LIASD
Université Paris8
2, rue de la Liberté
93526 Saint-Denis Cedex
France
http://www.ai.univ-paris8.fr/

Université Paris10 Nanterre
200 Avenue République
92001 Nanterre
France

Rapport de Stage

Portage de module Fuerte vers Groovy

Maîtres de stage
Nicolas Jouandeau

Bahar OZDEMIR
11294108

Remerciements

Tout d'abord, j'exprime ma profonde gratitude envers Nicolas JOUANDEAU pour m'avoir pris en stage au LIASD. Je le remercie également pour ses nombreux conseils, soutiens techniques qui permis d'organiser mon travail d'un point de vue professionnel.

Je tient à remercier Jean-Noël VITTAUT sans qu'il aurait été plus dur d'être acceptée à ce stage ainsi que Adrien Revault d'ALLONNES pour ses conseils.

Je me dois également de remercier le laboratoire de l'Université Paris Nanterre pour l'intérêt qu'il ont porté à mon stage.

J'adresse aussi mes remerciements aux autres collègues pour la bonne ambiance et les quelques conseils toujours bienvenus, plus particulièrement à Nizar ABAK-KALI et Belgacem KHALED.

C'est un réel plaisir de travailler dans une telle équipe.

Sommaire

Table des figures

1 Introduction

1.1 Généralités

Etudiante en dernière année de licence Informatique à l'Université Paris8, on a été amené à réaliser un stage d'un mois afin de découvrir le monde de la recherche et le domaine d'activités. Le stage a été effectué au sein du LIASD du 16 juin au 25 juillet 2014. Ce stage a été une opportunité de travailler dans un centre de recherche. Le but étant non pas le profit mais l'avancée scientifique, cela crée une réelle motivation.

C'est dans ce cadre que l'on a mis en pratique ses connaissances acquises ces dernières années, mais également appris à s'adapter pour optimiser et non plus appliquer pour réaliser.

2 Deroulement du stage

2.1 Planning du temps

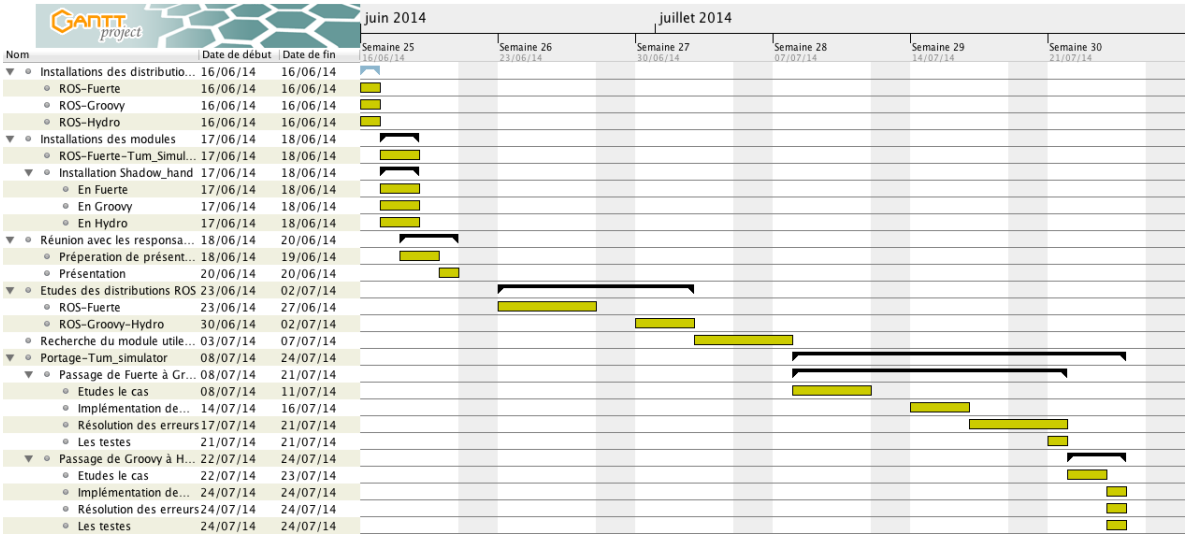
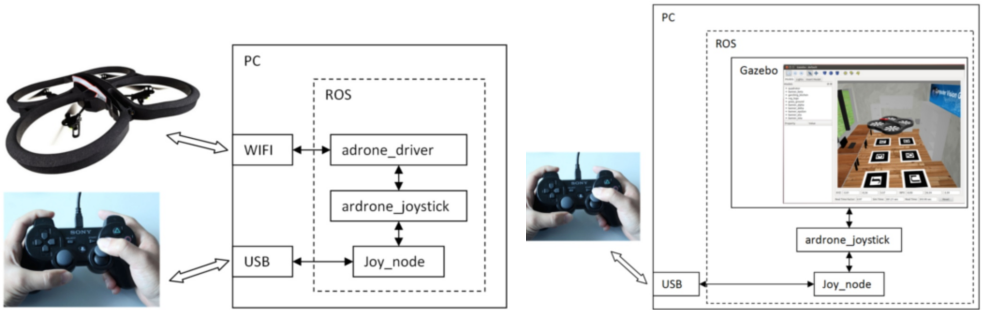


FIGURE 2.1 – Deroulement de stage

Le stage se réalise entre le 16 et 25 juin 2014 à l ?université Paris 8. Les premiers jours sont passés par les installations des distribution des ROS ainsi la préparation pour la réunion avec les responsables de stages. Par la suite, les distributions sont étudiées afin de repérer les differences entre eux. Une fois que les distributions sont examinés, les modules à porter sont installés et les indices sont cherchées pour avoir le bon résultat du portage. Après la compilation, le module est testé.

3 Problematique

L'objectif final est de mettre en place un module, « tum_simulator », dans les deux autres distributions ROS tels que Groovy et Hydro. Ce module permettant aux développeurs robotiques de tester rapidement les algorithmes en utilisant des scénarios pour AR.Drone2.0 basé sur Gazebo qui lui offre la possibilité de simuler avec précision et efficacité le quadrocopter dans des environnements intérieurs et extérieurs complexes.



Les changements au niveau de configuration de système, architecture et la configuration d'un package se différencient entre les distributions ROS.

3.1 Objectif final du stage

Le sujet de stage est d'abord de répondre si le module sera portable vers les versions plus récentes ,puis de constituer un nouveau package du module « tum_simulator » compatible aux versions.

En effet, les buts sont :

- Développer l'ergonomie dans la version Fuerte
- Repérer les changements entre Fuerte-Groovy et Fuerte-Hydro
- Former les grandes lignes du portage
- Tester le module

3.2 Lien avec les deux autres sujets

4 Existants

4.1 ROS

ROS (Robot Operating System) est un framework open source pour le développement de logiciel robotique. Elle a une fonctionnalité du système d'exploitation. En effet ,elle fournit abstraction matérielle, des pilotes de périphériques, les bibliothèques, visualiseurs, transmission de messages, la gestion des paquets, et plus encore. ROS a été créée en 2007 par le Laboratoire d'intelligence artificielle de Stanford avec l'appui du projet l'AI Robot Stanford (STAIR).

Ubuntu est la plate-forme de développement principal pour ROS.

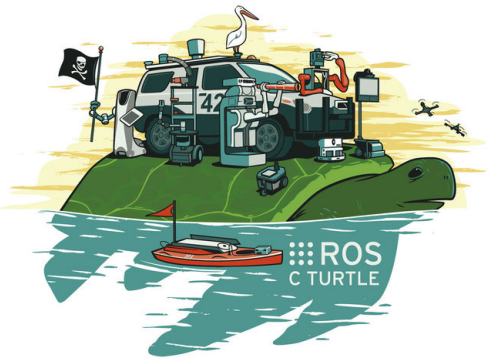


FIGURE 4.1 – Robot Operating Système

4.1.1 Les fonctionnalités

1. Perception
2. Dentification d'objet
3. Segmentation et reconnaissance
4. Reconnaissance faciale
5. De reconnaissance gestuelle (compatible kinect Voir installation , voir en vidéo)
6. Suivi de mouvement
7. Ego-motion
8. Compréhension des mouvements
9. Structure du mouvement (SFM)

10. Stereopsis stéréo vision : perception de la profondeur de 2 caméras (3D)
11. Motion
12. Robotique mobile
13. Contrôle
14. Planification
15. Saisit

4.1.2 Les champs de ROS (information technique sur l'OS)

1. Coordination d'un noeud maître
2. Edition ou abonnement à des flux de données (images, stéréo, laser, le contrôle, l'actionneur, contact ...)
3. Multiplexage Informations
4. Création des noeuds et la destruction
5. Les noeuds sont distribués de façon transparente, permettant un fonctionnement distribué plus que du multi-core, les clusters multi-processeur, multi-GPU multi-core (calcul distribué).
6. Connexion
7. Paramètre de serveur
8. Les systèmes de test et de simulation

4.2 Les distributions de ROS

Il existe plusieurs distributions de ROS notamment ROS Diamondback, ROS Emys électriques, ROS Fuerte Tortue , ROS Groovy Galapagos, ROS Hydro Méduse , ROS Indigo Igloo etc...

Ce projet consiste au passage de Fuerte à Groovy et à Hydro.

(a) Avril 2012 (b) December 2012 (c) Septembre 2013

FIGURE 4.2 – Les versions ROS

4.2.1 ROS Fuerte Tortue

ROS Fuerte Tortue est le cinquième distribution ROS et a été libéré le 23 Avril 2012. Fuerte a des améliorations importantes qui le rendent plus facile à intégrer avec d'autres cadres et des outils logiciels. Cela comprend une réécriture du système de construction, la migration vers le framework Qt, et passage au mode autonome bibliothèques. ROS vise à rendre le code plus réutilisable de robotique, et cette version est une base solide de nouveau pour la prochaine génération de grandes bibliothèques de la robotique.

Installation : Fuerte

Depuis : : http ://wiki.ros.org/fuerte/Installation/Ubuntu

1. sudo sh -c 'echo "deb http ://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'
2. wget http ://packages.ros.org/ros.key -O - | sudo apt-key add -
3. sudo apt-get update
4. sudo apt-get install ros-fuerte-desktop-full
5. source /opt/ros/fuerte/setup.bash
6. sudo apt-get install python-rosinstall python-rosdep
7. export | grep ROS

4.2.2 ROS Groovy Galapagos

ROS Groovy Galapagos qui est la sixième version de la distribution ROS a été sorti le 31 Décembre 2012. Dans cette version,l'accent est mis sur l'infrastructure de base de ROS pour le rendre plus facile à utiliser, plus modulaire, plus évolutive, travail sur un plus grand nombre de systèmes d'exploitation / architectures matérielles / robots et surtout d'impliquer davantage la communauté ROS.

ROS Groovy Galapagos sera principalement destiné à la presse Ubuntu 12.04 (Precise), bien que la compatibilité avec d'autres systèmes Linux ainsi que Mac OS X, Android et Windows soit prévue.

Installation : Groovy

http ://wiki.ros.org/groovy/Installation/Ubuntu

1. sudo sh -c 'echo "deb http ://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'
2. wget http ://packages.ros.org/ros.key -O - | sudo apt-key add -
3. sudo apt-get update
4. sudo apt-get install ros-groovy-desktop-full
5. sudo rosdep init
6. rosdep update

4.2.3 ROS Hydro Méduse

Etant la septième version de la distribution ROS, ROS Hydro Medusa a été libéré le 4 Septembre 2013. Elle est la version améliorée de Groovy. En effet, la plupart des paquets de ROS ont été changés en fonction du nouveau système catkin en améliorant les composants de base de ROS. En outre, des nombreuses améliorations dans les outils comme rviz (Outil de visualisation 3D pour ROS) et rqt (rqt est un cadre pour le développement graphique pour ROS.) ainsi ont été faites.

En outre, cette version de ROS a fait des progrès à dépendre de versions canoniques des dépendances, plutôt que l'emballage des versions. C'est le cas de PCL, scène et Gazebo et autres petites bibliothèques qui servent à personnaliser et publié séparément par la communauté ROS. Hydro a également une meilleure intégration avec Gazebo parmi les trois versions.

Installation : Hydro

http ://wiki.ros.org/groovy/Installation/Ubuntu

1. sudo sh -c 'echo "deb http ://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'
2. wget http ://packages.ros.org/ros.key -O - | sudo apt-key add -
3. sudo apt-get update
4. sudo apt-get install ros-hydro-desktop-full
5. apt-cache search ros-hydro

6. sudo rosdep init
7. rosdep update
8. echo ”source /opt/ros/hydro/setup.bash” > > /.bashrc
9. source /.bashrc

4.3 Les grands lignes du portage

Le système de configuration de Fuerte est « rosbuild » tandis que celui de Groovy et Hydro est un nouveau système de configuration appelé « catkin ». « catkin » a été développé pour répondre à plusieurs inconvénients dans « rosbuild » . Contrairement à rosbuild catkin favorise système de fichiers Linux (FHS). Ceci rend beaucoup plus facile de distribuer des paquets ROS sur d'autres systèmes d'exploitation et architectures. Il existe une version prototype de « catkin » introduite dans « fuerte » seulement pour quelque paquets. « catkin » est maintenant le système de compilation officielle de ROS.

Dans « rosbuild » les packages dépendent de package, les stacks(piles) dépendent de stack. Le stack qui contient un package doit depend de tous les stacks qui contiennent les dependences des packages.

La concept de pile « stack » est supprimé dans « catkin ». Une des principales raisons est qu'il y a une duplication de suivi des dépendances entre paquets et des piles.Les paquets ne sont plus regroupés dans les piles. Au lieu de cela, ils sont regroupés dans des collections de paquets appelés « metapackage ».Un « metapackage » n'est pas un conteneur des packages comme celui de « rosbuild » mais un ensemble nommé de reference des packages. Le « metepackage » peut être depend des autres packages qui ne sont pas forcement dans le meme dossier ou la meme source de repertoire.

Fuerte –> Groovy

.

1. Le fichier *stack.xml* est remplacé par un fichier *package.xml* qui indique un méta-paquet. Ce fichier package.xml est déplacé dans un dossier sur le même niveau que les autres paquets, de *my_stack / stack.xml* à *my_stack / my_stack /package.xml*.
2. Les paquets contiennent encore un fichier *CMakeLists.txt*, mais le contenu et le format de ce fichier a radicalement changé.
3. Les paquets contiennent plus d'un *Makefile*.

4.4 Les differences applicatives

4.4.1 Le concept des packages

il existe des différences dans le concept des paquets « rosbuild » et « catkin ». Afin de transformer un paquet « rosbuild » à un paquet « catkin » :

1. Créer un espace de travail de catkin pour tenir les projets migrés
2. Apporter tous projets dans l'espace de travail de catkin « catkin workspace » une fois que tous ces dependences sont catkinisé (configurés) .
3. Convertir le stack ayant au moins un paquet au metapackage.
   (a) Créer un paquet catkin simple qui remplacera le stack .Ceci sera un meta-package.
   (b) Ajouter dans « package.xml » tous les autre packages de stack à l'aide de « run_depend » .
   (c) Supprimer « stack.xml » et remplacer ses informations vers le « package.xml »
   (d) Donc « Stack » devient à « Metapackage »
4. Pour chaque paquet qui contient un fichier « manifest.xml »
   (a) Renommer « manifest.xml » par « package.xml »
   (b) Ajouter un tag avec le nom de package qui sera le nom de repertoire.
   (c) S'il manque, créer un fichier « CMakeLists.txt » contenant « catkin_package() »
   (d) Supprimer « Makefile »
5. Dans chaque « CMakeLists.txt »
   (a) Si macros « rosbuild » ont été utilisés, passer de macros « rosbuild » aux commandes de CMake sous-jacents
   (b) Déclarer comment les cibles (les binaires C++) sont installées
6. Dans chaque executable C++ ou bibliothèque
   (a) Ajouter un add_dependencies() pour le cible pour qu'il génère les messages d'en-têtes.
      i. Par exemple, un paquet foo qui construit un exécutable foo_node et qui dépend foo_msgs nécessite d'être ajoutée foo_node et foo_msgs_gencpp à l'aide de add_dependencies au « CMakeLists.txt ». (foo_node foo_msgs_gencpp) à la fin de CMakeLists.txt dossier de foo
   (b) Ajouter un TARGET_LINK_LIBRARIES () avec $ {catkin_LIBRARIES}
7. Pour chaque executable python dans le package
   (a) Installer le script en utilisant « cmake » pas en utilisant « setup.py » . Ainsi il est installé dans « lib/ PKGNAME » au lieu de global « bin » repertoire.

(b) Si le script n'est pas utilisé par le « rosrun » mais aussi dans les fichiers « launch » .Ceci peut-être trouvé par : "$(find pkgname)path/to/script » :

8. Si le package contient des paquets en python , créer un « setup.py » , déclarer les paqutes et faire appel à « catkin_python_setup() » dans le « CMakeLists.txt »

4.4.2 Les macros

Les changement plus remarquables entre les systèmes des configurations , rosbuild-catkin sont dans le fichier CMakeLists.txt. Dans les versions plus récentes que rosbuild les macros de rosbuild sont remplacés par les autres.

Build Macros

Rosbuild	**Catkin**
rosbuild_init()	supprimé
rosbuild_add_library(...)	add_library(...)
rosbuild_add_executable(...)	add_executable(...)
rosbuild_add_compile_flags(...)	set_target_properties(target PROPERTIES COMPILE_FLAGS new_flags)
rosbuild_remove_compile_flags(...)	set_target_properties(target PROPERTIES COMPILE_FLAGS new_flags)
rosbuild_add_link_flags(...)	set_target_properties(target PROPERTIES LINK_FLAGS new_flags
rosbuild_remove_link_flags(...)	set_target_properties(target PROPERTIES LINK_FLAGS new_flags)
rosbuild_add_boost_directories(...) ; rosbuild_link_boost(target components)	find_package(Boost REQUIRED COMPONENTS components) ; include_directories(${Boost_INCLUDE_DIRS}) ; target_link_libraries(target ${Boost_LIBRARIES})
rosbuild_add_openmp_flags(...)	find_package(OpenMP)
rosbuild_invoke_rospack(...)	X
rosbuild_find_ros_package(...)	X
rosbuild_find_ros_stack()	X
rosbuild_include(package module)	include(module) (peut nécessiter quelques travaux spéciaux pour trouver le chemin du module)
rosbuild_add_lisp_executable()	pas de support pour le moment

Test Macros

Rosbuild	Catkin
rosbuild_add_gtest(...)	catkin_add_gtest(...)
rosbuild_add_gtest_labeled	if (${LABEL}) catkin_add_gtest(...) endif()
rosbuild_add_gtest_future	comment it out
rosbuild_add_gtest_build_flags	use set_target_properties on test target
rosbuild_add_pyunit	migrate to catkin_add_nosetests(...)
rosbuild_add_pyunit_labeled	similar to rosbuild_add_gtest_labeled
rosbuild_add_rostest(...)	add_rostest(...)
rosbuild_add_rostest_labeled	similar to rosbuild_add_gtest_labeled
rosbuild_add_roslaunch_check	roslaunch_add_file_check (trouver d'abord le paquet roslaunch)
rosbuild_declare_test	add_dependencies(tests <test-target>)

Message/Service Macros

Rosbuild	Catkin
rosbuild_add_generated_msgs(...)	add_message_files(DIRECTORY msgFILES ...)
rosbuild_add_generated_srvs(...)	add_service_files(DIRECTORY srv FILES ...)
rosbuild_genmsg()	generate_messages() (un seul pour CMakeLists.txt)
rosbuild_gensrv()	generate_messages() (un seul pour CMakeLists.txt)

Version Macros

Rosbuild	Catkin
rosbuild_get_stack_version	obsolete
rosbuild_get_package_version	obsolete

Data Macros

Rosbuild	Catkin
rosbuild_download_data(url filename [md5sum])	catkin_download_test_data
rosbuild_download_test_data	catkin_download_test_data

Actionlib

Rosbuild	Catkin
include(${actionlib_msgs_PACKAGE_PATH}/cmake/actionbuild.cmake)	Ajouter actionlib_msgs pour le paquet appelé par find_package
genaction()	add_action_files(DIRECTORY msg FILES ...)

Python

S'il existe des scripts python dans le paquet , on a besoin d'ajouter un **setup.py** :

Pour une mise en page standard avec le module python dans le *src* sous-répertoire de votre paquet de catkin :

setup.py/ Python

.

```
1. #!/usr/bin/env python
2. from distutils.core import setup
3. from catkin_pkg.python_setup import generate_distutils_setup
4. d = generate_distutils_setup(
5. don't do this unless you want a globally visible script
6. scripts=['bin/myscript'],
7. packages=['PYTHON_PACKAGE_NAME'], package_dir=' ' : 'src' )
8. setup(**d)
```

dans le fichier CMakeLists.txt

CMakeLists.txt/Macro

catkin_python_setup ()
installer (PROGRAMMES scripts / foo_script DESTINATION $ {} CATKIN_PACKAGE_BIN_DESTINATION)

dynamic_reconfigure

Pour chaque fichier de **.cfg**

Rosbuild	**Catkin**
import roslib ;roslib.load_manifest(PACKAGE)	N'EXISTE PAS
from dynamic_reconfigure.parameter_generator import *	from dynamic_reconfigure.parameter_generator_catkin import *

Dans CMakeLists.txt, ajouter ce qui suit avant l'appel à *catkin_package ()* pour générer du code pour les fichiers reconfigure dynamiques :

CMakeLists.txt

generate_dynamic_reconfigure_options (cfg / DynFile1.cfg cfg / DynFile2.cfg ...)

De plus, dans le CMakeLists.txt, ajouter la dépendance suivante pour chaque cible qui utilise le code généré :

CMakeLists.txt

```
# Assurez-vous que les en-têtes de configuration sont construits avant que n'importe quel noeud les utilise.

add_dependencies(example_node ${${PROJECT_NAME}_EXPORTED_TARGETS})
```

5 Etude de cas

5.1 L'architecture de tum_simulator en Fuerte

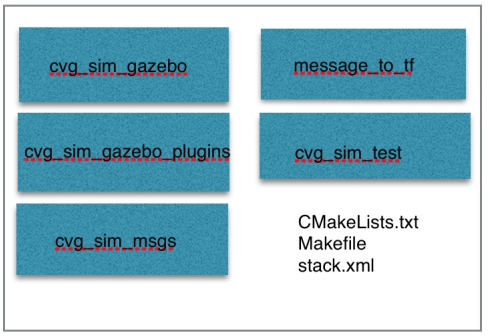
Le module « tum_simulator » ayant en forme de stack regroupe des paquets tels que *cvg_sim_gazebo , cvg_sim_gazebo_plugins, cvg_sim_msgs, cvg_sim_test, message_to_tf.*

cvg_sim_gazebo : contient des modèles d'objets, les modèles de capteurs, les modèles de Quadrocopter, les informations d'environnement de vole et des fichiers de lancement dessiné volontairement pour chaque objets et l'environnement pur sans aucun autre objet.

cvg_sim_gazebo_plugins : contient plugins gazebo pour le modèle quadrocopters. quadrotor_simple_controller est utilisé pour contrôler le mouvement du robot et de fournir des informations de navigation, telles que : / ardrone / navdata. D'autres sont des plugins pour les capteurs dans le quadricoptère, tels que : capteur IMU, capteur sonar, capteur GPS.

cvg_sim_msgs : contient les formulaires de message pour le simulateur.

message_to_tf : un emballage utilisé pour créer un noeud de ros, qui transfère le ros topic/ground_truth/state à un / tf topic.

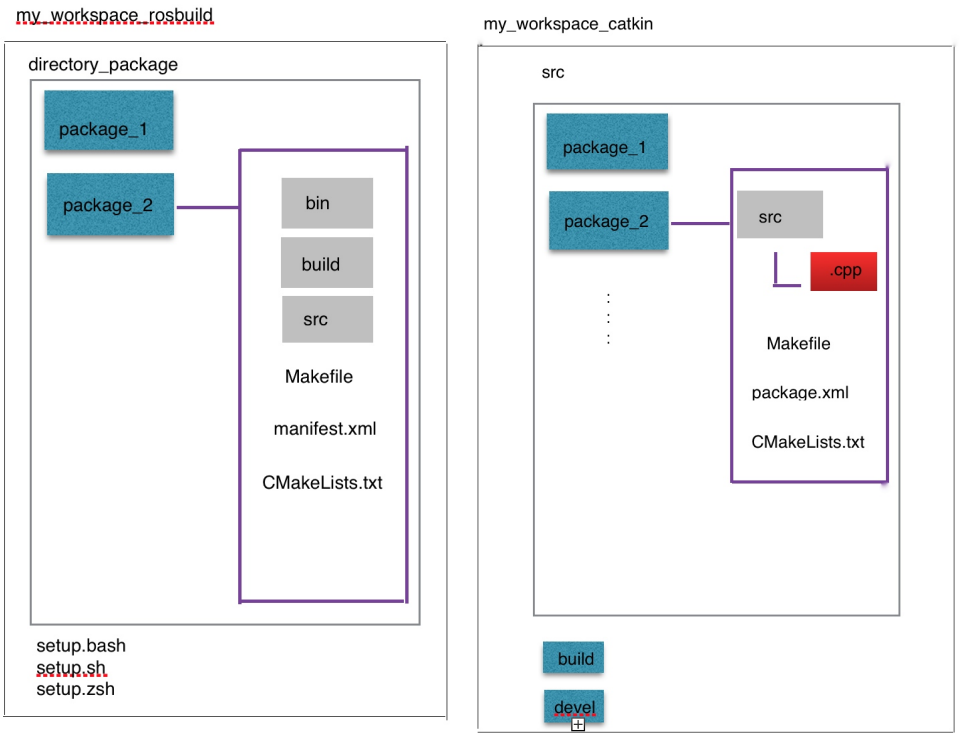


(a) Rosbuild

FIGURE 5.1 – L'architecture du « tum_simulator » en Fuerte

5.1.1 Sous parties obsoletes

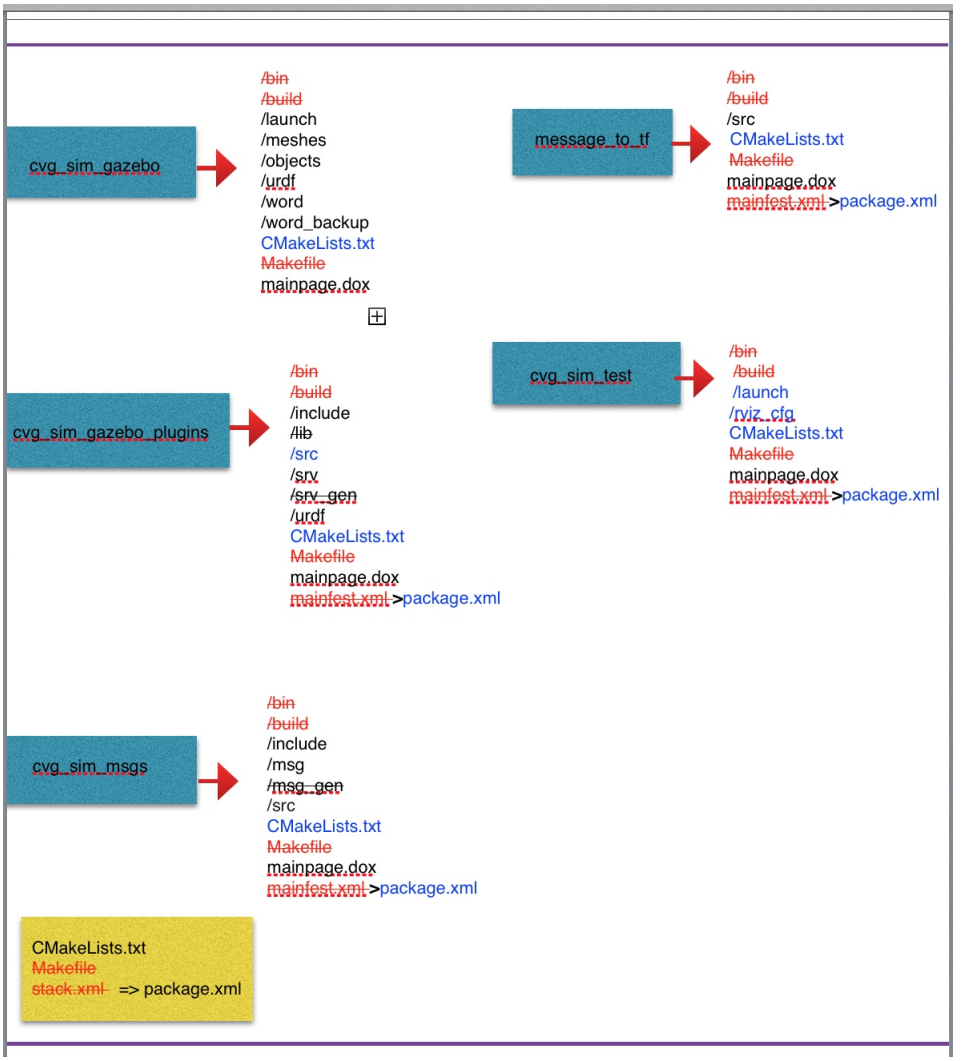
Transformation d'un paquet Fuerte à un paquet Groovy nécessite des changements :



(a) Rosbuild (b) Catkin

FIGURE 5.2 – Le concept des packages

La figure 5.2 montre la transformation de rosbuild workspace à catkin workspace.



(a) De « rosbuild » à « catkin »

FIGURE 5.3 – Sous parties obsoletes

La figure 5.3 montre les parties obsoletes et les parties modifiées dans le paquet de tum_simulator de version Groovy. Ce qui est en rouge n'existe pas et ce qui est en bleu exprime les modifications des contenus effectuées dans la nouvelle version du paquet.

En revanche la configuration des paquets ne sont pas fait à l'intérieur des paquets mais dans un repertoire à part du workspace appelé « build ». Exécution de l'ensemble des paquets font avec la commende « catkin_make » :

Execution 1

```
$my_workspace_catkin/catkin_make
```

Si seulement un paquet est exécuté , on utilise la commende « rosmake »

Execution 2

```
$my_workspace_catkin/src/rosmake my_package
```

En outre, en tant que tous les fichiers *manifest.xml* devient *package.xml* , *stack.xml* se transforme aussi à *package.xml*

6 Modifications de l'architecture

La distribution Groovy et Hydro possède CMake qui est un « moteur de production » multiplate-forme. Son processus de construction logicielle est entièrement contrôlé par des fichiers de configuration, appelés CMakeLists.txt mais CMake ne produit pas directement le logiciel final,il s'occupe de la génération de fichiers de construction standards. CMakeLists.txt décrit comment construire le code et où l'installer.

6.1 CMakeLists.txt

Afin d'effectuer la configuration des paquets, le fichier CMakeLists.txt nécessite quelques tags :

6.1.1 Version de CMake *cmake_minimum_required()*

Chaque fichier CMakeLists.txt de catkin doit commencer par la version CMake. Groovy nécessite la version 2.8.3 ou supérieur.

cmake_minimum_required (VERSION 2.8.3)

6.1.2 Nom du paquet par *project()*

La précision de nom du paquet est faite par project() . La référence du nom du projet dans le script CMake est effectuée par la variable $ {PROJECT_NAME} .

6.1.3 Les dépendances avec *find_package ()*

Les dépendances de type CMake/catkin doivent être indiquées avec find_package (). Ce macro trouve les chemins des dépendances puis les inclus aux variables d'environnement.

S'il le projet depend des autres paquets catkin, find_package () aide à les transformer en composants de catkin. Au lieu d'appeler la fonction un par un pour chaque paquet, on les spécifie en tant que composants et on trouve toutes les dépendances en même temps.

find_package ()

```
find_package(catkin REQUIRED COMPONENTS dep dep2 etc)
```

soit il faut le faire un par un :

find_package ()

```
    find_package(catkin REQUIRED)
find_package(dep REQUIRED)
```

Que fait find_package() ?

Une fois que le paquet est trouvé par find_package, plusieurs variables d'environnement qui donnent les informations sur le paquet sont constituées .Les variables d'environnement décrivent où les packages exportent les fichiers d'en-tête sont, où sont les fichiers de source, quelles sont les bibliothèques dépendantes du paquet et leur chemin.

Les noms suivent toujours la convention de <nom du paquet> _ <PROPRIÉTÉ> :

1. <NOM>_FOUND - la valeur true si la bibliothèque est trouvé, sinon tomber
2. <Nom>_INCLUDE_DIRS ou <NOM>_INCLUDES - Les chemins inclus exportés par le paquet
3. <Nom>_LIBRARIES ou <NOM>_LIBS - Les bibliothèques exportées par le paquet

find_package ()

```
find_package(catkin REQUIRED COMPONENTS composant)
```

Cela signifie que les chemins inclus , les bibliothèques, etc exportés par composant sont également annexé aux variables de catkin_.
Par exemple, catkin_INCLUDE_DIRS contient les chemins inclus , non seulement pour catkin mais aussi pour composant.

find_package ()

```
find_package (composant)
```

Cela signifie que les chemins composant, les bibliothèques et ainsi de suite ne serait pas ajouté à catkin_ des variables. Il en résulte composant_INCLUDE_DIRS, composant_LIBRARIES, et ainsi de suite.

Note ! Si C ++ et Boost sont utilisés dans le paquet, il faut invoquer find_package () sur Boost et préciser quels aspects de Boost utilisés en tant que composants. Par

exemple, si l'on utilise « thread » de Boost :
find_package(Boost REQUIRED COMPONENTS thread)

6.1.4 catkin_package()

catkin_package () est une CMake macro et est fourni par catkin . Cela est nécessaire pour préciser les informations spécifiques au système de construction qui à son tour est utilisé pour générer pkg-config et les fichiers CMake. Cette fonction doit être appelée avant de déclarer toutes les cibles avec add_library () ou add_executable(). La fonction dispose de 5 arguments optionnels :

1. INCLUDE_DIRS - Les chemins inclus exportés(c.-à-CFLAGS) pour le paquet
2. INCLUDE_DIRS - Les chemins inclus exportés(c.-à-CFLAGS) pour le paquet
3. CATKIN_DEPENDS - Autres projets dépendants de catkin
4. DEPEND - Non-catkin projets que ce projet dépend
5. CFG_EXTRAS - options de configuration supplémentaires

Par exemple :

catkin_package()

```
catkin_package(
# INCLUDE_DIRS include
# LIBRARIES cvg_sim_gazebo
# CATKIN_DEPENDS gazebo
# DEPENDS system_lib
)
```

6.1.5 Spécifier les cibles à construire

il existe deux possibilités :

1. target exécutable - programmes que nous pouvons exécuter
2. target bibliothèque - bibliothèques qui peuvent être utilisés par des targets exécutables lors de la compilation et / ou de l'exécution

Inclure les chemins de bibliothèques et chemins

On a besoin de spécifier où les ressources peuvent être trouvées pour les targets,les fichiers header et bibliothèques :

1. Inclure les chemins - Où le code pour les headers se trouvent (le plus courant en C / C ++)
2. Les chemins des bibliothèque - Où sont situés les bibliothèques que cible exécutable a besoin

3. include_directories (<dos1>, <dir2>, ..., <dirn>)
4. link_directories (<dos1>, <dir2>, ..., <dirn>)

include_directories()

Apres l'appel de find_package() *_INCLUDE_DIRS est généré. Par contre , c'est possible qu'on ait besoin d'inclure les répertoires supplémentaires. Ici, c'est le cas : catkin et Boost nécessite cet appel include_directories ()

include_directories(include ${Boost_INCLUDE_DIRS} $catkin_INCLUDE_DIRS etc)

cvg_sim_gazebo

```
.
include_directories(
include
${catkin_INCLUDE_DIRS}
${GAZEBO_INCLUDE_DIRS}
)
```

cvg_sim_gazebo_plugins

```
.
include_directories(
include
${catkin_INCLUDE_DIRS}
${Boost_INCLUDE_DIRS}
${GAZEBO_INCLUDE_DIRS}
${SDFormat_INCLUDE_DIRS}
)
```

Targets exécutables

— Pour spécifier une cible exécutable qui doit être construit,on doit utiliser la fonction de CMake notamment **add_executable()**.

add_executable

```
add_executable(myProgram src/main.cpp src/some_file.cpp src/another_file.cpp)
```

add_executable / cvg_sim_gazebo_plugins

```
add_executable(test_trajectory src/test_trajectory.cpp)
```

— La macro **target_link_libraries()** peut être utilisée pour ajouter des chemins de bibliothèques supplémentaires. Ce macro est utilisé plutôt après

l'appel add_executable() . Exemple :

target_link_libraries

```
target_link_libraries(<executableTargetName>, <lib1>, <lib2>, ... <libN>)
```

target_link_libraries/ cvg_sim_gazebo_plugins

```
add_executable(test_trajectory src/test_trajectory.cpp)
target_link_libraries(test_trajectory ${catkin_LIBRARIES})
```

6.1.6 Messages, Services et Action

Les fichiers de messages (.msg), services (.srv) et actions (.action) de ROS demandent une étape spéciale pour la construction avant d'être construits et utilisés par les paquets ROS. Le but est de générer le fichier de la programmation d'un langage spécifique de sorte que l'on puisse utiliser les messages, les services, et les actions dans leur langage de programmation de choix. Le système de construction générera les liaisons à l'aide des générateurs disponibles (par exemple gencpp, genpy, genlisp, etc).

Il ya trois macros prévues pour gérer les messages, de services, et les actions respectivement :

- add_message_files
- add_service_files
- add_action_files

Les fichiers des messages, services, actions doivent être suivis par un appel à la macro qui invoque la génération :

Générateur des messages

```
generate_messages()
```

Ajouter les services

```
add_service_files( DIRECTORY srv FILES SetBias.srv )
```

Ici on indique auquel repertoire il faut entrer et quels fichiers doivent être utilisés.

Ajouter les messages / cvg_sim_msgs

```
.
add_message_files(
DIRECTORY msg
FILES
Altimeter.msg
Altitude.msg
AttitudeCommand.msg
Compass.msg
```

```
:
)
```

cvg_sim_msgs

```
generate_messages( DEPENDENCIES geometry_msgs )
```

Ici on génère les messages avec le paquet dépendant indiqué.

Pré-requis / Contraintes importants

— Ces macros doivent venir avant la macro catkin_package() pour que la production fonctionne correctement.

```
.
find_package (composants requis de catkin ...)
add_message_files (...)
add_service_files (...)
add_action_files (...)
generate_messages (...)
catkin_package (...)
:
```

— catkin_package() doit avoir une dépendance CATKIN_DEPENDS sur message_runtime.

```
catkin_package (CATKIN_DEPENDS message_runtime ... )
```

— find_package() pour le paquet message_generation, soit seul ou comme composant de catkin

```
find_package (catkin REQUIRED COMPONENTS message_generation)
```

— package.xml doit contenir une dépendance de construction sur message_generation et une dépendance d'exécution sur message_runtime . Ce n'est pas nécessaire si les dépendances sont tirés dans d'autres paquets transitives.

— Un paquet ayant des messages , services ainsi que des exécutables exige de créer une dépendance explicite sur la cible du message pour qu'il généré automatiquement de sorte qu'ils sont construits dans le bon ordre.

```
add_dependencies(some_target ${PROJECT_NAME}_generate_messages_cpp)
```

6.2 package.xml

Un fichier de XML doit être inclus dans la racine du paquet. Ce fichier définit les propriétés concernant le paquet telles que description,nom d'auteur, licence et

les dépendances du paquet dans rosbuild. Dans catkin,package.xml remplace manifest.xml. Contrairement au manifest.xml, package.xml contient les informations supplémentaires qui sont obligatoires notamment version ,maintenaire. En plus, les paquets dépendants non seulement nécessitent d'être configurés mais aussi d'être exécutés.

Configurer l'outil de dépendances est de spécifier les outils du système de construction ce dont le paquet a besoin pour se construire. Typiquement le seul outil de construction nécessaire est catkin.

Configurer les dépendances est de spécifier les paquets qui sont requis au moment de la construction du paquet actuel.

Cela peut être inclure les headers des paquets au moment de la compilation, lier les bibliothèques de ces paquets ou exiger toutes les autres ressources lors de la construction. En particulier les paquets qui sont trouvés par find_package() dans CMakeLists.txt sont aussi appelés dans package.xml.

Executer les dépendances précise les paquets qui sont nécessaires pour exécuter le code ou de construire des bibliothèques dans ce package.Ceci aide à partager les bibliothèques ou à inclure les headers dans les headers public de ce paquet.

Ces trois types de dépendances sont spécifiées en utilisant les balises suivantes :

1. <buildtool_depend>
2. <build_depend>
3. <run_depend>

Méta-paquet

La précision d'un metapaquet est fait en ajoutant le tag suivant dans package.xml :

```
<export>
<metapackage/>
</export>
```

6.3 Application

```
cmake_minimum_required(VERSION 2.8.3)
project(cvg_sim_gazebo_plugins)

find_package(catkin REQUIRED COMPONENTS roscpp  std_msgs geometry_msgs
             sensor_msgs image_transport nav_msgs cvg_sim_msgs ardrone_autonomy)
find_package(Boost REQUIRED COMPONENTS thread)
find_package(gazebo REQUIRED )

add_service_files(
    DIRECTORY srv
    FILES
    SetBias.srv
)


generate_messages(
  DEPENDENCIES
 geometry_msgs nav_msgs sensor_msgs std_msgs
)

catkin_package(
#  INCLUDE_DIRS include
    LIBRARIES
    diffdrive_plugin_6w
    reset_plugin
    hector_gazebo_ros_imu
    hector_gazebo_ros_magnetic
    hector_gazebo_ros_gps
    hector_gazebo_ros_sonar
    hector_gazebo_ros_baro
    hector_gazebo_quadrotor_simple_controller
    hector_gazebo_quadrotor_state_controller
 CATKIN_DEPENDS image_transport
 DEPENDS system_lib
)

include_directories(
  include
  ${catkin_INCLUDE_DIRS}
  ${Boost_INCLUDE_DIRS}
  ${GAZEBO_INCLUDE_DIRS}
  ${SDFormat_INCLUDE_DIRS}
)

add_library(diffdrive_plugin_6w src/diffdrive_plugin_6w.cpp)
target_link_libraries(diffdrive_plugin_6w ${Boost_LIBRARIES})
add_library(reset_plugin src/reset_plugin.cpp)
add_library(hector_gazebo_ros_imu src/gazebo_ros_imu.cpp)
add_dependencies(hector_gazebo_ros_imu cvg_sim_gazebo_plugins_gencpp)
target_link_libraries(hector_gazebo_ros_imu ${Boost_LIBRARIES})
add_library(hector_gazebo_ros_magnetic src/gazebo_ros_magnetic.cpp)
add_library(hector_gazebo_ros_gps src/gazebo_ros_gps.cpp)
add_library(hector_gazebo_ros_sonar src/gazebo_ros_sonar.cpp)
add_library(hector_gazebo_ros_baro src/gazebo_ros_baro.cpp)
add_dependencies(hector_gazebo_ros_baro ardrone_autonomy_gencpp)
add_library(hector_gazebo_quadrotor_simple_controller src/quadrotor_simple_controller.cpp)
add_dependencies(hector_gazebo_quadrotor_simple_controller ardrone_autonomy_gencpp)
add_library(hector_gazebo_quadrotor_state_controller src/quadrotor_state_controller.cpp)
add_dependencies(hector_gazebo_quadrotor_state_controller ardrone_autonomy_gencpp)
target_link_libraries(hector_gazebo_quadrotor_state_controller ${image_transport_LIBRARIES})

add_executable(test_trajectory src/test_trajectory.cpp)
target_link_libraries(test_trajectory ${catkin_LIBRARIES})

add_subdirectory(urdf)
```

FIGURE 6.1 – CMakeLists.txt (cvg_sim_gazebo_plugins)

```
<package>
  <name>cvg_sim_gazebo_plugins</name>
  <version>1.0.0</version>
  <description>
       . . .
  </description>

  <maintainer email="bahar.ozdemir2335@gmail.com">Bahar OZDEMIR</maintainer>
  <license>BSD</license>
  <url type="website">http://ros.org/wiki/cvg_sim_gazebo_plugins</url>
  <author>Hongrong Huang</author>

  <!-- Dependencies which this package needs to build itself. -->
  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>ardrone_autonomy</build_depend>
  <build_depend>gazebo</build_depend>
  <build_depend>geometry_msgs</build_depend>
  <build_depend>image_transport</build_depend>
  <build_depend>nav_msgs</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>cvg_sim_msgs</build_depend>

  <run_depend>cvg_sim_msgs</run_depend>
  <run_depend>ardrone_autonomy</run_depend>
  <run_depend>gazebo</run_depend>
  <run_depend>geometry_msgs</run_depend>
  <run_depend>image_transport</run_depend>
  <run_depend>nav_msgs</run_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>sensor_msgs</run_depend>
  <run_depend>std_msgs</run_depend>


<export>
    <cpp lflags="-Wl,-rpath,${prefix}/lib -L${prefix}/lib" cflags="-I${prefix}/
        include"/>
    <gazebo plugin_path="${prefix}/lib"/>
</export>

</package>
```

FIGURE 6.2 – package.xml (cvg_sim_gazebo_plugins)

6.4 Pré-requis / Contraintes importants

Tous ces modifications ne suffisent pas pour l'execution et compilation du tum_simulator car il y a des changements dans les bibliothèques utilisées par le module. Certaines bibliothèques ne sont pas compatible avec nouvelles versions de ROS. Donc ces erreurs sont apparus.

6.4.1 Les bibliothèques

physics.h

cvg_sim_gazebo_plugins/src/gazebo_ros_gps.cpp
cvg_sim_gazebo_plugins/src/gazebo_ros_magnetic.cpp
cvg_sim_gazebo_plugins/src/gazebo_ros_sonar.cpp
cvg_sim_gazebo_plugins/src/diffdrive_plugin_6w.cpp
erreur fatale : physics/physics.h : Aucun fichier ou dossier de ce type

Fuerte –> Groovy

include "physics/physics.h" –> include "physics/physics.hh"

gzmath.hh

cvg_sim_gazebo_plugins/include/hector_gazebo_plugins/sensor_model.h
erreur : 'math' does not name a type

Pour les fonctions mathématiques il faut ajouter gzmath.hh

Time.hh

cvg_sim_gazebo_plugins/include/hector_gazebo_plugins/gazebo_ros_gps.h
cvg_sim_gazebo_plugins/include/hector_gazebo_plugins/gazebo_ros_magnetic.h
cvg_sim_gazebo_plugins/include/hector_gazebo_plugins/gazebo_ros_sonar.h
cvg_sim_gazebo_plugins/include/hector_gazebo_plugins/diffdrive_plugin_6w.h
cvg_sim_gazebo_plugins/include/hector_gazebo_plugins/gazebo_ros_imu.h

include "common/Time.hh"

6.4.2 Les changements des fonctions

class gazebo : :math : :Vector3

cvg_sim_gazebo_plugins/src/gazebo_ros_imu.cpp :
erreur :'class gazebo : :math : :Vector3' has no member named *'GetDotProd'*
erreur : 'class gazebo : :math : :Vector3' has no member named *'GetCrossProd'*

Origine

.
double cos_alpha=(gravity_body+accelModel.getCurrentError()).**GetDotProd**(gravity_body)/normalization_constant ;
.
math : :Vector3 normal_vector(gravity_body.**GetCrossProd**(accelModel.getCurrentError())) ;

Modifié

.
double cos_alpha = (gravity_body+accelModel.getCurrentError()).**Dot**(gravity_body)/normalization_constant ;
.
math : :Vector3 normal_vector(gravity_body.**Cross**(accelModel.getCurrentError())) ;

cvg_sim_gazebo_plugins/src/quadrotor_simple_controller.cpp :
erreur : 'class gazebo : :math : :Vector3' has no member named **'GetDotProd'**

Origine

.
double load_factor = gravity * gravity / world->GetPhysicsEngine()->GetGravity().**GetDotProd**(gravity_body) ; // Get gravity

Modifié

.
double load_factor = gravity * gravity / world->GetPhysicsEngine()->GetGravity().**Dot**(gravity_body) ; // Get gravity

class gazebo : :math : :Angle

cvg_sim_gazebo_plugins/src/gazebo_ros_sonar.cpp :
erreur : 'class gazebo : :math : :Angle' has no member named **'GetAsRadian'**

Origine

.

range_.field_of_view=std : :min(fabs((sensor_->GetAngleMax()-sensor_->GetAngleMin()).**GetAsRadian()**),fabs((sensor_->GetVerticalAngleMax()-sensor_-> GetVerticalAngleMin()).**GetAsRadian()**)) ;

Modifié

.

range_.field_of_view = std : :min(fabs((sensor_->GetAngleMax()-sensor_ ->GetAngleMin()).**Radian()**), fabs((sensor_->GetVerticalAngleMax()-sensor_ ->GetVerticalAngleMin()).**Radian()**)) ;

6.4.3 RVIZ

RVIZ est un outil de visiualiseur 3D de ROS. Fuerte, Groovy et Hydro utilise même visualiser par contre le format du fichier de configuration est changé. En effet, Fuerte possède les fichiers « .vcg » de format « INI » tandis que Groovy et Hydro disposent « .rviz » de format « YAML ».

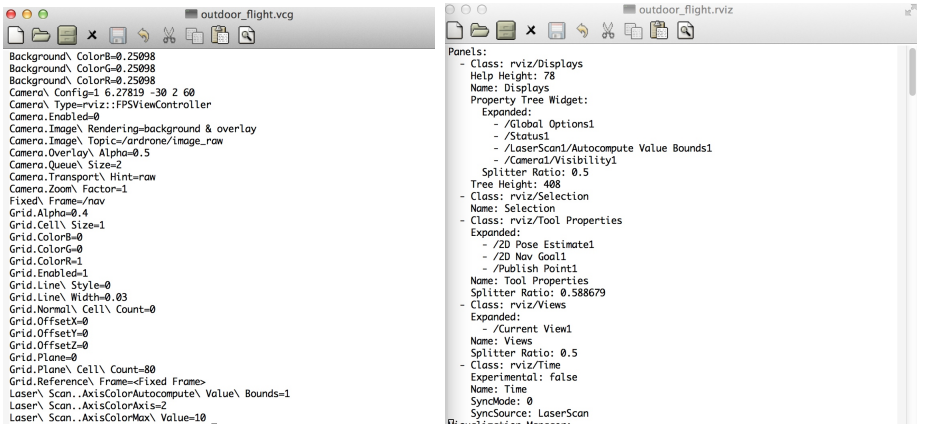


FIGURE 6.3 – Les formats RVIZ

Le fichier .launch dans lequel est défini les caractéristiques d'un noeud comprend une partie sur la configuration de rviz.

FIGURE 6.4 – RVIZ dans .launch

6.4.4 Ergonomie

Pour régler le problème d'ergonomie il faut faire des modifications sur le fichier :

/fuerte_workspace/sandbox/ardrone_helpers/ardrone_joystick/src/main.cpp

Dans **void joyCb(const sensor_msgs : :JoyConstPtr joy_msg)**

Origine

.

```
if (is_flying && !dead_man_pressed){
```

```
ROS_INFO("L1 was released, landing");
pub_land.publish(std_msgs : :Empty());
is_flying = false;
}
```

Correction

.

```
if (is_flying && dead_man_pressed){
ROS_INFO("L1 was released, landing");
pub_land.publish(std_msgs : :Empty());
is_flying = false;
}
```

Cette modification permet au drone de rester s'envoler une fois qu'il a décollé.S'il est au mode vole, l'appuie sur la manette fait atterrir le drone.

7 Expérimentation

7.1 tum_simulator

7.1.1 Mode d'utilisation

1. Metter tous les paquets nécessaires dans la workspace de catkin : tum_simulator et paquet de ardrone_autonomy (qui contient certaines formes de message standard pour simulateur de ARDrone).
2. construire des paquets cvg_sim_gazebo_plugins et message_to_tf.
   catkin_ws/src/$ rosmake cvg_sim_gazebo_plugins
   catkin_ws/src/$ rosmake message_to_tf
3. Lancer un maître de ros en tapant la commande suivante dans la console
   roscore
4. Exécutez une simulation en exécutant un fichier de lancement dans le paquet cvg_test_sim :
   roslaunch cvg_sim_test outdoor_flight.launch

7.1.2 Resultats

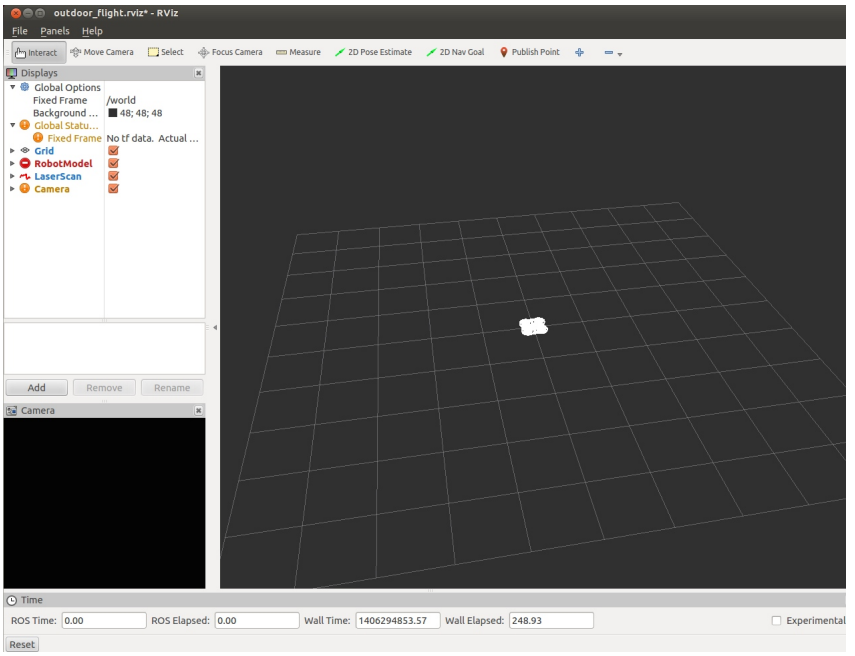


FIGURE 7.1 – tum_simulator

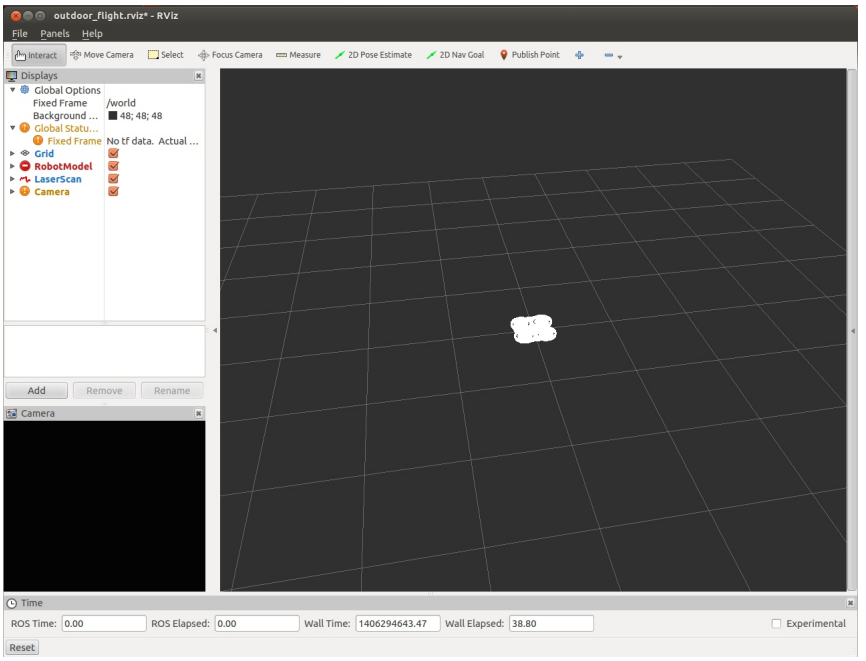


FIGURE 7.2 – tum_simulator

8 Conclusion

Cette article présente un portage d ?un module d ?une version ROS vers les nouveaux. Tous les travaux requis afin d'atteindre le but sont faits en succès.En effet, les dépendances et les libraries sont repérées, l'architecture est modifiée, l'ergonomie de joystick est améliorée dans le paquet «ardrone helpers ». Le simulateur est lancé avec Rviz.

Ce travail a nécessité une grande partie de recherche , comparaison,consultation et de réalisation de documentation, mais m'a permis également d'effectuer un travail recherche intéressent.

D'après ces expériences, ce stage m'a été bénéfique, car il m'a permis de découvrir et apprécier le monde du travail, et d'utiliser mes connaissances acquises au cours de ma formation professionnelle en Informatique.il m'a permis également d'acquérir de nouvelles connaissances et d'entretenir des relations professionnelles et humaines.