ELE3000 — Projet personnel en génie électrique

PROJET :
Contrôle d'un AR Drone via une interface Leap Motion

Par :
BOUCHER, Éric (1618053)

Présenté à :
SAUSSIÉ, David

Date de la remise du rapport : 17 août 2015
École Polytechnique de Montréal

Table des matières

REMERCIEMENTS

Pour commencer, j'aimerais remercier Élikos d'avoir soumis ce projet. Sans eux, jamais je n'aurai pu me lancer dans un projet aussi intéressant et instructif que celui-ci. Sans avoir directement participé au projet, ils m'ont initialement dirigé en me présentant brièvement ROS et Leap Motion.

J'aimerais aussi remercier mon directeur de projet, David Saussié, pour sa présence durant la session pour nos rencontres autant que pour sa présence par courriel lorsque des problèmes de dernières minutes sont survenus.

Finalement, j'aimerais aussi remercier quelques autres personnes qui m'ont non seulement aidé à tester le côté intuitif de mon projet en effectuant quelques vols d'essai, mais aussi en me pointant certains problèmes que je pourrais corriger. De plus, ces personnes ont gentiment accepté d'être filmées durant leurs vols pour faire partie d'un montage que j'ai présenté le 11 août dernier lors de la présentation finale. Bref, un grand merci à Loïc Sanschagrin-Thouin, Léonardo Tjandra et ma douce moitié, Laurence Miron-Lefort pour m'avoir tant aidé.

I. INTRODUCTION

Lors des premiers cours de la session d'été, Élikos, une société technique de Polytechnique qui travaille principalement avec des quadricopters, est venu faire une courte présentation des projets qu'ils proposaient dans le cadre de leur défi à relever. Des projets aussi variés que la caractérisation des propulseurs d'un quadricopter, le suivi d'un robot équipé d'un marqueur AR ou même le développement d'un chargeur de batteries ont été proposés aux élèves du cours ELE3000.

Dans une liste d'environ sept projets soumis, un seul ressortait du lot à mes yeux et venait vraiment me chercher : le contrôle d'un drone Parrot via une interface Leap Motion. Malgré qu'à priori, je ne connaissais ni les drones ni ce qu'était le système Leap Motion, la simple lecture du résumé du projet m'intéressait au plus haut point. Il s'agissait, en résumé, de piloter un drone en modélisant la main d'une personne.

Une chose m'échappait encore avant de décider de me lancer dans ce projet. Pourquoi? Une télécommande est pourtant très efficace. Alors, quel était l'avantage de piloter un drone par modélisation?

La réponse : Pour des présentations grand public! Par exemple, laisser des enfants piloter un drone de grande valeur à l'aide d'une télécommande tout aussi dispendieuse peut sembler absurde. Par contre, s'ils ne touchent strictement à rien et que leurs commandes par modélisation sont soumises à un programme de surveillance, cela semble plus plausible. Sachant qu'Élikos obtient une grande partie de leur soutien financier par ces apparitions grand public, un tel système permettrait d'attirer encore plus l'attention du public.

Bref, avec une meilleure idée des motifs du projet, j'ai décidé de me lancer dans ce projet pour de bon. Malgré l'ampleur qu'il semblait avoir, ça semblait être une idée très intéressante, qui une fois complétée serait définitivement très amusante à présenter.

II. BESOINS ET CONTRAINTES

Lors de la soumission du projet par Élikos, qui est dans le cadre de ce projet l'équivalent d'un client, certains besoins et certaines contraintes furent clairement identifiés pour permettre d'atteindre le résultat désiré.

Premièrement, le système devra pouvoir effectuer un ***décollage et un atterrissage automatique****. Il n'est pas nécessaire que le pilote ait à tout faire le travail du décollage et de l'atterrissage puisqu'il s'agit d'une manœuvre récurrente plutôt simple à automatiser. Ainsi, on désire qu'une fois le système en marche, une simple commande de l'utilisateur qui agira comme une commande "ON/OFF" permettra un décollage et un atterrissage automatique.*

Deuxièmement, une fois en vol, l'AR Drone de Parrot qui sera utilisé supporte des commandes suivant ***quatre axes de liberté*** *soit avant/arrière, gauche/droite, haut/bas et une rotation horaire/antihoraire par rapport à l'axe haut/bas. Ainsi, le système devra être capable de capter au minimum 4 axes de liberté parvenant de la main du pilote. S'il y a plus de quatre axes de liberté qui sont captés, un choix s'imposera sur ceux qui semblent les plus intuitifs.*

Troisièmement, lors d'un vol piloté par des gens du public, ***un périmètre de vol*** *devra être érigé, mais surtout respecté pour une question de sécurité. En effet, non seulement le matériel a une grande valeur et ne doit pas être endommagé mais, en plus, la puissance des moteurs d'un quadricopter est telle qu'il est possible de blesser gravement quelqu'un s'il y a contact avec une des quatre hélices. Bref, le système devra être en mesure d'obtenir des informations sur la position du drone et le restreindre à un périmètre très précis. S'il venait à s'approcher à moins de deux mètres des limites, le système devrait modifier les commandes envoyées par le pilote pour corriger la trajectoire.*

Quatrièmement, lors d'une présentation, un copilote sera toujours présent pour surveiller si le vol se passe bien. Si quelque chose cloche et que le pilote semble soit avoir perdu le contrôle ou qu'il semble vouloir faire une manoeuvre risquée, le copilote doit toujours avoir la possibilité d'enclencher ***un arrêt d'urgence manuel****.*

Dans le cas où ces quatre contraintes sont respectées et qu'un système de pilotage par modélisation via une interface Leap Motion est bel et bien fonctionnel pour contrôler un AR Drone de Parrot, le projet pourra être considéré comme étant réussi et complété.

Dans un dernier temps, si toutes ces contraintes sont remplies, un besoin optionnel intéressant à travailler le plus possible serait que le système soit ***le plus intuitif possible****. Si possible, qu'aucune formation et/ou explication préalable ne soit nécessaire pour bien piloter le drone.*

III. SPÉCIFICATIONS FONCTIONNELLES

Entrée :

- *Leap Motion via USB*

Sortie :

- *AR Drone 1.0 (Parrot) via WIFI*

Fonction :

- *Analyser les mouvements d'une main pour piloter un drone*

Facteurs humains :

- *L'utilisateur doit être limité à un pilotage sécuritaire (périmètre de vol)*

Réactions aux erreurs :

- *Arrêt d'urgence manuel en cas d'erreur grave (humain ou logiciel)*

Pour remplir le mandat, voici certaines spécifications qui seront utilisées :

Note : La position « initiale » ou « de repos » est lorsque la main est directement au-dessus du contrôleur Leap Motion à environ 6 pouces, à plat (normale à la paume perpendiculaire au contrôleur Leap Motion), les doigts pointant vers l'avant.

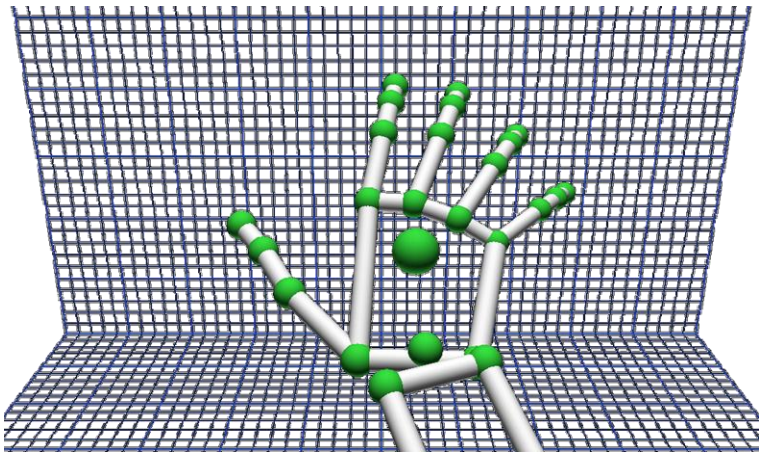

Figure 1 : Position « initiale » ou « de repos » vu lors de la modélisation

1) Lors de la détection d'une main, il y aura décollage automatique. Tant que la main est détectée, le vol se poursuit. Si la main n'est plus détectée à un moment où le drone est en vol, ce dernier passe en pilotage automatique pour se stabiliser puis il procède à un atterrissage automatique.

2) Si la main se rapproche du contrôleur Leap Motion, le drone diminue son altitude tant et aussi longtemps que la main ne revient pas à la position « initiale ». Si au contraire, celle-ci s'éloigne du contrôleur Leap Motion, le drone augmentera progressivement son altitude tant qu'elle ne revient pas à sa position de repos. (ALTITUDE)

3) Si la main avance ou recule par rapport au contrôleur Leap Motion ou si celle-ci pivote par rapport à l'axe gauche/droite, le drone avancera ou reculera en conséquence jusqu'au retour à la position de repos.

4) Si la main se déplace vers la gauche/droite ou qu'elle pivote par rapport à l'axe avant/arrière, le drone se déplacera latéralement jusqu'à ce que la main revienne à la position initiale.

5) Si la main pivote par rapport à l'axe haut/bas, le drone tournera par rapport au même axe jusqu'à ce que la main revienne à la position de repos.

6) Si une commande est envoyée du clavier (i.e. la touche « Enter »), l'arrêt d'urgence manuelle devra provoquer une stabilisation du drone (pilotage automatique) jusqu'à nouvel ordre. Ensuite, une seconde commande pourra soit redonner le contrôle au pilote soit provoquer un atterrissage d'urgence.

IV. CONTRÔLEUR LEAP MOTION

Le contrôleur Leap Motion est un dispositif que l'on peut acheter par internet au coût d'approximativement 100 $ qui permet de modéliser facilement une ou plusieurs mains à l'aide de trois caméras infrarouge. Ces caméras n'ayant pas exactement le même angle ni la même position, le microcontrôleur intégré au dispositif peut, par la suite, calculer à l'aide d'un modèle 3D d'une main préalablement enregistré toutes les informations suivantes :

1) La position de toutes les extrémités de chaque phalange, la position de la paume ainsi que la position de quelques points importants du poignet. Bref, on peut obtenir en temps réel toutes les positions que nous pourrions désirer concernant une main (comme on peut le voir par les points verts dans la figure 1).

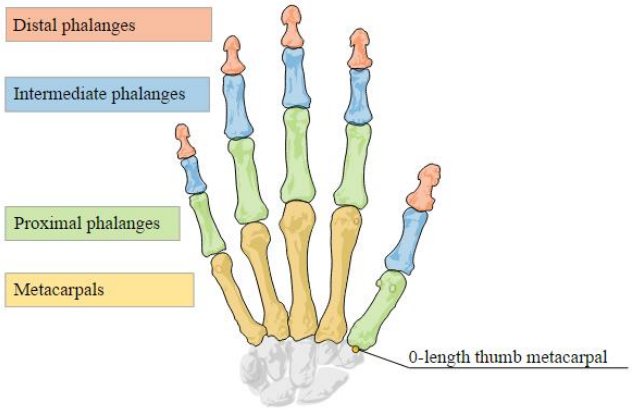

Figure 2 : Phalanges telles que reconnues et identifiées par le Leap Motion

2) La direction de chaque doigt donnée sous forme de vecteurs (tel qu'on l'observe dans la figure 2 ci-dessous) ainsi que la normale à la paume.

Figure 3 : Vecteurs de directeur de chaque doigt tel que fourni par le Leap Motion

3) Certains gestes très précis (voir figure 3). Une fois reconnu, un simple avertissement est envoyé par le Leap Motion. La liste complète des gestes peut être retrouvée sur le site

officiel de Leap Motion. Elle n'est pas fournie dans ce rapport puisque ce n'est pas utilisé dans le cadre de ce projet.

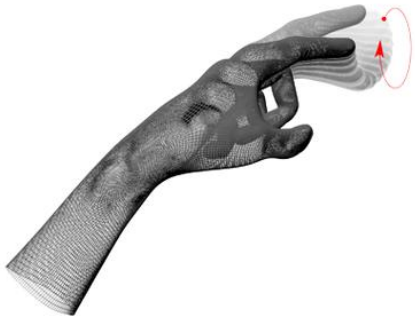

Figure 4 : Exemple de geste reconnu par le Leap Motion (rotation de l'index)

4) Certains accessoires peuvent aussi être détectés par le Leap Motion pour permettre une interaction différente, lorsqu'utilisé dans le cadre de modélisation d'une réalité virtuelle.

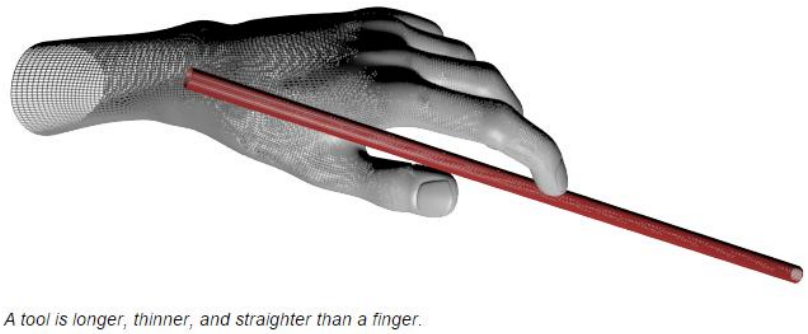

Figure 5 : Exemple d'accessoire qui serait détecté par le Leap Motion

5) Finalement, il est aussi possible de détecter certains « mouvements » particuliers. Par exemple, une translation de la main, une rotation, une homothétie, etc.

Note : Dans le cadre de ce projet, seulement les informations 1 et 2 sont utilisées. Les gestes, les accessoires ainsi que les mouvements ne sont même pas pris en compte lors de la collecte des informations. L'analyse peut être faite simplement à l'aide des positions et des directions.

V. AR DRONE ET SES CARACTÉRISTIQUES

Le drone que l'on pilotera lors de la présentation de ce projet sera un AR Drone 1.0 de Parrot. Il est vrai que ce projet qui servira principalement à Élikos ne sera pas forcément jumelé à un AR Drone dans le futur mais, à ce moment-là, il sera possible de modifier facilement le projet pour l'adapter au nouveau drone. Pour l'instant, nous nous en tiendrons simplement à ce drone (voir figure 6 ci-dessous).

Figure 6 : Exemple d'AR Drone de Parrot. Le style et les couleurs de la protection peuvent varier.

Les avantages d'utiliser ce drone par rapport à d'autres sont nombreux. Pour n'en nommer que quelques un : l'autonomie du contrôle du drone, sa protection/sa sécurité, son faible coût, l'autonomie de sa batterie et surtout sa simplicité.

En effet, la stabilisation du vol étant déjà faite directement sur le contrôleur du drone, si l'on désire le piloter, on n'a qu'à envoyer les commandes que l'on souhaite. Le nombre de commandes est assez limité : décollage automatique, atterrissage automatique et vitesse selon ses quatre axes de liberté (translation en altitude, translation latérale, translation frontale et rotation par rapport à l'axe d'altitude).

Ce drone est aussi très sécuritaire lorsqu'on y ajoute sa protection en mousse. Les hélices étant protégées, s'il y a une collision, non seulement le drone a peu de chance de se briser mais, si le drone frappe quelqu'un, la personne aussi ne sera pas blessée.

Finalement, avec une autonomie de vol d'environ 15 minutes sans arrêt et son coût d'environ 400 $, ce drone étant sans contredit un des meilleurs qu'il était possible de se permettre. En plus, il s'agissait d'un drone que la Polytechnique de Montréal possédait déjà.

VI. ROS : SYSTÈME D'EXPLOITATION IDÉAL

Une fois les composantes essentielles du système choisies, il était temps de se pencher sur la partie logiciel. Avant tout, il était nécessaire de choisir un système d'exploitation sur lequel on programmerait. Le choix n'a pas été très long puisqu'Élikos travaille déjà sur ROS, acronyme pour Robotic Operating System, qui est sans contredit un des systèmes d'exploitation les plus importants à connaître en robotique.

ROS est un système d'exploitation qui est facile à utiliser sur Linux et sur n'importe quel robot qui permet entre autres une communication facile et performante entre différents systèmes. Dans notre cas, puisque l'on veut communiquer avec le Leap Motion et avec l'AR Drone tout en effectuant nos propres calculs, il s'agit de toute évidence d'un bon choix.

Une des grandes forces de ROS tient du fait qu'il permet à des programmes codés en différents langages de programmation de communiquer ensemble. Par exemple, nous verrons bientôt que ce projet nécessitera 2 programmes/noeuds déjà faits soit leapmotion et ardrone_autonomy. Nous communiquerons avec ces deux entités facilement malgré que le premier soit codé en Python et le second en C++.

L'apprentissage de ROS n'a pas été de tout repos. Par contre, une fois appris complètement, ROS est plutôt intuitif et efficace. Des tutoriaux officiels ont été suivis pour parvenir à comprendre ROS dans son entièreté. Le lien vers ces tutoriaux est la troisième référence dans la liste des références en annexe à ce document.

ROS peut facilement être décortiqué en quatre composantes fondamentales :

1) **Les nœuds.** Ceux-ci sont en fait les programmes, les exécutables, que l'on va nous-mêmes programmer pour effectuer les tâches que l'on désire accomplir. Il s'agit d'un élément essentiel à la mise sur pied d'un système sous ROS.

2) **Les sujets.** Ces derniers sont tout simplement des files d'information. Les nœuds étant les programmes, les sujets sont le moyen par lequel les nœuds peuvent facilement communiquer entre eux. Un nœud peut s'abonner à un sujet ou y publier. S'il s'y abonne, il recevra tout message qui sera publié sur celui-ci. S'il y publie, son message sera, par la suite, envoyé à tous les nœuds abonnés à ce même sujet.

3) **Les messages.** Il s'agit en fait d'une structure de donnée qui indique à quiconque publie ou s'abonne à un sujet quel genre de message circule sur celui-ci et surtout comment il est structuré.
4) **Les services.** Ceux-ci agissent presque comme des nœuds, mais ils ne sont pas actifs en tout temps. Ils sont l'équivalent d'un serveur. On leur envoie une demande, ils vont la traiter puis y répondre.

Dans le cadre de ce projet, seulement les trois premières composantes seront utilisées (composantes essentielles à ROS) tandis que la dernière nous serait inutile.

Une dernière chose est essentielle pour bien saisir ROS. Parmi tous les nœuds, un est différent : Le ROS Master. Celui-ci est créé en même temps que ROS démarre. Il s'agit du nœud de base. Chaque fois qu'un autre nœud/sujet « naît », il s'identifie d'abord au ROS Master pour que celui-ci connaisse son existence et sa position. Dès qu'un second nœud est démarré et qu'il partage le même sujet qu'un premier nœud, le ROS Master les met en relation via une connexion « Peer To Peer ».

À partir de ce point, ROS est plutôt facile à comprendre. Lors de la compilation des nœuds, ROS utilise la structure des « messages » pour que, peu importe le langage de programmation utilisé, les données puissent être comprises lorsqu'elles sont publiées sur un sujet ou reçues par un sujet. Par la suite, les « noeuds » sont démarrés manuellement ou via un fichier « lunch », ils s'identifient au ROS Master, s'abonnent à leurs « sujets », établissent leur connexion P2P avec les autres nœuds grâce aux informations du ROS Master puis ils s'exécutent comme ils devraient le faire.

VII. DESIGN PRÉLIMINAIRE

Avec toutes ces informations, le design du système pouvait enfin être fait. Malgré qu'il s'agisse d'un design plutôt simple à priori, toute l'information que l'on devait acquérir pour bien le comprendre fut très ardue à bien saisir.

Le système est simple. Le Leap Motion communiquera via USB avec un nœud déjà semi-codé que l'on ajoutera à ROS qui s'appelle tout simplement « leapmotion ». Ce noeud publiera toute l'information sur la main sur un sujet auquel on s'abonnera à partir de notre nœud principal : noeud d'analyse. Celui-ci fera, ensuite, tous les calculs nécessaires pour, au final, publier les commandes envoyé au drone sur différents sujets (ardrone_autonomy/land, ardrone_autonomy/takeoff et cmd_vel). Un dernier nœud sera installé puis démarré qui s'appellera « ardrone_autonomy ». Il s'abonnera aux trois sujets précédents tout en publiant les données de vol sur un nouveau sujet. Ce noeud déjà codé s'occupera de communiquer avec l'AR Drone via WIFI pour envoyer les commandes.

Tout ce système peut être résumé en un schéma plutôt simple que voici.

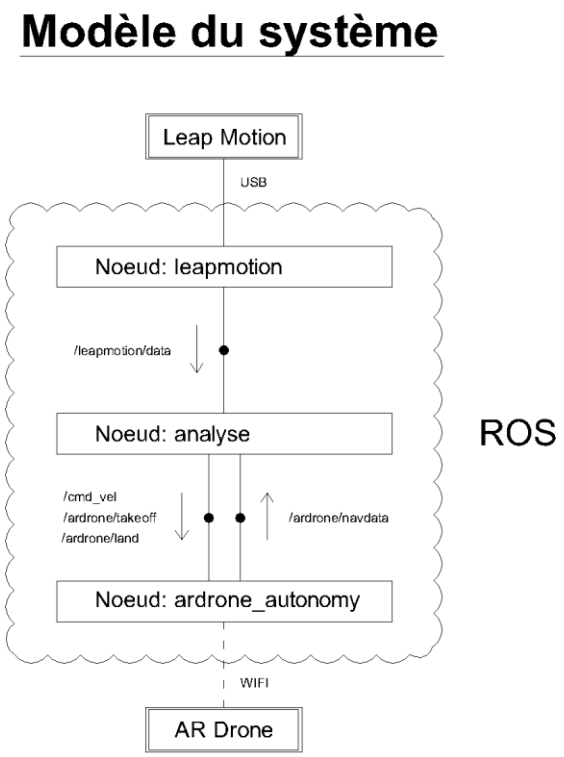

Figure 7 : Design préliminaire du système.

VIII. ROS INDIGO, POURQUOI?

Ce système est très simple et élégant à priori, mais il manque une information pertinente. Quelle version/distribution de ROS sera utilisée?

En effet, de nombreuses distributions existent et différents nœuds ne sont pas forcément compatibles avec chacune d'entre elles. Certaines distributions plus récentes comme « Jade » offrent de nouvelles fonctionnalités, mais elles limitent aussi les nœuds à ceux qui ont été adaptés à cette distribution.

Inversement, une version trop vieille n'offre pas forcément les fonctionnalités nécessaires au bon fonctionnement de notre projet.

Après en avoir essayé plusieurs et m'être informé sur chacune d'entre elles, j'en ai conclu que la distribution la plus **récente** qui est **compatible** avec les noeuds « leapmotion » et « ardrone_autonmy » est en fait nul autre que ROS Indigo.

Cette version de ROS a été mise en ligne le 22 juillet 2014 (assez récent) mais, contrairement à Jade qui est en ligne depuis le 23 mai 2015, ROS Indigo est rétro compatible avec les nœuds qui nous intéressent.

IX. PRÉPARATION ET INSTALLATIONS

Pour que le système soit prêt en emploi, en supposant que Linux (Ubuntu) est déjà installé, trois paquets doivent être installés : ROS Indigo, Leapmotion, Ardrone_autonomy.

ROS Indigo

Trois étapes sont nécessaires pour installer ROS :

1) Préparer « sources.list ».
2) Préparer les clés.
3) Faire l'installation

Pour effectuer ces trois étapes, sous Ubuntu, dans un terminal :

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Puis,

```
sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net --recv-key 0xB01FA116
```

Et, finalement, les commandes suivantes.

```
sudo apt-get update
sudo apt-get install ros-indigo-desktop-full
```

À ce point-ci, ROS Indigo est installé. Pour accélérer l'utilisation de l'environnement de ROS ainsi qu'avoir certaines fonctionnalités supplémentaires :

```
sudo rosdep init
rosdep update
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
source ~/.bashrc
sudo apt-get install python-rosinstall
```

Pour démarrer ROS à partir d'ici, il suffit d'initialiser le ROS Master avec la commande :

```
roscore
```

Leapmotion

Une fois ROS Indigo installé, rien de plus simple qu'installer le noeud « Leapmotion ».

Par contre, celui-ci utilise le SDK officiel de Leap Motion. Alors, il faudra préalablement aller sur le site officiel de Leap Motion soit « https://www.leapmotion.com/ » puis installer le kit de développement. Cette installation demande de s'inscrire sur le site.

Une fois installé, une simple commande permet d'initialiser le pilote du Leap Motion s'il n'est pas déjà en marche :

```
leapd
```

Maintenant, pour le nœud qui nous intéresse sous ROS, il suffit d'effectuer la commande suivante :

```
sudo apt-get install ros-indigo-leap-motion
```

C'est fait, le noeud « leapmotion » est installé.

Pour démarrer ce noeud, il suffit dans un terminal de faire les trois commandes suivantes (pour un système 64 bits) :

```
export PYTHONPATH=$PYTHONPATH:$HOME/LeapSDK/lib:$HOME/LeapSDK/lib/x64
source /opt/ros/hydro/setup.bash
rosrun leap_motion sender.py
```

Ardrone_autonomy

De la même façon qu'on installe le noeud « leapmotion », « ardrone_autonomy » nécessite qu'une commande très simple :

```
sudo apt-get install ros-indigo-ardrone-autonomy
```

Et voilà! Maintenant, il suffit de se connecter sur le réseau du drone, et démarrer le nœud dans un nouveau terminal :

```
rosrun ardrone_autonomy ardrone_driver
```

X. CRÉATION DE NOTRE PAQUET SOUS ROS

Maintenant que tout l'essentiel est installé, la conception va bientôt commencer.

D'abord, il faut créer notre nouveau paquet qui contiendra entre autres notre nœud d'analyse.

Pour ce faire, nous allons nous créer un répertoire de travail comme il est conseillé de le faire sur le tutoriel de ROS :

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

Et, construire le tout malgré que ce soit vide pour vérifier le bon fonctionnement de ROS.

```
cd ~/catkin_ws/
catkin_make
```

Ensuite, charger le « .bash » pour configurer notre terminal convenablement.

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

Nous pouvons maintenant créer le paquet.

```
cd ~/catkin_ws/src
catkin_create_pkg leap_ardrone_controller std_msgs rospy roscpp
```

Négligeons les dépendances pour l'instant.

Le paquet devrait maintenant se compiler.

```
cd ~/catkin_ws
catkin_make
```

Maintenant que le paquet est créé, si jamais on fermait le terminal, on devrait toujours refaire la commande suivante dans le nouveau terminal dans lequel nous voudrions programmer notre paquet :

```
source ~/catkin_ws/devel/setup.bash
```

Pour configurer notre paquet convenablement, quelques petits détails restent à régler.

Nous devons régler les dépendances. Pour ce faire, ouvrir le fichier « package.xml » qui se trouve « `~/catkin_ws/src/leap_ardrone_controller/` » et remplacer TOUT par le code suivant :

```
<?xml version="1.0"?>
<package>
  <name>leap_ardrone_controller</name>
  <version>1.0.0</version>
  <description>Fichier de configuration</description>
  <maintainer email="eric.boucher@polymtl.ca">Éric Boucher</maintainer>

  <license>BSD</license>

  <build_depend>message_generation</build_depend>
  <run_depend>message_runtime</run_depend>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>leap_motion</build_depend>
  <build_depend>geometry_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>leap_motion</run_depend>
  <run_depend>geometry_msgs</run_depend>

  <export> </export>
</package>
```

Maintenant, on doit faire de même avec le fichier qui se trouve à côté soit « CMakeLists.txt ».

Il faut l'ouvrir puis tout remplacer par :

```
cmake_minimum_required(VERSION 2.8.3)
project(leap_ardrone_controller)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  geometry_msgs
  message_generation
  leap_motion
)
add_message_files(
   FILES
   leap.msg
   leapros.msg
 )
generate_messages(
   DEPENDENCIES
   std_msgs
   geometry_msgs
 )

catkin_package(
  CATKIN_DEPENDS message_runtime
)
include_directories(
  ${catkin_INCLUDE_DIRS}
)

add_executable(analyse src/analyse.cpp)
target_link_libraries(analyse ${catkin_LIBRARIES})
add_dependencies(analyse ${catkin_EXPORTED_TARGETS})
```

À présent, les fichiers de configurations étant bien codés, nous allons créer les messages. Dans un terminal,

```
cd ~/catkin_ws/src/leap_ardrone_controller
mkdir msg
touch leap.msg
touch leapros.msg
```

Les fichiers étant créés, ouvrez « leap.msg » et mettre le code suivant dans le fichier :

```
Header header


float64[3] hand_direction
float64[3] hand_normal
float64[3] hand_palm_pos
float64 hand_pitch
float64 hand_roll
float64 hand_yaw
```

Ouvrez « leapros.msg » et mettre le code suivant dans le fichier :

```
Header header
geometry_msgs/Vector3 direction
geometry_msgs/Vector3 normal
geometry_msgs/Point palmpos
geometry_msgs/Vector3 ypr
```

Maintenant, que les messages existent et sont bien configurés, nous pouvons passer au morceau principal. Nous allons créer le fichier source « analyse.cpp ».

Effectuer les commandes suivantes pour commencer :

- `cd ~/catkin_ws/src/leap_ardrone_controller`
- `mkdir src`
- `touch analyse.cpp`

Il faut maintenant ouvrir le fichier analyse.cpp et coller le code du programme principal (le code va jusqu'à la page 27) :

```
#include "ros/ros.h"
#include "ros/time.h"

#include "std_msgs/Empty.h"
#include "std_msgs/String.h"
#include "visualization_msgs/MarkerArray.h"
#include "geometry_msgs/PoseStamped.h"
#include "geometry_msgs/Twist.h"
#include "sensor_msgs/Image.h"

#include "leap_motion/leapros.h"
#include "ardrone_autonomy/Navdata.h"

#include "std_msgs/Time.h"
#include <math.h>

#include <sstream>

ros::Publisher droneCommandTakeOff;
ros::Publisher droneCommandLand;
ros::Publisher droneCommandVel;

int droneStatus = -1;

          // -1 - No Status yet
          // 0 - Initialized, ready to update actual status
          // 2 - Landed
          // 3 - Flying
          // 4 - Flying / stable
```

```
          // 6 - Taking off
          // 7 - Flying
          // 8 - Landing

// Tableau de restriction des commandes (périmètre de vol)
int blockingCmd[8] = {0}; // if "0", no block. if "1", blocking the command.

          // 0 - Front command
          // 1 - Back command
          // 2 - Left command
          // 3 - Right command
          // 4 - Lift command
          // 5 - Land command
          // 6 - Rotate left command
          // 7 - Rotate right command

// Le programme donne 3 avertissements pour le niveau de la batterie
bool batteryLowCalled[3] = {false, false, false};

// Arrêt d'urgence (toutes les commandes sont coupées)
bool paused = false;

// Fonction de gestion de l'arrêt d'urgence
void pause(const std_msgs::Empty){
          if(paused){
                    paused = false;
                    ROS_INFO("Resume.");
                    for(int i = 0; i < 8; i++)
                              blockingCmd[i] = 0;
          }
          else{
                    paused = true;
                    ROS_INFO("Paused.");
                    for(int i = 0; i < 8; i++)
                              blockingCmd[i] = 1;
          }
}

// Fonction d'analyse des données de vol
```

```
void navAnalyses(const ardrone_autonomy::Navdata& msg)
{
	// Vérification du status du drone
	int initStatus = droneStatus;

	// Alerte potentielle du niveau de la batterie
	if(msg.batteryPercent == 0.0){}
	else if(msg.batteryPercent < 5.0 && batteryLowCalled[2] == false){
		batteryLowCalled[0] = true;
		batteryLowCalled[1] = true;
		batteryLowCalled[2] = true;
		ROS_INFO("Battery is dead (under 5%%)");
	}
	else if(msg.batteryPercent < 15.0 && batteryLowCalled[1] == false){
		batteryLowCalled[0] = true;
		batteryLowCalled[1] = true;
		ROS_INFO("Battery critically low for flight (under 15%%)");
	}
	else if(msg.batteryPercent < 30.0 && batteryLowCalled[0] == false){
		batteryLowCalled[0] = true;
		ROS_INFO("Battery low (under 30%%)");
	}

	// Blocage en altitude (seul blocage possible avec NavData)
	if(!paused){
		if(msg.altd > 1600)
			blockingCmd[4] = 1;
		if(msg.altd < 1600)
			blockingCmd[4] = 0;

		if(msg.altd < 100)
			blockingCmd[5] = 1;
		if(msg.altd > 100)
			blockingCmd[5] = 0;
	}

	// Si tout est initialisé, on met toujours à jour le status du drone
	if(droneStatus != -1)
		droneStatus = msg.state;
```

```
		// S'il y a un changement de status, on averti l'utilisateur (debugging).
		if(initStatus != droneStatus){
			if(droneStatus == 6)
				ROS_INFO("Taking off...");
			if(droneStatus == 8)
				ROS_INFO("Landing...");
			if(droneStatus == 2)
				ROS_INFO("Landed.");
			if(initStatus != 3 && initStatus != 4 && initStatus != 7 && (droneStat
us == 3 || droneStatus == 4 || droneStatus == 7)){
				ROS_INFO("Flying.");
				//timeToScanImage = true;
			}
		}
}


// Fonction d'analyse des données du Leap Motion
void leapAnalyses(const leap_motion::leapros& msg)
{


		// Takeoff et Landing si les données de la main ne changent plus depuis quelques
 millisecondes
		// Lorsqu'aucune main n'est détectée, les dernières valeurs restent bloquées...
		static float lastDetectedHand = 0.0;
		static ros::Time lastDetectionStamp = (ros::Time)0;
		ros::Time currentStamp = ros::Time::now();


		if(currentStamp - lastDetectionStamp > (ros::Duration)0.1){
			float detectingHand = msg.direction.x;


			if(droneStatus == -1){
				droneStatus = 2;
				lastDetectedHand = detectingHand;
			}


			if(detectingHand == lastDetectedHand && (droneStatus != 2 && droneStat
us != -1 && droneStatus != 0) && !paused){


				geometry_msgs::Twist velocityCmd;
```

```
                        velocityCmd.linear.x = 0.0;
                        velocityCmd.linear.y = 0.0;
                        velocityCmd.linear.z = 0.0;
                        velocityCmd.angular.x = 0.0;
                        velocityCmd.angular.y = 0.0;
                        velocityCmd.angular.z = 0.0;

                        droneCommandVel.publish(velocityCmd);

                        ros::Rate loop_rate2(2);
                        loop_rate2.sleep();

                        std_msgs::Empty msg;
                        droneCommandLand.publish(msg);
                }

                if(detectingHand != lastDetectedHand && droneStatus == 2 && !paused){
                        std_msgs::Empty msg;
                        droneCommandTakeOff.publish(msg);
                }

                lastDetectedHand = detectingHand;
                lastDetectionStamp = currentStamp;
        }

        // Analyse des commandes de vol

        if(droneStatus == 3 || droneStatus == 4 || droneStatus == 7){ // It`s flying and
being controlled

                // Cueillette des données avec mise en forme (de -1.0 à 1.0)
                float frontSpeed = ((-1.0)*msg.palmpos.z)/100.0;
                float frontSpeed2 = msg.normal.z;
                float heightSpeed = (msg.palmpos.y - 150.0)/100.0;
                float sideSpeed = ((-1.0)*msg.palmpos.x)/100.0;
                float sideSpeed2  = msg.normal.x;
                float rotatingSpeed = ((-1.0)*msg.direction.x)/0.5;
```

```
                    // Au cube pour garder le signe, mais avoir un modèle non linéaire.
                    frontSpeed = pow(frontSpeed,3.0);
                    frontSpeed2 = pow(frontSpeed2,3.0);
                    sideSpeed = pow(sideSpeed,3.0);
                    sideSpeed2 = pow(sideSpeed2,3.0);
                    heightSpeed = pow(heightSpeed,3.0);
                    rotatingSpeed = pow(rotatingSpeed,3.0);

                    // Commandes négligeables si trop petites (pas de vibration envoyée au
drone)
                    if(fabs(frontSpeed) < 0.04)
                              frontSpeed = 0.0;

                    if(fabs(frontSpeed2) < 0.04)
                              frontSpeed2 = 0.0;

                    if(fabs(sideSpeed) < 0.04)
                              sideSpeed = 0.0;

                    if(fabs(sideSpeed2) < 0.04)
                              sideSpeed2 = 0.0;

                    if(fabs(heightSpeed) < 0.04)
                              heightSpeed = 0.0;

                    if(fabs(rotatingSpeed) < 0.04)
                              rotatingSpeed = 0.0;

                    // Combiner inclinaison de la main et translation de la main.
                    // Permet d'avoir un système encore plus intuitif.
                    frontSpeed = frontSpeed + frontSpeed2;
                    sideSpeed = sideSpeed + sideSpeed2;

                    // Limitation des commandes entre -1.0 et 1.0...
                    if(frontSpeed > 1.0)
                              frontSpeed = 1.0;
                    if(frontSpeed < -1.0)
                              frontSpeed = -1.0;
                    if(heightSpeed > 1.0)
```

```
                heightSpeed = 1.0;
        if(heightSpeed < -1.0)
                heightSpeed = -1.0;
        if(sideSpeed > 1.0)
                sideSpeed = 1.0;
        if(sideSpeed < -1.0)
                sideSpeed = -1.0;
        if(rotatingSpeed > 1.0)
                rotatingSpeed = 1.0;
        if(rotatingSpeed < -1.0)
                rotatingSpeed = -1.0;

        // Création de la commande officielle
        geometry_msgs::Twist velocityCmd;

        velocityCmd.linear.x = frontSpeed;
        velocityCmd.linear.y = sideSpeed;
        velocityCmd.linear.z = heightSpeed;
        velocityCmd.angular.x = 0.0;
        velocityCmd.angular.y = 0.0;
        velocityCmd.angular.z = rotatingSpeed;

        // Vérification des commandes bloquées
        if(blockingCmd[0] == 1 && velocityCmd.linear.x > 0)
                velocityCmd.linear.x = 0;
        if(blockingCmd[1] == 1 && velocityCmd.linear.x < 0)
                velocityCmd.linear.x = 0;

        if(blockingCmd[3] == 1 && velocityCmd.linear.y > 0)
                velocityCmd.linear.y = 0;
        if(blockingCmd[2] == 1 && velocityCmd.linear.y < 0)
                velocityCmd.linear.y = 0;

        if(blockingCmd[4] == 1 && velocityCmd.linear.z > 0)
                velocityCmd.linear.z = 0;
        if(blockingCmd[5] == 1 && velocityCmd.linear.z < 0)
                velocityCmd.linear.z = 0;

        if(blockingCmd[6] == 1 && velocityCmd.angular.z < 0)
```

```
                        velocityCmd.angular.z = 0;
                if(blockingCmd[7] == 1 && velocityCmd.angular.z > 0)
                        velocityCmd.angular.z = 0;

                // Envoi des commandes de vol
                droneCommandVel.publish(velocityCmd);
        }
}

// Loupe principale d'initialisation!
int main(int argc, char **argv)
{
        // Initialisation de ROS
        ros::init(argc, argv, "leapDroneController");

        // Création du noeud (avertissement au ROS Master)
        ros::NodeHandle n;

        // Préparation à la publication sur 3 sujets.
        droneCommandTakeOff = n.advertise<std_msgs::Empty>("/ardrone/takeoff",10);

        droneCommandLand = n.advertise<std_msgs::Empty>("/ardrone/land",10);

        droneCommandVel = n.advertise<geometry_msgs::Twist>("/cmd_vel",10);

        // Abonnement à 3 sujets
        ros::Subscriber leapData = n.subscribe("/leapmotion/data", 10, leapAnalyses);

        ros::Subscriber navData = n.subscribe("/ardrone/navdata", 10, navAnalyses);

        ros::Subscriber pauseSub = n.subscribe("/security/pause", 10, pause);

        // BOUCLE PRINCIPALE

        ros::Rate loop_rate(10);
        ROS_INFO("Node Initialized.");
        ROS_INFO("Ready to take off as Leap Motion detects hand.");

        while(ros::ok()){
```

```
                    ros::spinOnce();
                    loop_rate.sleep();
            }
}
```

Tout est fait. Le paquet est prêt à être compilé.

Deux simples commandes vont conclure le tout :

- `cd ~/catkin_ws`
- `catkin_make`

Pour démarrer notre noeud, il suffit maintenant d'ouvrir un terminal et d'utiliser rosrun :

- `rosrun leap_ardrone_controller analyse.cpp`

XI. DESIGN DÉTAILLÉ : NOEUD D'ANALYSE

Ça fait beaucoup de commandes et de code à coller. Il est vrai qu'une fois tout cela fait, le système fonctionne, mais pourquoi ? Que fait le noeud d'analyse en fait?

Tout ce code est énorme mais, en fait, plutôt simple: 3 fonctions secondaires et une principale.

La fonction principale ne fait qu'initialiser ROS, crée le nœud, avertit ROS que ce nœud publiera sur 3 sujets et s'abonne à 3 autres sujets (d'où les 3 fonctions secondaires). Ensuite, une simple boucle vérifie si des messages sont reçus. Si oui, la fonction liée au sujet est appelée, sinon, le noeud dort pendant quelques millisecondes avant de revérifier.

La première fonction secondaire, « pause », permet simplement d'utiliser l'arrêt d'urgence. Si elle est appelée, soit elle bloque toutes les commandes, soit elle les débloque.

La deuxième fonction secondaire, « navAnalyse », vérifie les données de vol. Elle permet de mettre à jour le status du drone, de faire un blocage des commandes en altitude (à l'aide des données du sonar) et avertit l'utilisateur lorsque la batterie est faible (3 avertissements à 3 niveaux différents).

La troisième et dernière fonction secondaire, « leapAnalyse », est la plus importante. Celle-ci vérifie en permanence si une main est détectée ou pas pour procéder au décollage et à l'atterrissage automatique. Lorsque le drone est en vol, elle obtient les données de la main, **elle traite ces données** puis elle envoie les commandes de vol au drone.

Le traitement est plutôt simple, on vérifie la distance de la paume par rapport à un point de référence (0,0,150) soit centré avec le Leap Motion à environ 6 pouces au dessus de celui-ci. La distance de ce point dans toutes les directions provoque une commande jusqu'à une commande maximale de -1.0 ou 1.0. Cette commande maximale est généralement atteinte par n'importe quel mouvement s'éloignant de 100 unités du centre. Aussi, dans le traitement, on met les commandes au cube et on leur ajoute une limite basse de non-fonctionnement. Cela enlève les vibrations lorsqu'on veut stabiliser le drone et crée approximativement 3 zones de fonctionnement : une zone stable, une zone de mouvement précis et une zone de très grande vitesse.

Aussi, pour les mouvements frontaux et latéraux, j'ai ajouté des commandes qui vérifient la normale à la paume puisque plusieurs personnes pilotent **intuitivement** avec l'angle de la paume à la place de sa position. J'additionne les deux commandes et j'applique le maximum (-1.0/1.0)

XII. TESTS ET VALIDATIONS

Bref, le système est maintenant installé et semble fonctionnel.

Les tests seront plutôt simples parce qu'en fait, si une erreur s'est produite avant la compilation, il faut corriger le code. Si la compilation a réussi, on lance le nœud et vérifie s'il publie les bonnes données.

Pour démarrer le système au complet (4 nœuds), on lance 4 terminaux :

```
1 -       roscore

2 -       export PYTHONPATH=$PYTHONPATH:$HOME/LeapSDK/lib:$HOME/LeapSDK/lib/x64
          source /opt/ros/hydro/setup.bash
          rosrun leap_motion sender.py

3 -       (Connexion wifi au drone puis...)
          rosrun ardrone_autonomy ardrone_driver

4 -       source ~/catkin_ws/devel/setup.bash
          rosrun leap_ardrone_controller analyse.cpp
```

Ensuite, pour effectuer nos tests, on peut vérifier en temps réel les données qui sont publiées sur certains sujets. Par exemple, si on veut tester « leapmotion/data », dans un nouveau terminal :

- `rostopic echo /leapmotion/data`

De même,

- `rostopic echo /ardrone_autonomy/land`
- `rostopic echo /ardrone_autonomy/takeoff`
- `rostopic echo /cmd_vel`
- `rostopic echo /ardrone_autonomy/navdata`
- `rostopic echo /security/pause`

Pour déclencher/arrêter l'arrêt d'urgence, il suffit de publier manuellement sur le sujet « /security/pause ».

```
rostopic pub /security/pause std_msgs/Empty
```

Toutes ces techniques permettent facilement de diagnostiquer des problèmes potentiels.

Dans mon cas, j'ai pu remarquer des bogues dans mon programme au tout début grâce à ces techniques et, aujourd'hui, elle me permette de prouver le fonctionnement du système sans même forcément avoir accès à un drone.

Suite à tous mes tests, j'ai pu en conclure que mon système est fonctionnel et pratiquement sans faille. Il est vrai que le périmètre de vol n'est pas complet, mais avec la caméra de basse qualité du drone, il m'était impossible de connaître ma position latérale et frontale.

En effet, même en intégrant les données de l'accéléromètre, une erreur majeure finissait par s'accumuler assez rapidement. La seule autre manière de localiser le drone, c'était par caméra. Sans une caméra de haute qualité et surtout avec un environnement répétitif (filet de la volière), le système de localisation par analyse d'image n'était pas fonctionnel.

Pour corriger le tout, j'ai modifié l'arrêt d'urgence manuel. Elle permet, en effet, d'avoir un copilote qui bloque le drone (stabilisation) si quelque chose n'est pas normal pour potentiellement redonner le contrôle au pilote au même point si le problème peut se régler. Ainsi, lors d'une présentation grand public, la sécurité reste présente (avec un copilote) et le périmètre de vol est respecté de toute manière en altitude (impossible de sortir de la volière par le haut).

Avec ces dernières modifications, le système fut parfaitement fonctionnel pour la présentation grand public du 11 août dernier et, effectivement, tout s’est passé pour le mieux (incluant l'arrêt d'urgence).

XIII. CONCLUSION ET PERSPECTIVES

Pour conclure, je suis très satisfait non seulement du résultat du projet, qui fut un grand succès, mais aussi et surtout de tout ce que j'ai pu apprendre au cours de celui-ci. Apprendre ROS fut sans contredit la plus grande difficulté de ce projet mais, en même temps, c'était aussi la plus grande satisfaction. Ça semble être tellement important comme système d'exploitation en robotique et, en plus, il s'agit vraisemblablement du domaine dans lequel je désire aller.

Bref, un projet réussi et satisfaisant dans son ensemble malgré le périmètre de vol incomplet. Mais est-il vraiment terminé? Dans le cadre de ce cours, oui. Pour des présentations grand public, potentiellement. D'un point de vue de son potentiel, pas du tout.

Plusieurs améliorations pourraient y être faites qui le rendraient vraiment intéressant!

Premièrement, utiliser l'image de la caméra frontale avec un nouveau nœud qui serait relié à des lunettes d'immersion. Ça serait génial. Non seulement, le pilotage du drone serait plus simple, mais ce serait tout simplement plus agréable. En effet, pour piloter le drone actuellement, il faut toujours penser comme si on était dans son référentiel d'inertie. Son « devant » n'est pas forcément le nôtre. Avec des lunettes d'immersion, plus de problème, nous serions à la place du drone!

D'ailleurs, lors de la présentation du 11 août dernier, j'ai eu la chance de discuter avec un amateur de drone. Ce dernier a été jusqu'à fabriquer son propre drone. Il m'a appris que c'est quelque chose qui se fait déjà (lunette d'immersion) et que c'est définitivement une des meilleures expériences à vivre avec un drone.

Deuxièmement, il serait intéressant de régler le problème du périmètre de vol. Deux choix sont possibles, utiliser une meilleure caméra avec peut-être même un meilleur algorithme ou utiliser un système externe comme un GPS. La première méthode est plus efficace, mais plus souvent portée à l'erreur. La seconde est fiable, mais moins précise. Une solution pourrait être de combiner les deux. Si l'analyse d'image fonctionne, l'utiliser. Si elle plante, passer au mode GPS.

Il s'agit de deux améliorations majeures qui pourraient être faites relativement facilement, mais il y en a certainement plusieurs autres. Bref, ce projet a beaucoup de potentiel pour être amené encore plus loin.

Pour l'instant, il demeure qu'il est assez fonctionnel pour déclarer qu'il remplit le mandat d'Élikos et qu'il pourrait servir lors de présentation grand public!

LISTE DE RÉFÉRENCES

Leap Motion, Inc. (2015). API Overview. Tiré de https://developer.leapmotion.com/documentation/cpp/devguide/Leap_Overview.html

ROS wiki. (2015). ROS Distributions. Tiré de http://wiki.ros.org/Distributions

ROS wiki. (2015). ROS Tutorials. Tiré de http://wiki.ros.org/ROS/Tutorials

ROS wiki. (2015). Leap Motion. Tiré de http://wiki.ros.org/leap_motion

ROS wiki. (2015). AR Drone Autonomy. Tiré de http://wiki.ros.org/ardrone_autonomy

ROS wiki. (2015). Installation de ROS Indigo sous Ubuntu. Tiré de http://wiki.ros.org/indigo/Installation/Ubuntu