Robot Operating System

Introduction

Olivier Stasse

PUBLISHED BY PUBLISHER

BOOK-WEBSITE.COM

First printing, xxx

Table des matières

Introduction à ROS

1. Introduction

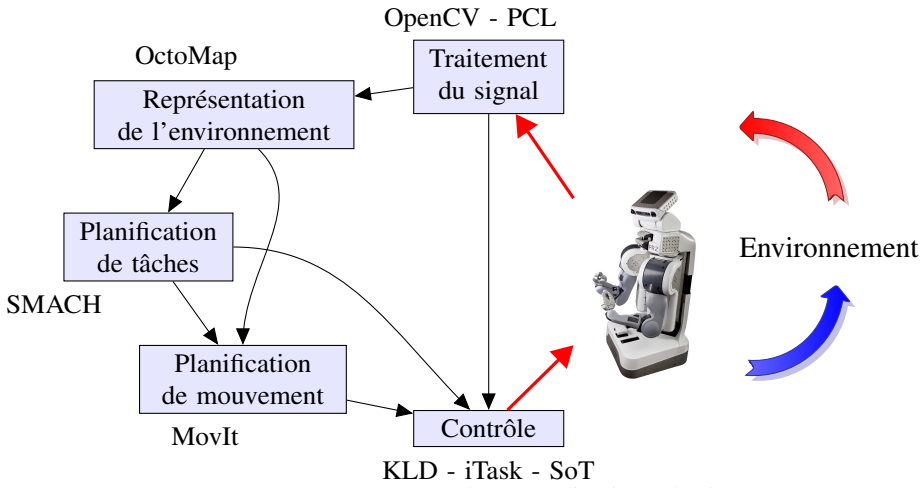


FIGURE 1.1 – Structure d'une application robotique

1.1 Motivations

1.1.1 Complexité des systèmes robotiques

Les systèmes robotiques font appel à de nombreuses compétences telles que la mécanique, l'électrotechnique, le contrôle, la vision par ordinateur, et de nombreux pans de l'informatique comme l'informatique temps-réel, le parallélisme, et le réseau. Souvent les avancées en robotique dépendent de verrous technologiques résolus dans ces champs scientifiques. Un système robotique peut donc être complexe et faire intervenir de nombreuses personnes. Afin de pouvoir être efficace dans la conception de son système robotique il est donc impératif de pouvoir réutiliser au maximum des outils existants et de pouvoir collaborer efficacement avec d'autres équipes.

En physique, des systèmes à grande échelle comme le *Large Hadron Collider* ont vu le jour grâce à des collaborations internationales à grande échelle. En informatique des projets comme le système d'exploitation Linux n'existe qu'à travers une collaboration internationale. Celle-ci peut impliquer des individus, mais on trouve également le support de sociétés qui ne souhaitent pas s'engager dans le développement complet

d'un système d'exploitation mais qui souhaiteraient de nouvelles fonctionnalités. En effet cette approche est souvent moins coûteuse car le développement d'un système d'exploitation est évaluer à 100 années pour un seul homme.

Le système ROS pour Robotics Operating System est le premier projet collaboratif robotique à grande échelle qui fournit un ensemble d'outils informatique permettant de gagner du temps dans la mise au point d'un robot, ou d'un système robotique.

1.1.2 Application Robotique

Une application robotique suit en général la structure représentée dans la figure 1. Elle consiste à générer une commande afin qu'un robot puisse agir sur un environnement à partir de sa perception de celui-ci. Il existe plusieurs boucles de *perception-action*. Elles peuvent être :

$40\ Khz$ très rapide par exemple pour le contrôle du courant,

$1\ Khz - 200\ Hz$ rapide, par exemple pour le contrôle du corps complet d'un robot humanoïde,

$1\ s - 5\ mn$ lente, par exemple dans le cadre de la planification de mouvements réactifs dans des environnements complexes.

$2 - 72h$ trés lente, par exemple dans le cadre de planification à grande échelle, ou sur des systèmes avec de trés grands nombre de degrés de libertés.

Dans tous les cas, il est nécessaire de transporter l'information entre des composants logiciels largement hétérogènes et faisant appel à des compétences très différentes. Chacun de ces composants logiciels a une structure spécifique qui dépend du problème à résoudre et de la façon dont il est résolu. Chacun est encore très actif du point de vue de la recherche scientifique, ou demande des niveaux de technicité relativement avancés. La réprésentation de l'environnement par exemple a connu récémment beaucoup de changements, et on trouve maintenant des prototypes industriels permettant de construire en temps réel des cartes denses de l'environnement sur des systèmes embarqués [2]. Il est donc nécessaire de pouvoir facilement tester des structures logiciels implémentant des algorithmes différents pour une même fonctionalité sans pour autant changer l'ensemble de la structure de l'application. Par exemple on veut pouvoir changer la boîte "Représentation de l'environnement" sans avoir à modifier les autres boîtes. Ceci est réalisé en utilisant des messages standards et les mêmes appels de service pour accéder aux logiciels correspondants à ces boîtes. Enfin, on souhaiterait pouvoir analyser le comportement du robot à différents niveaux pour investiguer les problèmes potentiels ou mesurer les performances du système. Dans le paragraphe suivant, les concepts de ROS permettant de répondre à ces challenges sont introduits.

1.2 Concepts de ROS

La clef de ROS est la mise en place de 4 grands types de mécanismes permettant de construire une application robotique comme celle décrite dans le paragraphe précédent. Ces mécanismes, décrits dans la figure. 1.2.1, sont :

- la *Plomberie*, c'est à dire la connection des composants logiciels entre eux quelque soit la répartition des noeuds de calcul sur un réseau,
- les *Outils*, c'est à dire un ensemble de logiciels permettant d'analyser, d'afficher et de débugger une application répartie,
- les *Capacités*, c'est à dire des librairies qui implémentent les fonctionnalités telles que la planification de tâches (SMACH), de mouvements (OMPL), la construction d'un modèle de l'environnement (Octomap),
- l'*Ecosystème*, c'est à dire un nombre suffisant d'utilisateurs tels que ceux-ci ont plus intérêt à collaborer plutôt qu'à reconstruire les mêmes outils.

1.2.1 Plomberie

La plomberie, représentée dans la figure 1.2.1, est implémentée grâce à un middleware. Un middleware fournit un bus logiciel qui permet à des objets localisés sur différents ordinateurs d'interagir. L'interaction s'effectue en transmettant des données sous une forme normalisée (les *messages*) via ce que pourrait voir comme des post-its (les *topics*). Un noeud de calcul (un *node*) produisant des données peut ainsi inscrire des données qu'il produit sur un post-it en utilisant le nom de celui-ci. Un autre noeud lui lira les données sur le post-it sans savoir quel noeud a produit ces données. La normalisation des données est réalisée grâce à un langage de description d'interface. Des programmes permettent de générer automatiquement la transcription

FIGURE 1.2 – Concepts généraux sous-jacents à ROS

entre ce langage de description et des langages de programmation comme le C++, python ou java. Afin de pouvoir communiquer entre les objets, le système fournit un annuaire sur lequel viennent s'enregistrer chacun des noeuds de calcul. Enfin un noeud peut demander un service à un autre noeud en suivant un mécanisme de d'appel de service à distance. Ces mécanismes très généraux existent aussi chez d'autre middlewares comme Corba, ou YARP un autre middleware robotique. En général un middleware fournit les mécanismes d'appels pour différents langagues et différents systèmes d'exploitation. On peut également stockés des paramètres qui correspondent à des données spécifiques à une application, à un robot ou des données qui d'une manière générale n'évoluent pas au cours de l'exécution d'une application.

1.2.2 Outils

Les outils sont une des raisons du succés de ROS. Nous trouvons ainsi :

— *rviz* : Une interface graphique permettant d'afficher les modèles des robots, des cartes de navigation reconstruites par des algorithmes de SLAM, d'interagir avec le robot, d'afficher des images, des points 3D fournis par des caméras 3D.

— *rqt_graph* : Une interface graphique permettant d'analyser le graphe d'applications et les transferts de données via les topics.

— *rosbag* : Un programme permettant d'enregistrer et de rejouer des séquences topics. La fréquence, la durée les topics à enregistrer peuvent être spécifiés.

— *rqt* : Une interface de contrôle incrémentale. Elle se base sur le système de plugins de la bibliothèque de GUI Qt.

— *catkin* : Un système de gestion de paquets, de génération de code automatique et compilation.

Chacun de ces outils a également ses propres limites, et de nombreux alternatives existent dans les paquets fournis sur ROS.

1.2.3 Capacités

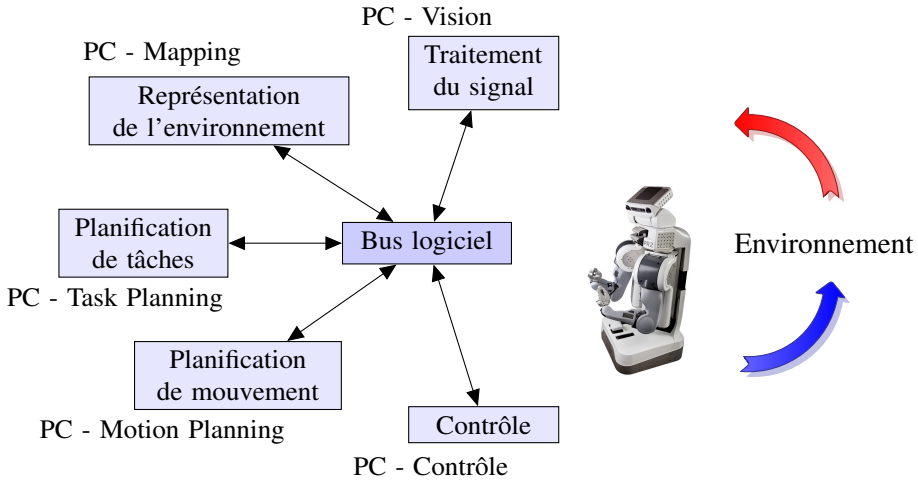


FIGURE 1.3 – Structure d'une application robotique

Les différents fonctionnalités décrites dans la figure 1 sont implémentés par des librairies spécialisés décrites dans la figure 1.2.3. Chacune est issue de plusieurs décennies de recherche en robotique et souvent encore l'objet de nombreuses avancées. Parmi celles-ci on peut trouver :

— Pour la vision :

  - La point cloud library
  - La librairie de vision par ordinateur opencv.
- Pour la reconstruction d'environnements :
  - En 2D :
  - En 3D : KinFu.
- Pour la planification de mouvements :
  - MoveIt !
- Pour la planification de mission :
  - SMASH :
  -
- Pour le contrôle :
  - iTask
  - ControlIt
  - StackOfTasks

1.2.4 Exemples

ROS a été utilisé comme support pour des challenges robotiques comme le DARPA Robotics Challence et le challenge robotique de la NASA. Dans le cade du DARPA Robotics Challenge qui a pris place en Juin 2015, de nombreuses équipes ont du faire face à des défis d'intégration. ROS a été une des briques fondamentales pour permettre aux équipes utilisant ATLAS de pouvoir fonctionner.

Examinons maintenant un exemple simple qui consiste à mettre en place un asservissement visuel sur un robot TurtleBot 2.

1.3 Ecosystème

1.3.1 Historique

ROS a été écrit à partir de 2008 par la société Willow Garage fondée par Larry Page, également fondateur de Google. En plus de ROS, cette société a construit le PR-2, un robot très sophistiqué équippé de deux bras, d'une base mobile et avec des moyens de calcul importants. Les implémentations effectués par cette société ont été nombreux et ont permis d'atteindre une masse critique d'utilisateurs sans commune mesure avec les actions concertées précédentes. Pour cette raison on retrouve dans les documentations des paquets historiques souvent la référence à Willow Garage. Cette société a cependant été fermée en 2013 et a fait place à l'Open Source Robotics Foundation financé par plusieurs société. Un des faits récents les plus notables est le don de 50 millions de dollars effectués par Toyota, suivi d'un autre investissement de 50 millions de dollars pour construire une structure privée adossée à l'OSRF.

L'historique des releases de ROS est résumé dans le tableau 1.1. Si on note au démarrage un manque de régularité, la politique actuelle consiste à caler les releases de ROS sur celles d'Ubuntu en Long Time Support (LTS) et celles plus courtes des années impaires. D'une manière générale on constate un décalage d'au moins une année entre la release de ROS et celle de la mise à jour des applications robotiques. C'est en général le temps qu'il faut pour résoudre les erreurs des releases en LTS et porter le code des applications robotiques.

1.3.2 Fonctionnalités

Il existe un certain nombre de logiciels très bien supportés par des projets et des communautés qui offre des fonctionnalités très intéressantes pour la robotique. Certains existent par ailleurs ROS et d'autres sont supportés directement par l'OSRF. En voici une liste non-exhaustive :

- **Définition de messages standards pour les robots**.
  ros_comm est le projet qui définit et implémente le mécanisme de communication initial et les types initiaux. De nombreux paquets viennent complèter les types nécessaires dans les applications robotiques.
- **Librairie pour la géométrie des robots**.
  La librarie KDL est utilisée pour implémenter le calcul de la position de tous les corps du robot en fonction de ces actionneurs et du modèle des robots.
- **Un language de description des robots**.
  URDF pour Universal Robot Description Format est un spécification en XML décrivant l'arbre

2008	Démarrage de ROS par Willow Garage
2010 - Janvier	ROS 1.0
2010 - Mars	Box Turtle
2010 - Aout	C Turtle
2011 - Mars	Diamondback
2011 - Aout	Electric Emys
2012 - Avril	Fuerte
2012 - Décembre	Groovy Galapagos
2013 - Février	Open Source Robotics Fundation poursuit la gestion de ROS
2013 - Aout	Willow Garage est absorbé par Suitable Technologies
2013 - Aout	Le support de PR-2 est repris par Clearpath Robotics
2013 - Septembre	Hydro Medusa (prévu)
2014 - Juillet	Indigo Igloo (EOL - Avril 2019)
2015 - May	Jade Turtle (EOL - Mai 2017)
2016 - May	Kinetic Kame (EOL - Mai 2021)
2017 - May	Lunar Loggerhead

Tableau 1.1 – Historique de ROS et des releases

cinématique d'un robot, ses caractéristiques dynamiques, sa géométrie et ses capteurs.
- **Un système d'appel de fonctions à distance interruptible**.
- **Un système de diagnostiques**.
  Il s'agit ici d'un ensemble d'outils permettant de visualiser des données (rviz), d'afficher des messages (roslog/rosout), d'enregister et de rejouer des données (rosbag), et d'analyser la structure de l'application (rosgraph)
- **Des algorithmes d'estimation de pose**.
  Des implémentations de filtre de Kalman, des systèmes de reconnaissance de pose d'objets basé sur la vision (ViSP).
- **Localisation - Cartographie - Navigation**.
  Des implémentations d'algorithmes de SLAM (Self Localization and Map building) permettent de construire une représentation 2D de l'environnement et de localiser le robot dans cet environnement.
- **Une structure de contrôle**.
  Afin de pouvoir normaliser l'interaction entre la partie haut-niveau d'une application robotique que sont la planification de mouvements, la planification de missions, avec partie matérielle des robots, le framework ros-control a été proposé.

1.3.3 Systèmes supportés

ROS ne supporte officiellement que la distribution linux Ubuntu de la société Canonical. La raison principale est qu'il est très difficile de porter l'ensemble de ROS sur tous les systèmes d'exploitation cela nécessiterait de la part de tous les contributeurs de faire l'effort de porter tous les codes sur plusieurs systèmes. Certains logiciels sont très spécifiques à des plateformes particulières et ont peu de sens en dehors de leur cas de figure. A l'inverse le coeur de ros constitués des paquets roscomm a été porté sur différents systèmes d'exploitations comme OS X, Windows, et des distributions embarquées de Linux (Angstrom, OpenEmbedded/Yocto).

Les releases de ROS sont spécifiées par la REP (ROS Enhancement Proposal) : `http://www.ros.org/reps/rep-0003.html`

Ceci donne pour les dernières releases :
- Lunar Loggerhead (Mai 2017 - Mai 2019)
  - Ubuntu Xenial (16.04 LTS)
  - Ubuntu Yakkety (16.10)
  - Ubuntu Zesty (17.04)
  - C++11, Boost 1.58/1.61/1.62/, Lisp SBCL 1.2.4, Python 2.7, CMake 3.5.1/3.5.2/3.7.2
  - Ogre3D 1.9.x, Gazebo 7.0/7.3.1/7.5, PCL 1.7.2/1.8.0, OpenCV 3.2, Qt 5.5.1/5.6.1/5.7.1, PyQt5
- Kinetic Kame (Mai 2016 - Mai 2021)

  - Ubuntu Wily (15.10)
  - Ubuntu Xenial (16.04 LTS)
  - C++11, Boost 1.55, Lisp SBCL 1.2.4, Python 2.7, CMake 3.0.2
  - Ogre3D 1.9.x, Gazebo 7, PCL 1.7.x, OpenCV 3.1.x, Qt 5.3.x, PyQt5
- Indigo Igloo (May 2014)
  - Ubuntu Saucy (13.10)
  - Ubuntu Trusty (14.04 LTS)
  - C++03, Boost 1.53, Lisp SBCL 1.0.x, Python 2.7, (Test additionels contre Python 3.3 recommendés) CMake 2.8.11

1.4 Exemple : Asservissement visuel sur TurtleBot 2

Afin d'illustrer le gain à utiliser ROS considérons l'application illustrée dans la Fig.1.4. On souhaite utiliser un robot Turtlebot 2 équipé d'une Kinect et d'une base mobile Kobuki. Le but est de contrôler le robot de telle sorte à ce qu'il reconnaisse un objet par sa couleur et qu'il puisse génére une commande à la base mobile de telle sorte à ce que l'objet soit centré dans le plan image de la caméra du robot. Pour cela on doit extraire une image I du robot et générer un torseur cinématique $\begin{pmatrix} v \\ w \end{pmatrix}$ pour la base mobile.

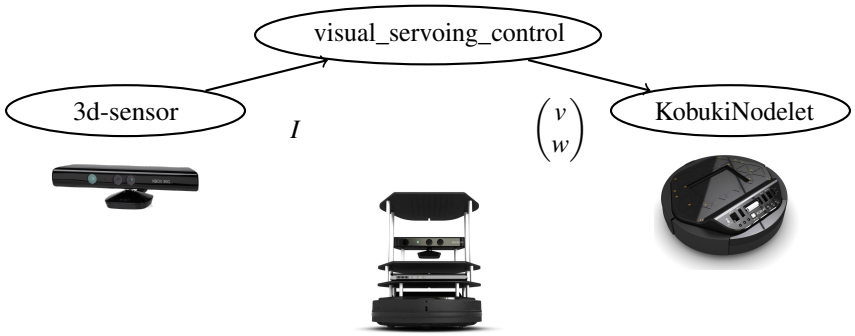


FIGURE 1.4 – Transfert d'une image I vers un node de contrôle qui calcule un torseur cinématique pour contrôler la base mobile du robot.

L'implémentation de cette structure logicielle s'effectue de la façon suivante sous ROS. En premier on utilise un noeud de calcul qui s'occupe de l'extraction de l'image d'une caméra Kinect. Dans la Fig.1.5 il est représenté par l'ellipse nommée **3d-sensor**. L'image I est alors publiée dans un topic appelé **/camera/image_color** qui peut-être vu comme un post-it. Celui-ci peut-être lu par un autre noeud de calcul appelé **visual_servoing_control** qui produit le torseur cinématique $\begin{pmatrix} v \\ w \end{pmatrix}$. Ce torseur est publié sur un autre topic appellé **/mobilebase/command/velocity**. Finalement celui-ci est lu par un troisième noeud de calcul appellé **KobukiNodelet**. Il est en charge de lire le torseur donné par rapport au centre de la base et le transformer en commande pour les moteurs afin de faire bouger la base mobile dans la direction indiquée.

Il est intéressant de noter que chacun des noeuds de calcul fournissent des fonctionnalités qui peuvent être indépendantes de l'application considérée. On peut par exemple utilise le noeud pour la Kinect sur un autre robot qui utiliserait le même capteur. Le programmateur de robot n'est ainsi pas obligé de réécrire à chaque fois un programme sensiblement identique. De même la base mobile peut-être bougée suivant une loi de commande différente de celle calculée par le noeud **visual_servoing_control**. Si un noeud est écrit de façon générique on peut donc le mutualiser à travers plusieurs applications robotiques, voir à travers plusieurs robots. Enfin chacun de ces noeuds peut-être sur un ordinateur différent des autres noeuds. Cependant il est souvent nécessaire de pouvoir changer des paramètres du noeud afin de pouvoir l'adapter à l'application visée. Par exemple, on souhaiterait pouvoir contrôler les gains du noeud **visual_servoing_control**. Il est alors possible d'utiliser la notion de paramétres pour pouvoir changer les gains proportionnels K_p et dérivés K_d du contrôleur implémenté dans le noeud **visual_servoing_control** (cf Fig.1.6).

Comme l'illustre la Fig.1.7 dans ROS, les programmes permettant à tous les noeuds d'interagir ensemble soit à travers des topics soit à travers des services sont **roscomm** (pour la communication) et **rosmaster**

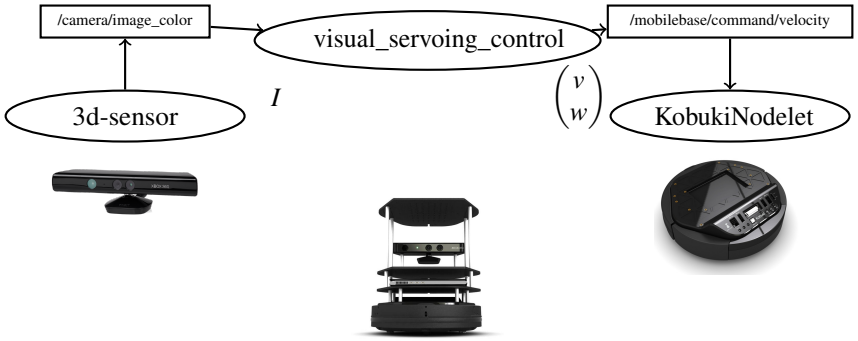


FIGURE 1.5 – Graphe d'une application ROS. Les ellipses représentent les noeuds de calcul ROS, et les rectangles les post-its où sont stockés des données disponibles pour tous les noeuds. Les flèches indiquent les flux d'informations échangées dans l'application.

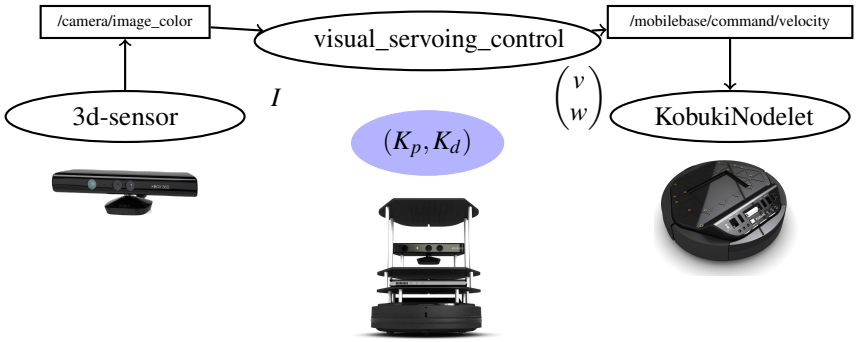


FIGURE 1.6 – Utilisation des paramètres pour moduler le comportement d'une loi de commande. Ici les paramètres K_p et K_d correspondent respectivement au gain proportionnel et au gain dérivé

(pour les références entre noeuds). Deux autres programmes s'occupent de fournir les accés aux topics **rostopic** et aux paramètres **rosparam**.

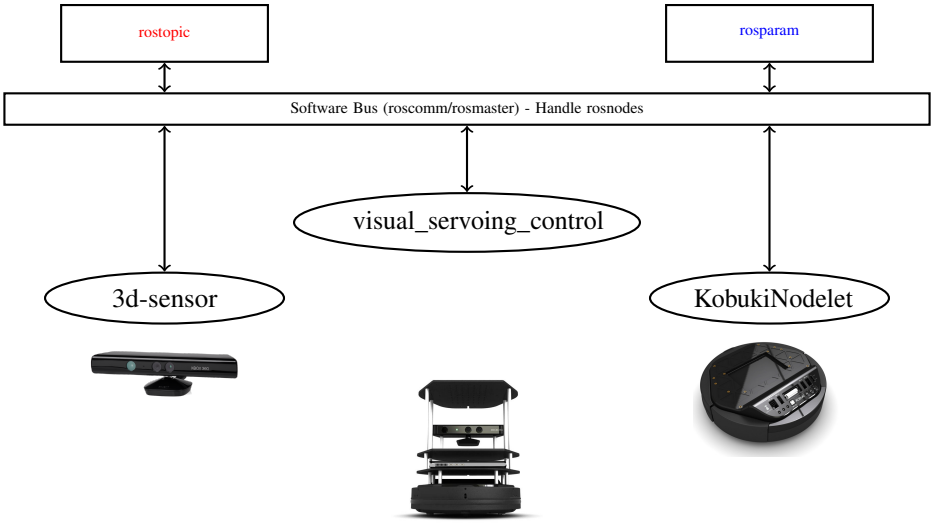


FIGURE 1.7 – Organisation interne de ROS pour implémenter les différents concepts

2. Les paquets ROS

Le système de gestion des paquets ROS est probablement un élèment clé de la réussite de ROS. Il est bien plus simple que le système de gestion des paquets par Debian. La contre-partie est que ce système ne fonctionne pas au-delà de l'ecosystème ROS.

2.1 Configuration de l'environnement

2.1.1 Initialiser les variables d'environnement

Afin de pouvoir utiliser ROS il est nécessaire de configurer son environnement shell. Sous linux, bash est le shell le plus utilisé. Il suffit donc d'ajouter la ligne suivante au fichier *.bashrc*

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

Pour créer son espace de travail il faut d'abord créer le répertoire associé

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

Puis ensuite initialisé l'espace de travail

```
catkin_init_workspace
```

Le tutoriel associé est : Tutorial Installing and Configuring Your ROS Environment : `http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment`

Il est également recommandé de rajouter la ligne suivante dans le fichier *.bashrc* :

```
source $HOME/catkin ws/devel/setup.bash
```

Cependant le répertoire **devel** n'existe qu'une fois le script **catkin_make** utilisé (voir section2.5 pour plus de détails).

Une fois le fichier **setup.bash** sourcé les variables d'environnement il y a deux variables d'environnement trés importantes :

```
env | grep ROS
ROS_MASTER_URI=http://localhost:11311
ROS_PACKAGE_PATH=/opt/ros/indigo/share
```

La variable **ROS_MASTER_URI** indique comment trouver l'annuaire des objets. Elle suit la structure suivante :

```
ROS_MASTER_URI=http://hostname:port_number
```

hostname indique l'ordinateur où **roscore** est lancé. **port_number** indique le port où **roscore** attend les connections.

La variable d'environnment *ROS_PACKAGE_PATH* indique les répertoires où chercher les paquets. Par défaut elle indique les paquets systèmes de la release ROS.

2.1.2 Navigation dans le système de paquets

On peut chercher un paquet avec le script **rospack**. Par exemple pour trouver le paquet **roscpp** :

```
rospack find roscpp
```

Comme il arrive souvent de devoir naviguer entre des paquets systèmes et des paquets de l'environnement de développement, il existe un équivalent de **cd** qui permet de naviguer directement entre les paquets :

```
roscd roscpp
roscd roscpp/cmake
```

Finalement il existe un répertoire particulier qui contient les logs de tous les nodes. On peut y accéder directement par :

```
roscd log
```

Le tutoriel associé est : Tutorial Navigation dans le système de fichiers
`http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem`

2.2 Structure générale des paquets ROS

Pour l'organisation des paquets ROS une bonne pratique consiste à les regrouper par cohérence fonctionnelle. Par exemple si un groupe de roboticiens travaille sur des algorithmes de contrôle, il vaut mieux les regrouper ensemble. Il est possible de faire cela sous forme d'un espace de travail (appelé également *workspace*). La figure Fig.2.1 représente une telle architecture. Dans un workspace, il faut impérativement créer un repertoire **src** dans lequel on peut créer un repertoire par paquet qui contient les fichiers d'un paquet.

Par exemple le paquet **package_1** contient au minimum deux fichiers :

— CMakeLists.txt : Le fichier indiquant comment compiler et installer le paquet
— package.xml : Le fichier ROS décrivant l'identité du paquet et ses dépendences.

Jusqu'à ROS groovy, il était possible de mettre ensemble des paquets dans des Stacks. Ceci a été simplifié et depuis l'introduction de **catkin** les Stacks ne sont plus supportées.4 Il suffit de mettre des paquets dans le même répertoire sous le répertoire **src** et sans aucun autre fichier pour faire des regroupements. Dans la figure Fig.2.1 on trouve par exemple le répertoire **meta_package_i** qui regroupe les paquets de **meta_package_i_sub_0** à **meta_package_i_sub_j**. On trouve par exemple des projets sous forme de dépôts qui regroupe de nombreux paquets légers ou qui contiennent essentiellement des fichiers de configuration. C'est par exemple le cas pour les dépots qui contiennent les paquets décrivant un robot. C'est le cas du robot TIAGO de la société PAL-Robotics https://github.com/pal-robotics/tiago_robot.

Au même niveau que **src**, la plomberie de ROS va créer trois répertoires : **build**, **devel**, **install**.

Le répertoire **build** est utilisé pour la construction des paquets et à ce titre contient tous les fichiers objets. Le répertoire **devel** contient les exécutables et les librairies issus de la compilation des paquets et donc spécifiés comme cible dans le fichier **CMakeLists.txt**. Le répertoire **install** ne contient que les fichiers qui sont explicitement installés à travers les directives d'installation spécifiées dans le fichier **CMakeLists.txt** des paquets.

2.3 Création d'un paquet

Pour la création d'un paquet il faut donc se placer dans le répertoire **src** de l'espace de travail choisi. Dans la suite on supposera que l'espace de travail est le répertoire **catkin_ws**.

```
cd ~/catkin_ws/src
```

Pour créer un nouveau package et ses dépendences il faut utiliser la commande **catkin_create_pkg** en spécifiant le nom du paquet et chacune des dépendences.

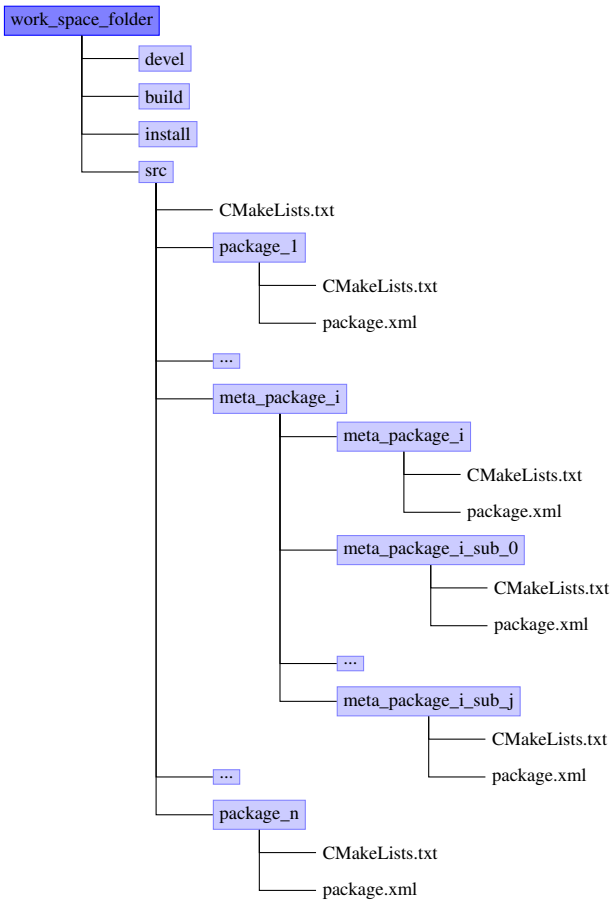


FIGURE 2.1 – Organisation de l'espace de travail : les paquets sources n'ont pas de structure hiérarchique (ou à un seul niveau lorsque les dépôts contiennent plusieurs paquets ROS).

```
catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
catkin_create_pkg [package name] [depend1] [depend2] [depend3]
```

On peut alors afficher les dépendences du premier ordre en utilisant la commande **rospack**.

```
rospack depends1 beginner_tutorials
std_msgs
rospy
roscpp
```

Les dépendences sont stockées dans le fichier **package.xml**.

```
roscd beginner_tutorials
cat package.xml
<package>
...
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
...
</package>
```

2.4 Description des paquets : le fichier package.xml

Jusqu'à ROS groovy les paquets devait inclure un fichier de description appelé **manifest.xml**. Depuis l'introduction de **catkin** le fichier qui doit être inclus est **package.xml**.

Il existe deux versions du format spécifié dans le champ <package>. Pour les nouveaux paquets, le format recommandé est le format 2.

```
<package format=2>
</package>
```

Il y 5 champs nécessaires pour que le manifeste du paquet soit complet :

— <name> : Le nom du paquet.
— <version> : Le numéro du version du paquet.
— <description> : Une description du contenu du paquet.
— <maintainer> : Le nom de la personne qui s'occupe de la maintenance du paquet ROS.
— <license> : La license sous laquelle le code est publié.

La license utilisée le plus souvent sous ROS est BSD car elle permet d'utiliser ces codes également pour des applications commerciales.

Il est important de faire attention au contexte dans lequel le code peut-être licensié notamment dans un cadre professionnel.

```
<?xml version="1.0"?>
<package>
  <name>beginner_tutorials</name>
  <version>0.0.0</version>
  <description>The beginner_tutorials package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <maintainer email="jane.doe@example.com">Jane Doe</maintainer>

  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>
```

Le site web du paquet peut-être indiqué dans le champ <url>. C'est l'endroit où on trouve généralement la documentation.

```
  <!-- Url tags are optional, but mutiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/beginner_tutorials</url> -->
```

Certains paquets servent à utiliser des logiciels tierces dans le cadre de ROS. Dans ce cas le paquet ROS encapsule ce logiciel tierce. C'est dans le champ <author> que l'on peut spécifier le nom de l'auteur de ce logiciel tierce. Il est possible de mettre plusieurs champs <author>.

```
<!-- Author tags are optional, mutiple are allowed, one per tag -->
<!-- Authors do not have to be maintianers, but could be -->
<!-- Example: -->
<!-- <author email="jane.doe@example.com">Jane Doe</author> -->
```

Les dépendences sont de 6 ordres :

— build dependencies (format 1 & 2) : spécifie les paquets nécessaires pour construire ce paquet. Il peut s'agir des paquets qui contiennet des en-têtes nécessaires à la compilation.
— build export dependencies (format 2) : spécifie les paquets nécessaires pour construire des librairies avec ce paquet. Notamment lorsque les en-têtes publiques de ce paquet font référence à ces paquets.
— execution dependencies (format 1 & 2) : spécifie les paquets nécessaires pour exécuter des programmes fournis par ce paquet. C'est le cas par exemple lorsque ce paquet dépend de librairies partagées.
— test dependencies (format 1 & 2) : spécifie uniquement les dépendences additionnelles pour les tests unitaires.
— build tool dependencies (format 1 & 2) : spécifie les outils de construction système nécessaires pour le construire. Typiquement le seul outil nécessaire est catkin. Dans le cas de la compilation croisée les outils sont ceux de l'architecture sur laquelle la compilation est effectuée.
— doc dependencies (format 2) : spécifie les outils pour générer la documentation.

Pour notre exemple voici les dépendences au format 1 avec ROS indigo exprimés dans le fichier **package.xml**.

```
<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>message_generation</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use run_depend for packages you need at runtime: -->
<!--   <run_depend>message_runtime</run_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!--   <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>
```

Lorsque l'on souhaite groupé ensemble des paquets nous avons vu qu'il était possible de créer un **metapackage**. Ceci peut-être accompli en créant un paquet qui contient dans le tag suivant dans la section export :

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

Les metapackages ne peuvent avoir des dépendences d'exécution que sur les paquets qu'il regroupe.

```
<!-- The export tag contains other, unspecified, tags -->
<export>
  <!-- Other tools can request additional information be placed here -->

</export>
</package>
```

2.5 Compilation des paquets indigo

La compilation de tous les paquets s'effectue avec une seule commande. Cette commande être exécutée dans le répertoire du workspace. C'est cette commande qui déclenche la création des répertoires **build** et

devel.

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

Les répertoires **build** et **devel** étant créés automatiquement ils peuvent être effacés sans problème tant que le fichier **CMakeLists.txt** est cohérent avec l'environnement de programmation. Il est indispensable de le faire si le workspace a été déplacé sur le disque.

3. Graphe d'application avec ROS

Dans ce chapitre nous allons introduire les concepts de base permettant la construction d'une application robotique distribuée sur plusieurs ordinateurs. Notons que les aspects temps-réel ne sont pas directement adressés ici.

3.1 Introduction

On retrouve dans ROS un certain nombre de concepts commun à beaucoup de middleware :

— Nodes : Un node est un exécutable qui utilise ROS pour communiquer avec d'autres nodes.
— Messages : Type de données ROS utilisés pour souscrire ou publier sur un topic.
— Topics : Les nodes peuvent *publier* des messages sur un topic aussi bien que *souscrire* à un topic pour recevoir des messages.
— Master : Nom du service pour ROS (i.e. aide les noeuds à se trouver mutuellement).
— Paramètres : Informations très peu dynamiques qui doivent être partagés dans l'application.
— rosout : Equivalent de stdout/stderr.
— roscore : Master+rosout+parameter server (serveur de paramètres).

3.1.1 Définition d'un node

On peut définir un node comme étant un fichier exécutable dans un paquet ROS. Les noeuds ROS utilisent une librairie client pour communiquer avec les autres noeuds. Les noeuds peuvent publier ou souscrire à des topics. Les noeuds peuvent fournir ou utiliser un service. Les librairies client sont *rospy* pour python et *roscpp* pour C++.

3.1.2 Le système des name services

Pour pouvoir localiser et interagir entre les nodes il faut un **annuaire**. C'est le rôle de **roscore**. Il doit être lancé *systématiquement* mais une seule fois.

```
1 roscore
```

Si **roscore** ne se lance pas, il faut :

— Vérifier la connection réseau.
— Si roscore ne s'initialise pas et évoque des un problème de droits sur le répertoire .ros

En général on ne lance pas **roscore** directement ou à des fins de debuggage. Le lancement d'un graphe d'application s'effectue en général par **roslaunch** qui lance lui-même **roscore**. L'utilisation de **roslaunch**

nécessite un fichier XML décrivant le graphe d'application. Son utilisation est vue plus en détails dans le paragraphe 3.3.

Pour obtenir la liste des nodes actifs on peut utiliser la commande **rosnode** :

```
rosnode list
```

Pour obtenir des informations sur un node :

```
rosnode info /rosout
```

Tutoriel associé : Tutorial Understanding ROS Nodes : `http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes`

3.1.3 Lancer des nodes

On peut lancer un fichier exécutable/node d'un paquet :

```
rosrun [package_name][node_name]
```

Par exemple pour lancer le node **turtlesim** :

```
rosrun turtlesim turtlesim_node
```

Une fenêtre bleue avec une tortue apparaît comme celle affichée dans la figure 3.1. La tortue peut-être différente car elles sont tirées aléatoirement dans un ensemble de tortues. Si vous souhaitez lancer le

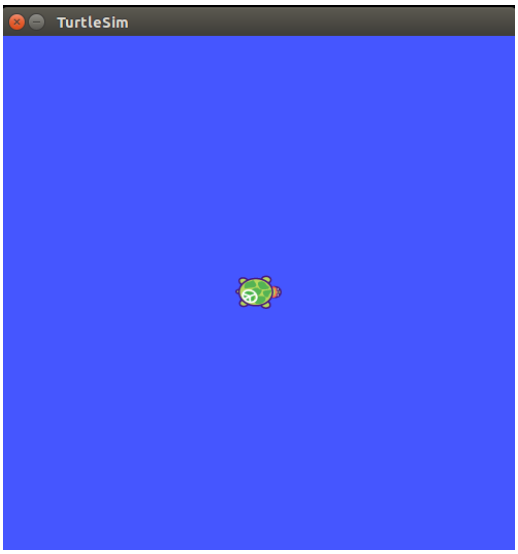


FIGURE 3.1 – Fenêtre résultat de rosrun turtlesim turtlesim_node

programme avec un nom différent il est possible de spécifier :

```
rosrun turtlesim turtlesim_node __name:=my_turtle
```

Pour vérifier si un node est actif on peut utiliser la commande **rosnode** :

```
rosnode ping my_turtle
```

Tutoriel associé : Tutorial Understanding ROS Nodes : `http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes`

Afin de pouvoir commander la tortue dans la fenêtre il faut utiliser le node **turtle_teleop_key** qui permet d'envoyer des commandes avec le clavier :

```
rosrun turtlesim turtle_teleop_key
```

Pour visualiser le graphe de l'application on peut lancer la commande suivante :

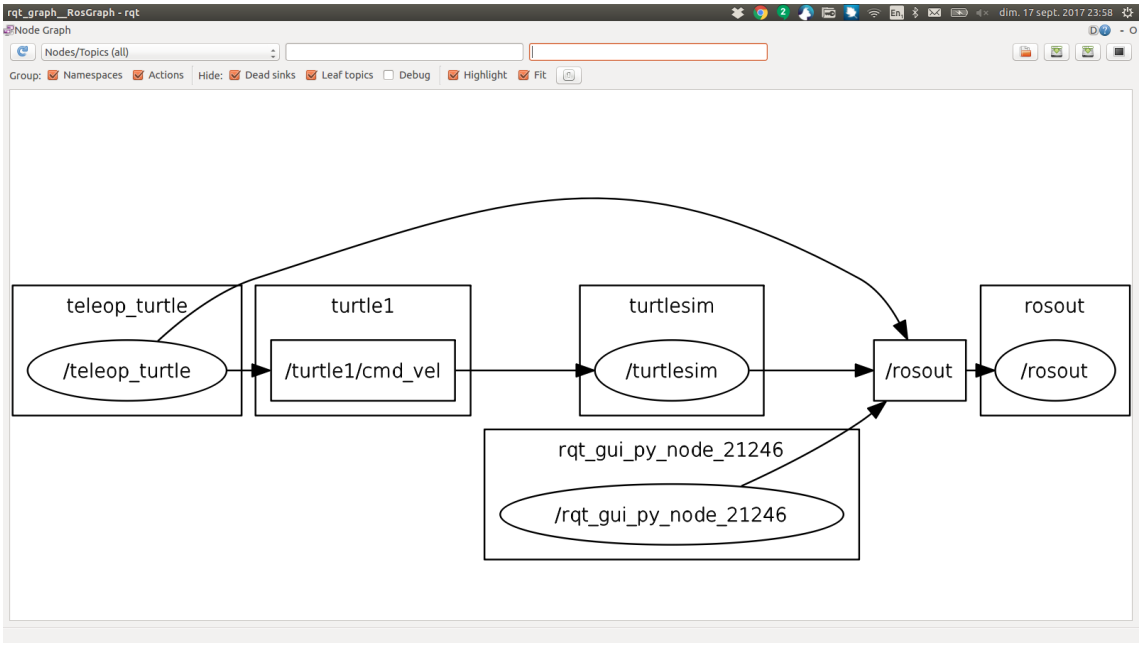


FIGURE 3.2 – Graphe de l'application **turtlesim** et **turtle_teleop_key**

```
rosrun rqt_graph rqt_graph
```

Pour démarrer le graphe de l'affichage des topics :

```
rosrun rqt_plot rqt_plot
```

On obtient le graphe affiché dans la figure Fig.3.3.

3.1.4 Topic

Les topics sont des données publiées par des noeuds et auquelles les noeuds souscrivent. L'exécutable permettant d'avoir des informations sur les topics est **rostopic**.

```
rostopic bw display bandwith used by topic
rostopic echo print messages to screen
rostopic hz display publishing rate of topic
rostopic list print information about active topics
rostopic pub publish data to topic
rostopic type print topic type
```

rostopic list

Cette commande affiche la liste des topics. Le résultat est le suivant :

```
/rosout
/rosout_agg
/statistics
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

rostopic bw

On obtient la bande passante utilisée par un topic en tapant :

```
rostopic bw /turtle1/pose
```

On obtient alors le résultat suivant :

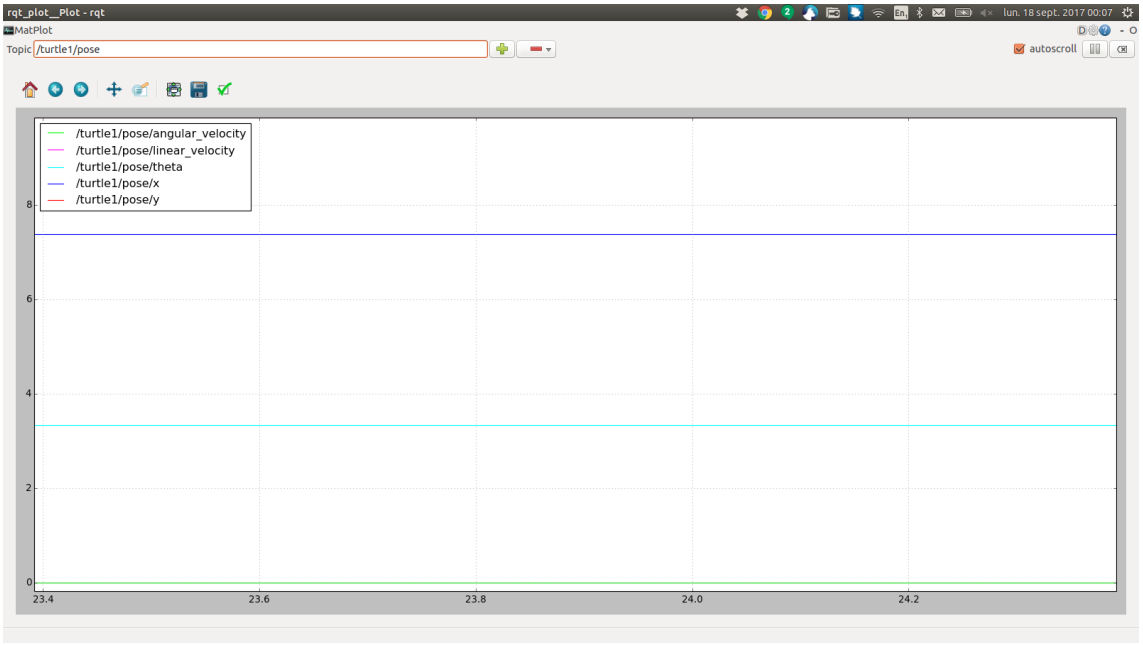


FIGURE 3.3 – Affichage de la position de la tortue dans la fenêtre

```
average: 2.52KB/s
  mean: 0.02KB min: 0.02KB max: 0.02KB window: 100
```

rostopic echo

Affiche le contenu d'un topic en fonction du message transmis. Par exemple :

```
rostopic echo /turtle1/pose
```

donne :

```
x: 5.544444561
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
```

rostopic type

Cette commande affiche le message (structure de données au sens de ROS) d'un topic. Par exemple :

```
rostopic type /turtle1/pose
```

affiche :

```
turtlesim/Pose
```

rostopic pub

Cette commande permet de publier des données sur un topic. Elle suit la structure suivante :

```
rostopic pub /topic type
```

Par exemple si on revient à notre exemple on peut essayer :

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
```

```
  x: 0.0
  y: 0.0
  z: 1.0"
```

Cette commande demande à la tortue de tourner sur elle-même à la vitesse angulaire de 1 rad/s. Grâce à la complétion automatique il est possible de taper uniquement le nom du topic qui étend ensuite le type du topic et la structure de la donnée à transmettre. La durée de la prise en compte de l'information est limitée, on peut utiliser l'option **-r 1** pour répéter l'émission à une fréquence d'1 Hz.

3.2 rqt_console

rqt est une interface d'affichage non 3D qui se peuple avec des plugins. Elle permet de construire une interface de contrôle incrémentalement. L'exécutable permettant d'afficher les messages des noeuds de façon centralisé est **rqt_console**.

```
rosrun rqt_console rqt_console
rosrun rqt_logger_level rqt_logger_level
```

La deuxième commande permet de lancer le système d'affichage des messages d'alertes et d'erreur. Les deux commandes créent les deux fenêtres suivantes affichées dans Fig. 3.4. Pour illustrer cet exemple

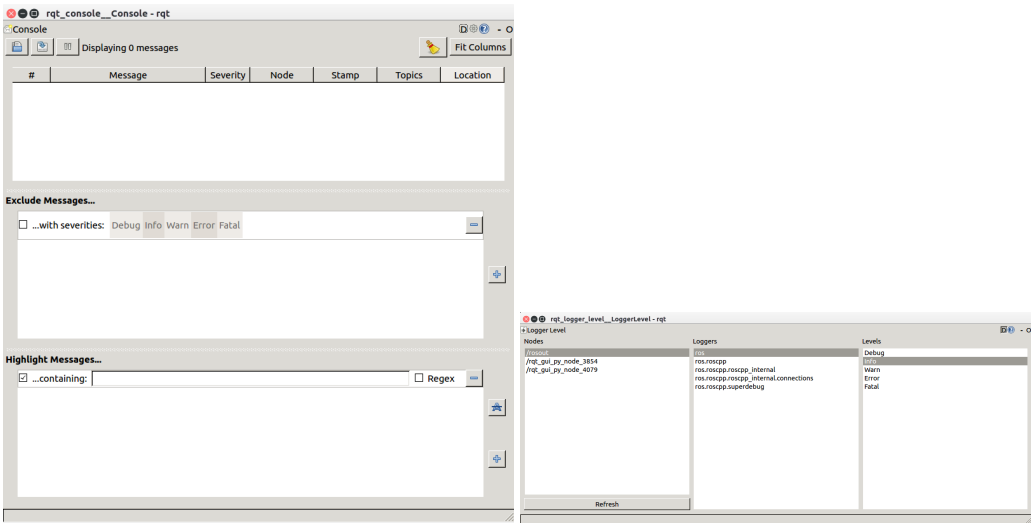


FIGURE 3.4 – Fenêtres correspondant à **rqt_console_console** et à **rqt_logger_level**

on peut lancer le noeud **turtlesim** avec :

```
rosrun turtlesim turtlesim_node
```

La position de la tortue va alors s'afficher dans la console car il s'agit d'un message de niveau INFO. Il est possible de changer le niveau des messages affichés. En le mettant par exemple à WARN, et en envoyant la commande suivant à **turtlesim** :

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
  x: 2000.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2000.0"
```

Elle pousse la tortue dans les coins de l'environnement et on peut voir un message affichant un warning dans la fenêtre du node **rqt_console_console** affichée dans la figure.3.5.

Tutoriel associé : Tutorial Using rqt console et roslaunch `http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch` ;

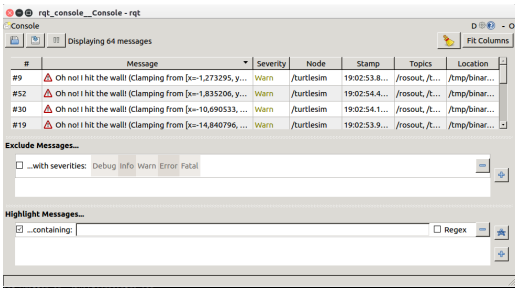


FIGURE 3.5 – Fenêtres correspondant à **rqt_console_console** lorsque la tortue s'écrase dans les murs

3.3 roslaunch

roslaunch est une commande qui permet de lancer plusieurs noeuds. **roslaunch** lit un fichier xml qui contient tous les paramètres pour lancer une application distribuée ROS.

3.3.1 Exemple

Par exemple considérons le fichier launch suivant :

```
<launch>

  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>

</launch>
```

Pour démarrer le graphe d'applications il suffit de lancer les commandes suivantes :

```
roslaunch [package] [filename.launch]
roscd beginner_tutorials
roslaunch beginner_tutorials turtlemimic.launch
```

On obtient alors deux fenêtres **turtlesim**.

Analysons maintenant le contenu du fichier **turtlemimic.launch**. La ligne suivante lance le node **sim** à partir de l'exécutable **turtlesim_node** qui se trouve dans le package **turtlesim** :

```
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
```

Parce que cette ligne se trouve dans le bloc

```
  <group ns="turtlesim1">
  </group>
```

le node se trouve dans le namespace **turtlesim1** et se nommera :

```
/turtlesim1/sim
```

De même le deuxième groupe avec le namespace **turtlesim2** va créer un node qui se nommera :

```
/turtlesim2/sim
```

On peut le vérifier avec **rqt_graph** qui donne la figure Fig.3.6. On trouve également deux topics nommés **/turtlesim1/turtle1/pose** et **/turtlesim2/turtle1/cmd_vel**. Si on exécute :

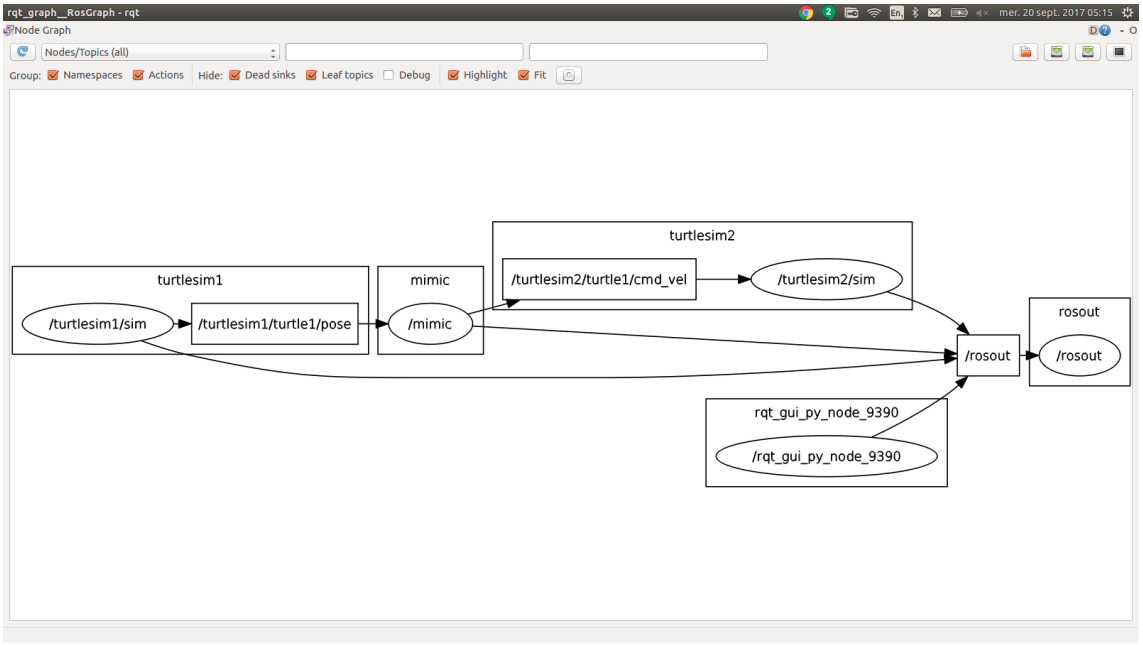


FIGURE 3.6 – Graphe correspondant au fichier turtlemimic.launch

```
rostopic list
```

On constate que l'on trouve deux ensembles de topics correspondants aux deux nodes de **turtlesim** et aux deux namespaces **turtlesim1** et **turtlesim2**.

Le bloc du fichier **turtlemimic.launch** :

```
<node pkg="turtlesim" name="mimic" type="mimic">
  <remap from="input" to="turtlesim1/turtle1"/>
  <remap from="output" to="turtlesim2/turtle1"/>
</node>
```

remplit deux fonctions. La première est de créer le node **mimic** à partir de l'exécutable **mimic** dans le paquet **turtlesim**. La deuxième avec les deux lignes correspondant aux balises **remap** est de changer respectivement les namespaces **input** en **turtlesim1/turtle1**, et **output** en **turtlesim2/turtle1**. Ceci permet de pouvoir connecter deux nodes qui ne publiez pas et ne souscrivez pas sur le même topic.

Tutoriel associé : Tutorial Using rqt console et roslaunch `http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch`

Les fichiers XML launch

roslaunch évalue le fichier XML en une passe. Les tags includes sont traités suivant un ordre en profondeur d'abord. Les tags sont évalués en série et la dernière affectation est celle qui est utilisée.

L'utilisation de la surcharge peut être instable. Il n'y a pas de garantie que la surcharge soit garantie (i.e. si un nom de paramètre change dans un fichier inclus). Il est plutôt recommander de surcharger en utilisant les directives $(arg)/<arg>.

Substitutions

Les attributs des tags peuvent utiliser des arguments de substitutions que roslaunch va résoudre avant de lancer les nodes. Les arguments de substitutions supportés à l'heure actuelle sont :

— $(env ENVIRONMENT_VARIABLE)
  Substitue la valeur d'une variable à partir de l'environnement. Le lancement échoue si la variable d'environnement n'est pas spécifiée. Cette valeur ne peut pas être surchargée par les tags <env>.

— $(optenv ENVIRONMENT_VARIABLE) $(optenv ENVIRONMENT_VARIABLE default_value)
  Substitue la valeur d'une variable d'environnement s'il est spécifiée. Si le champ default_value est

rempli il sera utilisé si la variable d'environnement n'est pas spécifiée. Si le champ default_value n'est pas fourni une chaîne de caractères vide sera utilisée. default_value peut-être constitué de multiples mots séparés par des espaces. Exemples :

```
<param name="foo" value="$(optenv NUM_CPUS 1)" />
<param name="foo" value="$(optenv CONFIG_PATH /home/marvin/ros_workspace)" />
<param name="foo" value="$(optenv VARIABLE ros rocks)" />
```

— $(find pkg)
i.e. $(find rospy)/manifest.xml. Spécifie un chemin relatif par rapport à un paquet. Le chemin dans le système de fichiers menant au paquet est substitué en ligne. L'utilisation de chemin relatifs à un paquet est hautement encouragé car des chemins hard-codés empêche la portabilité du fichier launch. L'utilisation de slashes ou anti-slashes est résolu suivant la convention du système de fichiers.
— $(anon name)
i.e. $(anon rviz-1), génére un identifiant anonyme basé sur name. name lui même est un identifiant unique : des usages multiples de $(anon foo) va créer le même name anonymisé. C'est utilisé pour les attributs name des blocs <node> de façon à créer des nodes avec des noms anonymes, car ROS force les nodes à avoir des noms uniques. Par exemple :

```
<node name="$(anon foo)" pkg="rospy_tutorials" type="talker.py" />
<node name="$(anon foo)" pkg="rospy_tutorials" type="talker.py" />
```

va générer une erreur car il y a deux nodes avec le même nom.
— $(arg foo)
$(arg foo) est évalué à la valeur spécifiée par le tag <arg>. Il doit y avoir un tag correspondant à <arg> dans le même fichier de launch qui déclare arg. Par exemple :

```
<param name="foo" value="$(arg my_foo)" />
```

affecte l'argument my_foo au paramètre foo.
Voici un autre exemple :

```
<node name="add_two_ints_server" pkg="beginner_tutorials" type="add_two_ints_server" />
<node name="add_two_ints_client" pkg="beginner_tutorials" type="add_two_ints_client" args="$(arg a) $(arg b)" />
```

Celui-ci lance le serveur et le client de l'exemple add_two_ints et passe en paramétres les valeurs *a* et *b*. Le fichier peut être alors utilisé de la façon suivante :

```
roslaunch beginner_tutorials launch_file.launch a:=1 b:=5
```

— $(eval <expression>) A partir de Kinetic
$(eval <expression>) permet d'évaluer des expressions en python arbitrairement complexes. Par exemple :

```
<param name="circumference" value="$(eval 2.* 3.1415 * arg('radius'))"/>
```

Note : Les expressions utilisant eval doivent analyser la chaîne de caractéres compléte. Il n'est donc pas possible de mettre d'autres expressions utilisant eval dans la chaîne en argument. L'expression suivante ne fonctionnera donc pas :

```
<param name="foo" value="$(arg foo)$(eval 6*7)bar"/>
```

Pour pallier à cette limitation toutes les commandes de substitution sont disponibles comme des fonctions dans eval. Il est donc possible d'écrire :

```
"$(eval arg('foo') + env('PATH') + 'bar' + find('pkg')"
```

A des fins de simplicité, les arguments peuvent aussi implicitement résolus. Les deux expressions suivantes sont identiques :

```
"$(eval arg('foo'))"
"$(eval foo)"
```

— $(dirname) A partir de Lunar
$(dirname) retourne le chemin absolu vers le répertoire du fichier launch où il est apparu. Ceci peut-être utilisé en conjonction avec eval et if/unless pour modifier des comportements basés sur le chemin d'installation ou simplement comme une façon de faire référence à des fichiers launch ou yaml relatifs au fichier courant plutôt qu'à la racine d'un paquet (comme avec $(find PKG)). Par exemple :

```
<include file="$(dirname)/other.launch" />
```

va chercher le fichier *other.launch* dans le même répertoire que le fichier launch dans lequel il est apparu.

Les arguments de substitution sont résolus sur la machine locale. Ce qui signifie que les variables d'environnement et les chemins des paquets ROS auront leurs valeurs déduites à partir de la machine locales même pour des processus lancés sur des machines distantes.

— if=value (optionel)
Si *value* est à vrai la balise et son contenu sont inclus.

— unless=value (optionel)
A moins que *value* soit à vrai la balise et son contenu sont inclus.

Par exemple :

```
<group if="$(arg foo)">
  <!-- Partie qui ne sera incluse que si foo est vrai -->
</group>

<param name="foo" value="bar" unless="$(arg foo)" /> <!-- Ce param\'etre ne sera pas affect\'e tant que "unless" est vrai -->
```

Les attributs if et unless

Toutes les balises comprennent les attributs *if* et *unless* qui inclus ou exclus une balise sur l'évaluation d'une valeur. "1" et "true" sont considérées comme des valeurs vraies, "0" et "false" sont considérées comme des valeurs fausses. D'autres valeurs sont considérées comme une erreur.

3.4 rosbag

Il est possible d'enregistrer et de rejouer des flux de données transmis par les topics. Cela s'effectue en utilisant la commande **rosbag**. Par exemple la ligne de commande suivante enregistre toutes les données qui passe par les topics à partir du moment où la commande est active (et si des évènements ou données sont génèrés sur les topics).

```
rosbag record -a
```

On peut par exemple lancer le node **turtlesim** et le noeud **teleop_key** pour envoyer des commandes.

Par défaut le nom du fichier **rosbag** est constitué de l'année, du mois et du jour et post fixé par **.bag**. Il est possible ensuite de n'enregistrer que certains topics, et de spécifier un nom de fichier. Par exemple la ligne de commande suivante n'enregistre que les topics **/turtle1/cmd_vel** et **/turtle1/pose** dans le fichier **subset.bag**.

```
rosbag record -O subset /turtle1/cmd_vel /turtle1/pose
```

3.5 rosservice

Les services sont une autre façon pour les noeuds de communiquer entre eux. Les services permettent aux noeuds d'envoyer des *requêtes* et et de recevoir des *réponses*. Dans la suite on supposera que le noeud **turtlesim** continue à fonctionner. Il va servir à illustrer les commandes vues dans ce paragraphe.

La commande **rosservice** permet d'interagir facilement avec le système client/serveur de ROS appelés *services*. **rosservice** a plusieurs commandes qui peuvent être utilisées sur les services comme le montre l'aide en ligne :

```
rosservice args print service arguments
rosservice call call the service with the provided args
rosservice find find services by service type
rosservice info print information about service
```

```
rosservice list list active services
rosservice type print service type
rosservice uri  print service ROSRPC uri
```

3.5.1 rosservice list

La commande

```
rosservice list
```

affiche la liste des services fournis par les nodes de l'application.

```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

On y trouve 9 services fournis par le node **turtlesim** : reset, clear, spawn, kill, turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/get_loggers, et turtlesim/set_logger_level. Il y a deux services relatifs au node **rosout** : /rosout/get_loggers and /rosout/set_logger_level.

3.5.2 rosservice type (service)

Il est possible d'avoir le type de service fourni en utilisant la commande :

```
rosservice type [service]
```

Ce qui donne pour le service clear :

```
rosservice type /clear
```

Le résultat est le suivant :

```
std_srvs/Empty
```

Ce service est donc vide car il n'y a pas d'arguments d'entrée et de sortie (i.e. aucune donnée n'est envoyée lorqu'une requête est effectuée et aucune donnée n'est transmise lors de la réponse).

3.5.3 rosservice call

Il est possible de faire une requête en utilisant la commande :

```
rosservice call [service] [args]
```

Ici aucun argument est nécessaire car le service est vide :

```
rosservice call /clear
```

Comme on peut s'y attendre le fond de la fenêtre du noeud **turtlesim** est effacé. Considérons maintenant un cas où le service a des arguments en examinant le service **spawn** :

```
rosservice type /spawn
```

Le résultat est la structure de la requête et de la réponse.

```
float32 x
float32 y
float32 theta
string name
---
string name
```

La requête est donc constituée de trois réels qui sont respectivement la position et l'orientation de la tortue. Le quatrième argument est le nom du node et est optionnel. Ce service nous permet de lancer une autre tortue, par exemple :

```
rosservice call /spawn 2 2 0.2 ""
```

On trouve maintenant une deuxième tortue dans la fenêtre.

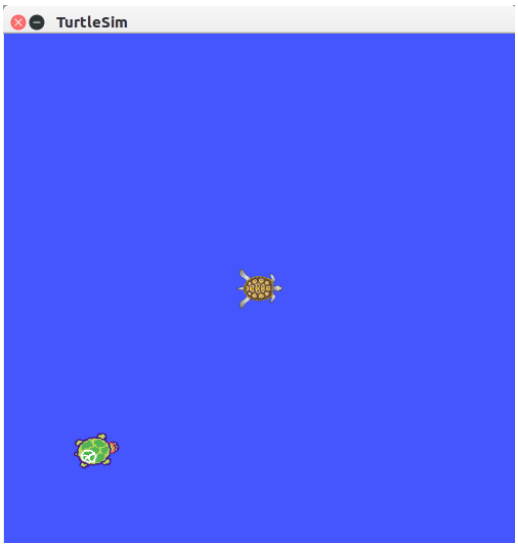


FIGURE 3.7 – Fenêtre résultat de rosrun call /spawn 2 2 0.2 ""

3.6 rosparam

ROS embarque un serveur de paramètres qui permet de stocker des données et de les partager sur toute l'application. Les mécanismes de synchronisation des paramètres est complètement différent des topics qui implémente un système de flux de données. Ils servent notamment à stocker : le modèle du robot (**robot_description**), des trajectoires de mouvements, des gains, et toutes informations pertinentes. Les paramètres peuvent se charger et se sauvegarder grâce au format YAML (Yet Another Language). Ils peuvent également être définis en ligne de commande. Le serveur de paramètre peut stocker des entiers, des réels, des booléens, des dictionnaires et des listes. rosparam utilise le langage YAML pour la syntaxe de ces paramètres. Dans des cas simples, YAML paraît trés naturel.

```
rosparam set  set parameter
rosparam get  get parameter
rosparam load load parameters from file
rosparam dump dump parameters to file
rosparam delete delete parameter
rosparam list list parameter names
```

3.6.1 rosparam list

Il est possible d'avoir la liste des paramètres en tapant la commande :

```
rosparam list
```

On obtient alors la liste suivante :

```
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_kinokoyama__38903
```

```
/rosversion
/run_id
```

Les 3 premiers paramètres permettent de contrôler la couleur du fond de la fenêtre du node **turtlesim**. La commande suivante change la valeur du canal rouge de la couleur de fond :

```
rosparam set /background_r 150
```

Pour que la couleur soit prise en compte il faut appeler le service **clear** de la façon suivante :

```
rosservice call /clear
```

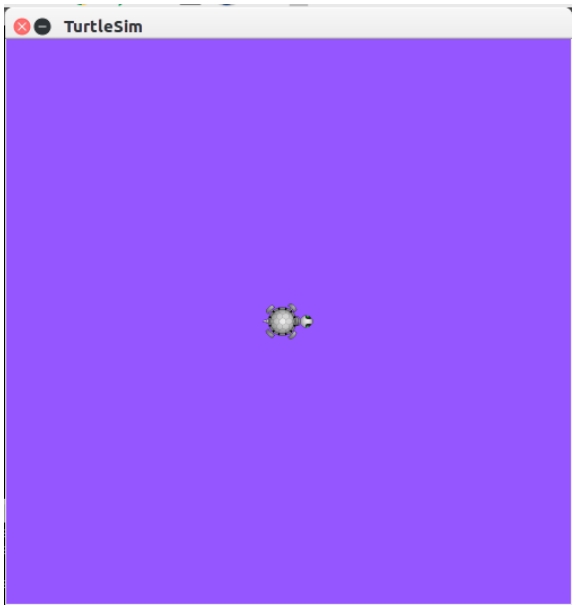


FIGURE 3.8 – Résultat de l'appel **rosparam set /background_r 150**

Le résultat est représenté dans la figure 3.8.

On peut voir la valeur des autres paramètres dans le serveur de paramètres. Pour avoir la valeur du canal vert de la couleur de fond on peut utiliser :

```
rosparam get /background_g
```

On peut aussi utiliser la commande suivante pour voir toutes les valeurs du serveur de paramètres :

```
rosparam get /
```

Le résultat est le suivant :

```
background_b: 255
background_g: 86
background_r: 150
rosdistro: 'kinetic

  '
roslaunch:
  uris: {host_kinokoyama__41913: 'http://kinokoyama:41913/'}
rosversion: '1.12.7

  '
run_id: 5250890c-ab42-11e7-9f88-104a7dbc5f2f
```

Il est possible de stocker ces informations dans un fichier qu'il est ensuite possible de recharger.

3.6.2 rosparam dump et rosparam load

La structure générale de la command est :

```
rosparam dump [file_name] [namespace]
rosparam load [file_name] [namespace]
```

Il est possible d'enregistrer tout les paramètres dans le fichier **params.yaml**

```
rosparam dump params.yaml
```

Il est ensuite possible de charger les fichiers dans de nouveaux espaces de noms par exemple copy :

```
rosparam load params.yaml copy
rosparam get /copy/background_b
255
```

3.7 Création de messages et de services

3.7.1 Introduction

Les messages sont définis par des structures appellées **msg**, tandis que les services le sont par des les structures **srv**.

— **msg** : msg files sont de simples fichiers textes qui donnent les champs d'un message ROS. Ils sont utilisés pour génèrer le code source des messages dans des languages différents.
— **srv** : un fichier srv décrit un service. Il est composé de deux parties : une requête et une réponse.

Les fichiers msg sont stockés dans le répertoire msg d'un paquet, et les fichiers srv sont stockés dans le répertoire srv.

Les fichiers msg sont de simples fichiers textes avec un type de champ et un nom de champ par ligne. Les types de champs possibles sont :

— int8, int16, int32, int64 (plus uint *)
— float32, float64
— string
— time, duration
— other msg files
— des tableaux à taille variables ([])
— des tableaux à taille fixes ([C], avec C la taille du tableau)

Il existe un type spécial dans ROS : **Header**, l'entête contient une trace temporelle (horodatage) et des informations sur les repères de réfèrence qui sont utilisés le plus couramment en ROS. Il est fréquent que la première ligne d'un message soit **Header header**.

Voici un exemple qui utilise un entête, une chaïne de caractères, et deux autres messages :

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

srv files sont comme des fichiers msg, à part qu'ils contiennet deux parties : une requête et une réponse. Les deux parties sont séparées par une ligne '—' Voici un exemple de fichier srv :

```
int64 A
int64 B
---
int64 Sum
```

3.7.2 Création d'un msg

Format du fichier

Créons maintenant un msg dans le paquet **beginnner_tutorials**

```
roscd beginner_tutorials
mkdir msg
echo "int64 num" > msg/Num.msg
```

Le message précèdent ne contient qu'une ligne. Il est possible de faire des fichiers plus compliqués en ajoutant plusieurs élèments (un par ligne) comme ceci :

```
string first_name
string last_name
uint8 age
uint32 score
```

Génération du code source

Afin de générer les codes sources (C++, Python) correspondant au fichier il est nécessaire d'effectuer un certain nombre de modifications sur les fichiers package.xml et CMakeLists.txt.

Dans le fichier package.xml il faut que les deux lignes suivantes sont décommentées :

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Durant la compilation du paquet seule la ligne build_depend (message_generation) est nécessaire, tandis qu'à l'execution du paquet c'est la ligne run_depend (message_runtime).

Il faut ensuite ouvrir le fichier CMakeLists.txt avec votre éditeur favori.

Il faut ajouter la dépendance au paquet message_generation à l'appel de find_package qui existe déjà dans le fichier de façon à pouvoir générer des messages. Il est possible de faire cela en ajoutant message_generation à la liste COMPONENTS de la façon suivante :

```
  # Do not just add this to your CMakeLists.txt, modify the existing text to add message_generation before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation
)
```

Il arrive parfois que le paquet puisse être construit sans mettre toutes les dépendances dans l'appel à find_package. Ceci est du au fait que lorsqu'on combine plusieurs paquets ensemble, si un paquet appelle find_package, celui-ci considéré est configué avec les mêmes valeurs. Mais ne pas faire cet appel empêche la compilation du paquet lorsque celui est isolé.

Il faut également exporté la dépendence à message_runtime.

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

Il faut ensuite indiquer au fichier CMakeLists.txt quel fichier contient le nouveau message que le système doit traiter. Pour cela il faut modifier le bloc suivant :

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

en

```
add_message_files(
  FILES
  Num.msg
)
```

En ajoutant les fichiers msg à la main dans le fichier CMakeLists.txt, cmake sait lorsqu'il doit reconfigurer le projet lorsque d'autres fichiers msg sont ajoutés.

Il faut ensuite effectuer les étapes décries dans le paragraphe 3.7.4.

Utilisation de rosmsg

On peut vérifier que ROS est capable de voir le fichier Num.msg en utilisant la commande rosmsg. L'utilisation de cette commande de façon générale est :

```
rosmsg show [type_message]
```

Par exemple :

```
rosmsg show [type_message]
```

On voit alors :

```
int64 num
```

Dans l'exemple précédent le type de message a deux parties :

— beginner_tutorials : le nom du paquet dans lequel le type est défini.

— Num : Le nom du msg Num

Si vous ne vous souvenez pas du paquet dans lequel le msg est défini, il est possible d'utiliser :

```
[beginner_tutorials/Num]:
int64 num
```

3.7.3 Création d'un srv

Format du fichier

Les fichiers srv se trouvent par convention dans le répertoire srv d'un paquet :

```
roscd beginner_tutorials
mkdir srv
```

Au lieu de créer à la main un fichier srv nous allons utiliser celui d'un paquet déjà existant : Pour cela l'utilisation de roscp est trés pratique pour copier un fichier d'un paquet :

```
roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts.srv
```

Le fichier *AddTwoInts.srv* a été copié du paquet *rospy_tutorials* vers le paquet *beginner_tutorials* et plus précisément dans le répertoire *srv*.

Il reste encore une étape qui comme précédemment consiste à transformer le fichier *srv* en fichier C++, Python et autres languages.

A moins de l'avoir fait dans l'étape précédente, il faut s'assurer que dans le fichier *package.xml* les deux lignes suivant aient été décommentées :

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Comme précédemment le système a besoin du paquet *message_generation* pendant la phase de compilation, et du paquet *message_runtime* pendant la phase d'execution.

De même si cela n'a pas été fait précédemment dans l'étape précédente, il faut ajouter la dépendence au paquet *message_generation* dans le fichier *CMakeLists.txt*.

```
# Do not just add this line to your CMakeLists.txt, modify the existing line
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
  )
```

Malgré son nom *message_generation* fonctionne à la fois pour msg et srv.

Les mêmes changements que pour l'étape précédente doivent être faits pour le fichier *package.xml*.

Il faut ensuite enlever les caractères # pour décommenter les lignes suivantes :

```
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )
```

Il faut ensuite remplacer les fichiers finissant par srv par les fichiers srv présents *AddTwoInts.srv* en l'occurence :

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

Il est maintenant possible de générer les fichiers liés aux srv. On peut aller directement au paragraphe 3.7.4.

3.7.4 Modifications CMakeLists.txt communes à msg et srv

Si le bloc suivant est toujours commenté :

```
# generate_messages(
#   DEPENDENCIES
# #  std_msgs  # Or other packages containing msgs
# )
```

Il faut décommenter les lignes et ajouter tout paquets nécessaires aux fichiers .msg par exemple dans notre cas, il faut ajouter std_msgs qui donne :

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Une fois que les nouveaux messages ou services ont été ajoutés il faut donc :

```
# In your catkin workspace
$ roscd beginner_tutorials
$ cd ../..
$ catkin_make install
$ cd -
```

Chaque fichier msg dans le répertoire msg va être traduit sous forme de fichier source dans tous les languages supportés. Les fichiers d'entête C++ seront générés dans le répertoire /catkin_ws/devel/include/beginner_tutorials. Le script python sera créé dans /catkin_ws/devel/lib/python2.7/dist-packages/beginner_tutorials/msg. The lisp file lui apparaît dans : /catkin_ws/devel/share/common-lisp/ros/beginner_tutorials/msg/.

De la même façon les fichiers .srv dans le répertoire srv seront générés dans les langages supportés. Pour le C++, les fichiers d'entête seront générés dans le même répertoire que pour les fichiers .msg.

4. Ecrire des nodes

Dans ce chapitre nous allons examiner l'écriture en C++ et en python des *nodes* implémentant les concepts vus dans le chapitre précédent. Les *nodes* est le mot dans la communauté ROS pour désigner un executable connecté au middleware ROS. Nous allons commencer par les *topics* puis examiner l'utilisation des *services*.

4.1 Topics

Dans ce paragraphe nous allons examiner l'écriture d'un node qui émet constamment un message et d'un autre node qui reçoit le message.

4.1.1 Emetteur

Dans l'émetteur nous devons faire les choses suivantes :

1. Initialiser le système ROS.
2. Avertir que le node va publier des messages *std_msgs/String* sur le topic "chatter".
3. Boucler sur la publication de messages sur "chatter" 10 fois par seconde.

C++

Il faut tout d'abord créer un répertoire src dans le paquet **beginner_tutorials** :

```
mkdir src
```

Ce répertoire va contenir tous les fichiers sources C++ du paquet.

Il faut ensuite créer le fichier **talker.cpp** et copier le code suivant :

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

/**
 * This tutorial demonstrates simple sending of messages over the ROS system.
 */
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can perform
```

```
13   * any ROS arguments and name remapping that were provided at the command line.
14   * For programmatic remappings you can use a different version of init() which takes
15   * remappings directly, but for most command-line programs, passing argc and argv is
16   * the easiest way to do it.  The third argument to init() is the name of the node.
17   *
18   * You must call one of the versions of ros::init() before using any other
19   * part of the ROS system.
20   */
21  ros::init(argc, argv, "talker");
22
23  /**
24   * NodeHandle is the main access point to communications with the ROS system.
25   * The first NodeHandle constructed will fully initialize this node, and the last
26   * NodeHandle destructed will close down the node.
27   */
28  ros::NodeHandle n;
29
30  /**
31   * The advertise() function is how you tell ROS that you want to
32   * publish on a given topic name. This invokes a call to the ROS
33   * master node, which keeps a registry of who is publishing and who
34   * is subscribing. After this advertise() call is made, the master
35   * node will notify anyone who is trying to subscribe to this topic name,
36   * and they will in turn negotiate a peer-to-peer connection with this
37   * node.  advertise() returns a Publisher object which allows you to
38   * publish messages on that topic through a call to publish().  Once
39   * all copies of the returned Publisher object are destroyed, the topic
40   * will be automatically unadvertised.
41   *
42   * The second parameter to advertise() is the size of the message queue
43   * used for publishing messages.  If messages are published more quickly
44   * than we can send them, the number here specifies how many messages to
45   * buffer up before throwing some away.
46   */
47  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
48
49  ros::Rate loop_rate(10);
50
51  /**
52   * A count of how many messages we have sent. This is used to create
53   * a unique string for each message.
54   */
55  int count = 0;
56  while (ros::ok())
57  {
58    /**
59     * This is a message object. You stuff it with data, and then publish it.
60     */
61    std_msgs::String msg;
62
63    std::stringstream ss;
64    ss << "hello world " << count;
65    msg.data = ss.str();
66
67    ROS_INFO("%s", msg.data.c_str());
68
69    /**
70     * The publish() function is how you send messages. The parameter
71     * is the message object. The type of this object must agree with the type
72     * given as a template parameter to the advertise<>() call, as was done
73     * in the constructor above.
74     */
75    chatter_pub.publish(msg);
76
77    ros::spinOnce();
78
79    loop_rate.sleep();
80    ++count;
81  }
82
```

```
83
84   return 0;
85 }
```

L'entête ros.h

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

permet d'inclurer toutes les en-têtes ROS nécessaires d'une manière efficace et compact.

L'entête "std_msgs/String.h" :

```
2 #include "std_msgs/String.h"
```

permet d'accéder à la déclaration C++ des messages std_msgs/String. Cette entête est générée automatiquement à partir du fichier **std_msgs/String.msg** qui se trouve dans le paquet std_msgs.

```
21   ros::init(argc, argv, "talker");
```

initialise ROS. Ceci permet à ROS de faire des changements de noms à travers la ligne de commande. Cette fonctionnalité n'est pas utilisé ici, mais c'est ici que l'on spécifie le nom du node, et pour cet exemple le nom est *talker*. Il ne faut oublier que les noms sont uniques dans le graphe en cours d'exécution.

Le nom doit être un nom de base sans /.

```
28   ros::NodeHandle n;
```

créer un lien entre ROS et le node. Le premier objet NodeHandle instancié s'occupe de l'initialisation du node. Le dernier détruit s'occupe de nettoyer toutes les ressources que le node a utilisé.

```
47   ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

Cette ligne dit au master (instancié par rosmaster) que le node va publier des messages de type std_msgs/String sur le topic "chatter". Cela permet de dire à tous les noeuds à l'écoute du topic que des données vont être publiées. Le second argument est la taille de la file d'attente au cas où le node publie plus données qu'elles ne peuvent être transmises. Après 1000 images, les données les plus anciennes sont abandonnées. La méthode `NodeHandle::advertise()` renvoit un objet de type `ros::Publisher`. Cet objet a deux fonctions : la première est de fournir une méthode `publish()` pour publier des données, et la deuxième est d'arrêter la publication des données lors que l'objet n'est plus dans le champ d'exécution. Note : Il faut être particulièrement attentif à la portée des variables notamment dans les classes.

```
49   ros::Rate loop_rate(10);
```

Un objet `ros::Rate` permet de spécifier une fréquence à laquelle on souhaite créer une boucle. L'objet chronomètre le temps entre deux appels à `Rate::sleep()` et faire dormir le thread courant le temps nécessaire pour réaliser la boucle (sauf si le retour sur `Rate::sleep()` n'est pas assez rapide).

Dans le cas présent, le thread est réveillé à 10 Hz.

```
55   int count = 0;
56   while (ros::ok())
57   {
```

Par défaut roscpp établit une redirection des signaux d'interruptions comme comme SIGINT (Ctrl-C). Si ce signal est envoyé alors `ros::ok()` retourne `false`.

Plus précisément `ros::ok()` renvoit `false` si :

- un signal SIGINT est reçu (Ctrl-C).
- un autre node avec le même nom a sorti le node du graphe d'application.
- `ros::shutdown()` a été appellé par une autre partie de l'application.
- tous les objets `ros::NodeHandles()` ont été détruits.

Une fois que `ros::ok()` retourne `false` tous les appels à ROS vont échouer.

```
61     std_msgs::String msg;
62
63     std::stringstream ss;
64     ss << "hello world " << count;
65     msg.data = ss.str();
```

Le node émet un message sur ROS en utilisant une classe adaptée aux messages, générée à partir d'un fichier *msg*. Des types de données plus compliqués sont possibles, mais pour l'instant l'exemple utilise un message de type "String" qui a un seul membre "data". Les lignes précédentes ne servent qu'à construire le message.

```
75  chatter_pub.publish(msg);
```

Cette ligne demande l'émission du message à tous les nodes connectés.

```
67  ROS_INFO("%s", msg.data.c_str());
```

Le contenu du message est affiché sur la console ROS du graphe d'application.

```
77    ros::spinOnce();
```

Appelé `ros::spinOnce()` n'est pas nécessaire ici, car aucune information ne nécessitant un traitement n'est attendue. Cependant si une souscription à un topic est ajoutée, l'appel à `ros::spinOnce()` est nécessaire pour les méthodes de callbacks soient appelées. Cet appel est donc rajouté pour éviter les oublis.

```
79    loop_rate.sleep();
```

C'est l'appel qui permet à l'objet `ros::Rate` de suspendre le thread courant afin d'atteindre la fréquence d'émission à 10 Hz.

Python

For the python code, all the files are put inside the **scripts** directory :

```
1  mkdir scripts
2  cd scripts
```

Il est possible de télécharger le script exemple **talker.py** dans le répertoire `scripts` et de le rendre executable :

```
1  wget https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/talker.py
2  chmod +x talker.py
```

```
1  #!/usr/bin/env python
2  # license removed for brevity
3  import rospy
4  from std_msgs.msg import String
5
6  def talker():
7      pub = rospy.Publisher('chatter', String, queue_size=10)
8      rospy.init_node('talker', anonymous=True)
9      rate = rospy.Rate(10) # 10hz
10     while not rospy.is_shutdown():
11         hello_str = "hello world %s" % rospy.get_time()
12         rospy.loginfo(hello_str)
13         pub.publish(hello_str)
14         rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20     pass
```

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

Cette ligne permet d'exécuter le script directement sans avoir à l'appeller par python.

```
3  import rospy
4  from std_msgs.msg import String
```

Importer le module rospy est nécessaire si on écrit un node ROS. L'import du module permet d'utiliser les messages de type String.

```
7 pub = rospy.Publisher('chatter', String, queue_size=10)
8 rospy.init_node('talker', anonymous=True)
```

Cette section définit l'interface talker au reste de ROS. `pub = rospy.Publisher("chatter", String, queue_size=10)` déclare que votre node publie sur le topic `chatter` en utilisant le message `String`. `String` est la classe `std_msgs.msg.String`. La variable `queue_size` limite le nombre de messages dans la file d'attente si le souscripteur ne les traite pas assez vite.

La ligne suivante `rospy.init_node(NAME,...)` est trés importante car elle donne à `rospy` le nom du node. Jusqu'à ce que `rospy` ait cette information le node ne peut pas communiquer avec ROS Master. Dans ce cas, le node va avoir le nom `talker`. NOTE : le nom doit avoir une nom de base, il ne peut pas contenir de slashes "/". `anonymous=True` assure donc que votre node a un nom unique en ajoutant des nombres aléatoires à la fin de `NAME`. Le lien suivant Initialization and Shutdown - Initializing your ROS Node fournit plus d'informations pour l'initialisation des nodes.

```
9 rate = rospy.Rate(10) # 10hz
```

Cette ligne crée un objet `rate` de type `Rate`. Avec la méthode `sleep()`, il est possible de faire des boucles à la fréquence désirée. Avec un argument de 10, il est prévu que la boucle se fasse 10 fois par seconde. Il faut cependant que le temps de traitement soit inférieur à un dixième de seconde.

```
10   while not rospy.is_shutdown():
11     hello_str = "hello world %s" % rospy.get_time()
12     rospy.loginfo(hello_str)
13     pub.publish(hello_str)
14     rate.sleep()
```

Cette boucle représente une utilisation standard de rospy : vérifier que le drapeau `rospy.is_shutdown()` et ensuite faire le travail. On doit vérifier que `is_shutdown()` pour vérifier que le programme doit sortir (par exemple s'il y a Ctrl-C ou autre). Dans ce cas, le travail est d'appeller `pub.publish(hello_str)` qui publie une chaîne de caractères sur le topic `chatter`. La boucle appelle `rate.sleep()` qui s'endort suffisamment longtemps pour maintenir la fréquence de la boucle.

Cette boucle appelle aussi `rospy.loginfo(str)` qui effectue trois tâches : les messages sont impriés sur l'écran, c'est écrit dans le fichier de log de Node, et c'est écrit dans `rosout`. `rosout` est utile pour le déboggage : il est possible d'analyser des messages en utilisant `rqt_console` au lieu de trouver la fenêtre d'affichage du node. `std_msgs.msg.String` est un message très simple, pour les types plus compliqués les arguments des constructeurs sont du même ordre que les fichiers msg. Il est également possible de ne pas passer des arguments et de passer des arguments directement.

```
1 msg = String()
2 msg.data = str
```

ou il est possible d'initialiser certains des champs et laisse le reste avec des valeurs par défaut :

```
1 String(data=str)
```

```
17   try:
18     talker()
19   except rospy.ROSInterruptException:
20     pass
```

En plus de la vérification standard de `__main__`, ce code attrape une exception `rospy.ROSInterruptException` qui peut être lancée par les méthodes `rospy.sleep()` et `rospy.Rate.sleep()` quand Ctrl-C est utilisé ou quand le node est arrêté. La raison pour laquelle cette exception est levée consiste à éviter de continuer à exécuter du code aprés l'appel à la méthode `sleep()`.

4.1.2 Souscripteur

Les étapes du souscripteur peuvent se résumer de la façon suivante :

— Initialiser le système ROS
— Souscrire au topic chatter
— Spin, attendre que les messages arrivent
— Quand le message arrive la fonction `chatterCallback()` est appellée.

C++

Pour le souscripteur il suffit de créer le fichier **./src/listener.cpp** dans le paquet **beginner_tutorials** et copier le code suivant : https://raw.github.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/listener/listener.cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"

/**
 * This tutorial demonstrates simple receipt of messages over the ROS system.
 */
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can perform
   * any ROS arguments and name remapping that were provided at the command line.
   * For programmatic remappings you can use a different version of init() which takes
   * remappings directly, but for most command-line programs, passing argc and argv is
   * the easiest way to do it.  The third argument to init() is the name of the node.
   *
   * You must call one of the versions of ros::init() before using any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "listener");

  /**
   * NodeHandle is the main access point to communications with the ROS system.
   * The first NodeHandle constructed will fully initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The subscribe() call is how you tell ROS that you want to receive messages
   * on a given topic.  This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing and who
   * is subscribing.  Messages are passed to a callback function, here
   * called chatterCallback.  subscribe() returns a Subscriber object that you
   * must hold on to until you want to unsubscribe.  When all copies of the Subscriber
   * object go out of scope, this callback will automatically be unsubscribed from
   * this topic.
   *
   * The second parameter to the subscribe() function is the size of the message
   * queue.  If messages are arriving faster than they are being processed, this
   * is the number of messages that will be buffered up before beginning to throw
   * away the oldest ones.
   */
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  /**
   * ros::spin() will enter a loop, pumping callbacks.  With this version, all
   * callbacks will be called from within this thread (the main one).  ros::spin()
   * will exit when Ctrl-C is pressed, or the node is shutdown by the master.
   */
  ros::spin();

  return 0;
}
```

En considérant uniquement les parties différentes avec l'émetteur, on trouve les lignes suivantes :

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

C'est la fonction qui sera appellée quand un nouveau message arrive sur le topic *chatter*. Le message est passé dans un `boost::shared_ptr`, ce qui signifie qu'il est possible de stocker le message sans se préoccuper de l'effacer et sans copier les données.

```
48 ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

Cet appel souscrit au topic *chatter* auprès du master. ROS appelle la fonction `chatterCallback()` à chaque fois qu'un nouveau message arrive. Le second argument est la taille de la file d'attente. Si les messages ne sont pas traités suffisamment rapidement, le système stocke jusqu'à 1000 messages puis commence à oublier les messages les plus anciens.

`NodeHandle::subscribe()` retourne un objet `ros::Subscriber` qui doit être conservé tout le temps de la souscription. Quand l'objet `ros::Subscriber` est détruit le node est automatiquement désabonné du topic *chatter*.

Il existe des versions de `NodeHandle::subscribe()` qui permette de spécifier une méthode d'une classe, ou même tout object appellable par un objet `Boost.Function`.

```
55 ros::spin();
```

`ros::spin()` entre dans une boucle et appelle les callbacks méthodes aussi vite que possible. Cette méthode est implémenté de façon à prendre peu de temps CPU. `ros::spin()` sort une fois que `ros::ok()` retourne `false` ce qui signifie que `ros::shutdown()` a été appelé soit par le gestionnaire de Ctrl-C, le master signifiant le shutdown, ou par un appel manuel.

Python

Il faut télécharger le fichier `listener.py` dans votre répertoire :

```
1  #!/usr/bin/env python
2  import rospy
3  from std_msgs.msg import String
4
5  def callback(data):
6      rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
7
8  def listener():
9
10     # In ROS, nodes are uniquely named. If two nodes with the same
11     # node are launched, the previous one is kicked off. The
12     # anonymous=True flag means that rospy will choose a unique
13     # name for our 'listener' node so that multiple listeners can
14     # run simultaneously.
15     rospy.init_node('listener', anonymous=True)
16
17     rospy.Subscriber("chatter", String, callback)
18
19     # spin() simply keeps python from exiting until this node is stopped
20     rospy.spin()
21
22 if __name__ == '__main__':
23     listener()
```

Il faut que le script soit exécutable :

```
1 chmod +x listener.py
```

Le code du script `listerner.py` est similaire à `talker.py`. On y a ajouté une méthode de callback pour souscrire aux messages.

```
15 rospy.init_node('listener', anonymous=True)
16
17 rospy.Subscriber("chatter", String, callback)
18
19 # spin() simply keeps python from exiting until this node is stopped
20 rospy.spin()
```

Ceci déclare que le node souscrit au topic `chatter` qui est de type `std_msgs.msg.String`. Quand de nouveaux messages arrivent, `callback` est appellé avec le message comme premier argument.

L'appel à `init_node` inclus `anonymous=True`. Cet argument génére un nombre aléatoire et l'ajoute au nom du noeud si celui existe déjà dans le graphe ROS. On peut ainsi lancer plusieurs fois le script `listener.py`.

`rospy.spin()` empêche simplement le node de sortir tant qu'on ne lui demande pas de s' arrêter. Contrairement à roscpp, `rospy.spin()` n'affecte pas les fonctions callbacks des souscripteurs car ils ont leurs propres threads.

Nous utilisons CMake comme notre système de construction, et il doit être utilisé même pour les nodes Pythons. En effet il est nécessaire de générer le code Python pour les messages et les services. Pour cela il faut aller dans le workspace catkin et lancer la commande catkin_make :

```
cd ~/catkin_ws
catkin_make
```

4.1.3 Lancer les nodes

Il suffit ensuite de lancer le listener dans un terminal :

```
rosrun beginner_tutorials listener.py
```

et le talker dans un autre terminal :

```
rosrun beginner_tutorials talker.py
```

4.2 Services

Dans cette partie nous allons écrire un serveur et un client pour le service **AddTwoInts.srv**. Le serveur va recevoir deux entiers et retourner la somme. Avant de commencer il faut s'assurer que le fichier **AddTwoInts.srv** existe bien dans le répertoire srv.

4.2.1 Serveur

C++

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
```

La première ligne correspond à la déclaration de l'entête ROS. La deuxième ligne correspond à la déclaration du service *AddTwoInts*, et l'en-tête générée lors de l'appel à **cmake_install**.

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
```

```
bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
```

La fonction fournit le service pour ajouter deux entiers, il utilise les types *request* et *response* défini le fichier srv and retourne un booléen.

```
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}
```

Ici les deux *ints* sont additionnés et stockés dans la réponse. Enfin certaines informations à propos de la requête et de la réponse sont loggées. Finalement le service retourne *true* quand la fonction est finie.

```
ros::ServiceServer service = n.advertiseService("add_two_ints", add);
```

Ici le service est crée et publié sur ROS.

Python

Pour ce client il faut créer le fichier **scripts/add_two_ints_server.py** dans le paquet **beginner_tutorials** en copiant le code suivant :

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

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

La partie spécifique à ROS pour la déclaration de service est très petite elle est limitée à la ligne :

```
s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
```

4.2.2 Client

C++

Pour le client, il faut créer le fichier *src/add_two_ints_client.cpp* et copier la suite dedans :

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
```

```
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

Le client s'architecture de la façon suivante :

```
ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
```

Cet appel crée un client pour le service *add_two_ints*. L'objet *client* de type `ros::ServiceClient` retourné est utilisé plus tard pour appeller le service.

```
beginner_tutorials::AddTwoInts srv;
srv.request.a = atoll(argv[1]);
srv.request.b = atoll(argv[2]);
```

Ici l'objet *srv* instancie la classe de service et on assigne des valeurs aux membres du champ *request*. Une classe service contient deux membres, `request` et `response` qui correspondent aux classes `Request` and `Response`.

```
if (client.call(srv))
```

Cette ligne appelle effectivement le service. Comme les appels aux services sont bloquants, il va retourner une fois que l'appel est fait. Si l'appel service fonctionne, `call()` retourne `true` et la valeur `srv.response` sera valide. Si l'appel n'a pas fonctionné `call()` retourne `false` et la valeur dans `srv.response` ne sera pas valide.

Construire les nodes en C++

Pour construire les nodes il faut modifier le fichier **CMakeLists.txt** qui se trouve dans le paquet **beginner_tutorials** :

```
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(add_two_ints_server beginner_tutorials_gencpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(add_two_ints_client beginner_tutorials_gencpp)
```

Ces lignes vont créer deux exécutables **add_two_ints_server** et **add_two_ints_client** qui vont aller par défaut dans le répertoire du paquet dans l'espace devel, localisé par défaut dans `/catkin_ws/devel/lib/<package_name>`. Il est possible de lancer les exécutables directement ou par rosrun. Ils ne sont pas placés dans '<prefix>/bin' car cela polluerait le chemin PATH lorsque l'on installe des paquets dans le système. Si l'on souhaite avoir l'exécutable dans le chemin à l'installation, il est possible de faire une cible d'installation.

Il faut maintenant lancer **catkin_make** :

```
cd ~/catkin_ws
catkin_make
```

Python

Pour ce client il faut créer le fichier **scripts/add_two_ints_client.py** dans le paquet **beginner_tutorials** en copiant le code suivant :

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

import sys
import rospy
```

```
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Il ne faut pas oublier de mettre les droits suivants pour faire du script un exécutable :

```
chmod a+x scripts/add_two_ints_client.py
```

Le code pour écrire un client de service est relativement simple. Pour les clients il n'est pas obligatoire d'appeller `init_node()`. Il faut d'abord attendre le service :

```
rospy.wait_for_service('add_two_ints')
```

Cette méthode bloque l'exécution jusqu'à ce que le service *add_two_ints* soit disponible. Ensuite on crée un objet pour pouvoir y accéder.

```
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```

On peut ensuite utiliser cet objet comme une fonction normale et l'utiliser ainsi :

```
resp1 = add_two_ints(x, y)
return resp1.sum
```

Parce que nous avons déclaré le service comme étant `AddTwoInts`, lorsque **catkin_make** est appelé deux classes sont créées : `AddTwoIntsRequest` et `AddTwoIntsResponse`. La valeur retournée est effectuée dans un objet `AddTwoIntsResponse`. Si l'appel échoue une exception de type `rospy.ServiceException` est lancée, et il faut donc mettre en place les blocs `try/except` correspondants.

Construire les nodes en python

Même pour les nodes python il est nécessaire de passer par le système de build. Cela permet de s'assurer que les codes pythons correspondants aux messages et services sont générés. Il faut donc aller dans son espace catkin et lancer **catkin_make** :

```
cd ~/catkin_ws
catkin_make
```

4.2.3 Lancer les nodes

Il suffit ensuite de lancer le client dans un terminal :

```
rosrun beginner_tutorials add_two_ints_client.py
```

et le server dans un autre terminal :

```
rosrun beginner_tutorials add_two_ints_server.py
```

II Travaux Pratiques

5. TurtleBot2

Ce chapitre est une version résumée des tutoriaux indiqués à chaque paragraphe.

5.1 Démarrage

Les robots de l'AIP sont accessibles via le Wifi. Le but est d'éviter d'être physiquement sur le robot et d'utiliser le réseau. Le nom des ordinateurs sur le réseau sont **turtlebot1**,**turtlebot2**, **turtlebot3**, **turtlebot4**.

Il faut donc vérifier que votre variable d'environnement ROS_MASTER_URI possède la bonne valeur. Si vous souhaitez vous connecter au robot **turtlebot2**, la variable d'environnement doit être initialisée de la façon suivante :

```
export ROS_MASTER_URI=http://turtlebot2:11311/
```

Pour avoir cette valeur à chaque fois, il faut inclure cette ligne dans votre fichier **.bashrc**.

5.1.1 Logiciel minimum

Pour accéder au robot **turtlebot2** il suffit de se connecter avec le login *turtlebot* en utilisant la commande **ssh** :

```
ssh turtlebot@turtlebot2
```

Pour lancer l'accés aux drivers permettant d'envoyer des commandes aux moteurs et lire les capteurs il faut lancer la commande suivante *sur le robot* :

```
roslaunch turtlebot_bringup minimal.launch --screen
```

Le robot fait un bruit qui monte dans les aigus pour indiquer le fichier minimum a été lancé. Il ne faut pas sortir de cette commande *tant que vous n'avez pas fini votre expérimentation*.

Note importante : il ne faut pas lancer 2 fois minimal.launch. Pour détecter qu'il a déjà été lancé faites :

```
ps aux | grep minimal
```

Si vous voyez une ligne identique à cela :

```
turtleb+  3762  0.3  0.3 300020 24840 pts/21   Sl+  15:23   0:02 /usr/bin/python /opt/ros/indigo/bin/roslaunch turtlebot_bringup
      minimal.launch
```

Pour visualiser le robot et son modèle 3D vous lancez la commande suivante *sur le PC* :

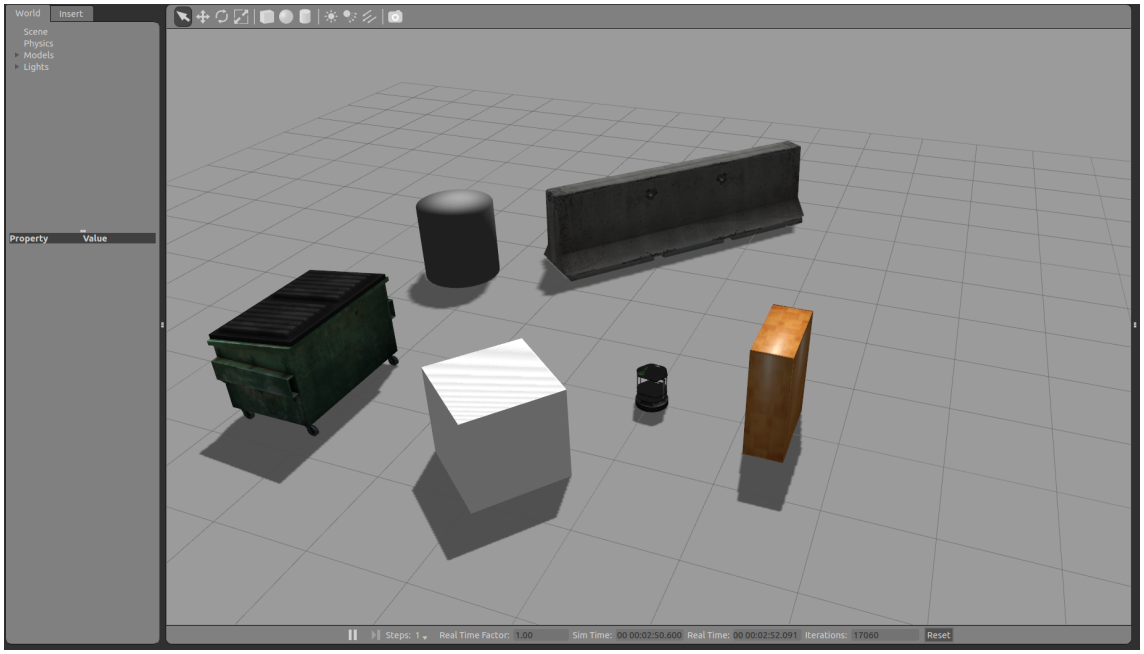


FIGURE 5.1 – Le robot turtlebot simulé avec Gazebo

```
roslaunch turtlebot_rviz_launchers view_robot.launch --screen
```

Vous obtenez alors une fenêtre qui ressemble à l'image dans la figure Fig. 5.1.

5.1.2 Vision 3d

Lorsque le logiciel minimum a été lancé il faut lancer également le driver pour pouvoir accéder à la caméra RGB-D du TurtleBot2 *sur le robot* :

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

Il faut donc effectuer cette commande dans un autre terminal.

Notes : Il ne faut pas lancer cette commande lorsque vous utilisez la cartographie et la navigation. En effet le fichier 3dsensor.launch est lancé par les fichiers de launch associés.

5.1.3 Gazebo

Pour simuler le robot sous Gazebo comme représenté dans la figure Fig.5.1, il faut lancer :

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

Dans la suite du document le lancement minimal du Turtlebot2 sous Gazebo fera référence à cette commande.

Notes :

— Si Gazebo est lancé en premier, il est possible que sa base de données des modèles se mette à jour. Cette opération peut prendre quelques minutes.

— Assurez vous que vos variables d'environnement soient bien à jour et notamment que le fichier **setup.bash** soit bien appellé dans le fichier **.bashrc** :

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

5.1.4 Téléopérer le turtlebot

Dans un deuxième terminal vous pouvez téléopérer le robot en tapant sur le PC (et non sur le turtlebot) :

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Dans le terminal vous devez voir apparaitre les lignes suivantes :

```
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
```

Les touches *i* et , permettent d'aller en avant et en arrière. Les touches *j* et *l* permettent de tourner à droite et à gauche.

Le tutorial associé est disponible ici.

5.1.5 Problèmes liés aux noms des nodes

Lorsque cette expérience est menée avec plusieurs robots, il est possible que rviz ou la téléopèration soit stoppée de façon inopinée. Un autre binôme peut avoir lancée la même commande en spécifiant votre robot.

Par exemple pour la téléopèration, le nom du noeud par défaut est <b>/turtlebot_teleop_keyboard<b>. Le nom du noeud est spécifié dans le fichier launch

5.2 Construction de cartes et navigation

5.2.1 Construction de cartes

Dans cette partie, le robot utilise une ligne de l'image de profondeur fournie par la kinect pour construire une carte de l'environnement. Pour lancer la construction de la carte il faut s'assurer que le logiciel minimal ait été lancé sur le robot (voir paragraphe 5.1.1). Et NE PAS LANCER le 3dsensor.launch ! Le tutorial associé est disponible ici.

1. Sur le robot il faut lancer le fichier de launch **gmapping_demo** sur le portable du turtlebot :

```
# From turtlebot laptop
roslaunch turtlebot_navigation gmapping_demo.launch
```

2. Sur le PC il faut lancer le fichier de launch **view_navigation** pour démarrer rviz et ses sorties capteurs :

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

3. Il faut alors explorer l'environnement pour construire la carte avec les nodes de téléopération (voir paragraphe 5.1.4).
4. Une fois que la carte est suffisante, il faut la sauver en utilisant la commande suivante dans un nouveau terminal :

```
rosrun map_server map_saver -f /tmp/my_map
```

Note : Ne coupez aucun serveur pour ne pas perdre la carte !

5.2.2 Navigation

Dans cette partie, le robot utilise une carte de l'environnement. Pour lancer la navigation dans la carte il faut s'assurer que le logiciel minimal ait été lancé sur le robot (voir paragraphe 5.1.1). Le tutorial associé est disponible ici.

1. Sur le robot il faut lancer le fichier de launch en spécifiant la carte générée :

```
export TURTLEBOT_MAP_FILE=/tmp/my_map.yaml
roslaunch turtlebot_navigation amcl_demo.launch
```

2. Sur le PC, on peut lancer rviz avec la commande suivante :

```
roslaunch turtlebot_rviz_launchers view_navigation.launch --screen
```

5.2.3 RVIZ

Localisation du turtlebot

Lorsqu'il démarre le turtlebot ne sait pas où il est. Pour lui fournir sa localisation approximative sur la carte :

— Cliquer sur le bouton "2D Pose Estimate"

— Cliquer sur la carte où le TurtleBot se trouve approximativement et pointer la direction du TurtleBot.

Vous devriez voir une collection de flèches qui sont des hypothèses de positions du TurtleBot. Le scan du laser doit s'aligner approximativement avec les murs de la carte. Si la procédure ne fonctionne pas, il est possible de répéter la procédure.

Teleoperation

Les opérations de téléopérations peuvent fonctionner en parallèle de la navigation. Elles sont toutefois prioritaires sur les comportements de navigation autonomes si une commande est envoyée. Souvent, il est utile de téléopérer le robot lorsque la localisation a été fournie au robot de façon à ce que l'algoritme converge vers une bonne estimation. En effet de nouvelles caractéristiques de l'environnement permettent de mieux discriminer les hypothèses.

Lancer un but de navigation

Une fois le robot localisé, il lui est possible de planifier un chemin dans l'environnement. Pour lui envoyer un but :

— Cliquer sur le boutton "2D Nav Goal".

— Cliquer sur la carte où l'on souhaite voir le Turtlebot et pointer la direction que celui-ci doit avoir à la fin.

La planification peut ne pas fonctionner s'il y a des obstacles où si le but est bloqué.

Construction de cartes avec Gazebo

Après avoir lancé la simulation comme expliquer au paragraphe 5.1.3, il est possible de lancer :

```
roslaunch turtlebot_gazebo gmapping_demo.launch
```

et on peut lancer le rviz pour visualiser le processus de construction de la carte :

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

La commande pour sauvegarder la carte est la même que pour le robot réel :

```
rosrun map_server map_saver -f <your map name>
```

Navigation avec Gazebo

Pour s'assurer que tout fonctionne correctement il suffit de refaire les mêmes étapes que précédemment mais sans lancer la construction de la carte et de faire à la place :

```
roslaunch turtlebot_gazebo amcl_demo.launch map_file:=<full path to your map YAML file>
```

6. Asservissement visuel

Dans ce chapitre on s'intéresse à la génération de mouvements pour deux robots : le robot Turtlebot2 et le robot Baxter illustré dans la figure Fig.6. Le but est générer un mouvement de façon à ce que l'objet soit centré dans l'image de la caméra du TurtleBot2 ou de la caméra d'un des bras du robot Baxter. Pour cela on commence par une courte introduction aux fonctions de tâches basée sur [1].

6.1 Introduction aux tâches

Une tâche du point de vue du contrôle permet de générer un loi de commande pour bouger un robot suivant un critère donné. La tâche consiste donc à réguler une erreur définie par la fonction suivante :

$$\mathbf{e}(t) = \mathbf{s}(t) - \mathbf{s}^*(t) \tag{6.1}$$

$\mathbf{s}(t)$ correspond à une quantité mesurable à partir des capteurs du robot. On notera en général $\mathbf{s}(t)$ son modèle en fonction de paramètres de l'environnement et d'un modèle, et $\hat{\mathbf{s}}(t)$ sa mesure.

Par exemple considérons le robot Baxter représenté dans la figure Fig.6, en utilisant les encodeurs du robot $\hat{\mathbf{q}} \in \mathbb{R}^{12}$ on peut calculer avec la position de son préhenseur avec le modèle géométrique. On obtient

FIGURE 6.1 – Le robot Baxter en rendu 3D utilisé dans le cadre de travaux pratiques. Le robot a trois caméra une au niveau de la tête, et deux dans chaque poignet.

alors :

$$\mathbf{s}_{pre}(t) : \mathbb{R}^{12} \rightarrow SE(3), \mathbf{q} \mapsto \begin{pmatrix} \mathbf{R}_{pre} & \mathbf{T}_{pre} \\ \mathbf{0}^{\top} & 1 \end{pmatrix} \tag{6.2}$$

avec $\mathbf{R}_{pre}$ l'orientation du préhenseur sous forme de matrice de rotation, et $\mathbf{T}_{pre}$ la position du préhenseur. Il peut s'agir de la position de son centre de masse :

$$\mathbf{s}_{com}(t) : \mathbb{R}^{12} \rightarrow \mathbb{R}^{3}, \mathbf{q} \mapsto \begin{pmatrix} x_{com} \\ y_{com} \\ z_{com} \end{pmatrix} \tag{6.3}$$

ou d'un point dans l'image de la caméra du robot.

6.1.1 Contrôleur en vitesse

Considérons le robot baxter regardant un objet de couleur unie avec sa caméra. En utilisant OpenCV il est possible de détecter sa couleur dans une image, et de calculer le centre de gravité (u_{cog}, v_{cog}) issu de la détection de la couleur. On peut relier la variation de la position de ce point dans l'image (en supposant que l'objet ne bouge pas) et la variation de la position de la caméra. Supposons la vitesse spatial de la caméra soit notée : $\mathbf{V}_c = \frac{\delta \mathbf{o}_c}{\delta t} = (\mathbf{v}_c, \mathbf{w}_c)$ alors on a :

$$\dot{\mathbf{s}} = \frac{\delta \mathbf{s}}{\delta \mathbf{o}_c} \frac{\delta \mathbf{o}_c}{\delta t} = \mathbf{L}_s \mathbf{V}_c \tag{6.4}$$

avec $\mathbf{L}_s \in \mathbb{R}^{3\times 6}$, appellée la *matrice d'interaction* liée à $\mathbf{s}$. En utilisant l'équation Eq.6.1, et en supposant que $\mathbf{s}^*(t)$ est constant on a donc :

$$\dot{\mathbf{e}} = \frac{\delta}{\delta t}(\mathbf{s}(t) - \mathbf{s}^*(t)) = \dot{\mathbf{s}} = \mathbf{L}_s \mathbf{V}_c \tag{6.5}$$

Supposons maintenant que nous souhaitions que l'erreur soit régulée suivant une descente exponentielle alors

$$\dot{\mathbf{e}} = -\lambda \mathbf{e} \tag{6.6}$$

En utilisant l'équation Eq.6.5 on a donc :

$$\mathbf{V}_c = -\lambda \mathbf{L}_s^{+} \mathbf{e} \tag{6.7}$$

Notons que ce problème est équivalent au problème d'optimisation suivant :

$$\begin{cases} \min_{\mathbf{V}_c} & \| \mathbf{V}_c \| \\ \text{tel que} & \mathbf{e} = -\lambda \mathbf{L}_s \mathbf{V}_c \end{cases} \tag{6.8}$$

En l'occurence, $\mathbf{e} = \hat{\mathbf{s}}(t) - \mathbf{s}^*$ où $\hat{\mathbf{s}}(t)$ est la *mesure* de la caractéristique dans le plan image. Il nous faut maintenant calculer $\mathbf{L}_s$ et générer $\mathbf{V}_c$.

6.1.2 Projection d'un point dans une image

La figure Fig. 6.1.2 illustre la projection du point $\mathbf{M} = (X_w Y_w Z_w)^{\top} = (X_c Y_c Z_c)^{\top}$ dans le plan image I au point $m = (Z_c x, Z_c y, Z_c)$. Ils sont reliés par la relation suivante :

$$Z_c \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} f\alpha & 0 & c_x & 0 \\ 0 & f & c_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} \tag{6.9}$$

avec f la distance du plan image I au centre de la caméra aussi appellée la *focale*, α le rapport entre le taille des pixels dans la direction horizontale et la direction verticale, et (c_x, c_y) le centre de l'image dans les coordonnées de la caméra.

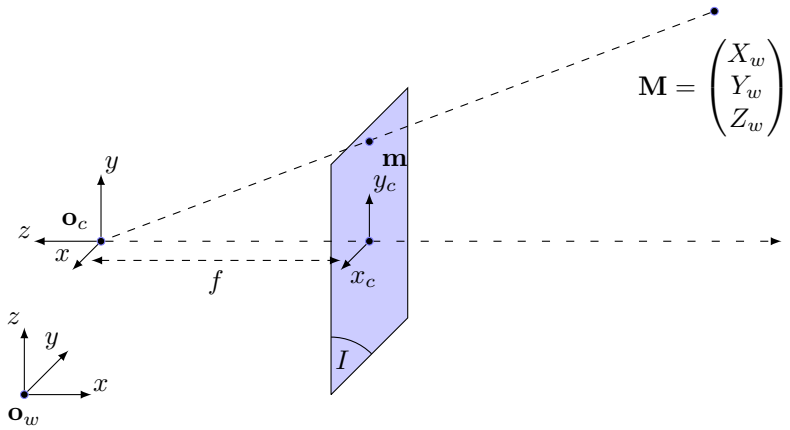


FIGURE 6.2 – o_c est l'origine du repère de la caméra, o_w l'origine du repère monde. La projection du point $\mathbf{M}$ dans le plan image I donne $\mathbf{m}$.

Si on remarque que :

$$\begin{pmatrix} f\alpha & 0 & c_x & 0 \\ 0 & f & c_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} f\alpha & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{6.10}$$

On peut poser :

$$K_f = \begin{pmatrix} f\alpha & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix}, \prod = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{6.11}$$

avec K_f une représentation canonique de la matrice des paramètres intrinséques de la caméra, et $\prod$ la matrice de projection canonique. En notant que :

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_c & \mathbf{T}_c \\ \mathbf{0}^\top & 1 \end{pmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} = \mathbf{M}_c\mathbf{M} \tag{6.12}$$

avec $\mathbf{R}_c$ l'orientation de la caméra dans le monde et $\mathbf{T}_c$ la position de la caméra dans le monde. Nous avons donc :

$$Z_c \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} f\alpha & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{R}_c & \mathbf{T}_c \\ \mathbf{0}^\top & 1 \end{pmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} = \mathbf{K}_f \prod \mathbf{M}_c\mathbf{M} \tag{6.13}$$

Finalement le modèle de la caractéristique visuelle mesurée est :

$$\mathbf{s} = \mathbf{m} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{Z_c} \mathbf{K}_f \prod \mathbf{M}_c\mathbf{M} \tag{6.14}$$

Notons que la caméra ne mesure pas $\mathbf{m}$ mais $\frac{\mathbf{m}}{Z_c}$.

6.1.3 Calcul de la matrice d'interaction

On cherche maintenant à relier la variation de la caractéristique visuelle avec celle de la caméra. En supposant que la focale de la caméra $f = 1$, que $\alpha = 1$ et que le centre de la caméra est à zéro $(c_x, c_y) = (0,0)$

alors on peut écrire :

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{Z_c} \begin{pmatrix} f\alpha & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} = (\frac{X_c}{Z_c} \ \frac{Y_c}{Z_c} \ 1)^\top \tag{6.15}$$

Donc :

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} \dot{X}_c/Z + X_c\dot{Z}_c/Z_c^2 \\ \dot{Y}_c/Z + Y_c\dot{Z}_c/Z_c^2 \end{pmatrix} = \begin{pmatrix} (\dot{X}_c + x\dot{Z}_c)/Z_c \\ (\dot{Y}_c + y\dot{Z}_c)/Z_c \end{pmatrix} \tag{6.16}$$

On peut relier la vitesse du point $\mathbf{M}$ avec la vitesse spatiale de la caméra $\mathbf{V}_c = (\mathbf{v}_c, \omega_c) = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$ par :

$$\dot{\mathbf{M}} = -\mathbf{v}_c - \omega_c \times M \Leftrightarrow \begin{cases} \dot{X}_c = -v_x - \omega_y Z_c + \omega_z Y_c \\ \dot{Y}_c = -v_y - \omega_z X_c + \omega_x Z_c \\ \dot{Z}_c = -v_z - \omega_x Y_c + \omega_y X_c \end{cases} \tag{6.17}$$

Cette relation peut-être écrite de la façon suivante :

$$\begin{cases} \dot{x} = -v_x/Z_c + xv_z/Z_c + xy\omega_x - (1+x^2)\omega_y + y\omega_z \\ \dot{y} = -v_y/Z_c + yv_z/Z_c + (1+y^2)\omega_x - xy\omega_y - x\omega_z \end{cases} \tag{6.18}$$

Maintenant la matrice la relation peut s'écrire :

$$\dot{\mathbf{s}} = \mathbf{L}_\mathbf{s}\mathbf{V}_c \tag{6.19}$$

avec

$$\mathbf{L}_\mathbf{s} = \begin{pmatrix} -1/Z_c & 0 & x/Z_c & xy & -(1+x^2) & y \\ 0 & -1/Z_c & y/Z_c & 1+y^2 & -xy & -x \end{pmatrix} \tag{6.20}$$

On peut voir que dans la formulation de la matrice d'interaction se trouve la profondeur du point M par rapport au repère de la caméra. Par conséquent tout schéma de contrôle qui utilise cette matrice d'interaction doit utiliser une estimation de Z_c. Enfin les hypothèses concernant les valeurs du centre de la caméra et de la focale ne sont généralement pas vérifiées et il est nécessaire de prendre en compte les valeurs de la matrice des paramètres intrinséques.

6.1.4 Génération de la commande

En utilisant une paramétrisation des matrices homogènes (par exemple Davit-Hartenberg) correspondant à la position des corps du robot on peut écrire pour le robot Baxter la fonction reliant la position des moteurs du robot à la position de la caméra :

$$\mathbf{s}_{cam}(\mathbf{q}) = \begin{pmatrix} \mathbf{R}_{cam} & \mathbf{T}_{cam} \\ \mathbf{0}^\top & 1 \end{pmatrix} \tag{6.21}$$

Par dérivation on peut alors calculer :

$$\mathbf{V}_c = \frac{\delta \mathbf{o}_c}{\delta \mathbf{q}} \frac{\delta \mathbf{q}}{\delta s} \tag{6.22}$$

On a donc :

$$\mathbf{V}_c = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \tag{6.23}$$

Note : lorsque le robot n'est pas fixé au sol $\mathbf{q}$ et $\dot{\mathbf{q}}$ ne sont pas obligatoirement de même taille. En effet la position du robot est paramétrée par un repére flottant. Pour $\dot{\mathbf{q}}$ la vitesse angulaire est exprimée suivant les axes $(x, y, z) \in \mathbb{R}^3$ mais l'orientation dans $\mathbf{q}$ peut être paramétrée soit par une matrice de rotation de taille 9, ou un quaternion de taille 4, ou des angles d'Euler de taille 3. Dans tous les cas, pour un robot non fixé au

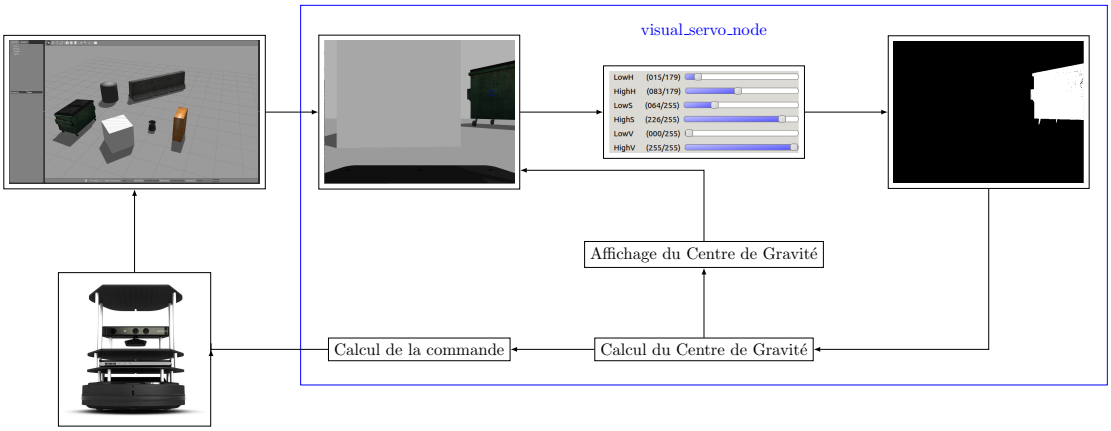


FIGURE 6.3 – Séquence d'opérations de base

sol, il n'est pas possible d'écrire Eq.6.22 mais on peut toujours avoir Eq.6.23 qui est l'application de l'espace tangent de la paramétrisation de la pose spatial du robot (et de ses moteurs) dans l'espace des vitesses. Par abus de language la matrice de l'équation Eq.6.23 est appellée Jacobienne articulaire.

Finalement en considérant le problème spécifié par l'équation Eq.6.24 on peut écrire :

$$\begin{cases} \min_{\dot{\mathbf{q}}} & \| \dot{\mathbf{q}} \| \\ \text{tel que} & \mathbf{e} = -\lambda \mathbf{L}_s \mathbf{J}(\mathbf{q}) \dot{\mathbf{q}} \end{cases} \tag{6.24}$$

ce qui peut s'écrire algèbriquement :

$$\dot{\mathbf{q}} = -\lambda (\mathbf{L}_s \mathbf{J}(\mathbf{q}))^{+} \mathbf{e} \tag{6.25}$$

Pour implémenter le contrôleur sur le robot Baxter il faut implémenter la matrice Jacobienne compléte et la matrice d'interaction. Pour le robot Turtlebot 2 on peut simplifier en considérant uniquement la vitesse angulaire ω_y et la matrice $\mathbf{J}(\mathbf{q})$ étant une matrice identité.

6.2 Traitement d'image pour le TurtleBot 2

Le but de ce programme est de générer une commande qui suit un objet en utilisant la couleur. La structure du programme est donc la suivante :

1. La première étape est d'obtenir une image couleur en souscrivant au topic approprié de la caméra du turtlebot.
2. La couleur est ensuite extraite de l'image en utilisant l'espace HSV et en testant l'appartenance d'un pixel à un intervalle.
3. En calculant la position du centre de gravité de l'objet dans l'image on calcule la commande à envoyer aux moteurs de la base mobile pour diminuer la différence entre le centre de l'image et le centre de gravité de l'objet. La commande est envoyée en publiant sur le topic approprié.

L'ensemble se basera sur la structure du programme fournie à la fin de ce sujet.

6.2.1 Extraction d'une image

La première étape est d'obtenir une image couleur à partir du robot. Il est donc nécessaire d'écrire la partie du code permettant de souscrire au topic fournissant cette image. On pourra pour cela s'inspirer de l'exemple du client souscripteur disponible à l'adresse suivante : http://wiki.ros.org/roscpp_tutorials/Tutorials/WritingPublisherSubscriber.

Il faudra penser à adapter le contexte au fait que nous considérons ici du C++. La fonction de callback devrait être notamment traité de façon différente par rapport à l'exemple.

La deuxième étape consiste à extraire les informations de la structure fournie par ROS. Les couleurs doivent être placées de façon appropriée dans l'image **cv_ptr**.

6.2.2 Extraction de la couleur

L'image est ensuite convertie au format HSV et placée dans la structure **imgHSV** (ligne 73). La structure **imgThresholded** contient le résultat du traitement de l'image HSV (ligne 76 – 77). Les pixels qui sont dans les intervalles spécifiés (lignes 42 – 48) par les ascenceurs sont mis à 255 tandis que les autres sont mis à 0. A partir de ces pixels il est possible de calculer le centre de gravité (lignes 80 – 96).

6.2.3 Calcul de la commande

Pour le Turtlebot 2 la commande se calcule ici trés simplement en effectuant la différence entre la position du centre de gravité et le centre de l'image. Cette différence est ensuite multipliée par un gain. Le but de cette partie est de reconstruire le message à envoyer sur le topic utilisé pour la commande du robot. La rotation va permettre au robot de centrer l'objet dans l'image.

Le publisher s'écrit en utilisant le code disponible à l'adresse suivante : http://wiki.ros.org/roscpp_tutorials/Tutorials/WritingPub

6.3 OpenCV - Code de démarrage

Le code est disponible sur github. Vous pouvez l'installer dans voter workspace :

```
cd ~/catkin_ws/src
git clone https://github.com/olivier-stasse/tp_ros_visual_servo.git
```

La première étape consiste à

```
#include <iostream>
#include <unistd.h>
#include <cv_bridge/cv_bridge.h>
#include <ros/ros.h>

#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <image_transport/image_transport.h>

#include <sensor_msgs/Image.h>
#include <sensor_msgs/image_encodings.h>
#include <geometry_msgs/Twist.h>

using namespace cv;
using namespace std;

static const std::string OPENCV_WINDOW = "Image window";

class ImageConverter
{
  image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;

  int iLowH_,iHighH_;
  int iLowS_,iHighS_;
  int iLowV_,iHighV_;

public:
  ImageConverter()
    : it_(nh_),iLowH_(15), iHighH_(83),
      iLowS_(190), iHighS_(226),
      iLowV_(0), iHighV_(255)
  {
    // Creates Node and subscribe to input video feed
    /*
     A COMPLETER
    */
    cv::namedWindow(OPENCV_WINDOW);


    //Create trackbars in "Control" window
    cvCreateTrackbar("LowH", "Control", &iLowH_, 179); //Hue (0 - 179)
    cvCreateTrackbar("HighH", "Control", &iHighH_, 179);

```

```
45    cvCreateTrackbar("LowS", "Control", &iLowS_, 255); //Saturation (0 - 255)
46    cvCreateTrackbar("HighS", "Control", &iHighS_, 255);
47    cvCreateTrackbar("LowV", "Control", &iLowV_, 255); //Value (0 - 255)
48    cvCreateTrackbar("HighV", "Control", &iHighV_, 255);
49
50    // Create publisher to send to control
51    /* A COMPLETER */
52   }
53
54   ~ImageConverter()
55   {
56    cv::destroyWindow(OPENCV_WINDOW);
57   }
58
59   void imageCb(const sensor_msgs::ImageConstPtr& msg)
60   {
61    cv_bridge::CvImagePtr cv_ptr;
62    try
63    {
64     cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
65    }
66    catch (cv_bridge::Exception& e)
67    {
68     ROS_ERROR("cv_bridge exception: %s", e.what());
69     return;
70    }
71
72    Mat imgHSV;
73    cvtColor(cv_ptr->image, imgHSV, COLOR_BGR2HSV);
74
75    Mat imgThresholded;
76    inRange(imgHSV, Scalar(iLowH_, iLowS_, iLowV_),
77     Scalar(iHighH_, iHighS_, iHighV_), imgThresholded); //Threshold the image
78
79    imshow("Thresholded Image", imgThresholded); //show the thresholded image
80    double cx=0.0,cy=0.0;
81    unsigned int nb_pts=0,idx=0;
82    for(unsigned int j=0;j<imgThresholded.rows;j++)
83      {
84   for(unsigned int i=0;i<imgThresholded.cols;i++)
85    {
86     if (imgThresholded.data[idx]>124)
87       {
88    cx+=(double)i;
89    cy+=(double)j;
90    nb_pts++;
91       }
92     idx+=1;
93    }
94      }
95    cx = cx/(double)nb_pts;
96    cy = cy/(double)nb_pts;
97
98    // Draw the CoG in the image
99    cv::circle(cv_ptr->image, cv::Point((int)cx,(int)cy), 10, CV_RGB(0,0,255));
100   // Update GUI Window
101   cv::imshow(OPENCV_WINDOW, cv_ptr->image);
102
103   // Control filling
104   /* A COMPLETER */
105   cv::waitKey(3);
106  }
107 };
108
109
110 int main( int argc, char** argv )
111 {
112
113  namedWindow("Control", CV_WINDOW_AUTOSIZE); //create a window called "Control"
114
```

```
115
116   Mat imgOriginal;
117   bool bSuccess;
118
119   /* Node initialization */
120   /* A COMPLETER */
121   ImageConverter ic;
122   ros::spin();
123
124   do
125     {
126       ros::spin();
127
128       usleep(30000);
129     } while(waitKey(30)==27);
130
131   return 0;
132
133 }
```

6.4 Aide pour la compilation avec OpenCV 2

OpenCV2 est la version officielle supportée par Indigo et Jade. Pour l'utiliser, vous devez juste ajouter une dépendence sur opencv2 et dans find_package.

6.5 Travail à faire

6.5.1 Turtlebot 2

Compilation

Compiler le code, rajouter les champs de classes manquants et compléter les fichiers **package.xml** et **CMakeLists.txt**.
Conseils :

1. Vérifiez le nom du code source. Utilisez le nom de l'exécutable pour changer un nom de fichier source si vous devez le faire.
2. Vérifiez les étapes nécessaires à la création d'un node en vous inspirant des souscripteurs et des publieurs sur des topics.
3. En cas de SEGFAULT pensez au fait qu'en C++ l'ordre de déclaration des variables est important.
4. Si vous avez un problème avec l'édition des liens n'oubliez pas que vous pouvez spécifier des dépendences vers d'autres paquets en modifiant les fichiers **package.xml** (blocks <build_depend> et <run_depend>) et **CMakeLists.txt**.

Souscription au topic fournissant l'image

Créer le souscripteur pour obtenir l'image et appeler la méthode **imageCb** lorsqu'une image est présente sur le topic. Conseils :

1. Cherchez quel est le nom du topic auquel vous souhaitez souscrire. Utilisez par exemple la commande rostopic.
2. Le nom du topic est relié au fait que l'on cherche à retrouver une image.
3. Utilisez les tutoriels permettant d'écrire des souscripteurs et des publieurs sur des topics.
4. Utilisez également les tutoriels qui font des callbacks sur des classes C++.
5. Recherchez des exemples sur Internet utilisant la classe **ImageTransport**.

Test du pipeline

Tester le pipeline de traitement d'image avec un gilet jaune.
Conseils :

1. Utilisez les paramètres ROS pour pouvoir publier les valeurs $H_{min}, H_{max}, S_{min}, S_{max}, V_{min}, V_{max}$. Regardez la documentation qui se trouve ici : http://wiki.ros.org/roscpp/Overview/Parameter%20Server
2. Dans le constructeur, testez si les paramètres existent si non alors les mettre dans l'arbre des paramètres. Pour cela utiliser un namespace. Par exemple :

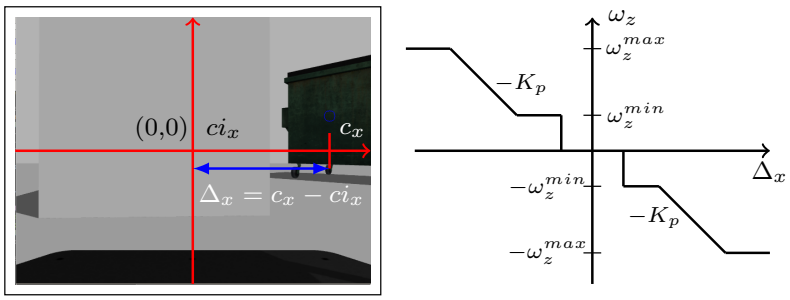


FIGURE 6.4 – Commande en vitesse à partir de la variation en pixels.

```
1  if (!nh_.hasParam("/visual_servo_initiales/Hmin"))
2    nh_.getParam("/visual_servo_initiales/Hmin",Hmin_);
3
```

Commande du robot

Etapes :

1. Trouver le nom du topic qui prend un message **<geometry_msgs : :Twist>** et qui fait bouger la base mobile du turtlebot 2.
2. Créer le publisheur permettant d'envoyer une commande en vitesse à la base mobile.
3. Calculer $|Delta_x|$ la différence entre le milieu de l'image ci_x égale à 320 et les coordonnées suivant l'axe x du centre de gravité.
4. Calculer la commande à envoyer sur le robot. Pour le moment, on ne fait pas de mouvement linéaire $\mathbf{v} = [v_x, v_y, v_z]^\top = [0.0, 0.0, 0.0]^\top$. Les vitesses angulaires autour de l'axe ω_x et de l'axe ω_y sont à zéros. La vitesse angulaire ω_z est celle qui nous intéresse. La figure Fig.6.5.1 illustre la forme de la commande qui dépend de la variation en pixels. Lorsque $|\Delta_x|$ est inférieure à une certain valeur la commande est nulle. Sinon on cherche à ce que la commande soit linéairement proportionnelle à Δ_x grâce au gain K_p. Lorsque Δ_x est positif le robot doit tourner vers la gauche et donc ω_z doit être négatif. Enfin si la commande est trop grande elle doit être saturée soit par $-\omega_z^{max}$ soit par ω_z^{max}. Si elle est positive mais trop petite alors il faut une commande minimale $-\omega_z^{min}$ soit par ω_z^{min} pour pouvoir vaincre la friction.
5. Construire le message qui est envoyé à la base mobile.
6. Implémenter un accés aux paramètres de façon à pouvoir changer les gains en ligne.
7. Tester le contrôleur sur gazebo.
8. Tester le contrôleur sur le robot.
9. Déplacer le contrôleur sur le robot.
10. Ecrire un fichier de launch qui initialise par défaut les gains du contrôleur soit sur le robot soit sur gazebo.

Utilisation de la kinect

Etapes :

1. Trouver le nom du topic qui fournit un message **<sensor_msgs : :Image>** issue de la caméra de profondeur.
2. Ecrire le prototype de la méthode capable de récupérer l'image de profondeur.
3. Créer le souscripteur permettant de récuperer l'image de profondeur.
4. Développer la méthode permettant d'extraire l'information de distance de l'image de profondeur.

III Modèles et Simulation

7. Universal Robot Description Format (URDF)

Le format URDF décrit au format XML le modèle d'un robot. Le paquet urdfm implémente le parser qui analyse ce modèle. Ce chapitre se base largement sur la page ros suivante http://wiki.ros.org/urdf/XML.

7.1 Capteurs - Sensors

7.1.1 Norme courante

L'élément **<sensor>** décrit les propriétés basiques d'un capteur visuel (caméra/capteur de rayons) Voici un exemple d'un élément décrivant une caméra :

```
<sensor name="my_camera_sensor" update_rate="20">
<parent link="optical_frame_link_name"/>
<origin xyz="0 0 0" rpy="0 0 0"/>
<camera>
  <image width="640" height="480" hfov="1.5708" format="RGB8" near="0.01" far="50.0"/>
</camera>
</sensor>
```

Et voici un exemple d'un élément représentant un laser :

```
<sensor name="my_ray_sensor" update_rate="20">
<parent link="optical_frame_link_name"/>
<origin xyz="0 0 0" rpy="0 0 0"/>
<ray>
  <horizontal samples="100" resolution="1" min_angle="-1.5708" max_angle="1.5708"/>
  <vertical samples="1" resolution="1" min_angle="0" max_angle="0"/>
</ray>
</sensor>
```

Cet élément a les attributs suivants :

- **name** : *(requis) (string)*
  Le nom du capteur
- **update_rate** : *(optional) (float) (Hz)*
  La fréquence a laquelle le capteur produit les données. Si cette valeur n'est pas spécifiée les données sont générées à chaque cycle.
- **type** : *(nouveau) (required) (string)*
  Le type du capteur peut être **camera**,**ray**,**imu**,**magnetometer**,**gps**, **force_torque**,**contact**,**sonar**,**rfidtag**,**rfid**.

Les éléments cet élément sont les suivants :

— **<parent>** : *(required)*
link*(required) (string)*
Le nom du corps auquel le capteur est attaché.
— **<origin>** : *(optional : à defaut l'identité si ce n'est pas spécifié*
Il s'agit de la position de l'origine du capteur optique relative au repère de référence du corps parent. Le repère du capteur optique adopte la convention suivante : z-vers le devant, x à droite, y en bas.
  — **xyz** : *(optionel : par défaut à 0*
  Représente la position suivant les axes x, y, z.
  — **rpy** : *(optionel : par défaut à l'identité*
  Représente les axes fixes roll, pitch et yaw en radians.
— **<camera>** : *(optionel)*

  — **<image>** : *(requis)*

    — **width** : *(requis) (unsigned int) (pixels)*
    Largeur de l'image en pixels
    — **height** : *(requis) (unsigned int) (pixels)*
    Hauteur de l'image en pixels
    — **format** : *(requis)(string)*
    Format de l'image. Peut-être chacune des chaînes définies dans le fichier **image_encodings.h** dans le paquet **sensors_msgs**.
    — **hfov** : *((requis)(float)(radians)*
    Largeur du champ de vue de la caméra (*rad*).
    — **near** : *(requis)(float)(m)*
    Distance la plus proche de la perception de la caméra. (*m*)

    — **far** : *(requis)(float)(m)*
    Distance la plus lointaine de la perception de la caméra. (*m*)

— **<ray>** : *(optionel)*

  — **<horizontal>** : *(optionel)*

    — **samples** : *(optionel : default 1)(unsigned int)*
    Le nombre de rayons simulés pour générer un cycle complet de perception laser.
    — **resolution** : *(optionel : default 1)(float)*
    Ce nombre est multiplié par **samples** pour déterminer le nombre de données produite. Si la résolution est inférieure à 1 les données de profondeur sont interpolées, si la résolution est supérieure à 1 les données sont moyennées.
    — **min_angle** : *(optionel : default 0)(float)(radians)* :

    — **max_angle** : *(optionel : default 0)(float)(radians)* :
    Doit être supérieur ou égale à **min_angle**.
  — **<vertical>** : *(optionel)*

    — **samples** : *(optionel : default 1)(unsigned int)*
    Le nombre de rayons simulés pour générer un cycle complet de perception laser.
    — **resolution** : *(optionel : default 1)(float)*
    Ce nombre est multiplié par **samples** pour déterminer le nombre de données produite. Si la résolution est inférieure à 1 les données de profondeur sont interpolées, si la résolution est supérieure à 1 les données sont moyennées.
    — **min_angle** : *(optionel : default 0)(float)(radians)* :

    — **max_angle** : *(optionel : default 0)(float)(radians)* :
    Doit être supérieur ou égale à **min_angle**.

7.2 Corps - Link

L'élément **link** décrit un corps rigide avec une inertie et des caractéristiques visuelles.

```
<link name="my_link">
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
</link>
```

7.2.1 Attributs

— **name** : *(requis) (string)*
Le nom du corps lui même.

7.2.2 Elements

— **<inertial>** *(optionel)*
Les propriétés inertielles du corps.
  — **<origin>** *(optionel : l'identité si rien n'est spécifié)*
  Il s'agit de la pose du repère de référence inertiel, relative au repère du corps de référence. L'origine du repère de référence inertiel doit être au centre de gravité. Les axes du repère de référence inertiel n'ont pas forcément besoin d'être alignés avec les axes principaux de l'inertie.
    — **xyz** *(optionel : par défaut à zéro*
    Représente le décalage selon les axes x, y, z
    — **rpy** *(optionel : par défaut égale à l'identité si non spécifié*
    Représente la rotation du corps selon les axes $(roll, pitch yaw)$ en radians.
  — **<mass>** La masse du corps est fixée par l'attribut **value** de cet élément
  — **<inertia>** La matrix d'inertie 3×3 représentée dans le repère inertiel. Parce que cette matrice d'inertie est symmétrique, seuls les 6 éléments diagonaux supérieurs sont donnés ici en utilisant les attributs **ixx**,**ixy**,**ixz**,**iyy**, **yz**,**izz**.
— **<visual>** *(optionel)* Les propriétés visuelles du corps. Cet élément spécifie la forme de l'objet (boite, cylindre, etc.) pour la visualisation. **Note** : il est possible d'avoir plusieurs blocks <visual> pour le même corps. Leur union définit la représentation visuelle du corps.
  — **name** *(optionel)* Spécifie un nom pour une partie de la géométrie d'un corps. Cela est particulièremen utile pour faire référence à une partie de la géométrie d'un corps.
  — **<origin>** Le repère de référence de l'élément visuel par rapport au repère de référence du corps.
    — **xyz** Représente la position de l'élément visuel dans le repère de référence du corps (en m).
    — **rpy** Représente l'orientation de l'élément visuel dans le repère de référence du corps (en *radians*).
  — **<geometry>** *(requis)* La forme de l'objet visuel. Cela peut-être fait l'un des élèments suivants :
    — **box** L'attribut **size** contients la taille (longueur, larger, hauteur) de la boîte. L'origine de la boîte est le centre.

- **cylinder** Spécifie le rayon (**<radius>**) et la hauteur (**length**) du cylindre. L'origine du cylindre est son centre.
- **sphere** Spécifie le rayon (**<radius>**). L'origine de la sphère est son centre.
- **mesh** Il s'agit d'un fichier décrivant une surface par des triangles. Il est spécifié par un nom de fichier (**<filename>**) et un facteur d'échelle (**<scale>**) aligné suivant l'axe de la boîte englobante de la surface. Le format recommandé est Collada (**.dae**) mais les fichiers STL (**.stl**) sont également supportés. Les fichiers de surfaces ne sont pas transférés entre les machines faisant référence au même modèle (/robot_description). Cela doit être un fichier local.

— **<material>** *(optionel)* Spécifie le matériel de l'élément visuel. Il est permis de spécifier un élément matériel à l'extérieur de l'élément **<link>** dans l'élément **<robot>**. Dans un élèment link il est possible de référencer le matériel par son nom.
- **name** *(optionel)* Spécifie le nom du matériel.
- **<color>** *(optionel)*
  rgba La couleur d'un matériel spécifié par un ensemble de quatre nombres représentant rouge/vert/blue/alpha, chacun dans l'intervalle $[0, 1]$.
- **<texture>** *(optionel)*
  La texture d'un matériel est spécifié par un fichier nommé **filename**.

— **<collision>** (optionel) Les propriétés de collision d'un corps. Elles peuvent être différentes des informations visuelles. Par exemple, on utilise souvent des modèles simplifés pour la collision afin de réduire les temps de calcul. Il peut y avoir des instances multiples de **collision** pour un même corps. L'union des géométries que ces instances définissent forment la représentation pour la collision de ce corps.
- **name** (optionel)
  Spécifie un nom pour une partie géométrique du corps. Ceci est utile pour faire référene à des parties spécifiques de la géométrie d'un corps.
- **<origin>** (optionel : défaut à l'identité si n'est pas spécifié) Le référentiel de référence de l'élément collision exprimé dans référentiel de référence du corps.
  - **xyz** (optionel : par défaut à zéro) Représente la position de l'élément de collision.
  - **rpy** (optionel : par défaut à l'identité) Représente les axes fixes roulis, tangage et lacet exprimés en radians.
- **<geometry>** Cette partie suit la même description que pour l'élément **visuel** décrit plus haut.

7.2.3 Résolution recommandée pour les mailles

Pour la détection de collision utilisant les paquets ROS de planification de mouvements le moins de face possible est recommandé pour les mailles de collision qui sont spécifiées dans l'URDF (idéalement moins que 1000). Si possible, l'approximation des éléments pour la collision avec des primitives géométriques est encouragée.

7.2.4 Elements multiples des corps pour la collision

Il a été décidé que les fichiers URDFs ne devait pas accepter des multiples groupes d'éléments de collision pour les corps, et cela même s'il y a des applications pour cela. En effet l'URDF a été conçu pour ne représenter que les propriétés actuelles du robot, et non pas pour des algorithmes extérieurs de détection de collision. Dans l'URDF les éléments **visual** doivent être le plus précis possible, les éléments de **collision** doivent être toujours une approximation la plus proche, mais avec bien moins de triangles dans les mailles.

Si des modèles de collision moins précis sont nécessaires pour le control ou la détection de collision il est possible de placer ces mailles/géométries dans des éléments XMLs spécialisés. Par exemple, si vos contrôleurs ont besoin de représentations particuliérement simplifiées, il est possible d'ajouter la balise **<collision_checking>** après l'élément **<collision>**.

Voici un exemple :

```
40 <link name="torso">
41   <visual>
42     <origin rpy="0 0 0" xyz="0 0 0"/>
43     <geometry>
44       <mesh filename="package://robot_description/meshes/base_link.DAE"/>
45     </geometry>
```

```
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="−0.065 0 0.0"/>
    <geometry>
      <mesh filename="package://robot_description/meshes/base_link_simple.DAE"/>
    </geometry>
  </collision>
  <collision_checking>
    <origin rpy="0 0 0" xyz="−0.065 0 0.0"/>
    <geometry>
      <cylinder length="0.7" radius="0.27"/>
    </geometry>
  </collision_checking>
  <inertial>
    ...
  </inertial>
</link>
```

Le parseur URDF ignorera ces éléments spécialisés et un programme particulier et spécialisé peut parser le XML pour obtenir cette information.

7.3 Transmission

Un élément de transmission est une extension du modèle URDF de description des robots qui est utilisé pour décrire la relation entre un actionneur et un joint. Cela permet de modéliser des engrenages ou des mécanismes parallèles. Une transmission transforme des variables d'efforts et de flots de telle sorte que leur produit (la puissance) reste constante. Plusieurs actionneurs peuvent être liés à de multiples joints à travers une transmission complexe.

Voici un exemple d'un élément de transmission :

```
<transmission name="simple_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="foo_joint">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="foo_motor">
    <mechanicalReduction>50</mechanicalReduction>
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </actuator>
  </transmission>
```

7.3.1 Attributs de transmission

L'élément de transmission a un attribut :

— **name** (requis)
Spécifie le nom unique d'une transmission.

7.3.2 Eléments de transmission

La transmission a les éléments suivants :

— **<type>** (une occurence)
Spécifie le type de transmission.

— **<joint>** (une ou plusieurs occurences)
Un joint auquel la transmission est connectée. Le joint est spécifié par l'attribut **name**, et les sous-éléments suivants :

  — **<hardwareInterface>** (une ou plusieurs occurences)
  Spécifie une interface matérielle supportée (par exemple **EffortJointInterface** ou **PositionJointInterface**). Notons que la valeur de cet élément doit être **EffortJointInterface** quand cette transmission est chargée dans Gazebo et **hardware_interface/EffortJointInterface** quand cette transmission est chargée dans RobotHW.

— **<actuator>** (une ou plusieurs occurences)
Il s'agit de l'actionneur auquel la transmission est connectée. L'actionneur est spécifié son attribut **name** et les sous éléments suivants :

— **<mechanicalReduction>** (optionel)
Spécifie la réduction mécanique de la transmission joint/actionneur. Cette balise n'est pas nécessaire pour toutes les transmissions.
— **<hardwareInterface>** (optionel) (une ou plusieurs occurences) Spécifie une interface matérielle supportée. Notons que la balise **<hardwareInterface>** ne doit être spécifiée que pour des releases précédent Indigo. L'endroit pour utiliser cette balise est la balise **<joint>**.

7.3.3 Notes de développement

Pour le moment seul le projet **ros_control** utilise ces éléments de transmissions. Développer un format de transmission plus complet qui est extensible à tous les cas est quelque chose de difficile. Une discussion est disponible ici.

7.4 Joint

7.4.1 Elément <joint>

L'élément joint décrit la cinématique et la dynamique d'un joint et spécifie les limites de sécurité. Voici un exemple d'un élément joint :

```
<joint name="my_joint" type="floating">
  <origin xyz="0 0 1" rpy="0 0 3.1416"/>
  <parent link="link1"/>
  <child link="link2"/>

  <calibration rising="0.0"/>
  <dynamics damping="0.0" friction="0.0"/>
  <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
  <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_limit="0.5" />
</joint>
```

7.4.2 Attributs

L'élément joint a deux attributs :
— **name** (requis)
Spécifie un nom unique pour le joint.
— **type** (requis)
Spécifie le type de joint qui peut prendre une des valeurs suivantes :
— *revolute* - une charnière qui tourne autour d'un axe spécifié par les attributs décrits ci-dessous et les limites inférieures et supérieures.
— *continuous* - une charnière qui tourne autour d'un axe spécifié sans aucune limite.
— *prismatic* - un joint coulissant le long d'un axe spécifié avec un intervalle d'utilisation limité par une valeur inférieure et une valeur supérieure.
— *fixed* - Ce n'est pas réellement un joint car il ne bouge pas. Tous les degrées de liberté sont fixés. Ce type de joint ne requiert pas d'axe et n'a pas d'intervalle d'utilisation spécifié par une valeur supérieure et inférieure.
— *floating* - Ce joint permet des mouvements suivant les 6 degrés de liberté.
— *planar* - Ce joint permet des mouvements dans un plan perpendiculaire à l'axe.

7.4.3 Elements

L'élément **joint** a les sous éléments suivants :
— **<origin>** (optionel : défaut à l'identité si non spécifié)
Il s'agit de la transformée du corps parent vers le corps enfant. Le joint est localisé à l'origine du référentiel enfant.
— **xyz** (optionel : par défaut à zéro)
Représente la position du joint.
— **rpy** (optionel : par défaut au vector zéro)
Représente la rotation autour un axe de rotation fixé : d'abord autour de l'axe de roulis (roll), puis autour de l'axe de tangage (pitch), puis celui de lacet (yaw). Tous les angles sont spécifiés en radians.

— **<parent>** *(requis)*
Le corps parent avec l'attribut nécessaire :
— **link** Le nom du corps qui est le parent du joint dans cet arbre.
— **<child>** *(requis)*
Le corps enfant avec l'attribut nécessaire :
— **link** Le nom du corps qui est l'enfant du joint dans cet arbre.

7.5 Gazebo

7.5.1 Eléments pour les corps/links

Nom	Type	Description
material	value	Le matériel de l'élément visuel
gravity	bool	Utilisation de la gravité
dampingFactor	double	La constante de décroissance de la descente exponentielle pour la vitesse d'un corps - Utilise la valeur et multiplie la vitesse précédente du corps pary $(1 - dampingFactor)$.
maxVel	double	Vitesse maximum contact d'un corps (corrigée par troncature).
minDepth	double	Profondeur minimum autorisée avant d'appliquer une impulsion de correction.
mu1,mu2	double	Coefficients de friction μ pour les directions principales de contact pour les surfaces de contacts comme définies par le moteur dynamique Open Dynamics Engine (ODE) (voir la description des paramètres dans le guide d'utilisateur d'ODE)
fdir1	string	3-tuple spécifiant la direction de mu1 dans le repère de référence local des collisions.
kp,kd	double	Coefficient de rigidité k_p et d'amortissement k_d pour les contacts des corps rigides comme défini par ODE (ODE utilisent erp et cfm mais il existe un mapping entre la erp/cfm et la rigidité/amorti)
selfCollide	bool	Si ce paramètre est à vrai, le corps peut entrer en collision avec d'autres corps du modèle.
maxContacts	int	Le nombre maximum de contacts possibles entre deux entités. Cette valeur écrase la valeur spécifiée la valeur de l'élément $max_contacts$ définit dans la section **physics**.
laserRetro	double	Valeur d'intensité retournée par le capteur laser.

7.5.2 Eléments pour les joints

Nom	Type	Description
stopCfm,stopErp	double	Arrêt de la constraint force mixing (cfm) et error reduction parameter (erp) utilisée par ODE
provideFeedback	bool	Permet aux joints de publier leur information de tenseur (force-torque) à travers un plugin Gazebo
implicitSpringDamper, cfmDamping	bool	Si ce drapeau est à vrai, ODE va utiliser ERP and CFM pour simuler l'amorti. C'est une méthode numérique plus stable pour l'amortissement que la méthode par défaut. L'élément **cfmDamping** est obsolète et doit être changé en **implicitSpringDamper**.
fudgeFactor	double	Met à l'échelle l'excés du moteur du joint dans les limites du joint. Doit-être entre zéro et un.

7.6 model_state

Note : Cette partie est un travail en progrès qui n'est pas utilisé à l'heure actuelle. La représentation des configurations pour des groupes de joints peut-être également utilisée en utilisant le format srdf.

7.6.1 Elément <model_state>

Cet élément décrit un état basic du modèle URDF correspondant. Voici un exemple de l'élément state :

```
<model_state model="pr2" time_stamp="0.1">
  <joint_state joint="r_shoulder_pan_joint" position="0" velocity="0" effort="0"/>
  <joint_state joint="r_shoulder_lift_joint" position="0" velocity="0" effort="0"/>
</model_state>
```

7.6.2 Model State

— **<model_state>**

— **model** (requis : string)
Le nom du modèle dans l'URDF correspondant.
— **timestamp** (optionnel : float, en secondes)
Estampillage temporel de cet état en secondes.
— **<joint_state>** (optionnel : string).

— **joint** (requis : string)
Le nom du joint auquel cet état se référe.
— **position** (optionnel : float ou tableau de floats)
Position pour chaque degré de liberté de ce joint.
— **velocity** (optionnel : float ou tableau de floats)
Vitesse pour chaque degré de liberté de ce joint.
— **effort** (optionnel : float ou tableau de floats)
Effort pour chaque degré de liberté de ce joint.

7.7 model

Le format de description unifié des robots (URDF pour Unified Robot Description Format) est une spécificiation XML pour décrire un robot. La spécification ne peut pas décrire tous les formats bien que celui-ci ait été conçu pour être le plus général possible. La limitation principale est que seule des structures en arbre peuvent être représentée ce qui ne permet pas de représenter des robots parallèles. De plus les specifications supposent que le robot est constitué de corps rigides connectés par des joints, les éléments flexibles ne sont pas supportés. La spécification couvre :

— La description cinématique et dynamique du robot
— La représentation visuelle du robot
— Modèle de collision du robot

La description d'un robot est constitué d'un ensemble de corps, et de joints connectant tous les corps ensembles. La description typique d'un robot ressemble donc à :

```
<robot name="pr2">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>

  <joint>  ....  </joint>
  <joint>  ....  </joint>
  <joint>  ....  </joint>
</robot>
```

Vous pouvez constater que la racine du format URDF est l'élément **<robot>**.

IV Appendices

8. Mots clefs pour les fichiers launch

8.1 <launch>

La balise tag est l'élèment racine de tout fichier roslaunch. Son unique rôle est d'être un conteneur pour les autres élèments.

8.1.1 Attributs

deprecated="deprecation message" ROS 1.1 : Indique aux utilisateurs que roslaunch n'est plus utilisé.

8.1.2 Elements

- <node> Lance un node.
- <param> Affecte un paramètre dans le serveur de paramètre.
- <remap> Renomme un espace de nom.
- <machine> Déclare une machine à utiliser dans la phase de démarrage.
- <rosparam> Affecte des paramètres en chargant un fichier rosparam.
- <include> Inclus d'autres fichiers roslaunch.
- <env> Spécifie une variable d'environnement pour les noeuds lancés.
- <test> Lance un node de test (voir rostest).
- <arg> Déclare un argument.
- <group> Groupe des élm̀ents partageant un espace de nom ou un renommage.

9. Rappels sur le bash

9.1 Lien symbolique

9.1.1 Voir les liens symboliques

Pour lister le contenu de votre répertoire vous pouvez utiliser la commande suivante :

```
ls
```

La sortie est la suivante :

```
catkin_ws            Music
catkin_ws_prenom_nom  Pictures
Desktop              Public
Documents            Templates
Downloads            Videos
examples.desktop
```

Pour savoir si le répertoire **catkin_ws** est un raccourci vers un autre répertoire vous pouvez utiliser la commande suivante :

```
ls -al catkin_ws
```

Le résultat est alors :

```
lrwxrwxrwx  1 pnom pnom       20 dec.   6 08:43 catkin_ws -> catkin_ws_prenom_nom
```

9.1.2 Créer un lien symbolique

Pour créer un lien de votre répertoire **catkin_ws_prenom_nom** vers **catkin_ws** :

```
ln -s catkin_ws_prenom_nom catkin_ws
```

9.1.3 Enlever un lien symbolique

Pour enlever un lien, il faut faire comme pour enlever un fichier :

```
rm catkin_ws
```

9.2 Gestion des variables d'environnement

La liste des variables d'environnements peut s'obtenir en faisant :

```
env
```

La liste des variables d'environnements peut s'obtenir en cherchant les lignes contenant la chaîne ROS :

```
env | grep ROS
```

Il est possible de changer la valeur de la variable d'environnement **ROS_MASTER_URI** pour pointer vers le robot **turtlebot2** sur le port **11311**.

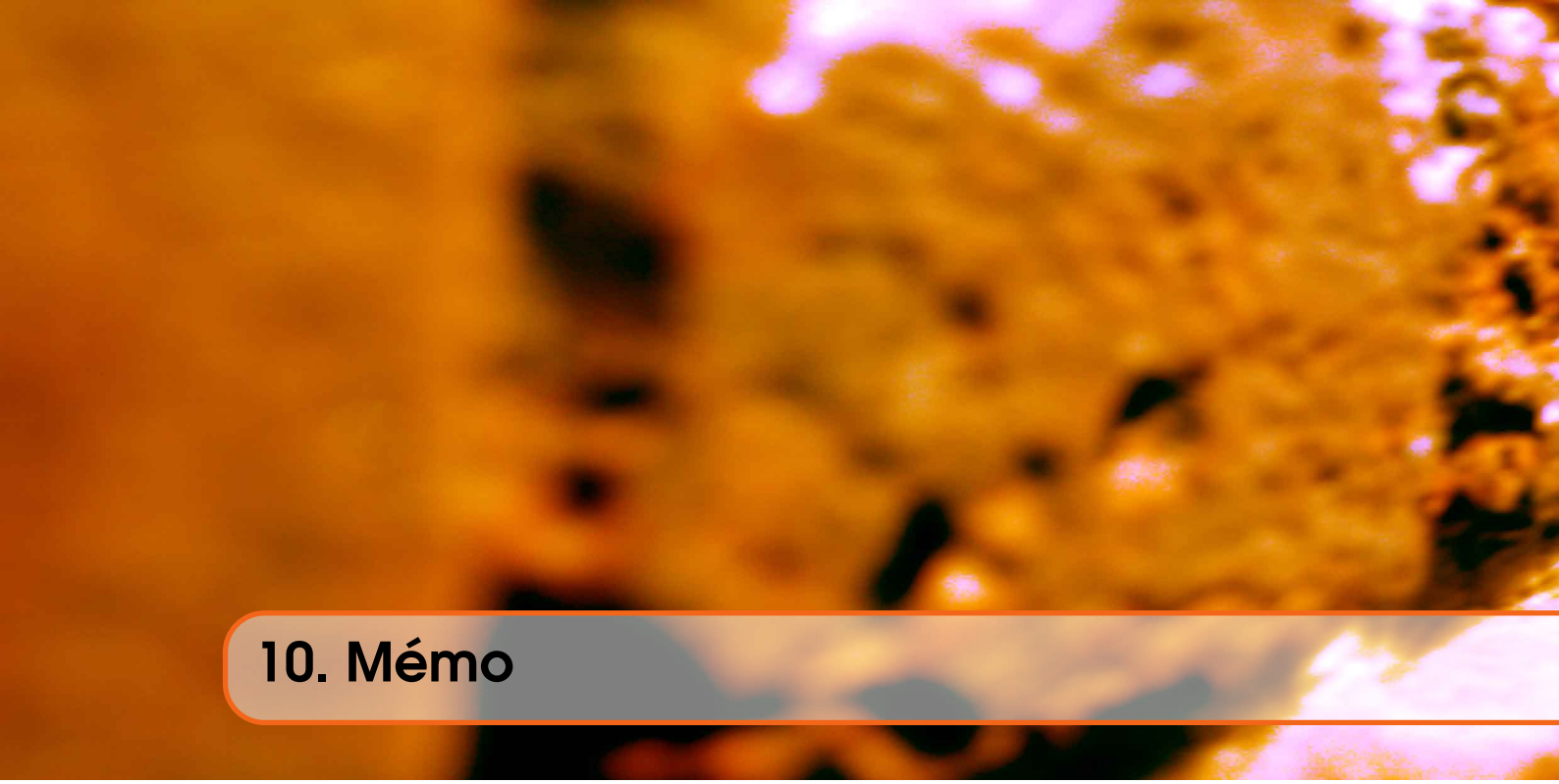
```
export ROS_MASTER_URI:=http://turtlebot2:11311
```

Les variables d'environnement sont locales à un terminal. Si vous avez ouvert plusieurs terminaux la modification de la valeur ne peut affecter que celui où la modification a été faites.

9.3 Fichier .bashrc

Le fichier **.bashrc** est lu à chaque fois qu'un terminal est lancé. La version qui est lue peut-être différente si le fichier est modifié d'un lancement à l'autre. Il est possible de forcer le fichier **.bashrc** à lire un autre fichier. Par exemple pour lire le fichier **setup.bash** qui se trouve dans votre répertoire **catkin_ws** :

```
source /home/etudiant/catkin_ws/devel/setup.bash
```



10. Mémo

ROS Mémo

Outils en ligne de commande pour le système de fichiers

rospack/rosstack	Un outil pour inspecter les packages/stacks
roscd	Change de répertoire vers le paquet ou la stack.
rosls	Liste les paquets et les stacks.
roscreate-pkg	Créer un nouveau paquet ROS.
rosdep	Installe le système des dépendences des paquets ROS.
catkin_make	Construit les paquets ROS.
roswtf	Affiche les erreurs et les alertes sur un système ROS ou un fichier de launch.
rxdeps	Affiche la structure de paquet et les dépendences

Usage :

```
$ rospack find [package]
$ roscd [package[/subdir]]
$ rosls [package[/subdir]]
$ roscreate-pkg [package name]
$ rosmake [package]
$ rosdep install [package]
$ roswtf or roswtf [file]
$ rxdeps [options]
```

Outils en ligne de commande les plus utilisés

roscore

Une collection de **blue**nodes et de programmes qui prérequis pour une application ROS. Il est nécessaire d'avoir roscore pour permettre aux nodes ROS de communiquer.

roscore contient
- master
- parameter server
- rosout

Usage :

```
$ roscore
```

rosmsg/rossrv

rossmsg/rossrv affiche les champs des Message/Service (msg/srv)

Usage :

`$ rosmsg show`	Affiche les champs dans le message
`$ rosmsg md5`	Affiche les valeurs md5 des messages
`$ rosmsg package`	Liste tous les messages d'un paquet
`$ rosmsg packages`	Liste tous les packages avec un message

Exemples :

Affiche le msg Pose

```
$ rosmsg show Pose
```

Liste les messages du paquet nav_msgs

```
$ rosmsg package nav_msgs
```

Liste les packages ayant des messages

```
$ rosmsg packages
```

rosrun

rosrun permet d'exécuter un programme sans changer de répertoire :

Usage :

```
$ rosrun package executable
```

Exemple :

Lancer turtlesim

```
rosrun turtlesim turtlesim_node
```

rosnode

Affiche des informations sur les nodes ROS, c.a.d. les publications, souscriptions et connections.

Usage :

`$ rosnode ping`	Test la connectivité du node
`$ rosnode list`	Liste les nodes actifs
`$ rosnode info`	Affiche les informations d'un node
`$ rosnode machine`	Liste les nodes exécutés sur une machine particulière
`$ rosnode kill`	Termine l'exécution d'un node

Exemples :

Termine tous les nodes

```
$ rosnode kill -a
```

Liste tous les nodes sur une machine

```
$ rosnode machine sqy.local
```

Test tous les nodes

```
$ rosnode ping -all
```

roslaunch

Démarre les nodes localement et à distance via SSH, et initialise les paramètres.

Exemples :

Lancer un fichier d'un package

```
roslaunch package filename.launch
```

Lancer sur un port différent :

```
roslaunch -p 1234 package filename.launch
```

Lancer uniquement les nodes locaux :

```
roslaunch -local package filename.launch
```

rostopic

Un outil pour afficher des informations à propos des topics c.a.d des publishers, des subscribers, la fréquence de publications et les messages.

Usage :

`$ rostopic bw`	Affiche la bande passante d'un topic
`$ rostopic echo`	Affiche le contenu du topic
`$ rostopic hz`	Affiche la fréquence de mise à jour d'un topic
`$ rostopic list`	Affiche la liste des topics
`$ rostopic pub`	Publie des données sur un topic
`$ rostopic type`	Affiche le type d'un topic (message)
`$ rostopic find`	Affiche les topics d'un certain type

Exemples :

Publie hello à une fréquence de $10\,Hz$

```
$ rostopic pub -r 10 /topic_name std_msgs/String hello
```

Affiche le topic /topic_name

```
$ rostopic echo -c /topic_name
```

Test tous les nodes

```
$ rosnode ping -all
```

rosparam

Un outil permet de lire et écrire des parameters ROS sur le serveur de paramètre, en utilisant notamment des fichiers YAML

Usage :

`$ rosparam set`	Spécifie un paramètre
`$ rosparam get`	Affiche un paramètre
`$ rosparam load`	Charge des paramètres à partir d'un fichier
`$ rosparam dump`	Affiche les paramètres disponibles
`$ rosparam delete`	Enlève un paramètre du serveur
`$ rosparam list`	Liste le nom des paramètres

Exemples :

Liste tous les paramètres dans un namespace

```
$ rosparam list /namespace
```

Spécifier un paramètre *foo* avec une liste comme une chaîne, un entier, un réel.

```
$ rosparam set /foo "['1', 1, 1.0]"
```

Sauve les paramètres d'un namespace spécifique dans le fichier **dump.yaml**

```
$ rosparam dump dump.yaml /namespace
```

rosservice

Cet outil permet de lister et d'appeller les services ROS.

Usage :

`$ rosservice list`	Affiche les informations relatives aux services actifs
`$ rosservice node`	Affiche le nom d'un node fournissant un service
`$ rosservice call`	Appelle le service avec les arguments donnés
`$ rosservice args`	Liste les arguments d'un service
`$ rosservice type`	Affiche le type de service (arguments d'entrées et de sorties)
`$ rosservice uri`	Affiche l'URI du service ROS
`$ rosservice find`	Affiche les services fournissant un type

Exemples :

Appel un service à partir de la ligne de commande

```
$ rosparam call /add_two_ints 1 2
```

Enchaîne la sortie de rosservice à l'entrée de rossrv pour voir le type du service

```
$ rosservice type add_two_ints | rossrv show
```

Affiche tous les services d'un type particulier

```
$ rosservice find rospy_tutorials/AddTwoInts
```

Outils en ligne de commande pour enregistrer des données

rosbag

C'est un ensemble d'outils pour enregistrer et rejouer des données des topics ROS. Il a pour but d'être très performant et éviter la déserialisation et la reserialisation des messages.

rosbag record génére un fichier ".bag" (ainsi nommé pour des raisons historiques) dans lequel le contenu de tous les topics passés en arguments sont enregistrés.

Exemples :

Record all topics :

```
$ rosbag record -a
```

Record select topics :

```
$ rosbag record topic1 topic2
```

rosbag play prend le contenu d'un fichier ou plusieurs fichiers ".bag" et le rejoue synchronisé temporellement.

Exemples :

Rejoue tous les messages sans attendre :

```
$ rosbag play -a demo_log.bag
```

Rejoue tous les fichiers de bag à la fois :

```
$ rosbag play demo1.bag demo2.bag
```

Outils graphiques

rqt_graph

Affiche le graphe d'une application ROS.

```
$ rosrun rqt_graph rqt_graph
```

rqt_plot

Affiche les données relatives à un ou plusieurs champs des topics ROS en utilisant matplotlib.

```
$ rosrun rqt_plot rqt_plot
```

rosbag

Un outil pour visualiser, inspecter et rejouer des historiques de messages ROS (non temps réel)

Enregistre tous les topics

```
$ rosbag record -a
```

Rejoue tous les topics

```
$ rosbag play -a topic_name
```

rqt_console

Un outil pour afficher et filtrer les messages sur rosout

Affiche les consoles

```
$ rosrun rqt_console rqt_console
```

tf_echo

Un outil qui affiche les informations relatives à propos d'une transformation particulière entre un repère source et un repère cible

Utilisation

```
$ rosrun tf tf_echo <source_frame>
<target_frame>
```

Examples :

Pour afficher la transformation entre /map et /odom :

```
$ rosrun tf tf_echo /map /odom
```

view_frames

Un outil pour visualiser l'arbre complet des transformations de coordonnées

Usage :

```
$ rosrun tf view_frames
$ evince frames.pdf
```

Bibliography

Books

Articles

Chapitre de livres

[CHC16] F. CHAUMETTE, S. HUTCHINSON et P. CORKE. “Handbook of Robotics 2nd Edition”. In : Spring Verlag, 2016. Chapitre Chapter 34 : Visual Servoing (cf. page 57).

Autres

[Goo16] GOOGLE. 2016. URL : https://developers.google.com/project-tango/ (cf. page 10).