 et qu'elle dispose d'une deuxième partie non allouée.

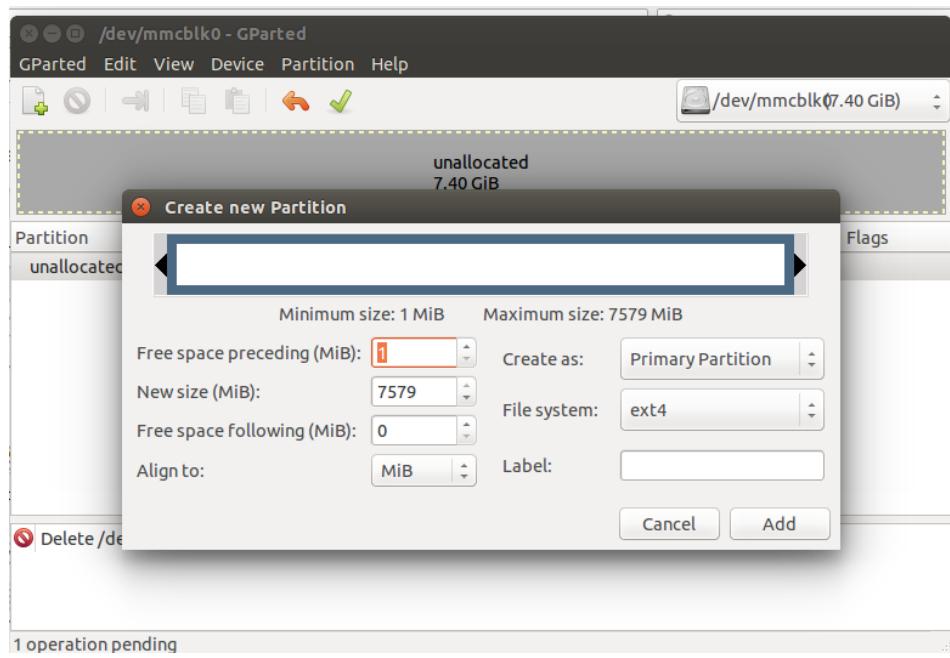
Faire un clique droit sur la partition et sélectionner "umount".

Puis clique droit Delete. On obtient alors une seule grande partition non allouée.



Faire clique droit new

Puis créez une partition ext4 de la taille de votre carte



Validez l'opération avec la croix verte en haut.

Vous devriez à nouveau voir apparaître le nom de la partition : /dev/mmcblk0p1  
Le point de montage de la carte SD est donc /dev/mmcblk0. (p1 signifie partition 1).

Votre carte est maintenant vierge, on peut maintenant flasher l'image sur la carte.

Pour flasher l'image il faut utiliser l'instruction suivante :

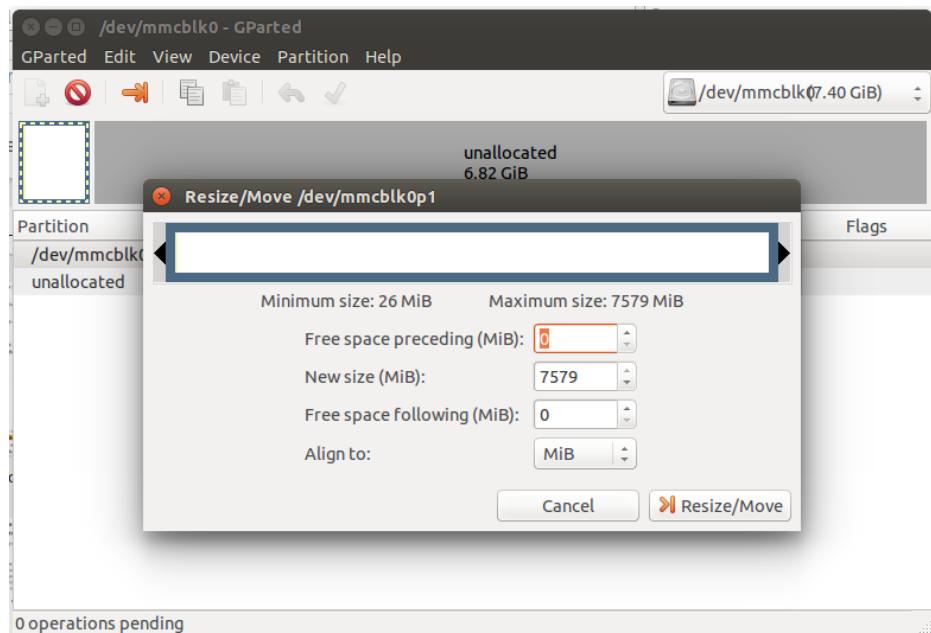
```
sudo dd if=image.img of=/dev/sdX bs=4M
```

/dev/sdX étant le point de montage de votre carte SD. (pour nous /dev/mmcblk0)

L'installation d'Ubuntu est alors terminée, mais il reste un dernier point utile à réaliser. Par défaut l'opération de flashage n'utilise pas toute la carte SD. Elle crée une partition de 5 Go. Donc si vous utilisez une carte de plus grande taille il est conseillé d'étendre la partition à la totalité de la carte.

Pour cela on retourne dans gparted.

Vous devriez voir deux partitions, votre partition principale de 5Go et une partition non allouée. Faire clique droit sur la partition principale puis Resize/Move et étendre la partition à toute la carte.



Valider avec la croix verte.

Vous pouvez maintenant introduire la carte SD dans la Odroid. Pensez à vérifier le switch de sélection de boot. Celui-ci doit être mis sur la position SD.

Le nom d'utilisateur est odroid et le mot de passe odroid.

## TUTORIEL 3 : Installation de ROS

### PRÉ-REQUIS :

Vous devez disposer d'un appareil avec Linux installé et d'une connexion internet.

Pour installer ROS il faut tout d'abord mettre à jour les variables locales :

```
sudo update-locale LANG=C LANGUAGE=C LC_ALL=C LC_MESSAGES=POSIX
```

Ensuite on indique la source des packages ROS :

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.list'
```

On configure les clés :

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add -
```

On met à jour l'utilitaire d'installation :

```
sudo apt-get update
```

Pour installer un package ROS il faut utiliser l'instruction :

```
apt-get install nom_du_package
```

Pour rechercher des packages nous devons utiliser l'instruction :

```
apt-cache search Nom_de_la_recherche
```

Dans notre cas : `apt-cache search turtlebot`

Nous avons installé les packages suivants :

- ros-indigo-turtlebot
- ros-indigo-turtlebot-navigation
- ros-indigo-turtlebot-follower
- ros-indigo-ros-base

Afin de pouvoir lancer les commandes relatives à ROS dans la console, il faut ajouter les sources au fichier de bashrc :

```
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc  
source ~/.bashrc
```

On peut également le faire localement dans chaque terminal avec l'instruction :

```
source /opt/ros/indigo/setup.bash
```

Enfin pour faire fonctionner les périphériques USB, il faut utiliser :

```
rosrun kobuki_ftdi create_udev_rules
```

ROS est maintenant fonctionnel et utilisable pour un Turtlebot.

## TUTORIEL 4 : Utilisation de ROS déporté :

### PRÉ-REQUIS :

Pour suivre ce tutoriel vous devez disposer de deux ordinateurs/appareils avec ROS installé et connectés au même réseau internet.

On utilise un ROS déporté si on veut accéder aux nœuds ROS d'un PC sur un autre PC. Pour cela la procédure à suivre est la suivante :

Sur le PC ou carte où le maître ROS va fonctionner, écrire les lignes de commande suivante avant de lancer ROS, et ce dans tous les terminaux où des commandes ROS vont être lancées :

```
export ROS_MASTER_URI=http://adresse_ip_du_maitre:11311  
export ROS_HOSTNAME=adresse_ip_du_maitre
```

Sur le deuxième PC ou carte, où l'on veut accéder aux topics ROS : (et ce dans tous les terminaux utilisant ROS) :

```
export ROS_MASTER_URI=http://adresse_ip_du_maitre:11311  
export ROS_HOSTNAME=adresse_ip_du_second_pc
```

L'adresse ip d'un PC peut être obtenue avec l'instruction : `ifconfig`

### Exemple :

Notre carte Odroid a l'adresse ip suivante : 192.168.0.2

Notre PC a l'adresse ip suivante : 192.168.0.3

On veut faire fonctionner le robot et sa caméra uniquement sur la carte Odroid et visualiser des images dans rqt sur notre PC.

Pour cela on ouvre deux terminaux sur la carte Odroid (en ssh par exemple) et on exécute les lignes suivantes :

#### Terminal 1 :

```
export ROS_MASTER_URI=http://192.168.0.2:11311  
export ROS_HOSTNAME=http://192.168.0.2  
roslaunch turtlebot Bringup minimal.launch
```

#### Terminal 2 :

```
export ROS_MASTER_URI=http://192.168.0.2:11311  
export ROS_HOSTNAME=http://192.168.0.2  
roslaunch turtlebot Bringup 3dsensor.launch
```

#### Sur le PC :

```
export ROS_MASTER_URI=http://192.168.0.2:11311  
export ROS_HOSTNAME=http://192.168.0.3  
rqt
```

**Remarque :** les roslaunch relatifs à la gestion de périphériques (comme la base du Turtlebot ou la caméra) doivent être lancés à partir du terminal du PC sur lequel ils sont branchés en USB.

## TUTORIEL 5 : Installation d'un package ROS via Catkin

### PRÉ-REQUIS :

Pour suivre ce tutoriel vous devez disposer d'un appareil avec Linux et les fonctions de base de ROS.

Ce tutoriel est appliqué à l'installation du package facedetection que nous utilisons sur le robot, mais peut être appliqué à tout package ROS utilisant Catkin trouvés sur Github par exemple.

Afin d'installer le package ROS qui nous servira à détecter les visages on doit créer un workspace Catkin et y télécharger les fichiers nécessaires puis les compiler.

Pour cela ouvrez un terminal et exécutez les instructions suivantes :

```
mkdir -p ~/catkin_ws/src
```

Le nom du workspace est ici catkin\_ws. Il peut avoir le nom que vous souhaitez et vous pouvez le placer à l'endroit que vous voulez. Par contre le dossier src doit absolument respecter ce nom.

Une fois le workspace catkin créé il faut mettre les packages ROS qui nous intéressent dans le dossier src. Ici nous le récupérons de Github :

```
cd catkin_ws/src  
git clone https://github.com/vicoslab/vicos_ros.git
```

Plusieurs des packages contenus dans ce repository ne nous intéressent pas, nous allons tout simplement les supprimer :

```
cd vicos_ros  
sudo rm -r ferms_ros/ julius_ros/ localizer/ turtlebot_vicos/ zbar_ros/
```

Une fois les packages supprimés, nous pouvons lancer la compilation. Pour cela il faut se placer à la racine de notre workspace catkin

```
cd ~/catkin_ws  
catkin_make
```

Vous devriez alors voir la compilation se faire jusqu'à arriver à 100%

Votre package est alors installé, cependant pour l'utiliser il faut sourcer le bash de setup. Vous avez plusieurs solutions, vous pouvez soit le faire à chaque fois, dans chaque terminal qui

utilisera des données relatives à ce package en utilisant depuis la racine de votre workspace la commande suivante :

```
source devel/setup.bash
```

Ou en ajoutant une fois pour toute la commande dans le fichier qui s'exécute à chaque ouverture d'un terminal de la manière suivante :

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

## TUTORIEL 6 : Création d'un package ROS

### PRÉ-REQUIS :

- **Connaissances :**

- Aucune connaissance n'est requise pour ce tutoriel puisqu'il va aborder tous les points de A à Z.
- Cependant des notions de Python peuvent aider

- **Matériel :**

- Appareil avec Linux installé
- Fonctions de base de ROS installées

### SOMMAIRE

- i. Création d'un package ROS
- ii. Création de messages ROS
- iii. Création d'un script python
- iv. Création d'un launch file
- v. Configuration du fichier package.xml
- vi. Configuration du fichier CMakeLists.txt
- vii. Compilation du package
- viii. Utilisation du package

Ce tutoriel a pour but d'apprendre à créer ses propres packages ROS. Tous les fichiers créés dans ce tutoriel sont disponibles sur notre Github à l'adresse suivante [https://github.com/TurtlebotMec3/example\\_package\\_ros](https://github.com/TurtlebotMec3/example_package_ros) et peuvent être installés et testés en suivant le tutoriel 5 : Installation d'un package ROS via Catkin

(`git clone https://github.com/TurtlebotMec3/example_package_ros.git`)

## 1. Création d'un package ROS

La création d'un package ROS permet différentes choses :

- Porter vos programmes très facilement sur un autre système
- Créer des messages personnalisés pour vos topics ROS
- Créer des launch file pour lancer vos nœuds codés en python par exemple

Afin de créer un package ROS il faut dans un premier temps créer un workspace catkin qui contiendra vos futurs packages. Catkin est une fonction de ROS qui permet de compiler des packages.

Un workspace catkin contient 3 dossiers : build/ devel/ et src/, les deux premiers sont générés par la compilation des packages, et le dernier contient tous vos programmes. On va donc le créer.

```
mkdir -p ~/workspace/src
```

Vous pouvez évidemment créer ce workspace avec le nom et la location que vous voulez.

Pour rappel “~/” est équivalent à “/home/votre\_nom\_d'utilisateur/”

Rendez vous maintenant dans le dossier src/ :

```
cd ~/workspace/src
```

Et créez votre propre package ROS :

```
catkin_create_pkg mon_package_ros
```

La fonction catkin\_create\_pkg va vous créer automatiquement deux fichiers à l'intérieur du dossier créé : package.xml et CMakeLists.txt. Ils sont tous les deux indispensables à la compilation de votre package et doivent être configurés correctement.

Dans un premier temps nous allons créer les dossiers qui vont contenir nos codes sources :

```
cd mon_package_ros  
mkdir launch/ msg/ src/
```

Le dossier launch contiendra tous vos launch files.

Le dossier msg contiendra la définition de vos messages ROS.

Le dossier src contiendra tous vos scripts.

Pour clarifier, dans toute la suite du tutoriel nous prendrons un exemple relativement simple. Nous créerons deux nœuds ROS ; l'un publiera des données sur un topic qui seront ensuite récupérées par un deuxième nœud qui s'abonnera à ce topic. Pour cela nous créerons un message personnalisé qui contient un nombre entier. On considérera que cet entier est le numéro d'une chanson à jouer par le topic qui s'y abonnera.

## 2. Création de messages ROS

Un message ROS est un fichier .msg se trouvant dans le dossier msg/ de votre package, et qui définit les variables que vous allez publier ou recevoir sur un topic ROS.

Pour le créer :

```
touch msg/MonMessage.msg
```

Il semble y avoir une convention ROS pour les messages avec une majuscule à la première puis à chaque mot comme ci-dessus.

Pour l'éditer :

```
vim msg/MonMessage.msg
```

Pour rappel pour écrire du texte dans vim, il faut dans un premier temps appuyer sur la touche i de votre clavier et passer ainsi en mode édition.

Voici quelques commandes vim qui vous seront très utiles, il en existe beaucoup d'autres :

- Pour enregistrer sans quitter appuyer sur échap puis tapez :w! et validez en appuyant sur entrer.
- La commande pour enregistrer et quitter est :x
- Pour quitter tout simplement :q
- Pour quitter sans enregistrer :q!

La forme du message est la suivante :

```
type_de_donnée nom_de_la_variable
```

Il existe de nombreux types de variables : string, bool, uint8, int16, float32, ...

Vous pouvez également utiliser comme variable des messages déjà créés. Par exemple une fois notre fichier message .msg créé, on peut écrire dans un autre fichier:

```
message nom_de_la_variable
```

**NB :** on peut également créer l'équivalent d'un "define" en affectant une valeur à une variable. Cela peut permettre de rendre le code de vos scripts plus clair.

**NB :** Si vous faites un copier/coller du code tel quel il se peut que ça ne marche pas, parce que certains caractères ne seront pas encodés de la bonne manière (exemple les : " et ' )

### Exemple :

```
# Message d'exemple
# Contient le numéro de la chanson à jouer
# Notre variable entière pourra prendre plusieurs valeurs
#   0 : Chanson numéro 0
#   1 : Chanson numéro 1
#   2 : Chanson numéro 2
#   3 : Bruit désagréable

uint8 TITRE0          = 0
uint8 TITRE1          = 1
uint8 TITRE2          = 2
uint8 BRUIT_DESAGREEABLE = 3

# Déclaration de la variable
uint8 value
```

Le signe # signifie comme en python que la ligne est un commentaire.

### 3. Crédation d'un script python

Le script python définit ce que votre nœud va faire. Nous allons en créer deux, un pour le récepteur et un pour l'émetteur.

```
touch src/émetteur.py src/recepteur.py
```

**NB :** Afin que notre script puisse s'exécuter correctement quand on l'appellera à partir d'un launch file, il faut **IMPÉRATIVEMENT** lui rajouter les droits d'exécutions :

```
chmod +x src/émetteur.py src/recepteur.py
```

La première ligne de chaque script python est la suivante, elle permet au système et aux éditeurs de textes de savoir que c'est un script écrit en python :

```
#!/usr/bin/env python
```

Ensuite les lignes qui suivent permettent de charger les bibliothèques dont on aura besoin. Pour utiliser des topics ROS, et créer des noeuds en python il faut utiliser rospy. Il se peut que vous ayez à l'installer sur votre système (sudo apt-get install ...)

```
import rospy
```

On va ensuite importer notre message ROS créé précédemment pour pouvoir l'utiliser dans le script. La convention est un peu particulière. L'extension .msg est ajoutée au nom du package et enlevée du nom du fichier du message.

Vous pouvez retrouver facilement cette information pour d'autres packages ROS, les messages se trouvent toujours dans le dossier msg du package et l'appel se fait toujours de la même manière. Pour accéder facilement à un package ROS dont vous ne connaissez pas le chemin vous pouvez utiliser la fonction `roscd nom_du_package` qui vous amènera directement dans le dossier du package ROS.

Il vous faudra également consulter les fichiers .msg que vous voulez utiliser pour connaître le nom des variables etc.

```
from nom_du_pckage_ros.msg import nom_du_message
```

Enfin on trouve également dans tout script ROS les lignes suivantes qui sont en fait le point d'entrée d'un script python :

```
if __name__ == '__main__':
    main()
```

En python on définit une fonction de la manière suivante :

```
def fonction(variables):
```

Les fonctions principales de rospy que vous pouvez être amenés à utiliser sont les suivantes :

```
rospy.init_node('nom_du_noeud')
```

Le nom du nœud doit être unique.

```
rospy.Subscriber('nom_du_topic', type_du_message, callback)
```

Callback est une fonction qui sera appelée à chaque fois qu'un message est publié sur le topic.

```
rospy.Publisher('nom_du_topic', type_du_message)
```

```
rospy.Rate(fréquence) qui permet de publier des messages à une fréquence fixe
```

```
rospy.is_shutdown() permet de savoir de savoir si l'utilisateur veut interrompre le programme (ctrl-c par exemple)
```

```
rospy.spin() permet de boucler le programme mais d'en sortir en cas de CTRL-C
```

Un exemple vaut mieux que la leçon :

**NB :** Si vous faites un copier/coller du code tel quel il se peut que ça ne marche pas, parce que certains caractères ne seront pas encodés de la bonne manière (exemple les : “ et ‘ )

### Code de l'émetteur : (Disponible sur le Github)

```
#!/usr/bin/env python
import rospy
from random import *
from mon_package_ros.msg import MonMessage

def main():
    rospy.init_node('noeud_emetteur')
    pub = rospy.Publisher('mon_topic/message', MonMessage)
    rate = rospy.Rate(10) #10 Hz

    while not rospy.is_shutdown():
        variable = MonMessage()

        # La fonction choice permet de choisir une valeur du tableau de
        # manière aléatoire
        variable.value = choice([variable.TITRE0, variable.TITRE1,
                               variable.TITRE2, variable.BRUIT_DESAGREABLE])

        # on publie le message
        pub.publish(variable)

        # Pour fonctionner à fréquence fixe
        rate.sleep()

if __name__ == '__main__':
    main()
```

### Code du récepteur :

```
#!/usr/bin/env python
import rospy
from mon_package_ros.msg import MonMessage
```

```

# fonction appelée quand on reçoit un message
def callback(data):
    if data.value == data.TITRE0:
        print('Je joue la chanson numero 0')
    elif data.value == data.TITRE1:
        print('Je joue la chanson numero 1')
    elif data.value == data.TITRE2:
        print('Je joue la chanson numero 2')
    elif data.value == data.BRUIT_DESAGREEABLE:
        print('Je joue un bruit désagréable')
    else:
        print('Je ne connais pas cette chanson')

def main():
    rospy.init_node('noeud_recepteur')
    rospy.Subscriber('mon_topic/message', MonMessage, callback)

    # on boucle tant que l'utilisateur n'arrête pas le programme
    rospy.spin()

if __name__ == '__main__':
    main()

```

## 4. Création d'un launch file

Le launch file permet le lancement automatique de tous les nœuds ROS. Les launch files sont en fait des fichiers XML.

Pour le créer `touch launch/mon_launch_file.launch`

Voici un launch file pour lancer un nœud python :

```

<?xml version="1.0"?>

<launch>
    <!-- Ceci est un commentaire -->
    <node  pkg='nom_du_package'  type='nom_du_script.py'  name='nom_du_noeud'
output='screen'>
    </node>
</launch>

```

On peut également appeler d'autres launch files dans un launch file :

```
<launch>
    <include file="$(find nom_du_package)/launch/nom_du_launch_file.launch" >
    </include>
</launch>
```

Il est également possible de passer des arguments aux nœuds ou aux fichiers inclus mais ce ne sera pas détaillé ici.

Pour plus d'informations vous pouvez ouvrir des fichiers launch file déjà existants et consulter le wiki : <http://wiki.ros.org/ROS/Tutorials/Roslaunch%20tips%20for%20larger%20projects>

**Notre launch file sera donc :**

```
<?xml version="1.0"?>

<launch>
    <node    pkg="mon_package_ros"    type="emetteur.py"    name="noeud_emetteur"
output="screen">
    </node>

    <node    pkg="mon_package_ros"    type="recepteur.py"   name="noeud_recepteur"
output="screen">
    </node>
</launch>
```

**NB :** Même remarque que précédemment, un simple copier/coller du code peut ne pas fonctionner car certains caractères ne seront codés correctement.

## 5. Configuration du fichier package.xml

Le fichier package.xml permet de donner toutes les informations utiles à savoir :

- Le nom du package
- Sa version
- Une description
- L'auteur avec un contact
- Une licence (opensource ou non)

- La méthode de compilation (catkin pour nous)
- Et enfin toutes les dépendances

Votre fichier package.xml doit ressembler à ça (attention s'il manque certains champs, la compilation échouera)

```
<?xml version="1.0"?>

<package>

<name>mon_package_ros</name>

<version>1.0.0</version>

<description>Ce package a pour but de vous apprendre à créer des packages ros</description>

<maintainer email="turtlebot.mec3@gmail.com"> Mec3</maintainer>

<license>BSD</license>

<url>https://github.com/TurtlebotMec3</url>

<author>Quentin Mercier</author>

<buildtool_depend>catkin</buildtool_depend>

<!-- Vous devez mettre ici les dépendances nécessaires à l'utilisation de votre package -->

<build_depend>rospy</build_depend>

<run_depend>rospy</run_depend>

</package>
```

## 6. Configuration du fichier CMakelists.txt

Le fichier CMakelists.txt est lui un peu plus complexe et complet. Il définit toute l'arborescence de votre package, c'est à dire :

- Les scripts python et leurs localisation
- Les messages ROS
- Les launch file et leurs locations

- Les dépendances (messages / packages)

Voici un exemple de CMakeLists avec les fonctionnalités principales. Il en existe beaucoup d'autres expliqués dans le fichier par des commentaires lors de sa création. Nous avons gardé ici et configuré que les principales :

```
cmake_minimum_required(VERSION 2.8.3)
project(mon_package_ros)

## package indispensable a la compilation
# genmsg permet d'executer la fonction generate_messages plus bas
find_package(catkin REQUIRED COMPONENTS rospy genmsg)

#####
## Declare ROS messages, services and actions ##
#####

# Il faut ajouter ici les messages que nous avons crees
add_message_files(
    FILES
    MonMessage.msg
)

# Il faut mettre ici nos dependances quand on en a
generate_messages(
    DEPENDENCIES
    # exemple std_msgs
)

#####

catkin_package(
)

#####
## Install ##
#####

# Il faut mettre ici la location de vos scripts pythons
install(PROGRAMS
    src/emetteur.py
    src/recepteur.py
```

```

DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)

# On met ici les autres fichiers à installer comme les launch files
install(FILES
    launch/mon_launch_file.launch
    DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
)

```

## 7. Compilation du package

Toute modification ou ajout de script, de message ROS ou du fichier CMakeLists.txt nécessite de recompiler le package.

Pour compiler le package il faut revenir à la racine du répertoire catkin

```
cd ~/workspace/
```

Puis exécuter la commande :

```
catkin_make
```

Vous devriez alors voir la compilation s'exécuter.

## 8. Utilisation du package

Pour utiliser le package ROS, il faut changer les informations du package dans le terminal que l'on utilise :

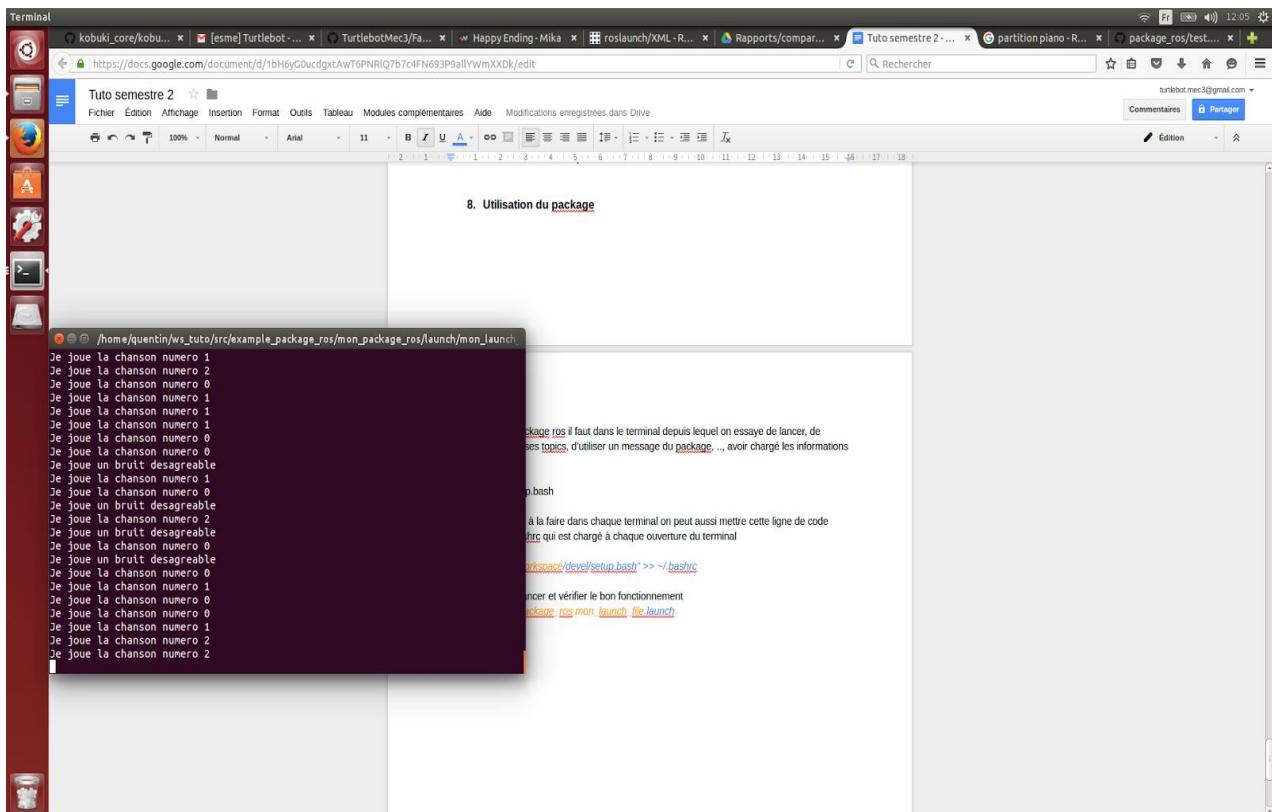
```
source devel/setup.bash
```

Pour ne pas avoir à la faire dans chaque terminal on peut aussi mettre cette ligne de code dans le fichier bashrc qui est chargé à chaque ouverture du terminal :

```
echo "source ~/workspace/devel/setup.bash" >> ~/.bashrc
```

Enfin on peut le lancer et vérifier le bon fonctionnement :

```
roslaunch mon_package_ros mon_launch_file.launch
```



### Vérification du bon fonctionnement du package

## TUTORIEL 7 : Utilisation du package sound\_publisher

### PRÉ-REQUIS :

- Un ordinateur avec Ubuntu et ros installés
- Un robot Turtlebot
- Notre package sound\_publisher installé
- Le package pyserial

### SOMMAIRE

- i. Notes définies en fréquence
- ii. Notes définies par leur nom et leur octave
- iii. Morceau de musique

Si vous n'avez pas le package pyserial, pour l'installer utilisez la commande :

`sudo apt-get install python-serial` dans un terminal

Le package sound\_publisher permet de jouer des sons personnalisés sur la base Kobuki. Il peut être installé en suivant notre tutoriel 5. Le code est disponible ici :

[https://github.com/TurtlebotMec3/package\\_ros](https://github.com/TurtlebotMec3/package_ros).

**NB :** les sons ne doivent être joués que quand le robot est à l'arrêt

**Il existe 3 modes différents pour jouer des sons personnalisés :**

- Jouer une série de notes définies par leur fréquence et leur durée
- Jouer une série de notes définies par leur nom, leur octave et leur durée
- Jouer un morceau prédéfini

Tout d'abord notez que vous trouverez les fichiers relatifs aux nœuds dans le dossier src, et ceux aux messages dans le dossier msg.

Pour lancer l'ensemble des nœuds nécessaires il faut exécuter la commande :

`roslaunch sound_publisher sound_publisher.launch`

Enfin veuillez noter que toutes les durées sont exprimées en ms.

## 1. Notes définies en fréquence

Pour jouer une note définie en fréquence vous devez publier la note en question sur le topic /sound/tones

Vous pouvez le faire directement dans un terminal :

```
rostopic pub /sound/tones sound_publisher/TonesArray
"score:
- frequency: 4800.0
time: 2000"
```

Ou alors dans un fichier python. Pour lancer des sons dans un script python, assurez-vous d'avoir au minimum les lignes de codes suivantes :

```
#!/usr/bin/env python
import roslib
import rospy
from sound_publisher.msg import Tones
from sound_publisher.msg import TonesArray

def main():
    pub = rospy.Publisher('sound/tones', TonesArray)
    partition = TonesArray()
    partition.score=[Tones(880, 500),
                    Tones(880, 500),
                    Tones(880, 500),
                    Tones(698.5, 376),
                    Tones(1046.5, 200),
                    Tones(880, 500),
                    Tones(698.5, 376),
                    Tones(1046.5, 200)]
    pub.publish(partition)
    rospy.spin()

if __name__ == '__main__':
#Initiate the node
    rospy.init_node('noeud qui publie', anonymous=True)
    main()
```

La variable partition, de type TonesArray est un tableau de Tones contenu dans la variable score.  
C'est pourquoi on écrit `partition.score = [Tones(), Tones(), ...]`

L'instruction `rospy.Publisher('sound/tones', TonesArray)` permet de définir sur quel topic on va publier.

Enfin l'instruction `pub.publish()` permet d'envoyer notre message sur le topic.

## 2. Notes définies par leur nom et leur octave :

Pour jouer une note définie en fréquence vous pouvez publier la note en question sur le topic /sound/musical\_note

```
rostopic pub /sound/musical_note sound_publisher/MusicalTonesArray
"score:
- tone: 'do#'
  octave: 6
  time_base: 8000"
```

Ou alors dans un fichier python. Pour lancer des sons dans un script python, assurez-vous d'avoir au minimum les lignes de codes suivantes :

```
#!/usr/bin/env python
import roslib
import rospy
from sound_publisher.msg import MusicalTones
from sound_publisher.msg import MusicalTonesArray

def main():
    pub = rospy.Publisher('sound/musical_note', MusicalTonesArray)
    partition = MusicalTonesArray()
    partition.score=[MusicalTones('do', 5, 500),
                    MusicalTones('re', 5, 500),
                    MusicalTones('mi', 5, 500),
                    MusicalTones('fa', 5, 500),
                    MusicalTones('sol', 5, 500),
                    MusicalTones('la', 5, 500),
```

```

        MusicalTones('si', 6, 500),
        MusicalTones('do', 5, 500)]
pub.publish(partition)
rospy.spin()

if __name__ == '__main__':
#Initiate the node
    rospy.init_node('noeud qui publie', anonymous=True)
    main()

```

La variable partition, de type MusicalTonesArray est un tableau de Tones contenu dans la variable score.

C'est pourquoi on écrit `partition.score = [Tones(), Tones(), ...]`

L'instruction `rospy.Publisher('sound/tones', TonesArray)` permet de définir sur quel topic on va publier.

Enfin l'instruction `pub.publish()` permet d'envoyer notre message sur le topic.

### **3. Morceau de musique :**

Pour jouer un morceau de musique il faut publier le numéro du morceau de musique à jouer sur le topic /sound/play\_song

Pour le faire en ligne de commande :

```
rostopic pub /sound/play_song sound_publisher/SongTitle "song: 0"
```

La procédure pour le faire dans un script python est relativement similaire aux deux autres et ne sera pas détaillée ici.

Voici par contre la procédure à suivre pour ajouter des nouveaux morceaux de musique. Utilisez la commande `roscore sound_publisher` pour vous rendre dans le package sound\_publisher.

Une fois dedans nous allons ouvrir le fichier msg contenant la liste des sons :

```
vim msg/SongTitle.msg
```

Ajoutez une nouvelle ligne avec le nom de votre chanson et le chiffre associé :

```
uint8 Nouvelle_Chanson      = chiffre_d'avant + 1
```

Toute modification d'un fichier msg implique de recompiler le package, revenez à la racine de votre workspace catkin (plus d'explications dans le tutoriel relatif à l'installation de package ROS via catkin), le dossier juste avant src et exécutez la commande :

```
catkin_make.
```

Une fois la compilation finie on peut ajouter le son dans le script python :

```
rosed sound_publisher song_pub.py
```

Appuyez sur la touche i pour ajouter du texte.

Ajouter un cas à la suite de tous les if :

```
elif data.song == data.Nouvelle_chanson:
```

## Il y a ensuite deux solutions

- Partition définie en fréquence :

```
partition.score=[Tones(659,750),  
                 Tones(698.5,250),  
                 Tones(784,1000),  
                 Tones(1046.5,2000),  
                 Tones(587,750),  
                 Tones(659,250),  
                 Tones(698.5,2000),  
                 Tones(784,750)]
```

```
pub_frequency_tones.publish(partition)
```

- Partition définie en Nom + octave :

```
music_score.score=[MusicalTones('do', 5, 500),  
                   MusicalTones('do', 5, 500),  
                   MusicalTones('do', 5, 500),  
                   MusicalTones('re', 5, 500),  
                   MusicalTones('mi', 5, 1000),  
                   MusicalTones('re', 5, 1000),  
                   MusicalTones('do', 5, 500),
```

```
    MusicalTones('mi', 5, 500),  
    MusicalTones('re', 5, 500),  
    MusicalTones('re', 5, 500),  
    MusicalTones('do', 5, 2000)]  
  
pub_musical_tones.publish(music_score)
```

Faites ensuite Echap puis :x entrée pour sauvegarder et quitter.  
Votre nouvelle chanson est maintenant disponible.

## TUTORIEL 8 : Utilisation de dynamic\_reconfigure

### PRÉ-REQUIS :

Pour certaines parties de ce tutoriel, il peut s'avérer nécessaire d'avoir compris le tutoriel 4 : Utilisation de ROS déporté et le tutoriel 6 : Création d'un package ROS.

### SOMMAIRE

- i. Utilisation de dynamic\_reconfigure via Rqt
- ii. Utilisation de dynamic\_reconfigure en ligne de commande
- iii. Modification d'une variable dynamic\_reconfigure dans un nœud ROS en python
- iv. Implémentation d'une variable dynamic\_reconfigure dans un nœud ROS en python

Dynamic reconfigure est un module utilisé par ROS qui permet de modifier à tout moment les paramètres de certains nœuds.

Par exemple, on peut l'utiliser pour modifier la résolution des différentes caméras (RGB, profondeur, infrarouge).

Les tutoriels d'utilisation et de création de variables dynamiques sont inspirés de la documentation officielle de ROS : [http://wiki.ros.org/dynamic\\_reconfigure/Tutorials](http://wiki.ros.org/dynamic_reconfigure/Tutorials) que nous avons voulu clarifier sur certains points. Vous pourrez donc potentiellement y trouver des informations complémentaires telles que l'application d'une variable dynamique à un nœud C++.

### 1. Utilisation de dynamic\_reconfigure via Rqt

Rqt est une interface graphique qui permet de visualiser de nombreux paramètres ROS. On peut y retrouver la liste de tous les topics, mesurer les débits, la fréquence de publication, etc... On peut également l'utiliser pour visualiser les variables modifiables dynamiquement.

Pour l'utiliser il est nécessaire d'avoir au moins un ROS master qui fonctionne sur le PC. Dans notre cas nous allons utiliser le minimal bringup de la camera 3D :

```
roslaunch turtlebot_bringup 3dsensor.launch
```

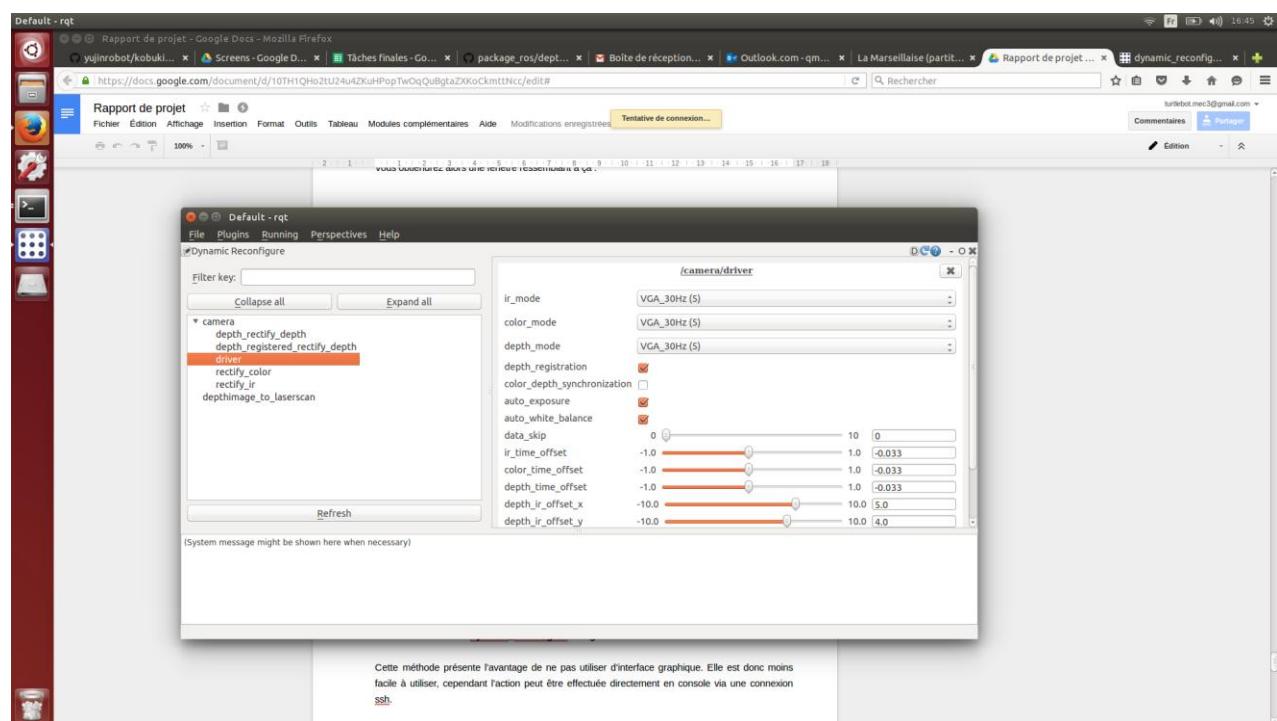
L'application Rqt étant une application "graphique", si vous êtes connecté en ssh via le PC/Carte sur lequel vous lancerez le minimal bringup alors il faudra utiliser un ros\_master déporté comme dans le tutoriel 4. La procédure ne sera pas réexpliquée ici.

Sur votre PC, dans un terminal tapez la commande `rqt`

Vous devriez alors voir apparaître une fenêtre.

Allez dans Plugins > Configuration > Dynamic\_reconfigure

Vous obtiendrez alors une fenêtre ressemblant à ça :



Vous pouvez alors cliquer dans la colonne de gauche sur les différentes rubriques pour accéder aux variables modifiables.

Les variables modifiables sont accessibles dans la fenêtre de droite.

## 2. Utilisation de dynamic\_reconfigure en ligne de commande :

Cette méthode présente l'avantage de ne pas utiliser d'interface graphique. Elle est donc moins facile à utiliser, cependant l'action peut être effectuée directement en console via une connexion ssh.

Pour connaître la liste des nœuds qui ont des paramètres reconfigurable (colonne de gauche dans rqt) il faut exécuter la commande suivante :

```
rosrun dynamic_reconfigure dynparam list
```

On peut également avoir accès aux valeurs actuelles et aux noms des variables des nœuds en tapant la commande :

```
rosrun dynamic_reconfigure dynparam get /nom_du_noeud
```

Par exemple pour le driver de la caméra on obtient la réponse suivante :

```
rosrun dynamic_reconfigure dynparam get /camera/driver
{'use_device_time': True, 'auto_exposure': True, 'color_mode': 5, 'auto_white_balance': True,
'ir_time_offset': -0.033, 'z_offset_mm': 0, 'depth_time_offset': -0.033, 'ir_mode': 5,
'color_time_offset': -0.033, 'depth_registration': True, 'z_scaling': 1.0, 'groups': {'use_device_time': True,
'auto_exposure': True, 'ir_time_offset': -0.033, 'parent': 0, 'z_scaling': 1.0, 'groups': {}},
'z_offset_mm': 0, 'depth_ir_offset_y': 4.0, 'data_skip': 0, 'id': 0, 'depth_ir_offset_x': 5.0, 'ir_mode': 5,
'color_mode': 5, 'auto_white_balance': True, 'name': 'Default', 'parameters': {}},
'depth_time_offset': -0.033, 'depth_mode': 5, 'color_time_offset': -0.033, 'depth_registration': True,
'state': True, 'type': '', 'color_depth_synchronization': False}, 'depth_ir_offset_y': 4.0, 'data_skip': 0,
'color_depth_synchronization': False, 'depth_ir_offset_x': 5.0, 'depth_mode': 5}
```

Le paragraphe est donc clairement moins compréhensible que par rqt, mais il peut vous permettre de retrouver le nom des paramètres et leur valeur actuelle. On peut par exemple identifier ici que le mode “auto\_exposure” est activé.

Pour modifier dynamiquement la variable il faut utiliser la commande :

```
rosrun dynamic_reconfigure dynparam set /nom_du_noeud nom_de_la_variable valeur
```

Pour désactiver le mode auto\_exposure il faudrait donc écrire :

```
rosrun dynamic_reconfigure dynparam set /camera/driver auto_exposure false
```

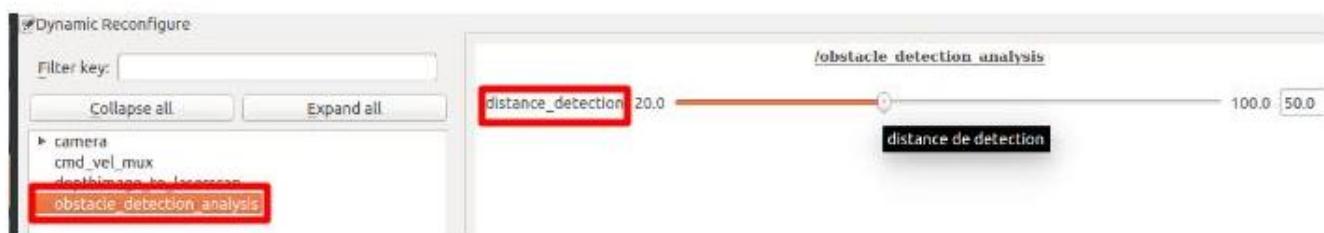
### **3. Modification d'une variable dynamic\_reconfigure dans un noeud ROS**

Pour modifier une variable reconfigurable de manière dynamique au sein d'un script python, il faut tout d'abord importer le client de dynamic\_reconfigure :

```
import dynamic_reconfigure.client
```

Une fois que cette ligne fait partie du code, il faut connaître le chemin du nœud à reconfigurer. Pour cela on peut utiliser la méthode vue précédemment, ou alors ouvrir dynamic\_reconfigure sous rqt et regarder dans la colonne de gauche.

Ensuite il faut connaître le nom de la variable que l'on souhaite modifier. Pour cela on peut regarder la liste des variables dans la colonne de droite.



Dans cet exemple on voit que le chemin du node est /obstacle\_detection\_analysis et la seule variable que l'on peut modifier s'appelle "distance\_detection".

Une fois ces informations acquises, on écrit la ligne :

```
client=dynamic_reconfigure.client.Client("/obstacle_detection_analysis")
```

Cela permet d'initialiser le client de reconfiguration avec l'adresse du nœud à reparamétriser.

La seconde ligne permet d'affecter à la variable sa nouvelle valeur :

```
client.update_configuration({"distance_detection":60})
```

Ici, on a mis distance\_detection à 60.

On peut modifier plusieurs variables en une seule ligne en utilisant la syntaxe :

```
client.update_configuration({"variable1":valeur1, "variable2":valeur2,...})
```

#### 4. Implémentation d'une variable dynamic\_reconfigure dans un nœud ROS

La création d'une variable dynamique pour un nœud ROS implique d'avoir créé, dans un premier temps, un package ROS via Catkin et donc potentiellement d'avoir suivi le tutoriel 5 : création d'un package ROS.

Pour commencer il faut vous rendre dans le package\_ros qui contient le nœud qui va utiliser des variables modifiables dynamiquement.

Pour cela on peut utiliser la commande `roscd nom_du_package`

Une fois dans le dossier du package, il faut créer un dossier cfg qui contiendra tous les fichiers .cfg qui définissent le type des variables modifiables dynamiquement.

Pour créer un fichier de config :

```
touch cfg/mon_fichier_config.cfg
```

Puis éditez-le :

```
vim cfg/mon_fichier_config.cfg
```

Ce fichier est assimilé à un script python, comme tout script python il doit commencer par la ligne suivante :

```
#!/usr/bin/env python
```

Il faut ensuite importer la librairie relative au dynamic reconfigure :

```
from dynamic_reconfigure.parameter_generator_catkin import *
```

Pour pouvoir modifier des paramètres dynamiquement, il faut créer un objet générateur de paramètres :

```
gen = ParameterGenerator()
```

Enfin vous pouvez ajouter autant de variables que vous voulez.

Pour ajouter une variable on utilise la fonction add comme ceci :

```
gen.add('nom_de_la_variable', type_de_la_variable, 0, "description",
        valeur_par_defaut, valeur_min, valeur_max)
```

**Il existe de nombreux types de variables dont :**

- int\_t : entier
- bool\_t : booléen
- str\_t : chaîne de caractère
- double\_t : nombre flottant
- etc. ...

Il est également possible de créer une variable de type énumération, c'est une variable qui peut prendre différents états. On utilise pour cela la fonction enum()

Je vous invite à regarder comment faire dans ce tutoriel :

[http://wiki.ros.org/dynamic\\_reconfigure/Tutorials/HowToWriteYourFirstCfgFile](http://wiki.ros.org/dynamic_reconfigure/Tutorials/HowToWriteYourFirstCfgFile)

Enfin la dernière ligne du fichier doit être la suivante :

```
exit(gen.generate('Nom du package', "nom_pour_la_doc",
"mon_fichier_config"))
```

mon\_fichier\_config correspond au nom donné au fichier.cfg

**Voici un exemple de fichier cfg :**

```
#!/usr/bin/env python
PACKAGE = "camera_detection"
from dynamic_reconfigure.parameter_generator_catkin import *
gen = ParameterGenerator()
gen.add("distance_detection", int_t, 0, "distance de detection", 50, 0, 300)
exit(gen.generate(PACKAGE, "depth_analysis", "depth_analysis"))
```

Ce fichier .cfg nécessite d'être pris en compte pour la compilation du package. Pour cela nous allons devoir éditer le fichier CMakelists.txt et package.xml qui est supposé être déjà à jour comme dans le tutoriel 5 : Création d'un package catkin

Pour cela éditez le fichier package.xml et ajoutez les deux lignes suivantes :

```
<build_depend>dynamic_reconfigure</build_depend>
<run_depend>dynamic_reconfigure</run_depend>
```

Dans le fichier CMakelists.txt :

Il faut dans un premier temps ajouter à la liste des “find\_package(catkin REQUIRED COMPONENTS ..)”  
dynamic\_reconfigure

Il faut ajouter ou décommenter juste après la fonction generate\_messages, la fonction suivante :

```
generate_dynamic_reconfigure_options(
    cfg/mon_fichier_config.cfg)
```

Vous pouvez maintenant recompiler le package avec catkin\_make.

Notez que toute modification du fichier.cfg implique de recompiler le package.

Maintenant pour mettre en place cette variable dynamique, il faut obligatoirement la lancer depuis un nœud ROS. Nous supposerons ici que vous avez déjà un nœud dans le package que nous utilisons et que vous souhaitez y implémenter une variable dynamique.

Pour cela éditez votre script python contenant le nœud et ajoutez-y les lignes suivantes :

```
from dynamic_reconfigure.server import Server
from mon_package_ros.cfg import mon_fichier_configConfig
```

Veuillez noter que pour importer votre fichier .cfg vous devez ajouter l'extension .cfg au nom du package comme nous l'avions fait pour les fichiers .msg. Mais vous devez également rajouter à la fin du nom du fichier sans l'extension le mot "Config" comme ci-dessus.

Il faut maintenant ajouter dans votre fonction main l'instruction qui permettra de lancer le serveur dynamic\_reconfigure. Le serveur est l'outil qui rend votre variable modifiable et qui vous permet surtout d'appeler une fonction dans votre code si elle est modifiée.

```
srv = Server(mon_fichier_configConfig, callback)
```

Enfin il faut définir la callback qui sera appelée en cas de modifications :

```
def callback(config, level):
    return config
```

Le return config est indispensable au bon fonctionnement du système. Vous pouvez également stocker la variable modifiée dans une variable globale en déclarant une variable au début de votre script python :

```
ma_variable = 0
```

Et en complétant la fonction comme ceci :

```
def callback(config, level):
    global ma_variable
    ma_variable = config.nom_de_la_variable_dans_le_fichier_cfg
    return config
```

Vous pouvez maintenant lancer votre script python soit via un roslaunch soit via l'instruction python votre\_script\_python et votre variable devrait être disponible dans rqt.

## TUTORIEL 9 : Manuel d'utilisation de notre Turtlebot

### PRÉ-REQUIS :

Pour suivre ce tutoriel vous devez avoir en votre possession :

- Le robot Turtlebot avec sa carte Odroid
- Le routeur wifi émettant le réseau : TurtlebotNetwork2
- Un PC avec linux et une connexion wifi

Première étape : branchez le routeur wifi

Une fois celui-ci actif, allumez le robot. La Odroid devrait démarrer automatiquement avec.

Sur votre PC connectez-vous au réseau wifi TurtlebotNetwork2

Ouvrez un terminal puis connectez-vous en ssh à la Odroid :

```
ssh odroid@odroid.local  
mot de passe : odroid
```

Vous remarquerez que le mot de passe ne s'affiche pas quand vous le saisissez. Pas d'inquiétude, c'est normal, continuez de le saisir.

Si lors de la connexion à la carte Odroid, le message d'erreur suivant apparaît :

```
quentin@quentin:~$ ssh odroid@odroid.local  
odroid@odroid.local's password:  
Last login: Tue Jan  1 02:48:23 2013 from 192.168.0.4  
terminate called after throwing an instance of 'std::runtime_error'  
  what():  locale::facet::_S_create_c_locale name not valid  
odroid@odroid:~$ █
```

Ou à tout autre moment (en particulier lors de l'utilisation de la compléction), celui-là :

```
odroid@odroid:~$ rosed camerterminate called after throwing an i  
:runtime_error'  
what():  locale::facet::_S_create_c_locale name not valid
```

Coupez alors la connexion ssh à la carte : CTRL-D dans le terminal

Puis tapez la commande suivante dans le terminal de votre ordinateur :

```
echo "export LC_ALL=\"en_US.UTF-8\" >> ~/.bashrc
```

Fermez le terminal et rouvrez en un nouveau.

Le message d'erreur ne devrait alors plus apparaître, vous pouvez vous reconnecter à la Odroid.

Enfin exécutez la commande :

```
roslaunch turtlebot_scenario turtlebot_scenario.launch
```

Cette commande lancera notre scénario.

Pour rappel notre scénario permet au robot de fonctionner de manière autonome, il va chercher à avancer tout droit et éviter les obstacles qu'il rencontrera. Pour cela il peut détecter des obstacles avec ses bumpers ou alors avec la caméra de profondeur pour faire des "évitements sans contacts".

Si vous tapez dans le bumper central, il lèvera la caméra pour essayer de reconnaître un visage. S'il repère un visage il effectuera un tour sur lui-même sinon il repartira.

De temps en temps, de manière aléatoire, le robot s'arrêtera pour chanter une chanson.

Si jamais vous tapez trois fois dans le bumper central alors le robot effectuera une action cachée.

## TUTORIEL 10 : Commandes de base ROS

### PRÉ-REQUIS :

Pour suivre ce tutoriel vous devez disposer d'un appareil avec Linux ainsi que les fonctions de base de ROS.

Ce tutoriel a pour but de vous apprendre quelques commandes qui vous simplifieront la tâche si vous souhaitez commencer à développer ou éditer des packages ROS. Il ne nécessite pas de compétence ou de matériel particulier si ce n'est un ordinateur avec ROS d'installé.

**Comme vous avez pu sans doute le découvrir, ROS est constitué de plusieurs éléments :**

- Des packages
- Des launchfiles
- Des messages
- Des scripts avec des nœuds
- Des fichiers de config .yaml
- ...

Tous ces fichiers étant situés à des endroits différents et regroupés par package dont on ne connaît pas forcément la localisation, il est parfois très difficile de s'y retrouver.

Pour lancer un package ROS et les nœuds associés :

`roslaunch nom_du_package nom_du_fichier_launch.launch`

Pour accéder au dossier contenant tous les fichiers d'un package ROS vous pouvez utiliser la commande :

`roscd nom_du_package`

Vous vous retrouverez directement dans le dossier du package

Pour éditer un fichier d'un package vous pouvez utiliser directement la commande

`rosed nom_du_package nom_du_fichier`

Vous pouvez même connaître les fichiers éditables dans le package en utilisant la complétion. Tapez plusieurs sur la touche Tab quand vous avez écrit rosed nom\_du\_package dans votre terminal. Il devrait alors vous proposer une liste de fichier comme ci-dessous :

```
quentin@quentin:~$ ssh odroid@odroid.local
odroid@odroid.local's password:
Last login: Tue Jan  1 02:57:12 2013 from 192.168.0.4
odroid@odroid:~$ 
odroid@odroid:~$ roscd camera_detection
camera_detection.launch      depth_analysis.py
CMakeLists.txt                depth_analysis.pyc
depth_analysis.cfg            ObstacleDetection.msg
depth_analysis_config.cfg    package.xml
depth_analysis.launch        server.py
odroid@odroid:~$ roscd camera_detection
camera_detection.launch      depth_analysis.py
CMakeLists.txt                depth_analysis.pyc
depth_analysis.cfg            ObstacleDetection.msg
depth_analysis_config.cfg    package.xml
depth_analysis.launch        server.py
odroid@odroid:~$ roscd camera_detection
```

Pour connaître le contenu d'un message ROS vous pouvez utiliser la commande :

`rosmsg show nom_du_message.msg`

```
quentin@quentin:~$ rosmsg show BumperEvent.msg
[kobuki_msgs/BumperEvent]:
uint8 LEFT=0
uint8 CENTER=1
uint8 RIGHT=2
uint8 RELEASED=0
uint8 PRESSED=1
uint8 bumper
uint8 state

quentin@quentin:~$
```

L'avantage de cette commande est qu'elle ne nécessite pas de connaître le package du message.

Pour retrouver le package vous pouvez utiliser la commande :

`locate NomDuMessage.msg`

Enfin une dernière fonction très utile est `rostopic` avec les paramètres suivants :

- `bw` : permet de connaître la bande passante prise par un topic  
`rostopic bw /nom_du_topic`

- *echo* : permet d'afficher les messages d'un topic  
`rostopic echo /nom_du_topic`
- *hz* : permet de connaître la fréquence de publication sur un topic  
`rostopic hz /nom_du_topic`
- *list* : permet de voir tous les topics actifs  
`rostopic list`
- *info* : permet de connaitre qui publie et qui souscrit sur un topic  
`rostopic info nom_du_topic`

Il existe d'autres commandes si cela vous intéresse, certaines d'entre elles sont expliquées ici : <http://wiki.ros.org/ROS/CommandLineTools>, mais les plus importantes ont été évoquées ci-dessus.