

Robot Operating System

Introduction

Olivier Stasse

Copyright © 2022 Olivier Stasse

PUBLISHED BY LAAS-CNRS

[HTTPS ://HOMEPAGES.LAAS.FR/ OSTASSE](https://homepages.laas.fr/ostasse)

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Edition, 2022/03/18

Table des matières

1	Introduction	11
1.1	Motivations	11
1.1.1	Complexité des systèmes robotiques	11
1.1.2	Application Robotique	12
1.2	Concepts de ROS	12
1.2.1	Plomberie	12
1.2.2	Outils	13
1.2.3	Capacités	13
1.2.4	Exemples	14
1.3	Ecosystème	14
1.3.1	Historique	14
1.3.2	Fonctionnalités	14
1.3.3	Systèmes supportés	16
1.4	Exemple : Asservissement visuel sur TurtleBot 2	17

I

Introduction à ROS-1

2	Les paquets ROS	23
2.1	Configuration de l'environnement	23
2.1.1	Initialiser les variables d'environnement	23
2.1.2	Navigation dans le système de paquets	24
2.2	Structure générale des paquets ROS	24
2.3	Création d'un paquet	24
2.4	Description des paquets : le fichier package.xml	26
2.5	Compilation des paquets melodic	27
2.6	Chaînage de workspace	28

3	Graphe d'application avec ROS-1	29
3.1	Introduction aux concepts de ROS	29
3.1.1	Définition d'un node	29
3.1.2	Le système des name services	29
3.1.3	Lancer des nodes	30
3.1.4	Topic	31
3.2	rqt_console	33
3.3	roslaunch	33
3.3.1	Exemple	33
3.4	rosbag	37
3.5	rosservice	37
3.5.1	rosservice list	37
3.5.2	rosservice type (service)	38
3.5.3	rosservice call	38
3.6	rosparam	38
3.6.1	rosparam list	39
3.6.2	rosparam dump et rosparam load	40
3.7	Création de messages et de services	40
3.7.1	Introduction	40
3.7.2	Création d'un msg	41
3.7.3	Création d'un srv	42
3.7.4	Modifications CMakeLists.txt communes à msg et srv	43
4	Ecrire des nodes ROS-1	45
4.1	Topics	45
4.1.1	Emetteur	45
4.1.2	Souscripteur	49
4.1.3	Lancer les nodes	51
4.2	Services	52
4.2.1	Serveur	52
4.2.2	Client	53
4.2.3	Lancer les nodes	55

II

Travaux Pratiques

5	TurtleBot3	59
5.1	Démarrage	59
5.1.1	Introduction Modèle du TurtleBot3	59
5.1.2	Installation	59
5.2	Gazebo	59
5.2.1	Téléopérer le turtlebot	60
5.3	Construction de cartes et navigation	60
5.3.1	Construction de cartes	60
5.3.2	Navigation	61
5.3.3	RVIZ	61

6	Asservissement visuel	63
6.1	Introduction aux tâches	63
6.1.1	Contrôleur en vitesse	64
6.1.2	Projection d'un point dans une image	64
6.1.3	Calcul de la matrice d'interaction	65
6.1.4	Génération de la commande	66
6.2	Traitement d'image pour le TurtleBot 2	67
6.2.1	Extraction d'une image	67
6.2.2	Extraction de la couleur	68
6.2.3	Calcul de la commande	68
6.3	OpenCV - Code de démarrage	68
6.4	Travail à faire	71
6.4.1	Turtlebot 3	71

III

Modèles et Simulation

7	Universal Robot Description Format (URDF)	75
7.1	Capteurs - Sensors	75
7.1.1	Norme courante	75
7.1.2	Limitations	77
7.2	Corps - Link	77
7.2.1	Attributs	77
7.2.2	Elements	77
7.2.3	Résolution recommandée pour les mailles	78
7.2.4	Elements multiples des corps pour la collision	78
7.3	Transmission	79
7.3.1	Attributs de transmission	79
7.3.2	Eléments de transmission	79
7.3.3	Notes de développement	80
7.4	Joint	80
7.4.1	Elément <joint>	80
7.4.2	Attributs	80
7.4.3	Elements	80
7.5	Gazebo	81
7.5.1	Eléments pour les corps/links	81
7.5.2	Eléments pour les joints	81
7.6	model_state	81
7.6.1	Elément <model_state>	82
7.6.2	Model State	82
7.7	model	82
8	Simulation	83
8.1	Introduction	83
8.2	Algorithme général d'un simulateur système	84
8.3	Algorithme général d'un moteur dynamique	84
8.3.1	Problème 1 : Réalité	84
8.3.2	Problème 2 : Flexibilité logicielle	84

9	Gazebo - Ignition	85
9.1	Introduction	85
9.2	SDF format	85
9.2.1	Quand utiliser le format SDF ou le format URDF ?	86
9.3	Plugin	86
9.3.1	Plugin Gazebo	86
9.3.2	Plugin : exemple de base	86
9.4	Modèle de robot en SDF	88
9.4.1	Introduction	88
9.4.2	Modèle du soleil	89
9.4.3	Modèle du monde pour notre robot	89
9.4.4	Modèle du robot turtlebot3_waffle modifié	90
9.4.5	Plugin du capteur solaire	98

IV

Introduction à ROS-2

10	Graphe d'application avec ROS-2	105
10.1	Turtlesim et rqt	105
10.1.1	Utilisation de turtlesim	105
10.1.2	Installer rqt	107
10.1.3	Utiliser rqt	107
10.1.4	Essayer le service spawn	107
10.1.5	Essayer le service set_pen	108
10.1.6	Redirection	109
10.1.7	Arrêt de turtlesim	110
10.2	Node	110
10.2.1	Définition d'un node	110
10.2.2	ros2 run	110
10.2.3	ros2 node list	110
10.2.4	Redirection	111
10.2.5	ros2 node info	111
10.2.6	rqt_graph	112
10.2.7	Topic	112
10.3	Service	116
10.3.1	ros2 service list	116
10.3.2	ros2 service type (service)	117
10.3.3	ros2 service list -t	117
10.3.4	ros2 interface show	117
10.3.5	ros2 service call	117
10.4	Paramètres	118
10.4.1	ros2 param list	119
10.4.2	ros2 param dump	119
10.4.3	ros2 param load	120
10.5	Actions	120
10.5.1	Introduction	120
10.5.2	Préparation	120
10.5.3	Utilisation des actions	121
10.5.4	ros2 node info	121
10.5.5	ros2 action list	122
10.5.6	ros2 action info	122
10.5.7	ros2 interface show	123

10.5.8	ros2 action send_goal	123
10.5.9	Résumé	124
10.6	rqt_console	124
10.6.1	rqt	124
10.6.2	rqt_console	124
10.6.3	Niveaux d'enregistrement	125
10.7	Bag : fichier de données	125
10.8	Launch : démarrer une application	126
10.8.1	Introduction	126
10.8.2	Un exemple	126
10.8.3	Préparer son package	127
10.9	Création de messages et de services	128
10.9.1	Introduction	128
10.9.2	Création d'un msg	129
10.9.3	Création d'un srv	129
10.9.4	CMakeList.txt	130
10.9.5	package.xml	130
10.9.6	Génération des codes	130
10.9.7	Vérifier la création du fichier msg et srv	130

11	Ecrire des nodes ROS-2	131
11.1	Topics	131
11.1.1	Création d'un paquet	131
11.1.2	Emetteur	132
11.1.3	Souscripteur	137
11.1.4	Lancer les nodes	139
11.2	Services	141
11.2.1	Création d'un paquet	141
11.2.2	Serveur	142
11.2.3	Client	144
11.2.4	Lancer les nodes	147

V

Tutoriaux sur TIAGO

12	TIAGO	151
12.1	Installation	151
12.1.1	Utilisation de la machine VMWare	151
13	Contrôle	153
13.1	Téléopérer la base mobile avec le clavier	153
13.1.1	But	153
13.1.2	Lancer la simulation	153
13.1.3	Faire bouger le robot	153
13.2	Faire bouger la base à travers des commandes en vitesse	153
13.2.1	But	153
13.2.2	Lancer la simulation	154
13.2.3	Faire bouger le robot en avant et en arrière	154

13.3 Contrôleur des trajectoires articulaires	154
13.3.1 But du tutoriel	155
13.3.2 Contrôleur du torse	155
13.3.3 Contrôleur de la tête	155
13.3.4 Contrôleur du bras	155
13.3.5 Contrôleur du grippage Hey5	155
13.3.6 Contrôleur du grippage	155
13.3.7 Lancer la simulation	155
13.3.8 Lancer le nœud	156
13.4 Bouger des articulations individuellement	157
13.4.1 Lancer la simulation	157
13.4.2 Bouger les articulations individuellement	158
13.4.3 Bouger les articulations à partir de la ligne de commande	158
13.5 Contrôle de la tête	158
13.5.1 But du tutoriel	158
13.5.2 Lancer la simulation	159
13.5.3 Lancer l'exemple d'un client d'action en C++	159
13.6 Rejouer des mouvements du haut du corps prédéfinis	159
13.6.1 Lancer la simulation	159
13.6.2 Jouer un mouvement via une interface graphique	161
13.6.3 Jouer un mouvement en ligne de commande	162
14 TIAGo Navigation	163
14.1 Créer une carte avec gmapping	163
14.1.1 But	163
14.1.2 Exécution	163
14.2 Localisation et planification de mouvement	164
14.2.1 But	165
14.2.2 Localisation	165
14.2.3 Navigation autonome avec rviz	166
15 Planification de mouvements	169
15.1 Planifier dans l'espace de configuration	169
15.1.1 But	169
15.1.2 Lancer la simulation	169
15.1.3 Lancer les nœuds	169
15.1.4 Inspection du code	170
15.2 Planifier dans l'espace cartésien	171
15.2.1 But	172
15.2.2 Lancer la simulation	172
15.2.3 Lancer les nœuds	172
15.2.4 Inspection du code	172
15.3 Planification avec Octomap	175
15.3.1 Objectif	175
15.3.2 Lancer la simulation	175
15.3.3 Lancer les nœuds	176
15.3.4 Démarrer la démonstration	176
15.3.5 Le plugin Rviz de MoveIt!	178

15.4	Démonstration d'une prise et d'un replacement d'un objet	178
15.4.1	But	178
15.4.2	Lancer la simulation	179
15.4.3	Lancer les nodes	179
15.4.4	Démarrer la démonstration	179

VI

Navigation

16	Navigation : Nav2	185
16.1	Introduction	185
16.2	Concepts de Navigation	186
16.2.1	Serveurs d'action	186
16.2.2	Les nodes avec cycles de vie	187
16.2.3	Arbre de Comportement - Behavior Trees	187
16.3	Serveurs de navigation	187
16.3.1	Planificateurs, Contrôleurs, Lisseurs et serveurs de récupération	188
16.3.2	Planificateurs	188
16.3.3	Contrôleurs	188
16.3.4	Comportements	189
16.3.5	Lisseurs	189
16.3.6	Suiveur de point intermédiaires	189
16.4	Estimation d'état	190
16.4.1	Standards	190
16.4.2	Localisation globale et SLAM	190
16.4.3	Odométrie	190
16.5	Représentation de l'environnement	191
16.5.1	Les cartes de coût et les couches	191
16.5.2	Filtres de cartes de coût	191
16.6	REP-105	191
16.6.1	Spécifications	191
16.6.2	Relation entre repères	192
17	Plugins pour la navigation	193
17.1	Introduction	193
17.1.1	Pré requis	193
17.2	Ecrire un nouveau plugin Costmap2D	193
17.3	Exporter et construire le plugin GradientLayer	195
17.4	Utiliser le plugin dans Costmap2D	196
17.5	Lancer le plugin GradientLayer	197
18	Plugin de planification	199
18.1	Introduction	199
18.2	Création d'un nouveau plugin de planification de mouvements	199
18.3	Exporter le plugin de planification	201
18.4	Passer le nom du plugin par le fichier de paramètres	202
18.5	Lancer le plugin StraightLine	202

19	Ecrire un nouveau plugin de contrôleur	203
19.1	Introduction	203
19.2	Créer un nouveau plugin contrôleur	203
19.3	Exporter le plugin du contrôleur	206
19.3.1	Passer le nom du plugin à travers le fichier de paramètres	207
19.3.2	Lancer le plugin de contrôleur en poursuite pure	207

VII

Appendices

20	Mots clefs pour les fichiers launch	211
20.1	<launch>	211
20.1.1	Attributs	211
20.1.2	Elements	211
21	Rappels sur le bash	213
21.1	Lien symbolique	213
21.1.1	Voir les liens symboliques	213
21.1.2	Créer un lien symbolique	213
21.1.3	Enlever un lien symbolique	213
21.2	Gestion des variables d'environnement	214
21.3	Fichier .bashrc	214
22	Utiliser docker pour ROS	215
22.1	Préparer votre ordinateur	215
22.2	Utiliser l'image docker	215
22.2.1	Lancer un bash	215
22.2.2	Stopper, redémarrer une image	217
22.2.3	Copier un fichier de et vers un docker	217
22.3	Installer les images docker de ROS	217
22.3.1	Mise à jour de l'image	217
22.3.2	Erreurs	217
22.4	Construire l'image docker	218
22.5	Mac	218
22.5.1	Interface X11 sous Mac	218
23	Mémo	219
	Bibliography	223
	Books	223
	Articles	223
	Chapitre de livres	223
	Autres	223

1. Introduction

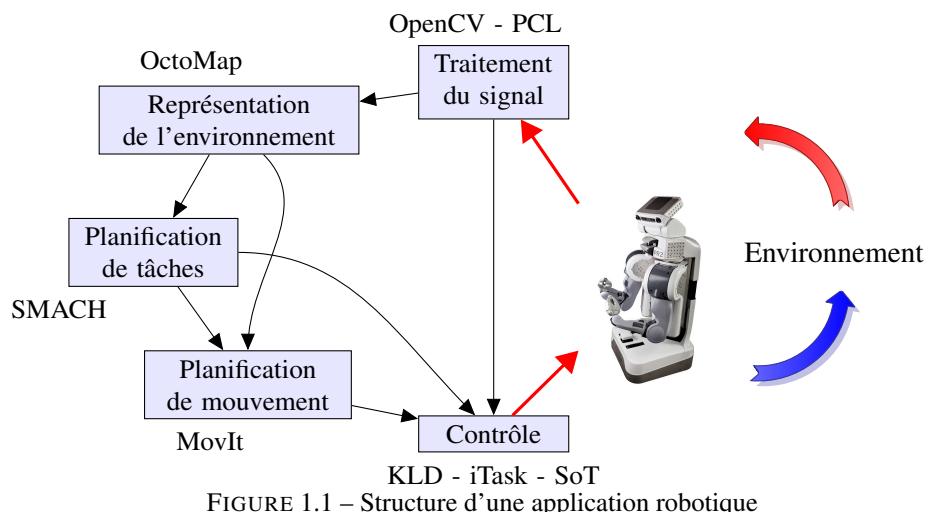


FIGURE 1.1 – Structure d'une application robotique

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 évalué 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 informatiques 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 très lente, par exemple dans le cadre de planification à grande échelle, ou sur des systèmes avec de très 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 représentation de l'environnement par exemple a connu récemment 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 [6]. Il est donc nécessaire de pouvoir facilement tester des structures logicielles implémentant des algorithmes différents pour une même fonctionnalité 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 clé 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 bibliothèques 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 langages 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 succès 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

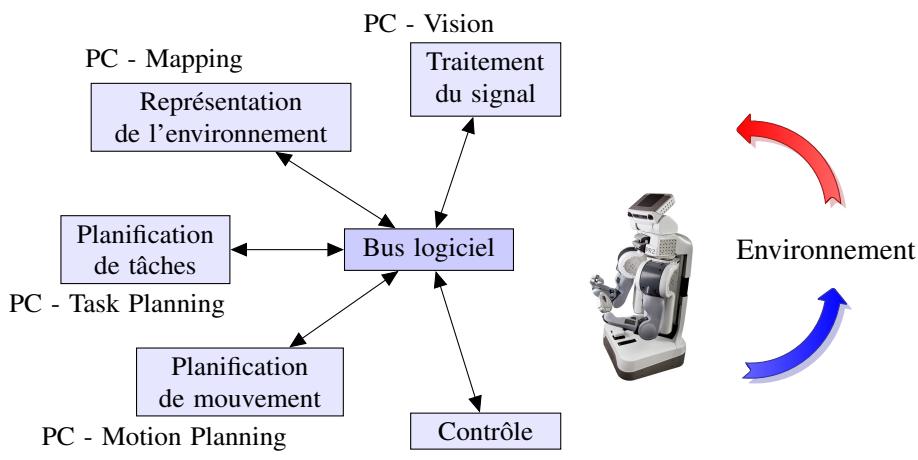


FIGURE 1.3 – Structure d'une application robotique

Les différents fonctionnalités décrites dans la figure 1 sont implémentés par des librairies spécialisées 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.
- Pour la reconstruction d'environnements :
 - En 2D : La pile de navigation pour ROS-1 et la <https://navigation.ros.org/pile> de navigation ROS-2
 - En 3D : ICP Mapping.
- Pour la planification de mouvements :
 - MoveIt!
- Pour la planification de mission :
 - SMASH avec de la documentation disponible ici
 - Mission Planner pour les drones
- Pour le contrôle :
 - iTask
 - ControlIt
 - Stack-Of-Tasks

1.2.4 Exemples

ROS-1 a été utilisé comme support pour des challenges robotiques comme le DARPA Robotics Challenge et le challenge robotique de la NASA. Dans le cadre du DARPA Robotics Challenge qui a pris place en Juin 2015, de nombreuses équipes ont du faire face à des défis d'intégration. ROS-1 a été une des briques fondamentales pour permettre aux équipes utilisant ATLAS de pouvoir fonctionner.

1.3 Ecosystème

1.3.1 Historique

ROS-1 a été écrit à partir de 2008 par la société Willow Garage fondée par Larry Page, également fondateur de Google. En plus de ROS-1, cette société a construit le PR-2, un robot très sophistiqué équipé 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 sociétés. 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 et ROS-2 sont résumé dans les tableaux 1.1 et 1.2. 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 compléter les types nécessaires dans les applications robotiques.
- **Librairie pour la géométrie des robots.**
La librairie 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 cinématique d'un robot, ses caractéristiques dynamiques, sa géométrie et ses capteurs.

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 - Mai	Jade Turtle (EOL - Mai 2017)
2016 - Mai	Kinetic Kame (EOL - Mai 2021)
2017 - Mai	Lunar Loggerhead (EOL - Mai 2019)
2018 - Mai	Melodic Morenia (EOL - Mai 2023)
2020 - Mai	Noetic Ninjemys (EOL - Mai 2025)

Tableau 1.1 – Historique de ROS et des releases - REP 3

2017 - Dec	Ardent Apalone (EOL - Dec 2018)
2018 - June	Bouncy Bolson (EOL - June 2019)
2018 - Dec	Crystal Clemminys (EOL - Dec 2019)
2019 - Mai	Dashing Diademata (EOL - Mai 2021)
2019 - Nov	Eloquent Elusor (EOL - Mai 2020)
2020 - May	Foxy Fitzroy (EOL - Mai 2023)
2021 - May	Galactic Geochelone (EOL - Nov 2022)
2022 - May	Humble Hawksbill (EOL - Mai 2027)
2020 - June	Rolling Ridley (EOL - Ongoing)

Tableau 1.2 – Historique des releases de ROS 2 - REP 2000

- **Un système d'appel de fonctions à distance interruptible.**
- **Un système de diagnostics.**
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-1 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-1 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 cœur de ros constitué des paquets roscomm a été porté sur différents systèmes d'exploitations comme OS X, Windows, et des distributions embarquées de Linux (Angstrom, OpenEmbedded/Yocto).

ROS-2 était initialement prévu pour être supporté sur Ubuntu, Windows 10 et MacOSX, mais il a été décidé récemment que le support de MacOS allait être moins prioritaire du à des difficultés de portage et une faible utilisation.

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 :

- Noetic Ninjems (Mai 2020 - Mai 2025)
 - Ubuntu Focal Fossa (20.04 LTS)
 - Debian Buster
 - Fedora 32
 - C++14, Boost 1.71/1.69/1.67, Lisp SBCL 1.4.16, Python 3.8, CMake 3.16.3/3.13.4/3.17
 - Ogre3D 1.9.x, Gazebo 9.0.0, PCL 1.10/1.9.1, OpenCV 4.2/3.2, Qt 5.13.2/5.12.5/5.11.3, PyQt5
- Melodic Melusa (Mai 2019 - Mai 2021)
 - Ubuntu Xenial (18.04 LTS)
 - Ubuntu Yakkety (17.10)
 - Debian Stretch
 - Fedora 28
 - C++14, Boost 1.62/1.65.1/1.66, Lisp SBCL 1.3.14, Python 2.7, CMake 3.7.2/3.9.1/3.10.2
 - Ogre3D 1.9.x, Gazebo 8.3.0/9.0.0, PCL 1.8.0/1.8.1, OpenCV 3.2, Qt 5.7.1/5.9.5/5.10.0, PyQt5
- 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

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 générer 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.

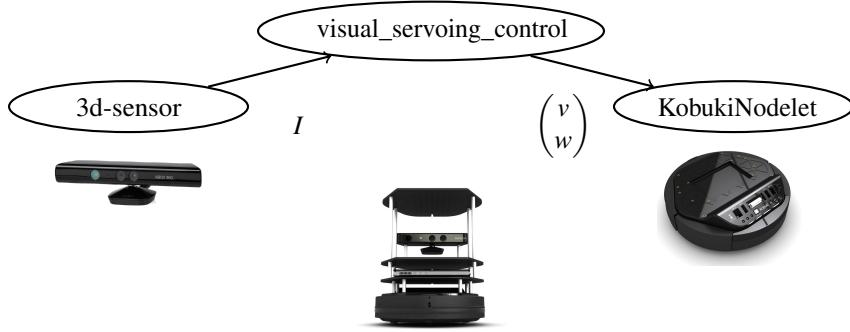


FIGURE 1.4 – Transfert d'une image I vers un node de contrôle qui calcule un torseur cinématique pour contrôler la base mobile du robot.

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.

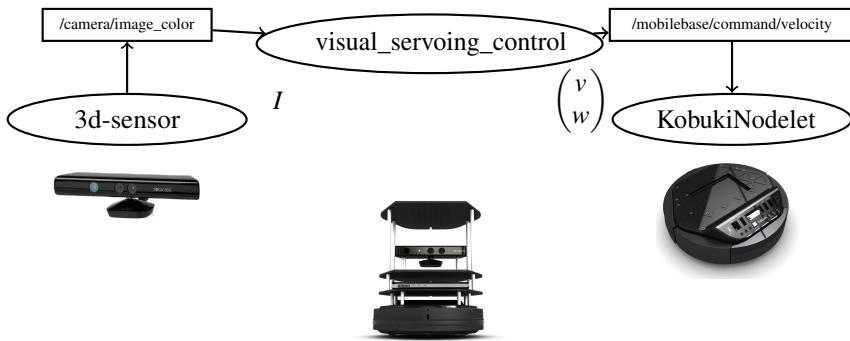


FIGURE 1.5 – Graphe d'une application ROS. Les ellipses représentent les noeuds de calcul ROS, et les rectangles les post-its où sont stockés des données disponibles pour tous les noeuds. Les flèches indiquent les flux d'informations échangées dans l'application.

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 utiliser le noeud pour la Kinect sur un autre robot qui utiliserait le même capteur. Le programmeur 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 paramètres 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).

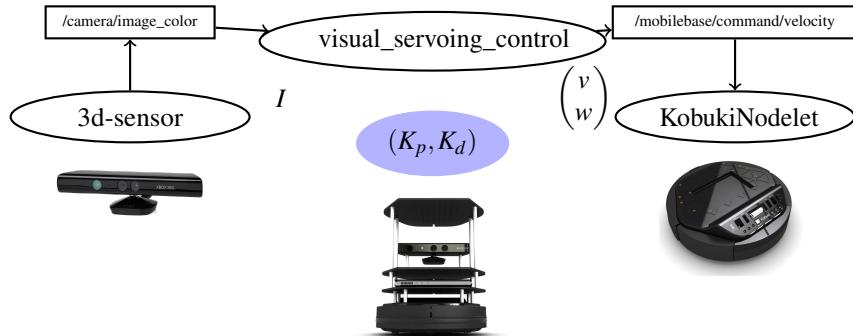


FIGURE 1.6 – Utilisation des paramètres pour modular le comportement d'une loi de commande. Ici les paramètres K_p et K_d correspondent respectivement au gain proportionnel et au gain dérivé

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** (pour les références entre noeuds). Un autre programme sert de serveur de paramètres **rosparam**. Le **rosmaster** enregistrent les nodes qui publient et qui souscrivent aux topics **rostopic**. Les publishers de topics sont informés de l'adresse des souscripteurs/subscribers par le **rosmaster**, et les communications entre les nodes par les topics s'effectuent directement. Une description technique détaillée est disponible à cette adresse : <https://wiki.ros.org/ROS/TechnicalOverview>.

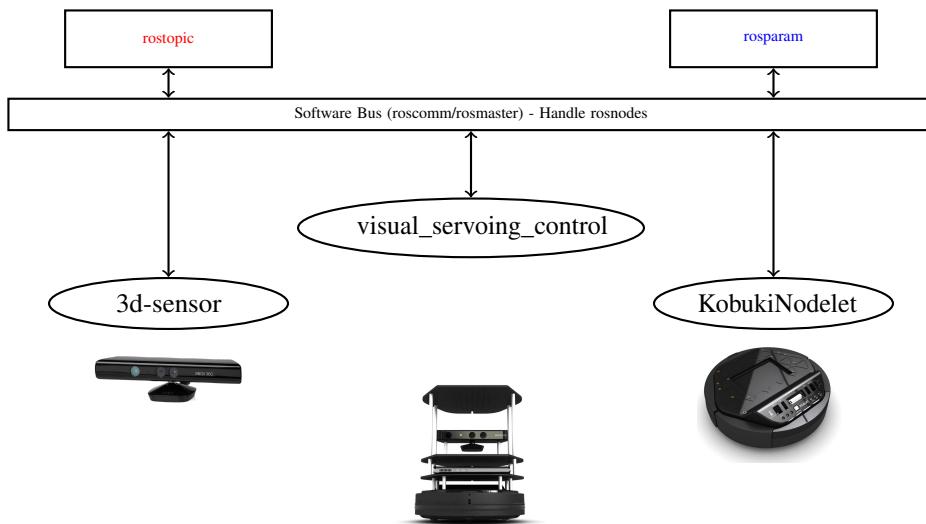


FIGURE 1.7 – Organisation interne de ROS pour implémenter les différents concepts

Une représentation plus générale du concept est donnée dans la figure Fig.1.8.

FIGURE 1.8 – Echange de données correspondant au mécanisme de service et de topic

FIGURE 1.9 – Echange de données correspondant au mécanisme d'action

Introduction à ROS-1

2	Les paquets ROS	23
2.1	Configuration de l'environnement	
2.2	Structure générale des paquets ROS	
2.3	Création d'un paquet	
2.4	Description des paquets : le fichier package.xml	
2.5	Compilation des paquets melodic	
2.6	Chaînage de workspace	
3	Graphe d'application avec ROS-1	29
3.1	Introduction aux concepts de ROS	
3.2	rqt_console	
3.3	roslaunch	
3.4	rosbag	
3.5	rosservice	
3.6	rosparam	
3.7	Création de messages et de services	
4	Ecrire des nodes ROS-1	45
4.1	Topics	
4.2	Services	

2. Les paquets ROS

Le système de gestion des paquets ROS est probablement un élément 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`

```
1 source /opt/ros/melodic/setup.bash
```

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

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

Puis ensuite initialiser l'espace de travail

```
1 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` :

```
1 source $HOME/catkin_ws/devel/setup.bash
```

Cependant le répertoire **devel** n'existe qu'une fois le script **catkin_make** utilisé (voir section 2.5 pour plus de détails).

Une fois le fichier **setup.bash** sourcé on peut trouver deux variables d'environnement très importantes :

```
1 env | grep ROS  
2 ROS_MASTER_URI=http://localhost:11311  
3 ROS_PACKAGE_PATH=/opt/ros/melodic/share
```

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

```
1 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'environnement *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** :

```
1 rosdep 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 :

```
1 roscd roscpp
2 roscd roscpp/cmake
```

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

```
1 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 répertoire **src** dans lequel on peut créer un répertoire 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 dépendances.

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 dépôts 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**.

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

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

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

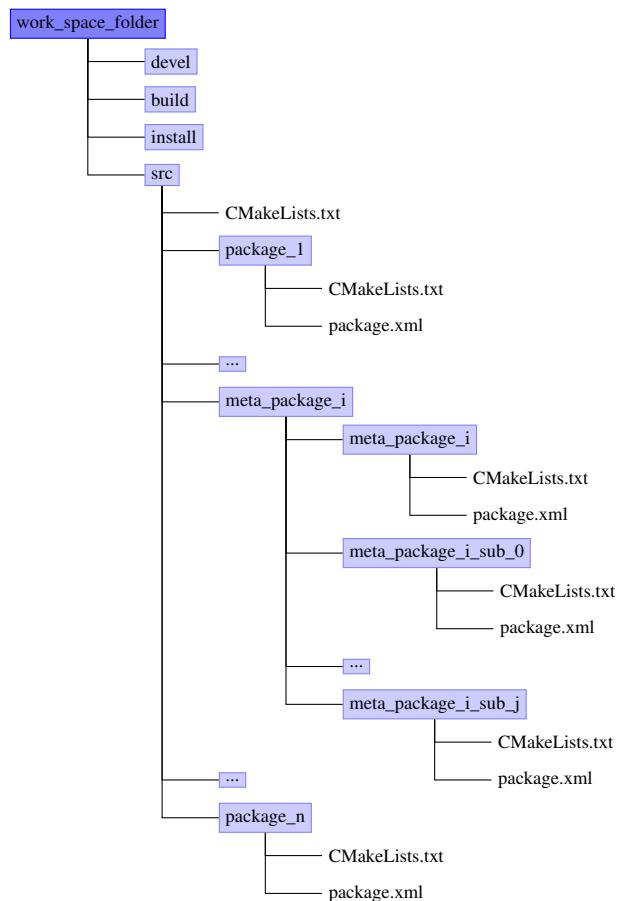


FIGURE 2.1 – Organisation de l'espace de travail : les paquets sources n'ont pas de structure hirarchique (ou  un seul niveau lorsque les dpts contiennent plusieurs paquets ROS).

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

```
1 rosdep depends beginner_tutorials
2 std_msgs
3 rospy
4 roscpp
```

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

```
1 roscd beginner_tutorials
2 cat package.xml
3 <package>
4 ...
5 <buildtool_depend>catkin</buildtool_depend>
6 <build_depend>roscpp</build_depend>
7 ...
8 </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.

```
1 <package format="2">
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.

```
1 <?xml version="1.0"?>
2 <package>
3   <name>beginner_tutorials</name>
4   <version>0.0.0</version>
5   <description>The beginner_tutorials package</description>
6
7   <!-- One maintainer tag required, multiple allowed, one person per tag -->
8   <!-- Example: -->
9   <maintainer email="jane.doe@example.com">Jane Doe</maintainer>
10
11  <!-- One license tag required, multiple allowed, one license per tag -->
12  <!-- Commonly used license strings: -->
13  <!-- BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
14  <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.

```
1 <!-- Url tags are optional, but mutiple are allowed, one per tag -->
2 <!-- Optional attribute type can be: website, bugtracker, or repository -->
3 <!-- Example: -->
4 <!-- <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>`.

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

Les dépendances sont de 6 ordres :

- build dependencies (format 1 & 2) : spécifie les paquets nécessaires pour construire ce paquet. Il peut s'agir des paquets qui contiennent 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 dépendances 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 dépendances au format 1 avec ROS melodic exprimées dans le fichier `package.xml`.

```

1  <!-- The *_depend tags are used to specify dependencies -->
2  <!-- Dependencies can be catkin packages or system dependencies -->
3  <!-- Examples: -->
4  <!-- Use build_depend for packages you need at compile time: -->
5  <!--   <build_depend>message_generation</build_depend> -->
6  <!-- Use buildtool_depend for build tool packages: -->
7  <!--   <buildtool_depend>catkin</buildtool_depend> -->
8  <!-- Use run_depend for packages you need at runtime: -->
9  <!--   <run_depend>message_runtime</run_depend> -->
10 <!-- Use test_depend for packages you need only for testing: -->
11 <!--   <test_depend>ttest</test_depend> -->
12 <buildtool_depend>catkin</buildtool_depend>
13 <build_depend>roscpp</build_depend>
14 <build_depend>rospy</build_depend>
15 <build_depend>std_msgs</build_depend>
16 <run_depend>roscpp</run_depend>
17 <run_depend>rospy</run_depend>
18 <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` :

```

1  <export>
2    <metapackage />
3  </export>
```

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

```

1  <!-- The export tag contains other, unspecified, tags -->
2  <export>
3    <!-- Other tools can request additional information be placed here -->
4
5  </export>
6 </package>
```

2.5 Compilation des paquets melodic

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

```

1 cd ~/catkin_ws/
2 catkin_make
3 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.

2.6 Chaînage de workspace

Lorsque l'une des commandes **catkin_make** ou **catkin_build** est utilisée celle-ci crée un fichier nommé **setup.bash**. Ce fichier une fois sourcé permet ensuite de travailler dans le workspace en initialisant correctement toutes les variables d'environnements. Un point important à noter est que ces variables d'environnements peuvent déjà avoir une valeur lors de l'appel à **catkin_make** ou **catkin_build**. Cette valeur est utilisée lors de la création du fichier **setup.bash**.

Ceci permet de faire du chaînage de workspace.

Par exemple dans le chapitre 12 il faut utiliser un workspace nommé **tiago_public_ws** permettant de simuler le robot Tiago. Avant de créer le répertoire **devel** il faut s'assurer que la variable d'environnement **ROS_PACKAGE_PATH** ne contient que la valeur

```
1 env | grep ROS
2 ROS_PACKAGE_PATH=/opt/ros/melodic/share
```

Une fois que la commande :

```
1 catkin build
```

a été utilisée, on peut sourcer le fichier **tiago_public_ws/devel/setup.bash**

On a alors :

```
1 env | grep ROS
2 ROS_PACKAGE_PATH=/home/etudiant/tiago_public_ws/devel/share...:/opt/ros/melodic/share
```

Tous les répertoires des paquets de **tiago_public_ws** se trouvent dans la variable **ROS_PACKAGE_PATH**.

Si on souhaite faire un workspace **app_for_tiago** qui utilise **tiago_public_ws** on peut alors après avoir sourcer le fichier **setup.bash** du workspace **tiago_public_ws** faire soit **catkin_make** ou **catkin_build**. On a alors :

```
1 env | grep ROS
2 ROS_PACKAGE_PATH=/home/etudiant/app_for_tiago/devel/share/...:/home/etudiant/tiago_public_ws/devel/share...:/opt/ro
   s/melodic/share
```

3. Graphe d'application avec ROS-1

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 aux concepts de ROS

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 std::cout/std::cerr.
- **roscore** : Master+rosout+parameter server (serveur de paramètres).

3.1.1 Définition d'un node

On peut définir un node comme étant le processus associé à l'exécution d'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 débogage. 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** :

```
1 rosnode list
```

Pour obtenir des informations sur un node :

```
1 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 :

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

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

```
1 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

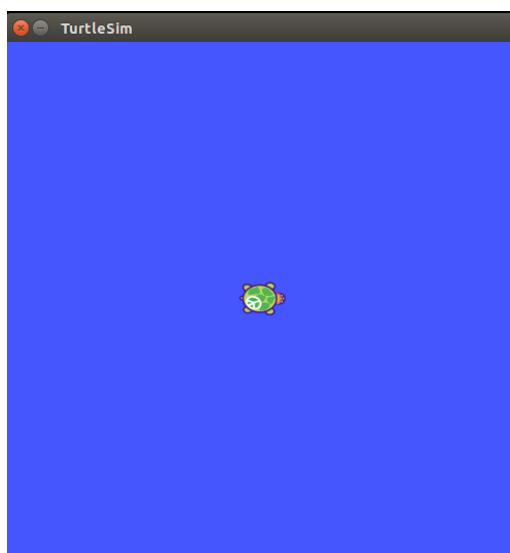


FIGURE 3.1 – Fenêtre résultat de rosrun turtlesim turtlesim_node

programme avec un nom différent il est possible de spécifier :

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

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

```
1 rosnode ping my_turtle
```

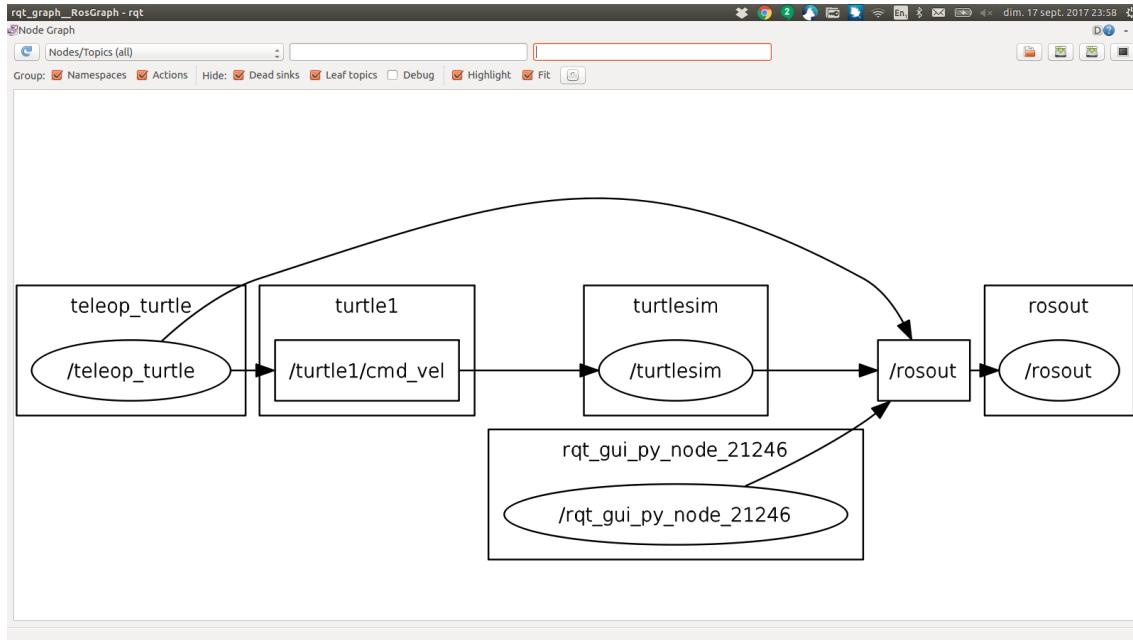
Tutoriel associé : Tutorial Understanding ROS Nodes : <http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes>

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 :

```
1 rosrun turtlesim turtle_teleop_key
```

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

```
1 rosrun rqt_graph rqt_graph
```

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

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 auxquelles les noeuds souscrivent. L’exécutable permettant d’avoir des informations sur les topics est **rostopic**. La liste des commandes en ROS-2 est similaire à celle de ROS-1 :

- bw : Affiche la bande passante prise par le topic
- echo : Affiche le contenu du topic en texte
- hz : Affiche la fréquence de publication du topic
- list : Affiche la liste des topics actifs
- pub : Publie des données sur le topic
- type : Affiche le type du topic

3.1.4.1 rostopic list

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

```
1 /rosout
2 /rosout_agg
3 /statistics
4 /turtle1/cmd_vel
5 /turtle1/color_sensor
6 /turtle1/pose
```

3.1.4.2 rostopic bw

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

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

On obtient alors le résultat suivant :

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

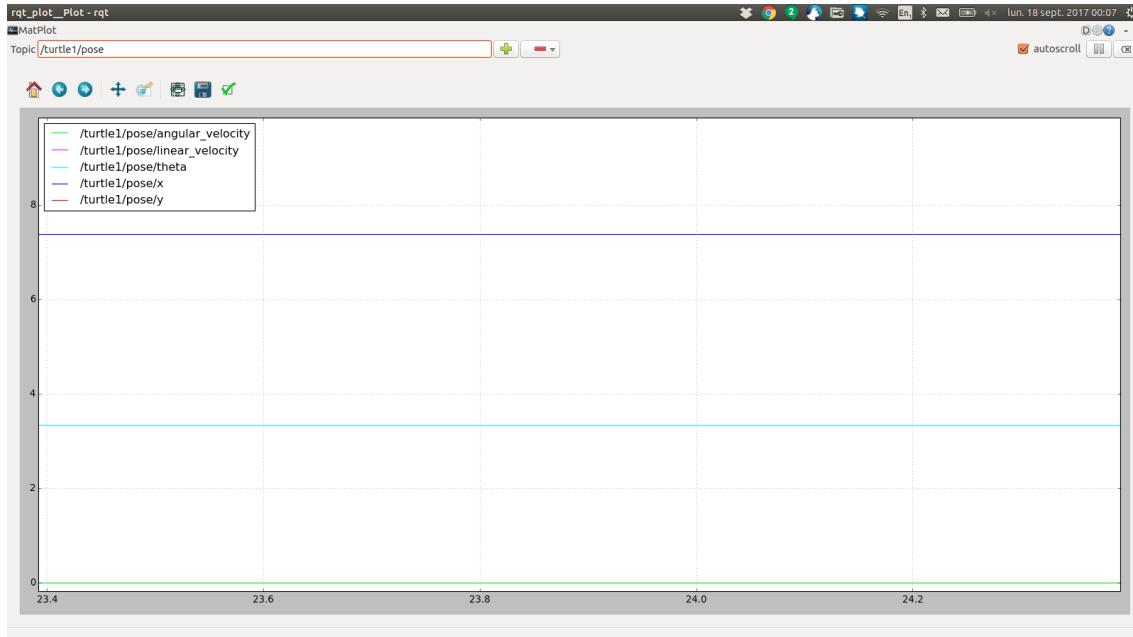


FIGURE 3.3 – Affichage de la position de la tortue dans la fenêtre

3.1.4.3 rostopic echo

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

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

donne :

```
1 x: 5.544444561
2 y: 5.544444561
3 theta: 0.0
4 linear_velocity: 0.0
5 angular_velocity: 0.0
```

3.1.4.4 rostopic type

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

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

affiche :

```
1 turtlesim/Pose
```

3.1.4.5 rostopic pub

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

```
1 rostopic pub /topic type
```

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

```
1 rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
2   x: 0.0
3   y: 0.0
4   z: 0.0
5   angular:
6     x: 0.0
7     y: 0.0
8     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 compléition 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**.

```
1 rosrun rqt_console rqt_console
2 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

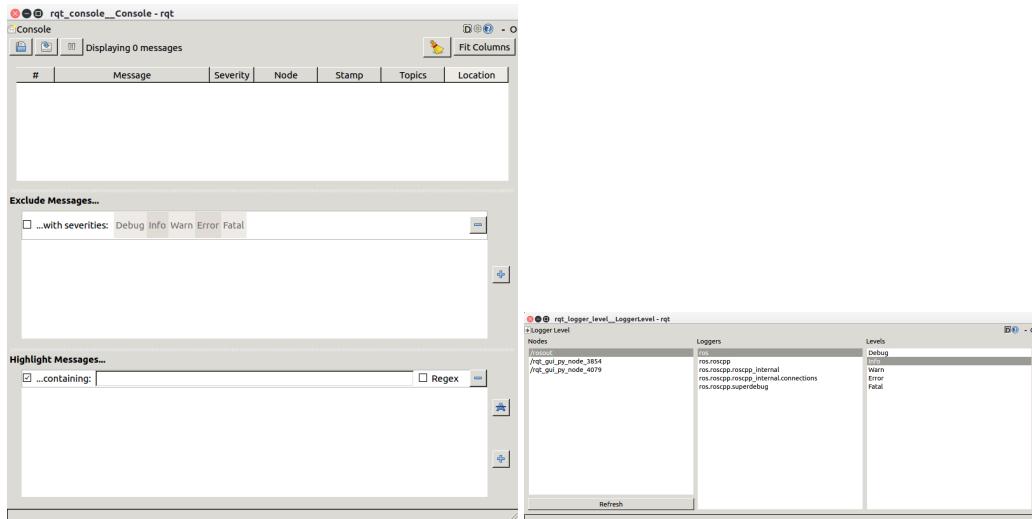


FIGURE 3.4 – Fenêtres correspondant à **rqt_console** et à **rqt_logger_level**

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

```
1 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** :

```
1 rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
2   x: 2000.0
3   y: 0.0
4   z: 0.0
5   angular:
6     x: 0.0
7     y: 0.0
8     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** affichée dans la figure.3.5.

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

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 :

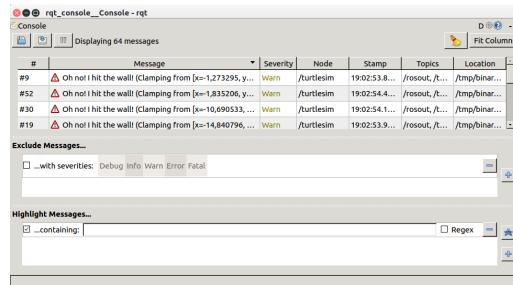


FIGURE 3.5 – Fenêtres correspondant à `rqt_console` lorsque la tortue s'écrase dans les murs

```

1 <launch>
2
3   <group ns="turtlesim1">
4     <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
5   </group>
6
7   <group ns="turtlesim2">
8     <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
9   </group>
10
11  <node pkg="turtlesim" name="mimic" type="mimic">
12    <remap from="input" to="turtlesim1/turtle1"/>
13    <remap from="output" to="turtlesim2/turtle1"/>
14  </node>
15
16 </launch>

```

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

```

1 roslaunch [package] [filename.launch]
2 rosrun beginner_tutorials
3 rosrun 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** :

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

Parce que cette ligne se trouve dans le bloc

```

1 <group ns="turtlesim1">
2   </group>

```

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

```
1 /turtlesim1/sim
```

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

```
1 /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 :

```
1 rostopic list
```

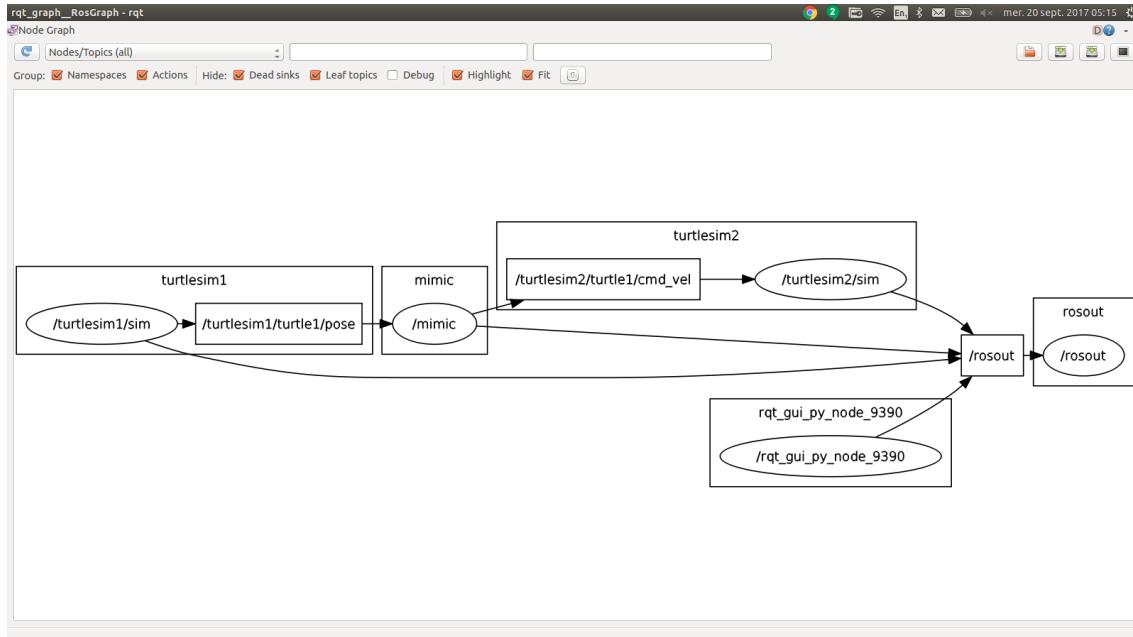
On constate que l'on trouve deux ensembles de topics correspondants aux deux nodes de **turtlesim** et aux deux namespaces **turtlesim1** et **turtlesim2**.

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

```

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

```

FIGURE 3.6 – Graphe correspondant au fichier `turtlemimic.launch`

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>

3.3.1.1 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>`.

3.3.1.2 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 :

```

1 <param name="foo" value="$(optenv NUM_CPUS 1)" />
2 <param name="foo" value="$(optenv CONFIG_PATH /home/marvin/ros_workspace)" />
3 <param name="foo" value="$(optenv VARIABLE ros_rocks)" />

```

— `$(find pkg)`

i.e. `$(find rospkg)/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. `$(anon rviz-1)`, génère un identifiant anonyme basé sur name. name lui même est un identifiant unique : des usages multiples de `$(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 :

```
1 <node name="$(anon foo)" pkg="rosy_tutorials" type="talker.py" />
2 <node name="$(anon foo)" pkg="rosy_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 :

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

affecte l'argument `my_foo` au paramètre `foo`.

Voici un autre exemple :

```
1 <node name="add_two_ints_server" pkg="beginner_tutorials" type="add_two_ints_server" />
2 <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 paramètres les valeurs `a` et `b`. Le fichier peut être alors utilisé de la façon suivante :

```
1 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 :

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

Note : Les expressions utilisant eval doivent analyser la chaîne de caractères complète. 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 :

```
1 <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 :

```
1 "$$(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 :

```
1 "$$(eval arg('foo'))"
2 "$$(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 :

```
1 <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 (optionnel)
Si `value` est à vrai la balise et son contenu sont inclus.
- unless=value (optionnel)
A moins que `value` soit à vrai la balise et son contenu sont inclus.

Par exemple :

```

1 <group if="$(arg foo)">
2   <!-- Partie qui ne sera incluse que si foo est vrai -->
3 </group>
4
5 <param name="foo" value="bar" unless="$(arg foo)" /> <!-- Ce param\`etre ne sera pas affect\`e tant que "unless"
   est vrai -->
```

3.3.1.3 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 générés sur les topics).

```
1 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**.

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

```

1 rosservice args print service arguments
2 rosservice call call the service with the provided args
3 rosservice find find services by service type
4 rosservice info print information about service
5 rosservice list list active services
6 rosservice type print service type
7 rosservice uri print service ROSRPC uri
```

3.5.1 rosservice list

La commande

```
1 rosservice list
```

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

```

1 /clear
2 /kill
3 /reset
4 /rosout/get_loggers
```

```

5 /rosout/set_logger_level
6 /spawn
7 /turtle1/set_pen
8 /turtle1/teleport_absolute
9 /turtle1/teleport_relative
10 /turtlesim/get_loggers
11 /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 :

```
1 rosservice type [service]
```

Ce qui donne pour le service clear :

```
1 rosservice type /clear
```

Le résultat est le suivant :

```
1 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 lorsqu'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 :

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

Ici aucun argument est nécessaire car le service est vide :

```
1 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** :

```
1 rosservice type /spawn
```

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

```

1 float32 x
2 float32 y
3 float32 theta
4 string name
5 ---
6 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 :

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

On trouve maintenant une deuxième tortue dans la fenêtre.

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 très naturel.



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

```

1 rosparam set set parameter
2 rosparam get get parameter
3 rosparam load load parameters from file
4 rosparam dump dump parameters to file
5 rosparam delete delete parameter
6 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 :

```

1 /background_b
2 /background_g
3 /background_r
4 /rostdistro
5 /roslaunch_uris/host_kinokoyama_38903
6 /rosversion
7 /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
```

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 :

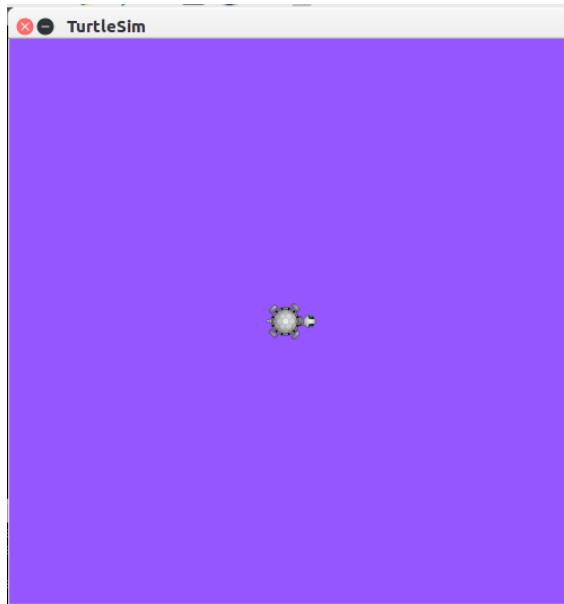


FIGURE 3.8 – Résultat de l'appel **rosparam set /background_r 150**

```

1 background_b: 255
2 background_g: 86
3 background_r: 150
4 rosdistro: 'kinetic'
5
6
7 roslaunch:
8   uris: {host_kinokoyama_41913: 'http://kinokoyama:41913/'}
9   rosversion: '1.12.7'
10
11
12 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 :

```

1 rosparam dump [file_name] [namespace]
2 rosparam load [file_name] [namespace]

```

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

```

1 rosparam dump params.yaml

```

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

```

1 rosparam load params.yaml copy
2 rosparam get /copy/background_b
3 255

```

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 générer le code source des messages dans des langages 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 référence 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 chaîne de caractères, et deux autres messages :

```
1 Header header
2 string child_frame_id
3 geometry_msgs/PoseWithCovariance pose
4 geometry_msgs/TwistWithCovariance twist
```

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

```
1 int64 A
2 int64 B
3 ---
4 int64 Sum
```

3.7.2 Création d'un msg

3.7.2.1 Format du fichier

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

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

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

```
1 string first_name
2 string last_name
3 uint8 age
4 uint32 score
```

3.7.2.2 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 :

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

Durant la compilation du paquet seule la ligne build_depend (message_generation) est nécessaire, tandis qu'à l'exécution 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 :

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

```

5     std_msgs
6     message_generation
7 )

```

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 dépendance à `message_runtime`.

```

1 catkin_package(
2 ...
3 CATKIN_DEPENDS message_runtime ...
4 ...)

```

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 :

```

1 # add_message_files(
2 #   FILES
3 #   Message1.msg
4 #   Message2.msg
5 # )

```

en

```

1 add_message_files(
2   FILES
3   Num.msg
4 )

```

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 décris dans le paragraphe 3.7.4.

3.7.2.3 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 :

```
1 rosmsg show [type_message]
```

Par exemple :

```
1 rosmsg show [type_message]
```

On voit alors :

```
1 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 :

```

1 [beginner_tutorials/Num]:
2 int64 num

```

3.7.3 Création d'un srv

3.7.3.1 Format du fichier

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

```

1 roscd beginner_tutorials
2 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 très pratique pour copier un fichier d'un paquet :

```
1 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 langages.

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 :

```
1 <build_depend>message_generation</build_depend>
2 <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'exécution.

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

```
1 # Do not just add this line to your CMakeLists.txt, modify the existing line
2 find_package(catkin REQUIRED COMPONENTS
3   roscpp
4   rospy
5   std_msgs
6   message_generation
7 )
```

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 :

```
1 # add_service_files(
2 #   FILES
3 #   Service1.srv
4 #   Service2.srv
5 # )
```

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

```
1 add_service_files(
2   FILES
3   AddTwoInts.srv
4 )
```

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

3.7.3.2

3.7.4 Modifications CMakeLists.txt communes à msg et srv

Si le bloc suivant est toujours commenté :

```
1 # generate_messages(
2 #   DEPENDENCIES
3 #   # std_msgs # Or other packages containing msgs
4 # )
```

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 :

```
1 generate_messages(
2   DEPENDENCIES
3   std_msgs
4 )
```

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

```
1 # In your catkin workspace
2 $ roscd beginner_tutorials
3 $ cd ../..
4 $ catkin_make install
5 $ cd -
```

Chaque fichier *msg* dans le répertoire *msg* va être traduit sous forme de fichier source dans tous les langages 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 ROS-1

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.

4.1.1.1 C++

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

```
1 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 :

```
1 #include <ros/ros.h>
2 #include <std_msgs/String.h>
3
4 #include <iostream>
5
6 /**
7  * This tutorial demonstrates simple sending of messages over the ROS system.
8  */
9 int main(int argc, char **argv)
10 {
11     /**
12      * 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'inclure 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() renvoie 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 SIGINT (Ctrl-C). Si ce signal est envoyé alors ros : :ok() retourne false .

- Plus précisément ros : :ok() renvoie 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 oubli.

```
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.

4.1.1.2 Python

Pour le code python, tous les fichiers sont mis dans le répertoire `scripts` :

```
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.githubusercontent.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'appeler 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 très 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'appeler `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 imprimés sur l'écran, c'est écrit dans le fichier de log de Node, et c'est écrit dans `rosout`. `rosout` est utile pour le débogage : 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 après 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.

4.1.2.1 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

```

1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 /**
5  * This tutorial demonstrates simple receipt of messages over the ROS system.
6  */
7 void chatterCallback(const std_msgs::String::ConstPtr& msg)
8 {
9     ROS_INFO("I heard: [%s]", msg->data.c_str());
10 }
11
12 int main(int argc, char **argv)
13 {
14     /**

```

```

15 * The ros::init() function needs to see argc and argv so that it can perform
16 * any ROS arguments and name remapping that were provided at the command line.
17 * For programmatic remappings you can use a different version of init() which takes
18 * remappings directly, but for most command-line programs, passing argc and argv is
19 * the easiest way to do it. The third argument to init() is the name of the node.
20 *
21 * You must call one of the versions of ros::init() before using any other
22 * part of the ROS system.
23 */
24 ros::init(argc, argv, "listener");

25 /**
26 * NodeHandle is the main access point to communications with the ROS system.
27 * The first NodeHandle constructed will fully initialize this node, and the last
28 * NodeHandle destructed will close down the node.
29 */
30 ros::NodeHandle n;

31 /**
32 * The subscribe() call is how you tell ROS that you want to receive messages
33 * on a given topic. This invokes a call to the ROS
34 * master node, which keeps a registry of who is publishing and who
35 * is subscribing. Messages are passed to a callback function, here
36 * called chatterCallback. subscribe() returns a Subscriber object that you
37 * must hold on to until you want to unsubscribe. When all copies of the Subscriber
38 * object go out of scope, this callback will automatically be unsubscribed from
39 * this topic.
40 *
41 * The second parameter to the subscribe() function is the size of the message
42 * queue. If messages are arriving faster than they are being processed, this
43 * is the number of messages that will be buffered up before beginning to throw
44 * away the oldest ones.
45 */
46 ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback,
47                                 ros::TransportHints().tcpNoDelay());
48
49 /**
50 * ros::spin() will enter a loop, pumping callbacks. With this version, all
51 * callbacks will be called from within this thread (the main one). ros::spin()
52 * will exit when Ctrl-C is pressed, or the node is shutdown by the master.
53 */
54 ros::spin();
55
56 return 0;
57 }
58 }
```

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

```

7 void chatterCallback(const std_msgs::String::ConstPtr& msg)
8 {
9     ROS_INFO("I heard: [%s]", msg->data.c_str());
10 }
```

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,
49                                     ros::TransportHints().tcpNoDelay());
```

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 permettent 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.

4.1.2.2 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 `listener.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 noeud 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 génère un nom unique 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 noeud 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 noeuds Python. 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` :

```
1 cd ~/catkin_ws
2 catkin_make
```

4.1.3 Lancer les nodes

Il suffit ensuite de lancer le `listener` dans un terminal :

```
1 rosrun beginner_tutorials listener.py
```

et le `talker` dans un autre terminal :

```
1 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

4.2.1.1 C++

```

1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3
4 bool add(beginner_tutorials::AddTwoInts::Request &req,
5          beginner_tutorials::AddTwoInts::Response &res)
6 {
7     res.sum = req.a + req.b;
8     ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
9     ROS_INFO("sending back response: [%ld]", (long int)res.sum);
10    return true;
11 }
12
13 int main(int argc, char **argv)
14 {
15     ros::init(argc, argv, "add_two_ints_server");
16     ros::NodeHandle n;
17
18     ros::ServiceServer service = n.advertiseService("add_two_ints", add);
19     ROS_INFO("Ready to add two ints.");
20     ros::spin();
21
22     return 0;
23 }
```

```

1 #include "ros/ros.h"
2 #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**.

```

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

```

1 bool add(beginner_tutorials::AddTwoInts::Request &req,
2          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** et retourne un booléen.

```

1 {
2     res.sum = req.a + req.b;
3     ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
4     ROS_INFO("sending back response: [%ld]", (long int)res.sum);
5     return true;
6 }
```

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.

```

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

Ici le service est crée et publié sur ROS.

4.2.1.2 Python

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

```

1#!/usr/bin/env python
2
3from beginner_tutorials.srv import *
4import rospy
5
6def handle_add_two_ints(req):
7    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
8    return AddTwoIntsResponse(req.a + req.b)
```

```

9 def add_two_ints_server():
10    rospy.init_node('add_two_ints_server')
11    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
12    print "Ready to add two ints."
13    rospy.spin()
14
15
16 if __name__ == "__main__":
17    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

4.2.2.1 C++

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

```

1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 #include <cstdlib>
4
5 int main(int argc, char **argv)
6 {
7     ros::init(argc, argv, "add_two_ints_client");
8     if (argc != 3)
9     {
10         ROS_INFO("usage: add_two_ints_client X Y");
11         return 1;
12     }
13
14     ros::NodeHandle n;
15     ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
16     beginner_tutorials::AddTwoInts srv;
17     srv.request.a = atol(argv[1]);
18     srv.request.b = atol(argv[2]);
19     if (client.call(srv))
20     {
21         ROS_INFO("Sum: %ld", (long int)srv.response.sum);
22     }
23     else
24     {
25         ROS_ERROR("Failed to call service add_two_ints");
26         return 1;
27     }
28
29     return 0;
30 }

```

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 appeler le service.

```

1 beginner_tutorials::AddTwoInts srv;
2 srv.request.a = atol(argv[1]);
3 srv.request.b = atol(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 dans `srv.response` sera valide. Si l'appel n'a pas fonctionné `call()` retourne `false` et la valeur dans `srv.response` ne sera pas valide.

4.2.2.2 Construire les nodes en C++

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

```
1 add_executable(add_two_ints_server src/add_two_ints_server.cpp)
2 target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
3 add_dependencies(add_two_ints_server beginner_tutorials_gencpp)
4
5 add_executable(add_two_ints_client src/add_two_ints_client.cpp)
6 target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
7 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'exécutible dans le chemin à l'installation, il est possible de faire une cible d'installation.

Il faut maintenant lancer **catkin_make** :

```
1 cd ~/catkin_ws
2 catkin_make
```

4.2.2.3 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 :

```
1 #!/usr/bin/env python
2
3 import sys
4 import rospy
5 from beginner_tutorials.srv import *
6
7 def add_two_ints_client(x, y):
8     rospy.wait_for_service('add_two_ints')
9     try:
10         add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
11         resp1 = add_two_ints(x, y)
12         return resp1.sum
13     except rospy.ServiceException, e:
14         print "Service call failed: %s"%e
15
16 def usage():
17     return "%s [x y]"%sys.argv[0]
18
19 if __name__ == "__main__":
20     if len(sys.argv) == 3:
21         x = int(sys.argv[1])
22         y = int(sys.argv[2])
23     else:
24         print usage()
25         sys.exit(1)
26     print "Requesting %s+%s"%(x, y)
27     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 :

```
1 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'appeler `init_node()`. Il faut d'abord attendre le service :

```
8     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.

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

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

```
11    resp1 = add_two_ints(x, y)
12    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 renvoyée 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.

4.2.2.4 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 python correspondants aux messages et services sont générés. Il faut donc aller dans son espace catkin et lancer `catkin_make` :

```
1 cd ~/catkin_ws  
2 catkin_make
```

4.2.3 Lancer les nodes

Il suffit ensuite de lancer le client dans un terminal :

```
1 rosrun beginner_tutorials add_two_ints_client.py
```

et le server dans un autre terminal :

```
1 rosrun beginner_tutorials add_two_ints_server.py
```


Travaux Pratiques

5	TurtleBot3	59
5.1	Démarrage	
5.2	Gazebo	
5.3	Construction de cartes et navigation	
6	Asservissement visuel	63
6.1	Introduction aux tâches	
6.2	Traitement d'image pour le TurtleBot 2	
6.3	OpenCV - Code de démarrage	
6.4	Travail à faire	

5. TurtleBot3

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

5.1 Démarrage

5.1.1 Introduction Modèle du TurtleBot3

Dans ce contexte particulier, le TurtleBot3 sera simulé dans le container du docker.

Il existe plusieurs modèles de TurtleBot3 : burger, waffle, waffle_pi. Dans ce TP nous utiliserons le modèle waffle.

Il faut donc vérifier que votre variable d'environnement TURTLEBOT3_MODEL possède la bonne valeur. Cette variable d'environnement doit être initialisée de la façon suivante :

```
1 export TURTLEBOT3_MODEL=waffle
```

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

5.1.2 Installation

Si vous ne parvenez pas à faire fonctionner gazebo, vous pouvez installer les paquets nécessaires avec :

```
1 sudo apt update
2 sudo apt install ros-foxy-turtlebot3-description ros-foxy-turtlebot3-simulations ros-foxy-turtlebot3-bringup ros-
foxy-turtlebot3-navigation2 ros-foxy-turtlebot3 ros-foxy-xacro ros-foxy-turtlebot3-teleop ros-foxy-turtlebot3-
cartographer
```

Si ROS ne trouve le package turtlebot3_bringup vous pouvez l'installer avec :

```
1 apt install ros-foxy-turtlebot3-bringup
```

5.2 Gazebo

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

```
1 ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Pour obtenir une maison (ATTENTION : il ne faut pas lancer deux fois gazebo) :

```
1 ros2 launch turtlebot3_gazebo turtlebot3_house.launch.py
```

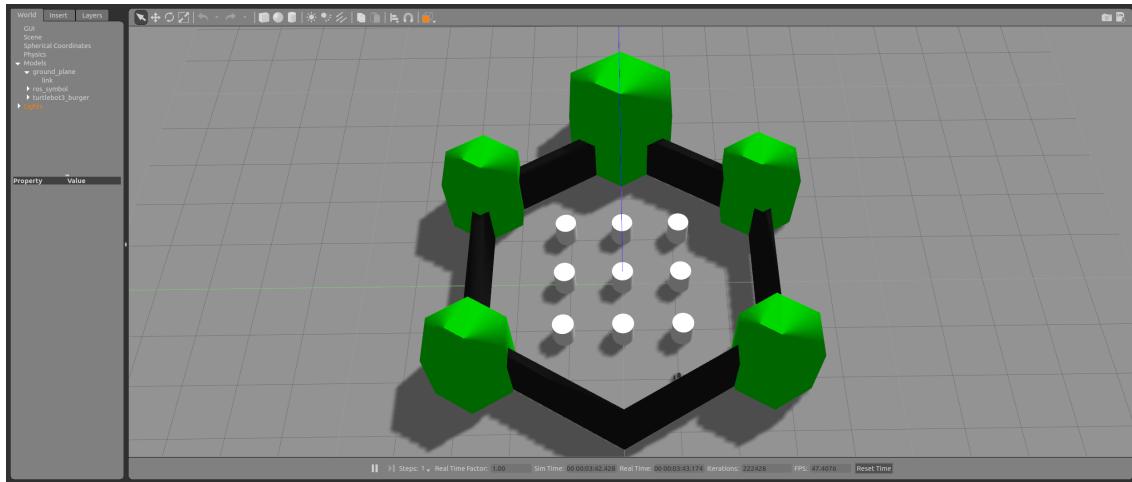


FIGURE 5.1 – Le turtlebot 3 dans une arène hexagonale avec des piliers verts

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
```

5.2.1 Téléopérer le turtlebot

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

```
| ros2 run turtlebot3_teleop teleop_keyboard
```

Dans le terminal vous devez voir apparaître les lignes suivantes :

```
1 Control Your Turtlebot!
2 -----
3 Moving around:
4   w
5   a   s   d
6   x
7
8 w/x : increase/decrease only linear speed by 10%
9 a/d : increase/decrease only angular speed by 10%
10 space key, s : force stop
11
12 CTRL-C to quit
13
14 currently: speed 0.2 turn 1
```

Les touches *w* et *x* permettent d'aller en avant et en arrière. Les touches *a* et *d* permettent de tourner à droite et à gauche.

Le tutorial associé est disponible ici.

5.3 Construction de cartes et navigation

5.3.1 Construction de cartes

Dans cette partie, le robot utilise une ligne de l'image de profondeur fournie par le laser pour construire une carte de l'environnement. Le tutorial associé est disponible ici.

1. En simulation il faut lancer le fichier de launch **gmapping_demo** sur un autre terminal. Remarque importante, cette commande lance un node **rviz** :

```
| # From turtlebot laptop
2 ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=True
```

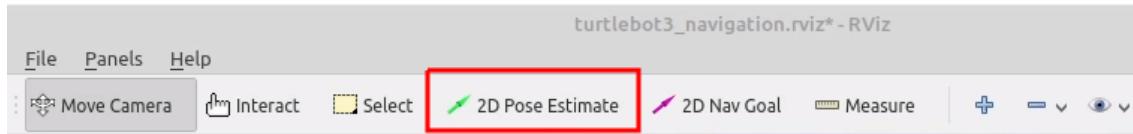


FIGURE 5.2 – Button pour l'estimation de la pose 2D

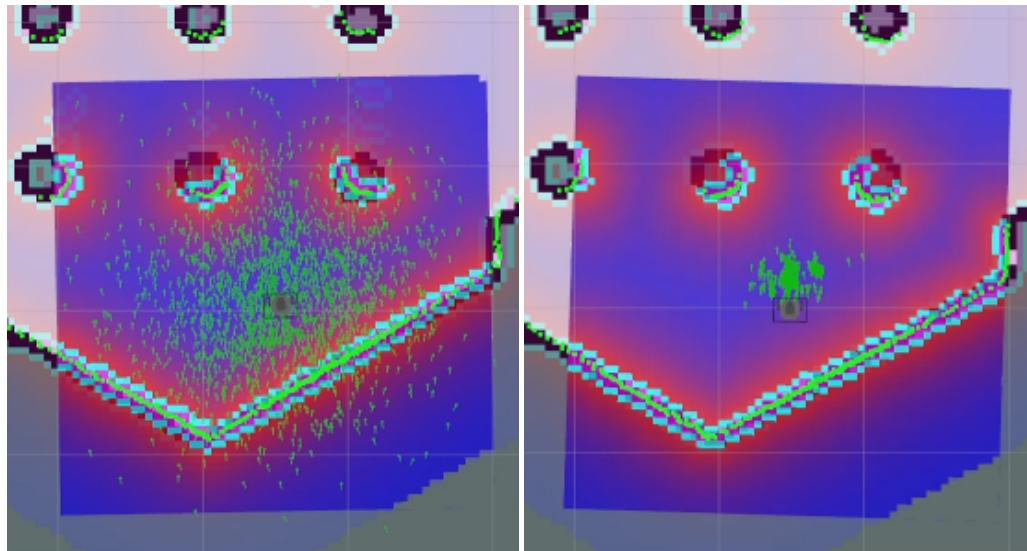


FIGURE 5.3 – AMCL partical

2. Il faut alors explorer l'environnement pour construire la carte avec les nodes de téléopération (voir paragraphe 5.2.1).
3. Une fois que la carte est suffisante, il faut la sauver en utilisant la commande suivante dans un nouveau terminal :

```
1 ros2 run nav2_map_server map_saver_cli -f ~/map
```

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

5.3.2 Navigation

Dans cette partie, le robot utilise une carte de l'environnement. Le tutorial associé est disponible ici.

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

```
1 export TURTLEBOT_MAP_FILE=/tmp/my_map.yaml
2 ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True map:=$HOME/map.yaml
```

5.3.3 RVIZ

5.3.3.1 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 l'algorithme ne converge pas, il est possible de répéter la procédure.

5.3.3.2 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



FIGURE 5.4 – Définir un but de navigation avec le bouton de RVIZ

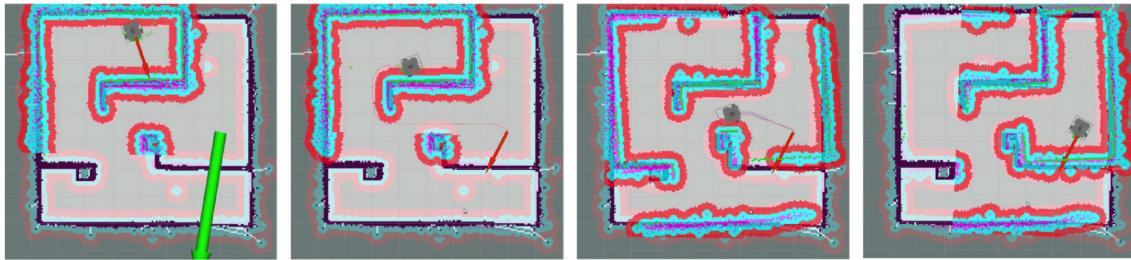


FIGURE 5.5 – Le système trouve un chemin à partir du but de navigation défini

utile de téléopérer le robot lorsque la localisation a été fournie au robot de façon à ce que l'algorithme converge vers une bonne estimation. En effet de nouvelles caractéristiques de l'environnement permettent de mieux discriminer les hypothèses.

5.3.3.3 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 bouton "2D Nav Goal" (voir Fig.5.4).
- Cliquer sur la carte où l'on souhaite voir aller le Turtlebot et pointer la direction que celui-ci doit avoir à la fin. Le système trouve par lui-même le chemin à suivre. Il s'arrête si des obstacles se trouvent sur son chemin. (voir Fig.5.5)

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

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 [3].

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 :

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

$\mathbf{s}(t)$ 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 :

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

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 :

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

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 :

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

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 :

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

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

$$\dot{\mathbf{e}} = -\lambda \mathbf{e} \quad (6.6)$$

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

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

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

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

En l'occurrence, $\mathbf{e} = \hat{\mathbf{s}}(t) - \mathbf{s}^*$ 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 :

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

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.

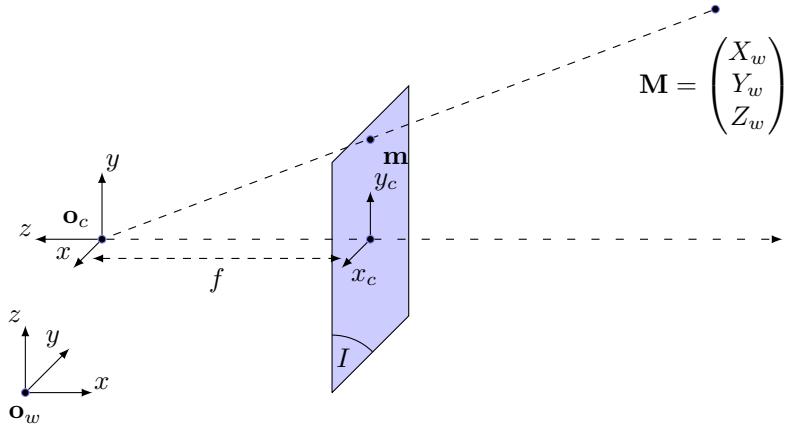


FIGURE 6.2 – \$o_c\$ est l'origine du repère de la caméra, \$o_w\$ l'origine du repère monde. La projection du point \$\mathbf{M}\$ dans le plan image \$I\$ donne \$\mathbf{m}\$.

Si on remarque que :

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

On peut poser :

$$K_f = \begin{pmatrix} f\alpha & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix}, \Pi = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad (6.11)$$

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

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

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 :

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

Finalement le modèle de la caractéristique visuelle mesurée est :

$$\mathbf{s} = \mathbf{m} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{Z_c} \mathbf{K}_f \Pi \mathbf{M}_c \mathbf{M} \quad (6.14)$$

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 :

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

Donc :

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

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

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

Cette relation peut-être écrite de la façon suivante :

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

Maintenant la matrice la relation peut s'écrire :

$$\dot{\mathbf{s}} = \mathbf{L}_s \mathbf{V}_c \quad (6.19)$$

avec

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

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 intrinsèques.

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 :

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

Par dérivation on peut alors calculer :

$$\mathbf{V}_c = \frac{\delta \mathbf{o}_c}{\delta \mathbf{q}} \frac{\delta \mathbf{q}}{\delta s} \quad (6.22)$$

On a donc :

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

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 repère 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

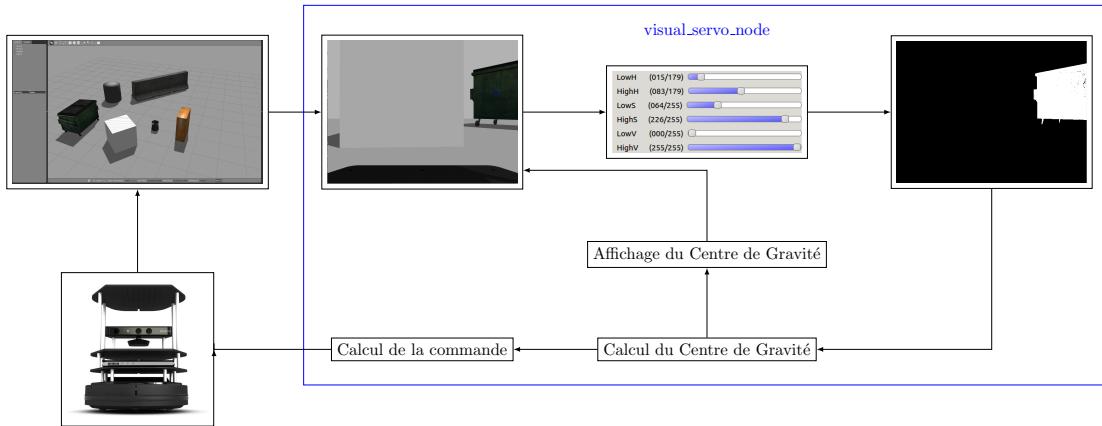


FIGURE 6.3 – Séquence d'opérations de base

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 langage la matrice de l'équation Eq.6.23 est appellée Jacobienne articulaire.

Finalement en considérant le problème spécifié par l'équation Eq.6.8 on peut écrire :

$$\left\{ \begin{array}{l} \min_{\dot{\mathbf{q}}} \quad \|\dot{\mathbf{q}}\| \\ \text{tel que} \quad \mathbf{e} = -\lambda \mathbf{L}_s \mathbf{J}(\mathbf{q}) \dot{\mathbf{q}} \end{array} \right. \quad (6.24)$$

ce qui peut s'écrire algèbriquement :

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

Pour implémenter le contrôleur sur le robot Baxter il faut implémenter la matrice Jacobienne complète 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/image_transport/Tutorials/SubscribingToImages.

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 (lignes 76 – 77). Les pixels qui sont dans les intervalles spécifiés (lignes 42 – 48) par les ascenseurs 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 très 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/WritingPublisherSubscriber.

6.3 OpenCV - Code de démarrage

Le code est disponible sur github. Vous pouvez l'installer dans votre espace de travail de la façon suivante :

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

Si vous êtes sur Noetic :

```
1 git checkout noetic
```

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <cv_bridge/cv_bridge.h>
4 #include <ros/ros.h>
5
6 #include "opencv2/highgui/highgui.hpp"
7 #include "opencv2/imgproc/imgproc.hpp"
8 #include <image_transport/image_transport.h>
9
10 #include <sensor_msgs/Image.h>
11 #include <sensor_msgs/image_encodings.h>
12 #include <geometry_msgs/Twist.h>
13
14 using namespace cv;
15 using namespace std;
16
17 static const std::string OPENCV_WINDOW = "Image window";
18
19 class ImageConverter
20 {
21     ros::NodeHandle nh_;
22     image_transport::ImageTransport it_;
23     image_transport::Subscriber image_sub_;
24     ros::Publisher cmd_vel_pub_;
25
26     int iLowH_, iHighH_;
27     int iLowS_, iHighS_;
28     int iLowV_, iHighV_;
29
30     // Commandes parameters are made with a tuple =
31     // [Parameter name, current value of the parameter, default value of the parameter]
32     typedef std::tuple<std::string, double, double> param_ltype;
33     // Dictionnary of the parameters
34     std::map<std::string, param_ltype> list_params;
35
36     // Center of gravity.
37     double cx_, cy_;
38     unsigned int nb_pts_;
39
40 public:
41     ImageConverter()
42         : it_(nh_), iLowH_(15), iHighH_(83),
43           iLowS_(190), iHighS_(226),
44           iLowV_(0), iHighV_(255),
45           cx_(0.0), cy_(0.0), nb_pts_(0)
46     {
47         // Creates subscriber to get the input video feed
```

```

48
49
50     // Window to show the image
51     cv::namedWindow(OPENCV_WINDOW);
52     // Window named "Control" to display sliders
53     cv::namedWindow("Control", WINDOW_AUTOSIZE);
54
55     // Create trackbars in "Control" window
56     cv::createTrackbar("LowH", "Control", &iLowH_, 179); //Hue (0 - 179)
57     cv::createTrackbar("HighH", "Control", &iHighH_, 179);
58
59     cv::createTrackbar("LowS", "Control", &iLowS_, 255); //Saturation (0 - 255)
60     cv::createTrackbar("HighS", "Control", &iHighS_, 255);
61     cv::createTrackbar("LowV", "Control", &iLowV_, 255); //Value (0 - 255)
62     cv::createTrackbar("HighV", "Control", &iHighV_, 255);
63
64     // Create publisher to send to control
65
66
67     list_params["Kp"] = std::make_tuple(std::string("/visual_servo/Kp"), 0.001, 0.001);
68     list_params["dx_clamp"] = std::make_tuple(std::string("/visual_servo/dx_clamp"), 10.0, 10.0);
69     list_params["wz_min"] = std::make_tuple(std::string("/visual_servo/wz_min"), 0.01, 0.01);
70     list_params["wz_max"] = std::make_tuple(std::string("/visual_servo/wz_max"), 1.7, 1.7);
71
72     initParameters();
73 }
74
75 void initParameters()
76 {
77     // For each parameters
78     for (auto it = list_params.begin(); it != list_params.end(); ++it)
79     {
80         // If the parameter is already set
81         if (nh_.hasParam(std::get<0>(it->second)))
82             // Read the current value.
83             nh_.getParam(std::get<0>(it->second), std::get<1>(it->second));
84         else
85             // Set the default value.
86             nh_.setParam(std::get<0>(it->second), std::get<1>(it->second));
87     }
88 }
89
90 ~ImageConverter()
91 {
92     cv::destroyWindow(OPENCV_WINDOW);
93 }
94
95 double computeCmdZ(double DeltaX)
96 {
97     double Kp = std::get<1>(list_params["Kp"]);
98     double DeltaX_Clamp = std::get<1>(list_params["dx_clamp"]);
99     double wz_min = std::get<1>(list_params["wz_min"]);
100    double wz_max = std::get<1>(list_params["wz_max"]);
101
102    // Compute simple linear relationship.
103    double cmdz = -Kp*(DeltaX);
104
105    ROS_INFO_STREAM("Kp: " << Kp);
106    ROS_INFO_STREAM("DeltaX: " << DeltaX);
107    ROS_INFO_STREAM("cmdz: " << cmdz);
108    // Lowest saturation
109    double signe_cmd = cmdz > 0.0 ? 1.0 : -1.0;
110
111    // Control saturation
112
113    // Set control to zero if the robot is near the target.
114
115    // Minimal value if the robot is not near enough but needs to move
116
117    return cmdz;
118 }
119
120 void processImages(const sensor_msgs::ImageConstPtr& msg,
121                     cv_bridge::CvImagePtr & cv_ptr,
122                     Mat & imgThresholded)
123 {
124     // From ros image to opencv
125     try
126     {
127         cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::RGB8);
128     }
129     catch (cv_bridge::Exception& e)
130     {
131         ROS_ERROR("cv_bridge exception: %s", e.what());
132     }
133 }
```

```

132     return;
133 }
134
135 Mat imgHSV;
136 // From RGB to HSV
137 cvtColor(cv_ptr->image, imgHSV, COLOR_RGB2HSV);
138
139 // Analysis object and provides a Thresholded image
140 inRange(imgHSV, Scalar(iLowH_, iLowS_, iLowV_),
141         Scalar(iHighH_, iHighS_, iHighV_), imgThresholded);
142 }
143
144 void displayImages(cv_bridge::CvImagePtr & cv_ptr,
145                     Mat & imgThresholded)
146 {
147     if (nb_pts_>0)
148     {
149         // Draw the CoG in the image
150         cv::circle(cv_ptr->image, cv::Point((int)cx_,(int)cy_), 10, CV_RGB(0,0,255));
151     }
152
153     // Show the thresholded image
154     imshow("Thresholded Image", imgThresholded);
155
156     // Update GUI Window
157     cv::imshow(OPENCV_WINDOW, cv_ptr->image);
158 }
159
160 void computeCoG(Mat & imgThresholded)
161 {
162     // Compute Center-Of-Gravity of the object.
163     cx_=0.0,cy_=0.0;
164     nb_pts_=0;
165     unsigned long int idx=0;
166     for(unsigned int j=0;j<imgThresholded.rows;j++)
167     {
168         for(unsigned int i=0;i<imgThresholded.cols;i++)
169         {
170             if (imgThresholded.data[idx]>124)
171             {
172                 cx_+=(double)i;
173                 cy_+=(double)j;
174                 nb_pts_++;
175             }
176             idx+=1;
177         }
178     }
179     if (nb_pts_>0)
180     {
181         cx_ = cx_/(double)nb_pts_;
182         cy_ = cy_/(double)nb_pts_;
183     }
184 }
185
186 void imageCb(const sensor_msgs::ImageConstPtr& imgMsg)
187 {
188
189     // 1 - Update each parameters by iterating over list_params.
190
191     // 2- Image processing
192     // Convert sensors_msg::Image to opencv image
193     cv_bridge::CvImagePtr cv_ptr;
194     // Result image
195     Mat imgThresholded;
196     processImages(imgMsg, cv_ptr, imgThresholded);
197
198     // Compute the Center-of-Gravity.
199     computeCoG(imgThresholded);
200
201     // 3 - Compute Control for base
202
203     // 4 - Send Control value to base
204
205     displayImages(cv_ptr, imgThresholded);
206
207     // 5 - Opencv update
208     cv::waitKey(3);
209 }
210
211 };
212
213
214 int main( int argc, char** argv )
215 {

```

```

216     ros::init(argc,argv,"tp_ros_visual_servo");
217     ImageConverter ic;
218     ros::spin();
219 }
```

6.4 Travail à faire

6.4.1 Turtlebot 3

6.4.1.1 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 dépendances vers d'autres paquets en modifiant les fichiers **package.xml** (blocks <build_depend> et <run_depend>) et **CMakeLists.txt**.

6.4.1.2 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**.

6.4.1.3 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 :

```

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

6.4.1.4 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 Tiago.
2. Créer le publieur permettant d'envoyer une commande en vitesse à la base mobile.
3. Calculer $|\Delta_x|$ la différence entre le milieu de l'image c_i 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.6.4.1.4 illustre la forme de la commande qui dépend de la variation en pixels. Lorsque $|\Delta_x|$ est inférieure à une certaine 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 droite et donc ω_z doit être négatif.

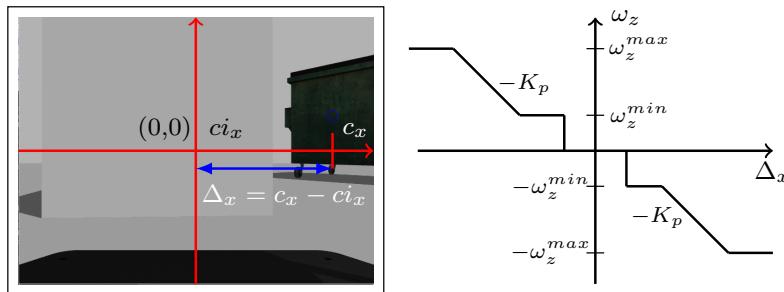


FIGURE 6.4 – Commande en vitesse à partir de la variation en pixels.

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 accès 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.

6.4.1.5 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 récupérer l'image de profondeur.
4. Développer la méthode permettant d'extraire l'information de distance de l'image de profondeur.

Modèles et Simulation

7	Universal Robot Description Format (URDF)	75
7.1	Capteurs - Sensors	
7.2	Corps - Link	
7.3	Transmission	
7.4	Joint	
7.5	Gazebo	
7.6	model_state	
7.7	model	
8	Simulation	83
8.1	Introduction	
8.2	Algorithme général d'un simulateur système	
8.3	Algorithme général d'un moteur dynamique	
9	Gazebo - Ignition	85
9.1	Introduction	
9.2	SDF format	
9.3	Plugin	
9.4	Modèle de robot en SDF	

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 :

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

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

```
8 <sensor name="my_ray_sensor" update_rate="20">
9   <parent link="optical_frame_link_name"/>
10  <origin xyz="0 0 0" rpy="0 0 0"/>
11  <ray>
12    <horizontal samples="100" resolution="1" min_angle="-1.5708" max_angle="1.5708"/>
13    <vertical samples="1" resolution="1" min_angle="0" max_angle="0"/>
14  </ray>
15 </sensor>
```

Cet élément a les attributs suivants :

- **name** : (requis) (string)
Le nom du capteur
- **update_rate** : (optional) (float) (Hz)
La fréquence à 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 défaut 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**.

7.1.2 Limitations

Peu de capteur peuvent exprimés dans le format URDF. C'est une faiblesse particulièrement problématique pour des programmes qui en ont besoin. C'est le cas pour la simulation et le contrôle. Cela démontre également que dans la pratique la plupart des développements utilisent très peu les simulations des capteurs ou le font à travers d'autres formats. C'est notamment le cas pour gazebo qui utilise le format SDF.

7.2 Corps - Link

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

```

16 <link name="my_link">
17   <inertial>
18     <origin xyz="0 0 0.5" rpy="0 0 0"/>
19     <mass value="1"/>
20     <inertia ixx="100" ixy="0" ixz="0" iyy="100" iyz="0" izz="100" />
21   </inertial>
22
23   <visual>
24     <origin xyz="0 0 0" rpy="0 0 0" />
25     <geometry>
26       <box size="1 1 1" />
27     </geometry>
28     <material name="Cyan">
29       <color rgba="0 1.0 1.0 1.0"/>
30     </material>
31   </visual>
32
33   <collision>
34     <origin xyz="0 0 0" rpy="0 0 0"/>
35     <geometry>
36       <cylinder radius="1" length="0.5"/>
37     </geometry>
38   </collision>
39 </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 symétrique, seuls les 6 éléments diagonaux supérieurs sont donnés ici en utilisant les attributs **ixx,ixy,ixz,iyy, iyz,izz**.
- **<visual>** (*optionel*) Les propriétés visuelles du corps. Cet élément spécifie la forme de l'objet (boîte, 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 particulièrement 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 éléments suivants :
 - **box** L'attribut **size** contiens la taille (longueur, largeur, 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>** (*optionnel*) 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 élément link il est possible de référencer le matériel par son nom.
 - **name** (*optionnel*) Spécifie le nom du matériel.
 - **<color>** (*optionnel*)
 - **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>** (*optionnel*)
 - La texture d'un matériel est spécifié par un fichier nommé **filename**.
- **<collision>** (*optionnel*) Les propriétés de collision d'un corps. Elles peuvent être différentes des informations visuelles. Par exemple, on utilise souvent des modèles simplifiés 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** (*optionnel*)
 - Spécifie un nom pour une partie géométrique du corps. Ceci est utile pour faire référence à des parties spécifiques de la géométrie d'un corps.
 - **<origin>** (*optionnel* : 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** (*optionnel* : par défaut à zéro) Représente la position de l'élément de collision.
 - **rpy** (*optionnel* : 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 **visual** 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 particulièrement 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>
46 </visual>
47 <collision>
48   <origin rpy="0 0 0" xyz="-0.065 0 0.0"/>
49   <geometry>
50     <mesh filename="package://robot_description/meshes/base_link_simple.DAE"/>
51   </geometry>
52 </collision>
53 <collision_checking>
54   <origin rpy="0 0 0" xyz="-0.065 0 0.0"/>
55   <geometry>
56     <cylinder length="0.7" radius="0.27"/>
57   </geometry>
58 </collision_checking>
59 <inertial>
60   ...
61 </inertial>
62 </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 :

```

63 <transmission name="simple_trans">
64   <type>transmission_interface/SimpleTransmission</type>
65   <joint name="foo_joint">
66     <hardwareInterface>EffortJointInterface</hardwareInterface>
67   </joint>
68   <actuator name="foo_motor">
69     <mechanicalReduction>50</mechanicalReduction>
70     <hardwareInterface>EffortJointInterface</hardwareInterface>
71   </actuator>
72 </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 Éléments de transmission

La transmission a les éléments suivants :

- **<type>** (une occurrence)
Spécifie le type de transmission.
- **<joint>** (une ou plusieurs occurrences)
Un joint auquel la transmission est connectée. Le joint est spécifié par l'attribut **name**, et les sous-éléments suivants :
 - **<hardwareInterface>** (une ou plusieurs occurrences)
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 occurrences)
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 occurrences) 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 Élément **<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 :

```

73 <joint name="my_joint" type="floating">
74   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
75   <parent link="link1"/>
76   <child link="link2"/>
77
78   <calibration rising="0.0"/>
79   <dynamics damping="0.0" friction="0.0"/>
80   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
81   <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_limit="0.5" />
82 </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 degrés 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 Éléments 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 par $(1 - dampingFactor)$.
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 <i>max_contacts</i> définit dans la section physics .
laserRetro	double	Valeur d'intensité renvoyée par le capteur laser.

7.5.2 Éléments 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 cfmDamper est obsolète et doit être changé en implicitSpringDamper .
fudgeFactor	double	Met à l'échelle l'excès 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 Élément <model_state>

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

```

83 <model_state model="pr2" time_stamp="0.1">
84   <joint_state joint="r_shoulder_pan_joint" position="0" velocity="0" effort="0"/>
85   <joint_state joint="r_shoulder_lift_joint" position="0" velocity="0" effort="0"/>
86 </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 réfère.
 - **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 spécification 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 spécifications 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 ensemble. La description typique d'un robot ressemble donc à :

```

87 <robot name="pr2">
88   <link> ... </link>
89   <link> ... </link>
90   <link> ... </link>
91
92   <joint> .... </joint>
93   <joint> .... </joint>
94   <joint> .... </joint>
95 </robot>

```

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

8. Simulation

8.1 Introduction

La simulation dans le contexte de la robotique permet en général de vérifier qu'un logiciel ou plus généralement une application distribuée, fait ce qui a été spécifié [7]. Le simulateur dans ce cas va remplacer le monde extérieur, et plus précisément il va calculer les actions des actionneurs et les signaux que devraient percevoir les capteurs. Il faut donc pour cela un simulateur de la dynamique du monde extérieur et un simulateur des phénomènes physique observés par les capteurs. La complexité des simulateurs vient du fait qu'il y a toujours une différence entre le monde réel et le monde simulé. Suivant le degré de réalité souhaité, il peut être nécessaire d'utiliser des algorithmes différents. Par exemple on ne simule pas de la même façon un robot qui évolue dans l'eau, un robot à roues, ou encore un drone.

Il existe cependant une confusion entre des architectures logicielles permettant de simuler

- une application logicielle complète. Elle permet d'utiliser le même logiciel en simulation et dans la réalité.
- la physique d'interaction entre le robot et son environnement

On appelle en général le premier un simulateur système tandis que le deuxième est appelé le moteur dynamique du simulateur. Gazebo et VREP rentre dans la première catégorie. ODE, Bullet et Newton sont dans la deuxième catégorie. Il arrive également que pour contrôler un robot il soit nécessaire de simuler la dynamique entre le robot et son environnement. Il s'agit encore d'une autre catégorie où le rôle essentiel du simulateur est de pouvoir fournir à un instant donné une commande à envoyer aux moteurs. Les formulations mathématiques peuvent être différentes car l'application est différente. Elles peuvent néanmoins partager un grand nombre de points communs.

On peut synthétiser la différence entre les simulateurs systèmes dont les buts sont :

- de simuler les capteurs, les actionneurs et l'environnement,
- pouvoir aller de la simulation au robot en minimisant les modifications,
- représenter de façon précise la réalité (grâce à un simulateur dynamique),
- maintenir la stabilité du calcul

tandis que les simulateurs de contrôle sont généralement prévus pour :

- le temps réel,
- capturer la dynamique principale du robot,
- être intégré dans la boucle de contrôle

Exemples de simulateurs systèmes : Gazebo, Stage, Morse, OpenHRP. Exemples de simulateurs de contrôle : MuJoCo, RBDL, Pinocchio, Robotran.

8.2 Algorithme général d'un simulateur système

Data: Modèle du robot, modèle du monde, état initial du robot et du monde

Résultat: Etat du monde, sortie des capteurs

Initialisation;

while La simulation n'est pas finie **do**

for chaque contrôleur c **do**

 | Lire les entrées capteurs pour le contrôleur c . Calculer la commande du contrôleur c .

end

 Calculer un pas de la dynamique. Récupérer l'état du monde. Afficher l'état du monde. **for** chaque contrôleur c **do**

 | Appliquer la commande du contrôleur c .

end

end

8.3 Algorithme général d'un moteur dynamique

Il existe de nombreuses façons de poser le problème [2] mais on trouve l'algorithme général suivant :

Data: Modèle du robot et du monde, état courant du robot et du monde, paire de collisions, Force externes

Résultat: Position du robot et du monde

En utilisant la commande du contrôleur;

Calcul de la position des corps par intégration (par exemple en utilisant Runge-Kutta 4-5) de la dynamique des corps;

Détection des collisions;

Calcul des forces de contacts;

(voir le rigid body pipeline 1 dans [5]).

Par exemple le simulateur système Gazebo et Morse utilisent tous les deux ODE pour la simulation. Mais Gazebo peut également utiliser 3 autres moteurs dynamiques : Bullet [5], DART [8] et Simbody [9].

8.3.1 Problème 1 : Réalité

La simulation de l'impact est très souvent difficile à effectuer de façon précise. Par exemple dans [1] la simulation du franchissement d'obstacles d'un robot humanoïde avec OpenHRP 2.x a fourni un impact de 800 N. Cependant sur le robot réel l'impact était de 1200 N qui aurait pu mener à la perte des capteurs de force localisés dans les chevilles du robot. La version d'OpenHRP 3.x a permis une meilleure simulation dynamique permettant de mieux discriminer les mouvements menant à des impacts trop violents des mouvements admissibles. La précision de la simulation nécessite cependant plus de calcul et donc prend plus de temps.

D'une manière générale il est difficile d'avoir un haut niveau de fiabilité en simulation. La compliance des actionneurs et des structures mécaniques est souvent difficile à réaliser car elle s'effectue à des fréquences nécessitant une très grande précision de simulation. La simulation de la vision est également souvent nécessaire afin de pouvoir simuler des algorithmes de traitement d'images.

8.3.2 Problème 2 : Flexibilité logicielle

Du point de vue logiciel on souhaiterait pouvoir passer de la simulation système du robot sans aucune modification ou simplement à travers de fichiers de configuration. Il est également souhaitable de pouvoir changer des parties du simulateur par exemple le moteur dynamique, afin de pouvoir faire des simulations dans des modes différents. Pour cela il est nécessaire d'avoir un middleware suffisamment puissant et pouvoir simuler les capteurs.

9. Gazebo - Ignition

9.1 Introduction

Gazebo est le simulateur historique sur ROS. La dernière version est Gazebo 11 est d'après <http://gazebosim.org/#features> la fin de vie de Gazebo est prévue pour le 29 Janvier 2025. Le successeur de Gazebo est Ignition. Gazebo est développé sous licence Apache 2.0. Gazebo est un simulateur système. Il permet d'interfacer les logiciels développés pour le robot avec le simulateur. Il est possible d'étendre les capacités du simulateur grâce à un système de plugins.

Pour la suite on peut cloner le package suivant dans un workspace **gazebo_ws** :

```
1 mkdir -p $HOME/gazebo_ws/src
2 cd $HOME/gazebo_ws/src
3 git clone https://github.com/olivier-stasse/gazebo_ros_demos.git
```

Pour le construire :

```
1 cd $HOME/gazebo_ws/
2 colcon build --packages-select
```

9.2 SDF format

Le format SDF est formalisé ici : <http://sdformat.org/>, il permet de décrire les différents éléments nécessaires à la simulation : le modèle des éléments du monde, le robot et les modèles de contact à utiliser.

Un exemple extrêmement simple est donné ici :

```
1 <?xml version="1.0" ?>
2 <sdf version="1.4">
3   <world name="default">
4     <include>
5       <uri>model://ground_plane</uri>
6     </include>
7
8     <include>
9       <uri>model://sun</uri>
10    </include>
11
12    <!-- reference to your plugin -->
13    <plugin name="gazebo_tutorials" filename="libgazebo_tutorials.so"/>
14  </world>
15</sdf>
```

Ce fichier XML est un exemple très simple. On peut spécifier le format grâce à la ligne :

```
1 <sdf version="1.4">
```

On décrit le monde entre les balises **world**.

On peut ensuite mettre des modèles :

```
1 <include>
2   <uri>model://ground_plane</uri>
3 </include>
4
5 <include>
6   <uri>model://sun</uri>
7 </include>
```

Ici le sol avec **ground_plane** et le soleil comme lumière (**sun**).

Enfin on peut spécifier un plugin nommé **gazebo_tutorials** inclus dans la bibliothèque **libgazebo_tutorials.so** grâce à :

```
1 <plugin name="gazebo_tutorials" filename="libgazebo_tutorials.so"/>
```

9.2.1 Quand utiliser le format SDF ou le format URDF ?

Lorsque l'on utilise ROS dans une application robotique une bonne pratique est de ne pas faire de fichier SDF spécifique pour son robot. En effet, gazebo est capable de lire un fichier URDF et d'en faire un fichier SDF. Les plugins de gazebo à utiliser peuvent être indiqués par des balises gazebo. Celles-ci seront ignorées par les autres logiciels, MoveIt par exemple, qui ne les utilise pas.

Afin de pouvoir contrôler le robot à la fois en simulation et en contrôle il est généralement conseillé d'utiliser ros-control. Ce projet offre une abstraction du robot qui permet de tester le même logiciel en simulation et sur le vrai robot.

Cependant il nécessite un démarrage asynchrone qui rend l'utilisation plus compliquée et faire perdre du temps.

Enfin si le format URDF permet de modéliser un robot il ne permet pas de modéliser un monde et de fournir les informations nécessaires à sa simulation. Donc pour modéliser un environnement il n'est pas possible d'utiliser le format SDF.

Si votre robot n'utilise pas ros-control, il peut-être également plus avantageux d'utiliser directement le format SDF.

9.3 Plugin

9.3.1 Plugin Gazebo

Il existe 3 sortes de plugin sous Gazebo :

- Les plugin Sensor qui peuvent être mis dans des balises XML **<sensor>**
- Les plugin Model qui peuvent être mis dans des balises XML **<model>** mais pas **<sensor>**
- Les plugin World qui peuvent être mis dans des balises XML **<world>** mais pas **<model>** ni **<sensor>**

Il existe un certains nombre de plugins déjà disponibles à cette adresse : https://github.com/ros-simulation/gazebo_ros_pkgs

9.3.2 Plugin : exemple de base

Ce plugin permet de faire le lien entre Gazebo et ROS. Il s'agit d'un plugin **physics : :World**. Il est également possible de faire un plugin de type **sensors : :Sensor** ou de type **rendering : :Visual**.

Le plugin suivant est très simple.

```
1 #include <gazebo/common/Plugin.hh>
2 #include <rclcpp/rclcpp.hpp>
3 #include <gazebo_ros/node.hpp>
4
5 namespace gazebo
6 {
7 class WorldPluginTutorial : public WorldPlugin
8 {
9 public:
10   WorldPluginTutorial() : WorldPlugin()
11   {
12   }
13 }
```

```

14 void Load(physics::WorldPtr _world, sdf::ElementPtr _sdf)
15 {
16     gazebo_ros::Node::SharedPtr ros_node = gazebo_ros::Node::Get(_sdf);
17
18     // Make sure the ROS node for Gazebo has already been initialized
19     if (!rclcpp::ok())
20     {
21         RCLCPP_FATAL(ros_node->get_logger(),
22                     "A ROS node for Gazebo has not been initialized, unable to load plugin. ");
23         return;
24     }
25
26     RCLCPP_INFO(ros_node->get_logger(), "Hello World!");
27
28 }
29
30 };
31 GZ_REGISTER_WORLD_PLUGIN(WorldPluginTutorial)
32 }
```

Il suffit d'hériter de la classe de base `WorldPlugin`. Puis à part le constructeur qui permet de récupérer un pointeur vers le node créé dans ce contexte, de vérifier que ros a bien été initialisé et qui affiche "Hello World!". Il suffit ensuite d'implémenter la méthode **Load**. La dernière opération à faire est d'utiliser une macro pour créer la factory du plugin.

Le fichier `CMakeLists.txt` est le suivant :

```

1 cmake_minimum_required(VERSION 3.5)
2 project(gazebo_tutorials)
3
4 # Default to C++14
5 if(NOT CMAKE_CXX_STANDARD)
6     set(CMAKE_CXX_STANDARD 14)
7 endif()
8
9 if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
10    # we dont use add_compile_options with pedantic in message packages
11    # because the Python C extensions dont comply with it
12    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -Wpedantic -Wno-inconsistent-missing-override")
13 endif()
14
15 # find dependencies
16 find_package(rclcpp REQUIRED)
17 find_package(ament_cmake REQUIRED)
18
19
20 # Depend on system install of Gazebo
21 find_package(gazebo REQUIRED)
22 find_package(gazebo_ros REQUIRED)
23
24 #warning: variable CMAKE_CXX_FLAGS is modified
25 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${GAZEBO_CXX_FLAGS}")
26
27 #warning: use of link_directories() is strongly discouraged
28 MESSAGE(STATUS "GAZEBO_INSTALL_LIB_DIR: ${GAZEBO_INSTALL_LIB_DIR}")
29 link_directories(${gazebo_dev_LIBRARY_DIRS})
30 link_directories(${GAZEBO_INSTALL_LIB_DIR})
31 include_directories(${Boost_INCLUDE_DIR} ${GAZEBO_INCLUDE_DIRS})
32
33
34 add_library(${PROJECT_NAME} SHARED src/simple_world_plugin.cpp)
35 target_include_directories(${PROJECT_NAME} PUBLIC
36     ${<BUILD_INTERFACE>${CMAKE_CURRENT_SOURCE_DIR}/include}
37     ${<INSTALL_INTERFACE>include})
38
39 ament_target_dependencies(
40     ${PROJECT_NAME}
41     "rclcpp"
42     "gazebo_ros"
43 )
44 ament_export_libraries(${PROJECT_NAME})
45
46 link_directories(${GAZEBO_INSTALL_LIB_DIR})
47
48 install(TARGETS ${PROJECT_NAME}
49     EXPORT export_${PROJECT_NAME}
50     ARCHIVE DESTINATION lib/${PROJECT_NAME}
51     LIBRARY DESTINATION lib/${PROJECT_NAME}
52     RUNTIME DESTINATION lib/${PROJECT_NAME}
53 )
54 install(DIRECTORY launch
```

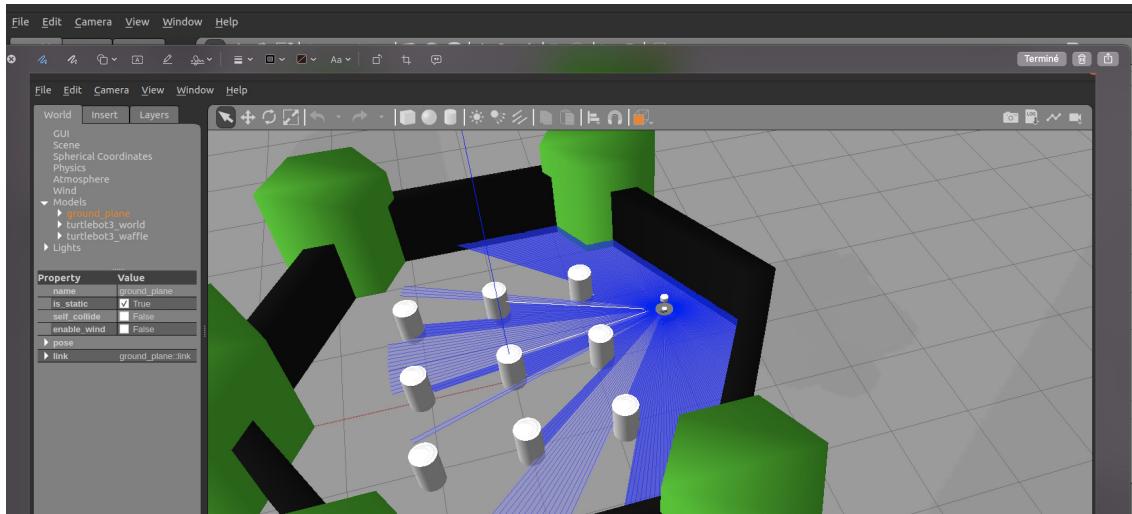


FIGURE 9.1 – Robot turtlebot3 modifié avec un capteur solaire

```

55 DESTINATION share/${PROJECT_NAME})
56
57 install(DIRECTORY worlds
58   DESTINATION share/${PROJECT_NAME})
59
60 ament_package()

```

9.4 Modèle de robot en SDF

9.4.1 Introduction

Dans cet exemple le but est de transformer le modèle du turtlebot3 waffle et lui rajouter un capteur. Il s'agit d'un capteur qui détecte la direction du soleil. Pour cela on utilise un plugin qui permet de calculer la direction du soleil en supposant qu'une lumière dans l'environnement s'appelle soleil.

9.4.1.1 Préparation

Le code est disponible sur github. Vous pouvez l'installer dans votre espace de travail de la façon suivante :

```

1 cd ~/dev_ws/src
2 git clone https://github.com/olivier-stasse/gazebo_ros_demos

```

Si vous êtes sur Foxy :

```

1 git checkout -b foxy -t origin/foxy

```

Cette commande permet de créer une branche locale nommée foxy qui suit la branche distance foxy qui se trouve sur le serveur origin.

Pour compiler le package :

```

1 cd ~/dev_ws/
2 colcon build --packages-select gazebo_tutorials

```

Il ensuite modifier la variable d'environnement **LD_LIBRARY_PATH** de la façon suivante :

```

1 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${HOME}/dev_ws/install/gazebo_tutorials/lib/gazebo_tutorials

```

Pour lancer la simulation :

```

1 ros2 launch gazebo_tutorials turtlebot3.launch.py

```

Le résultat est représenté dans la figure Fig.9.1. On peut y voir le capteur solaire représenté par un cylindre au dessus du robot.

9.4.1.2 Organization

Le paquet **gazebo_tutorials** possède deux répertoires nommés **models** et **worlds**. Le répertoire **models** contient deux répertoires : l'un se nomme **sunloin** et l'autre **turtlebot_waffle**. Le modèle **sunloin** est une version modifiée du modèle **sun** fourni par défaut par Gazebo. Le modèle **sunloin** place une lumière à $256e^9\ km$ au lieu de $10m$ à la verticale du sol. Le répertoire **turtlebot_waffle** contient le modèle du robot Turtlebot3-Waffle qui a été modifié pour inclure un capteur supplémentaire par rapport au modèle initial.

9.4.2 Modèle du soleil

Ce modèle est défini dans le répertoire **gazebo_tutorials/models/sunloin**. Le soleil est donc défini de la façon suivante :

```

1 <?xml version="1.0" ?>
2 <sdf version="1.5">
3   <light type="directional" name="sunloin">
4     <cast_shadows>true</cast_shadows>
5     <pose>0 0 256e9 0 0 0</pose>
6     <diffuse>0.8 0.8 0.8 1</diffuse>
7     <specular>0.2 0.2 0.2 1</specular>
8     <attenuation>
9       <range>1000</range>
10      <constant>0.9</constant>
11      <linear>0.01</linear>
12      <quadratic>0.001</quadratic>
13    </attenuation>
14    <direction>-0.5 0.1 -0.9</direction>
15  </light>
16 </sdf>
```

Après la spécification du xml et de la version de la balise sdf, la lumière est définie grâce à la balise **light**. Elle est de type directionnel et son nom est **sunloin**.

La position de la lumière est spécifiée par la balise **pose** sous le format $(x, y, z, \theta_x, \theta_y, \theta_z)$. Ici le champ a été modifié par rapport au modèle par défaut pour être plus proche de la distance réelle. Les deux champs **diffuse** et **specular** correspondent à :

- **diffuse** : La couleur que renvoie un objet sous une lumière blanche. Cette couleur est perçue comme la couleur réelle de l'objet plutôt que la réflexion de la lumière.
- **specular** : La couleur que renvoie la lumière suite à sa réflexion sur un objet. Elle dépend de la direction (balise **direction**) de la lumière, de la pose de la caméra, et de la normale de la surface de l'objet.

L'atténuation de la lumière est définie par la balise **attenuation** par les champs suivants :

- range : La distance sur laquelle l'atténuation s'effectue
- constant : La constante du facteur d'atténuation. 1.0 signifie qu'elle ne s'atténue jamais, 0.0 qu'elle est complètement atténuée.
- linear : Le facteur d'atténuation linéaire de la lumière. 1.0 signifie que l'atténuation s'effectue de façon homogène sur la distance.
- quadratic : Le facteur d'atténuation quadratique : ajoute une courbure à l'atténuation.

9.4.3 Modèle du monde pour notre robot

Le fichier **worlds/waffle/model.sdf** décrit le monde dans lequel le robot **turtlebot3_waffle** évolue.

```

1 <?xml version="1.0"?
2 <sdf version="1.6">
3   <world name="default">
4
5     <include>
6       <uri>model://ground_plane</uri>
7     </include>
8
9     <include>
10      <uri>model://sunloin</uri>
11    </include>
12
13    <scene>
14      <shadows>false</shadows>
15    </scene>
16
17    <gui fullscreen='0'>
18      <camera name='user_camera'>
19        <pose frame='>0.319654 -0.235002 9.29441 0 1.5138 0.009599</pose>
```

```

20      <view_controller>orbit</view_controller>
21      <projection_type>perspective</projection_type>
22    </camera>
23  </gui>
24
25  <physics type="ode">
26    <real_time_update_rate>1000.0</real_time_update_rate>
27    <max_step_size>0.001</max_step_size>
28    <real_time_factor>1</real_time_factor>
29  </ode>
30    <solver>
31      <type>quick</type>
32      <iters>150</iters>
33      <precon_iters>0</precon_iters>
34      <sor>1.400000</sor>
35      <use_dynamic_moi_rescaling>1</use_dynamic_moi_rescaling>
36    </solver>
37    <constraints>
38      <cfm>0.00001</cfm>
39      <erp>0.2</erp>
40      <contact_max_correcting_vel>2000.000000</contact_max_correcting_vel>
41      <contact_surface_layer>0.01000</contact_surface_layer>
42    </constraints>
43  </ode>
44 </physics>
45
46 <model name="turtlebot3_world">
47   <static>1</static>
48   <include>
49     <uri>model://turtlebot3_world</uri>
50   </include>
51 </model>
52
53   <include>
54     <pose>-2.0 -0.5 0.01 0.0 0.0 0.0</pose>
55     <uri>model://turtlebot3_waffle</uri>
56   </include>
57
58 </world>
59 </sdf>

```

Le monde utilisé posséde donc un sol (ligne 6) et le soleil décrit précédemment (ligne 10). La section comprise entre les balises **gui** spécifie la caméra de l'utilisateur avec sa pose, son type de vue et le type de projection 3D. Elle correspond aux lignes (17-23).

Les paramètres de la simulation de la physique sont spécifiés entre les balises **physics**. On peut spécifier la vitesse de mise à jour temps réelle, la taille maximale d'une étape et le facteur temps réel.

La simulation dynamique est réalisée par ODE, et ses paramètres sont spécifiés dans la balise **ode**. Le solveur spécifié ici essaie de résoudre rapidement le problème. Il utilise 150 itérations maximum. Les contraintes peuvent être relâchées en utilisant les paramètres **cfm** et **erp**.

L'environnement de **turtlebot3_world** est spécifié par le modèle **turtlebot3_world/model.sdf** (46- 51). Il s'agit des colonnes vertes, des piliers, et des murs.

Le robot que nous avons modifié est spécifié dans le modèle **turtlebot3_waffle** (lignes 53- 56).

9.4.4 Modèle du robot turtlebot3_waffle modifié

9.4.4.1 Le modèle

Ce modèle est défini dans le répertoire **gazebo_tutorials/models/turtlebot3_waffle**. Il est donné de façon complète ici par le fichier **model.sdf** :

```

1 <?xml version="1.0" ?>
2 <sdf version="1.5">
3   <model name="turtlebot3_waffle">
4     <pose>0.0 0.0 0.0 0.0 0.0 0.0</pose>
5
6     <link name="base_footprint"/>
7
8     <link name="base_link">
9
10    <inertial>
11      <pose>-0.064 0 0.048 0 0 0</pose>
12      <inertia>
13        <ixx>4.2111447e-02</ixx>
14        <ixy>0</ixy>
15        <ixz>0</ixz>
16        <iyy>4.2111447e-02</iyy>
17        <iyz>0</iyz>

```

```

18     <izz>7.5254874e-02</izz>
19   </inertia>
20   <mass>1.3729096e+00</mass>
21 </inertial>
22
23 <collision name="base_collision">
24   <pose>-0.064 0 0.048 0 0 0</pose>
25   <geometry>
26     <box>
27       <size>0.265 0.265 0.089</size>
28     </box>
29   </geometry>
30 </collision>
31
32 <visual name="base_visual">
33   <pose>-0.064 0 0 0 0 0</pose>
34   <geometry>
35     <mesh>
36       <uri>model://turtlebot3_waffle/meshes/waffle_base.dae</uri>
37       <scale>0.001 0.001 0.001</scale>
38     </mesh>
39   </geometry>
40 </visual>
41 </link>
42
43 <link name="imu_link">
44   <sensor name="tb3_imu" type="imu">
45     <always_on>true</always_on>
46     <update_rate>200</update_rate>
47     <imu>
48       <angular_velocity>
49         <x>
50           <noise type="gaussian">
51             <mean>0.0</mean>
52             < stddev>2e-4</ stddev>
53           </noise>
54         </x>
55         <y>
56           <noise type="gaussian">
57             <mean>0.0</mean>
58             < stddev>2e-4</ stddev>
59           </noise>
60         </y>
61         <z>
62           <noise type="gaussian">
63             <mean>0.0</mean>
64             < stddev>2e-4</ stddev>
65           </noise>
66         </z>
67       </angular_velocity>
68       <linear_acceleration>
69         <x>
70           <noise type="gaussian">
71             <mean>0.0</mean>
72             < stddev>1.7e-2</ stddev>
73           </noise>
74         </x>
75         <y>
76           <noise type="gaussian">
77             <mean>0.0</mean>
78             < stddev>1.7e-2</ stddev>
79           </noise>
80         </y>
81         <z>
82           <noise type="gaussian">
83             <mean>0.0</mean>
84             < stddev>1.7e-2</ stddev>
85           </noise>
86         </z>
87       </linear_acceleration>
88     </imu>
89     <plugin name="turtlebot3_imu" filename="libgazebo_ros_imu_sensor.so">
90       <ros>
91         <!-- <namespace>/tb3</namespace> -->
92         <remapping>~out:=imu</remapping>
93       </ros>
94     </plugin>
95   </sensor>
96 </link>
97
98 <link name="base_scan">
99   <inertial>
100    <pose>-0.052 0 0.111 0 0 0</pose>
101   <inertia>
```

```

102      <ixx>0.001</ixx>
103      <ixy>0.000</ixy>
104      <ixz>0.000</ixz>
105      <iyy>0.001</iyy>
106      <iyz>0.000</iyz>
107      <izz>0.001</izz>
108    </inertia>
109    <mass>0.114</mass>
110  </inertial>
111
112  <collision name="lidar_sensor_collision">
113    <pose>-0.052 0 0.111 0 0 0</pose>
114    <geometry>
115      <cylinder>
116        <radius>0.0508</radius>
117        <length>0.055</length>
118      </cylinder>
119    </geometry>
120  </collision>
121
122  <visual name="lidar_sensor_visual">
123    <pose>-0.064 0 0.121 0 0 0</pose>
124    <geometry>
125      <mesh>
126        <uri>model://turtlebot3_waffle/meshes/lds.dae</uri>
127        <scale>0.001 0.001 0.001</scale>
128      </mesh>
129    </geometry>
130  </visual>
131
132  <sensor name="hls_lfcfd_lds" type="ray">
133    <always_on>true</always_on>
134    <visualize>true</visualize>
135    <pose>-0.064 0 0.121 0 0 0</pose>
136    <update_rate>5</update_rate>
137    <ray>
138      <scan>
139        <horizontal>
140          <samples>360</samples>
141          <resolution>1.000000</resolution>
142          <min_angle>0.000000</min_angle>
143          <max_angle>6.280000</max_angle>
144        </horizontal>
145      </scan>
146      <range>
147        <min>0.120000</min>
148        <max>3.5</max>
149        <resolution>0.015000</resolution>
150      </range>
151      <noise>
152        <type>gaussian</type>
153        <mean>0.0</mean>
154        <stddev>0.01</stddev>
155      </noise>
156    </ray>
157    <plugin name="turtlebot3_laserscan" filename="libgazebo_ros_ray_sensor.so">
158      <ros>
159        <!-- <namespace>/tb3</namespace> -->
160        <remapping>~/out:=scan</remapping>
161      </ros>
162      <output_type>sensor_msgs/LaserScan</output_type>
163      <frame_name>base_scan</frame_name>
164    </plugin>
165  </sensor>
166</link>
167
168<link name="wheel_left_link">
169
170  <inertial>
171    <pose>0.0 0.144 0.023 -1.57 0 0</pose>
172    <inertia>
173      <ixx>1.1175580e-05</ixx>
174      <ixy>-4.2369783e-11</ixy>
175      <ixz>-5.9381719e-09</ixz>
176      <iyy>1.1192413e-05</iyy>
177      <iyz>-1.4400107e-11</iyz>
178      <izz>2.0712558e-05</izz>
179    </inertia>
180    <mass>0.1</mass>
181  </inertial>
182
183  <collision name="wheel_left_collision">
184    <pose>0.0 0.144 0.023 -1.57 0 0</pose>
185    <geometry>

```

```

186     <cylinder>
187         <radius>0.033</radius>
188         <length>0.018</length>
189     </cylinder>
190   </geometry>
191   <surface>
192     <!-- This friction parameter don't contain reliable data!! -->
193     <friction>
194       <ode>
195         <mu>100000.0</mu>
196         <mu2>100000.0</mu2>
197         <fdir1>0 0 0</fdir1>
198         <slip1>0.0</slip1>
199         <slip2>0.0</slip2>
200     </ode>
201   </friction>
202   <contact>
203     <ode>
204       <soft_cfm>0</soft_cfm>
205       <soft_erp>0.2</soft_erp>
206       <kp>1e+5</kp>
207       <kd>1</kd>
208       <max_vel>0.01</max_vel>
209       <min_depth>0.001</min_depth>
210     </ode>
211   </contact>
212   </surface>
213 </collision>
214
215 <visual name="wheel_left_visual">
216   <pose>0.0 0.144 0.023 0 0 0</pose>
217   <geometry>
218     <mesh>
219       <uri>model://turtlebot3_waffle/meshes/tire.dae</uri>
220       <scale>0.001 0.001 0.001</scale>
221     </mesh>
222   </geometry>
223 </visual>
224 </link>
225
226 <link name="wheel_right_link">
227
228   <inertial>
229     <pose>0.0 -0.144 0.023 -1.57 0 0</pose>
230     <inertia>
231       <ixx>1.1175580e-05</ixx>
232       <ixy>-4.2369783e-11</ixy>
233       <ixz>-5.9381719e-09</ixz>
234       <iyy>1.1192413e-05</iyy>
235       <iyz>-1.4400107e-11</iyz>
236       <izz>2.0712558e-05</izz>
237     </inertia>
238     <mass>0.1</mass>
239   </inertial>
240
241 <collision name="wheel_right_collision">
242   <pose>0.0 -0.144 0.023 -1.57 0 0</pose>
243   <geometry>
244     <cylinder>
245       <radius>0.033</radius>
246       <length>0.018</length>
247     </cylinder>
248   </geometry>
249   <surface>
250     <!-- This friction parameter don't contain reliable data!! -->
251     <friction>
252       <ode>
253         <mu>100000.0</mu>
254         <mu2>100000.0</mu2>
255         <fdir1>0 0 0</fdir1>
256         <slip1>0.0</slip1>
257         <slip2>0.0</slip2>
258     </ode>
259   </friction>
260   <contact>
261     <ode>
262       <soft_cfm>0</soft_cfm>
263       <soft_erp>0.2</soft_erp>
264       <kp>1e+5</kp>
265       <kd>1</kd>
266       <max_vel>0.01</max_vel>
267       <min_depth>0.001</min_depth>
268     </ode>
269   </contact>
```

```
270      </surface>
271  </collision>
272
273  <visual name="wheel_right_visual">
274    <pose>0.0 -0.144 0.023 0 0 0</pose>
275    <geometry>
276      <mesh>
277        <uri>model://turtlebot3_waffle/meshes/tire.dae</uri>
278        <scale>0.001 0.001 0.001</scale>
279      </mesh>
280    </geometry>
281  </visual>
282</link>
283
284 <link name='caster_back_right_link'>
285   <pose>-0.177 -0.064 -0.004 -1.57 0 0</pose>
286   <inertial>
287     <mass>0.001</mass>
288     <inertia>
289       <ixx>0.00001</ixx>
290       <ixy>0.000</ixy>
291       <ixz>0.000</ixz>
292       <iyy>0.00001</iyy>
293       <iyz>0.000</iyz>
294       <izz>0.00001</izz>
295     </inertia>
296   </inertial>
297   <collision name='collision'>
298     <geometry>
299       <sphere>
300         <radius>0.005000</radius>
301       </sphere>
302     </geometry>
303     <surface>
304       <contact>
305         <ode>
306           <soft_cfm>0</soft_cfm>
307           <soft_erp>0.2</soft_erp>
308           <kp>1e+5</kp>
309           <kd>1</kd>
310           <max_vel>0.01</max_vel>
311           <min_depth>0.001</min_depth>
312         </ode>
313       </contact>
314     </surface>
315   </collision>
316</link>
317
318 <link name='caster_back_left_link'>
319   <pose>-0.177 0.064 -0.004 -1.57 0 0</pose>
320   <inertial>
321     <mass>0.001</mass>
322     <inertia>
323       <ixx>0.00001</ixx>
324       <ixy>0.000</ixy>
325       <ixz>0.000</ixz>
326       <iyy>0.00001</iyy>
327       <iyz>0.000</iyz>
328       <izz>0.00001</izz>
329     </inertia>
330   </inertial>
331   <collision name='collision'>
332     <geometry>
333       <sphere>
334         <radius>0.005000</radius>
335       </sphere>
336     </geometry>
337     <surface>
338       <contact>
339         <ode>
340           <soft_cfm>0</soft_cfm>
341           <soft_erp>0.2</soft_erp>
342           <kp>1e+5</kp>
343           <kd>1</kd>
344           <max_vel>0.01</max_vel>
345           <min_depth>0.001</min_depth>
346         </ode>
347       </contact>
348     </surface>
349   </collision>
350</link>
351
352 <link name="camera_link"/>
```

```

354 <link name="camera_rgb_frame">
355   <inertial>
356     <pose>0.069 -0.047 0.107 0 0 0</pose>
357     <inertia>
358       <ixx>0.001</ixx>
359       <ixy>0.000</ixy>
360       <ixz>0.000</ixz>
361       <iyy>0.001</iyy>
362       <iyz>0.000</iyz>
363       <izz>0.001</izz>
364     </inertia>
365     <mass>0.035</mass>
366   </inertial>
367
368   <pose>0.069 -0.047 0.107 0 0 0</pose>
369   <sensor name="camera" type="camera">
370     <always_on>true</always_on>
371     <visualize>true</visualize>
372     <update_rate>30</update_rate>
373     <camera name="intel_realsense_r200">
374       <horizontal_fov>1.02974</horizontal_fov>
375       <image>
376         <width>1920</width>
377         <height>1080</height>
378         <format>R8G8B8</format>
379       </image>
380       <clip>
381         <near>0.02</near>
382         <far>300</far>
383       </clip>
384       <noise>
385         <type>gaussian</type>
386         <!-- Noise is sampled independently per pixel on each frame.
387             That pixel's noise value is added to each of its color
388             channels, which at that point lie in the range [0,1]. -->
389         <mean>0.0</mean>
390         <stddev>0.007</stddev>
391       </noise>
392     </camera>
393     <plugin name="camera_driver" filename="libgazebo_ros_camera.so">
394       <ros>
395         <!-- <namespace>test_cam</namespace> -->
396         <!-- <remapping>image_raw:=image_demo</remapping> -->
397         <!-- <remapping>camera_info:=camera_info_demo</remapping> -->
398       </ros>
399       <!-- camera_name>omit so it defaults to sensor name</camera_name-->
400       <!-- frame_name>omit so it defaults to link name</frameName-->
401       <!-- <hack_baseline>0.07</hack_baseline> -->
402     </plugin>
403   </sensor>
404 </link>
405
406 <link name="base_solar_sensor">
407   <inertial>
408     <pose>-0.052 0 0.111 0 0 0</pose>
409     <inertia>
410       <ixx>0.001</ixx>
411       <ixy>0.000</ixy>
412       <ixz>0.000</ixz>
413       <iyy>0.001</iyy>
414       <iyz>0.000</iyz>
415       <izz>0.001</izz>
416     </inertia>
417     <mass>0.114</mass>
418   </inertial>
419
420   <collision name="solar_sensor_collision">
421     <pose>-0.052 0 0.311 0 0 0</pose>
422     <geometry>
423       <cylinder>
424         <radius>0.0508</radius>
425         <length>0.055</length>
426       </cylinder>
427     </geometry>
428   </collision>
429
430   <visual name="solar_sensor_visual">
431     <pose>-0.064 0 0.321 0 0 0</pose>
432     <geometry>
433       <cylinder>
434         <radius>0.0508</radius>
435         <length>0.055</length>
436       </cylinder>
437     </geometry>

```

```
438     </visual>
439
440   </link>
441
442   <joint name="base_joint" type="fixed">
443     <parent>base_footprint</parent>
444     <child>base_link</child>
445     <pose>0.0 0.0 0.010 0 0 0</pose>
446   </joint>
447
448   <joint name="wheel_left_joint" type="revolute">
449     <parent>base_link</parent>
450     <child>wheel_left_link</child>
451     <pose>0.0 0.144 0.023 -1.57 0 0</pose>
452     <axis>
453       <xyz>0 0 1</xyz>
454     </axis>
455   </joint>
456
457   <joint name="wheel_right_joint" type="revolute">
458     <parent>base_link</parent>
459     <child>wheel_right_link</child>
460     <pose>0.0 -0.144 0.023 -1.57 0 0</pose>
461     <axis>
462       <xyz>0 0 1</xyz>
463     </axis>
464   </joint>
465
466   <joint name='caster_back_right_joint' type='ball'>
467     <parent>base_link</parent>
468     <child>caster_back_right_link</child>
469   </joint>
470
471   <joint name='caster_back_left_joint' type='ball'>
472     <parent>base_link</parent>
473     <child>caster_back_left_link</child>
474   </joint>
475
476   <joint name="imu_joint" type="fixed">
477     <parent>base_link</parent>
478     <child>imu_link</child>
479     <pose>-0.032 0 0.068 0 0 0</pose>
480     <axis>
481       <xyz>0 0 1</xyz>
482     </axis>
483   </joint>
484
485   <joint name="lidar_joint" type="fixed">
486     <parent>base_link</parent>
487     <child>base_scan</child>
488     <pose>-0.064 0 0.121 0 0 0</pose>
489     <axis>
490       <xyz>0 0 1</xyz>
491     </axis>
492   </joint>
493
494   <joint name="camera_joint" type="fixed">
495     <parent>base_link</parent>
496     <child>camera_link</child>
497     <pose>0.064 -0.065 0.094 0 0 0</pose>
498     <axis>
499       <xyz>0 0 1</xyz>
500     </axis>
501   </joint>
502
503   <joint name="camera_rgb_joint" type="fixed">
504     <parent>camera_link</parent>
505     <child>camera_rgb_frame</child>
506     <pose>0.005 0.018 0.013 0 0 0</pose>
507     <axis>
508       <xyz>0 0 1</xyz>
509     </axis>
510   </joint>
511
512   <joint name="solar_sensor_joint" type="fixed">
513     <parent>base_link</parent>
514     <child>base_solar_sensor</child>
515     <pose>-0.064 0 0.321 0 0 0</pose>
516     <axis>
517       <xyz>0 0 1</xyz>
518     </axis>
519   </joint>
520
521   <plugin name="turtlebot3_diff_drive" filename="libgazebo_ros_diff_drive.so">
```

```

522      <ros>
523        <!-- <namespace>/tb3</namespace> -->
524      </ros>
525
526      <update_rate>30</update_rate>
527
528      <!-- wheels -->
529      <left_joint>wheel_left_joint</left_joint>
530      <right_joint>wheel_right_joint</right_joint>
531
532      <!-- kinematics -->
533      <wheel_separation>0.287</wheel_separation>
534      <wheel_diameter>0.066</wheel_diameter>
535
536      <!-- limits -->
537      <max_wheel_torque>20</max_wheel_torque>
538      <max_wheel_acceleration>1.0</max_wheel_acceleration>
539
540      <command_topic>cmd_vel</command_topic>
541
542      <!-- output -->
543      <publish_odom>true</publish_odom>
544      <publish_odom_tf>true</publish_odom_tf>
545      <publish_wheel_tf>false</publish_wheel_tf>
546
547
548      <odometry_topic>odom</odometry_topic>
549      <odometry_frame>odom</odometry_frame>
550      <robot_base_frame>base_footprint</robot_base_frame>
551
552    </plugin>
553
554    <plugin name="turtlebot3_joint_state" filename="libgazebo_ros_joint_state_publisher.so">
555      <ros>
556        <!-- <namespace>/tb3</namespace> -->
557        <remapping>~/out:=joint_states</remapping>
558      </ros>
559      <update_rate>30</update_rate>
560      <joint_name>wheel_left_joint</joint_name>
561      <joint_name>wheel_right_joint</joint_name>
562    </plugin>
563
564    <plugin name="turtlebot3_solar_sensor" filename="libsolar_sensor.so">
565      <update_rate>30</update_rate>
566      <light_name>sunloin</light_name>
567      <model_name>turtlebot3_waffle</model_name>
568      <frame_name>base_solar_sensor</frame_name>
569    </plugin>
570
571  </model>
572</sdf>

```

9.4.4.2 Explications du modèle

Le modèle est spécifié entre deux balises **model**. Il est constitué de corps spécifiés par la balise **link** et d'articulations spécifiées par la balise **joint**.

La base du robot **turtlebot3_waffle** est nommé **base_link** (voir ligne 8).

Un corps, comme pour le format URDF, est défini par des paramètres inertIELS exprimés dans la balise **inertial**. Pour le corps **base_link** il s'agit des ligne 10-21. Le centre de masse est défini par la balise **pose** dans la balise **inertial**. Il s'agit de la ligne 11 pour le corps **base_link**. La matrice d'inertie du corps, supposé solide, est exprimée dans la balise **inertia**. (ligne 12). La masse du corps est exprimée dans la balise **mass** dans la balise **inertial** (ligne 20).

Afin de pouvoir calculer les collisions, et limiter le coût calculatoire il est préférable de ne pas utiliser un modèle compliqué de la géométrie de l'objet. On sépare pour cela la partie pour calculer les collisions et la partie pour visualiser l'objet. La partie collision est spécifiée avec la balise **collision** (ligne 23 pour **base_link**). La partie géométrie est spécifiée avec la balise **geometry** (ligne 25 pour **base_link**). Ici par exemple pour l'objet **base_link** la collision est modélisée par une boîte d'une taille de *26.5cm*. La balise **visual** est utilisée pour spécifier la géométrie de l'objet utilisée pour la visualisation du robot. Ici pour l'objet **base_link** la géométrie est spécifiée par un fichier DAE qui fournit une soupe de triangles, ou de polygones (ligne 34).

Comme il s'agit d'un robot mobile il n'est pas fixé dans le monde comme un robot industriel 6 axes classique. On ne trouve donc aucun joint de type **fixed** entre la base (**base_footprint** et le repère **world**).

Afin de pouvoir bouger le robot **turtlebot3_waffle** le robot utilise deux roues actionnées dont les

corps sont spécifiés par **wheel_right_link** (lignes 226 - 282) et **wheel_left_link** (lignes 168 - 224). Le robot conserve son équilibre grâce à deux roues folles : **cast_back_right_link** (lignes 284 - 316) et **caster_back_left_link** (lignes 318 - 350).

Les joints qui permettent de bouger le robot pour les roues sont : **wheel_left_joint** (lignes 448 - 455) et **wheel_right_joint** (lignes 457 - 464).

Les joints des roues libres sont : **caster_back_left_joint** (lignes 471 - 474) et **caster_back_right_joint** (lignes 466 - 469).

Les capteurs sont considérés comme des corps. On a ainsi une centrale inertie qui est définie par le corps **imu_link** (lignes 43-96). A l'intérieur des balises **link**, on trouve la définition de la simulation du capteur entre deux balises **sensor** (lignes 44-88). On y trouve la fréquence de mise à jour avec la balise **update_rate**. Des champs spécifiques permettent de définir la loi de probabilités du bruit du capteur. Dans ce cas, un bruit Gaussien est ajouté sur chacun des axes (x,y,z) de la vitesse angulaire et l'accélération linéaire. Il est défini par une valeur moyenne et une déviation standard. Chacun des axes est considéré comme une variable indépendante. Enfin pour que les informations du capteur soient disponibles via un topic ROS, il faut utiliser un plugin gazebo. Dans le cas de la centrale inertie on peut utiliser le plugin **libgazebo_ros_imu_sensor** (lignes 89-94).

9.4.4.3 Capteur solaire

Le capteur solaire est défini par le corps **base_solar_sensor**. Cela correspond aux lignes 406 - 440 du fichier donnant le modèle du robot. La géométrie pour la collision et la visualisation sont les mêmes. C'est à dire un cylindre de rayon 0.0508 et de longueur 0.055.

9.4.5 Plugin du capteur solaire

Le but de ce plugin est donner la direction du soleil sous la forme d'un topic de type **sensor_msgs** : **:msg** : **:Imu**. Le nom par défaut du topic sur lequel les informations seront publiées est **sunloin_imu**. La fréquence de mise à jour est donnée par le champ **update_rate**. Le nom du corps correspondant au capteur par défaut est **base_solar_sensor**. Il est possible de spécifier des offsets avec les balises **offsets** et **rpy_offsets**.

9.4.5.1 Le code

Le code implémentant cette stratégie est donné ici, la licence ayant été retirée par souci de compacté :

```

1 #include <gazebo_ros/node.hpp>
2 #include <gazebo_ros/utils.hpp>
3 #include <gazebo_ros/conversions/geometry_msgs.hpp>
4 #include <gazebo_ros/conversions/builtin_interfaces.hpp>
5 #include <gazebo_tutorials/solar_sensor.hpp>
6 #include <sensor_msgs/msg/imu.hpp>
7 #include <rclcpp/rclcpp.hpp>
8
9 #ifdef NO_ERROR
10 // NO_ERROR is a macro defined in Windows that's used as an enum in tf2
11 #undef NO_ERROR
12 #endif
13
14 #ifdef IGN_PROFILER_ENABLE
15 #include <ignition/common/Profiler.hh>
16 #endif
17
18 #include <string>
19 #include <memory>
20
21 namespace gazebo_plugins
22 {
23
24 class SolarSensorPrivate
25 {
26 public:
27
28     /// Callback to be called at every simulation iteration
29     /// \param[in] info Updated simulation info
30     void OnUpdate(const gazebo::common::UpdateInfo & info);
31
32     /// The link being tracked.
33     gazebo::physics::LightPtr light_{nullptr};
34
35     /// The body of the frame to display pose, twist
36     gazebo::physics::LinkPtr reference_link_{nullptr};
37
38     /// Pointer to ros node
39     gazebo_ros::Node::SharedPtr ros_node_{nullptr};

```

```

40  /// Odometry publisher
41  rclcpp::Publisher<sensor_msgs::msg::Imu>::SharedPtr pub_{nullptr};
42
43  /// Odom topic name
44  std::string topic_name_{"sunloin_imu"};
45
46  /// Frame transform name, should match name of reference link, or be world.
47  std::string frame_name_{"base_solar_sensor"};
48
49  /// Constant xyz and rpy offsets
50  ignition::math::Pose3d offset_;
51
52  /// Keep track of the last update time.
53  gazebo::common::Time last_time_;
54
55  /// Publish rate in Hz.
56  double update_rate_{0.0};
57
58  /// Gaussian noise
59  double gaussian_noise_;
60
61  /// Pointer to the update event connection
62  gazebo::event::ConnectionPtr update_connection_{nullptr};
63
64 };
65
66 SolarSensor::SolarSensor()
67 : impl_(std::make_unique<SolarSensorPrivate>())
68 {
69 }
70
71 SolarSensor::~SolarSensor()
72 {
73 }
74
75 // Load the controller
76 void SolarSensor::Load(gazebo::physics::ModelPtr model,
77                         sdf::ElementPtr sdf)
78 {
79     // Configure the plugin from the SDF file
80     impl_->ros_node_ = gazebo_ros::Node::Get(sdf);
81
82     // Get QoS profiles
83     const gazebo_ros::QoS & qos = impl_->ros_node_->get_qos();
84
85     if (!sdf->HasElement("update_rate")) {
86         RCLCPP_DEBUG(
87             impl_->ros_node_->get_logger(), "solar_sensor plugin missing <update_rate>, defaults to 0.0"
88             " (as fast as possible)");
89     } else {
90         impl_->update_rate_ = sdf->GetElement("update_rate")->Get<double>();
91     }
92
93     std::string light_name;
94     if (!sdf->HasElement("light_name")) {
95         RCLCPP_ERROR(impl_->ros_node_->get_logger(), "Missing <light_name>, cannot proceed");
96         return;
97     } else {
98         light_name = sdf->GetElement("light_name")->Get<std::string>();
99     }
100
101     gazebo::physics::WorldPtr world = model->GetWorld();
102     impl_->light_ = world->LightByName(light_name);
103     if (!impl_->light_) {
104         RCLCPP_ERROR(
105             impl_->ros_node_->get_logger(), "light_name: %s does not exist\n",
106             light_name.c_str());
107         return;
108     }
109
110     impl_->pub_ = impl_->ros_node_->create_publisher<sensor_msgs::msg::Imu>(
111         impl_->topic_name_, qos.get_publisher_qos(
112             impl_->topic_name_, rclcpp::SensorDataQoS().reliable()));
113     impl_->topic_name_ = impl_->pub_->get_topic_name();
114     RCLCPP_DEBUG(
115         impl_->ros_node_->get_logger(), "Publishing on topic [%s]", impl_->topic_name_.c_str());
116
117     if (sdf->HasElement("xyz_offsets")) {
118         RCLCPP_WARN(
119             impl_->ros_node_->get_logger(), "<xyz_offsets> is deprecated, use <xyz_offset> instead.");
120         impl_->offset_.Pos() = sdf->GetElement("xyz_offsets")->Get<ignition::math::Vector3d>();
121     }
122     if (!sdf->HasElement("xyz_offset")) {
123         if (!sdf->HasElement("xyz_offsets")) {

```

```

124     RCLCPP_DEBUG(impl_->ros_node_->get_logger(), "Missing <xyz_offset>, defaults to 0s");
125 }
126 } else {
127     impl_->offset_.Pos() = sdf->GetElement("xyz_offset")->Get<ignition::math::Vector3d>();
128 }
129
130 if (sdf->HasElement("rpy_offsets")) {
131     RCLCPP_WARN(
132         impl_->ros_node_->get_logger(), "<rpy_offsets> is deprecated, use <rpy_offset> instead.");
133     impl_->offset_.Rot() = ignition::math::Quaternionsd(
134         sdf->GetElement("rpy_offsets")->Get<ignition::math::Vector3d>());
135 }
136 if (!sdf->HasElement("rpy_offset")) {
137     if (!sdf->HasElement("rpy_offsets")) {
138         RCLCPP_DEBUG(impl_->ros_node_->get_logger(), "Missing <rpy_offset>, defaults to 0s");
139     }
140 } else {
141     impl_->offset_.Rot() = ignition::math::Quaternionsd(
142         sdf->GetElement("rpy_offset")->Get<ignition::math::Vector3d>());
143 }
144
145 if (!sdf->HasElement("gaussian_noise")) {
146     RCLCPP_DEBUG(impl_->ros_node_->get_logger(), "Missing <gassian_noise>, defaults to 0.0");
147     impl_->gaussian_noise_ = 0;
148 } else {
149     impl_->gaussian_noise_ = sdf->GetElement("gaussian_noise")->Get<double>();
150 }
151
152 impl_->last_time_ = world->SimTime();
153
154 if (!sdf->HasElement("frame_name")) {
155     RCLCPP_DEBUG(
156         impl_->ros_node_->get_logger(), "Missing <frame_name>, defaults to base_solar_sensor");
157 } else {
158     impl_->frame_name_ = sdf->GetElement("frame_name")->Get<std::string>();
159 }
160
161 if (model!=nullptr)
162 {
163     impl_->reference_link_ = model->GetLink(impl_->frame_name_);
164     if (!impl_->reference_link_) {
165         RCLCPP_WARN(
166             impl_->ros_node_->get_logger(), "<frame_name> [%s] does not exist.",
167             impl_->frame_name_.c_str());
168     }
169 }
170
171 // Listen to the update event. This event is broadcast every simulation iteration
172 impl_->update_connection_ = gazebo::event::Events::ConnectWorldUpdateBegin(
173     std::bind(&SolarSensorPrivate::OnUpdate, impl_.get(), std::placeholders::_1));
174 }
175
176 // Update the controller
177 void SolarSensorPrivate::OnUpdate(const gazebo::common::UpdateInfo & info)
178 {
179     if (!light_) {
180         return;
181     }
182 #ifdef IGN_PROFILER_ENABLE
183     IGN_PROFILE("SolarSensorPrivate::OnUpdate");
184 #endif
185     gazebo::common::Time current_time = info.simTime;
186
187     if (current_time < last_time_) {
188         RCLCPP_WARN(ros_node_->get_logger(), "Negative update time difference detected.");
189         last_time_ = current_time;
190     }
191
192     // Rate control
193     if (update_rate_ > 0 &&
194         (current_time - last_time_).Double() < (1.0 / update_rate_))
195     {
196         return;
197     }
198
199     // If we don't have any subscribers, don't bother composing and sending the message
200     if (ros_node_->count_subscribers(topic_name_) == 0) {
201         return;
202     }
203
204     // Differentiate to get accelerations
205     double tmp_dt = current_time.Double() - last_time_.Double();
206     if (tmp_dt == 0) {
207         return;
208     }

```

```

208     }
209 #ifdef IGN_PROFILER_ENABLE
210     IGN_PROFILE_BEGIN("fill ROS message");
211 #endif
212     sensor_msgs::msg::Imu imu_msg;
213
214     // Copy data into pose message
215     imu_msg.header.frame_id = frame_name_;
216     imu_msg.header.stamp = gazebo_ros::Convert<builtin_interfaces::msg::Time>(current_time);
217
218     // Get inertial rates
219     ignition::math::Vector3d vpos = light_->WorldLinearVel();
220     ignition::math::Vector3d veul = light_->WorldAngularVel();
221
222     // Get pose/orientation
223     auto pose = light_->WorldPose();
224
225     // Apply reference frame
226     if (reference_link_) {
227         // Convert to relative pose, rates
228         auto frame_pose = reference_link_->WorldPose();
229         auto frame_vpos = reference_link_->WorldLinearVel();
230         auto frame_veul = reference_link_->WorldAngularVel();
231
232         pose.Pos() = pose.Pos() - frame_pose.Pos();
233         pose.Pos().Normalize();
234         pose.Pos() = frame_pose.Rot().RotateVectorReverse(pose.Pos());
235         pose.Rot() *= frame_pose.Rot().Inverse();
236
237         vpos = frame_pose.Rot().RotateVector(vpos - frame_vpos);
238         veul = frame_pose.Rot().RotateVector(veul - frame_veul);
239     }
240
241     // Apply constant offsets
242
243     // Apply XYZ offsets and get position and rotation components
244     pose.Pos() = pose.Pos() + offset_.Pos();
245     // Apply RPY offsets
246     pose.Rot() = offset_.Rot() * pose.Rot();
247     pose.Rot().Normalize();
248
249     double pitch = std::asin(-pose.Pos().Z());
250     double yaw = std::atan2(pose.Pos().Y(), pose.Pos().X());
251
252     ignition::math::Quaternionsd compass_to_sunloin(0.0, pitch, yaw);
253     // Fill out messages
254     imu_msg.orientation = gazebo_ros::Convert<geometry_msgs::msg::Quaternion>(compass_to_sunloin);
255
256     // Fill in covariance matrix
257     /// @TODO: let user set separate linear and angular covariance values
258     double gn2 = gaussian_noise_ * gaussian_noise_;
259     imu_msg.orientation_covariance[0] = gn2;
260     imu_msg.orientation_covariance[4] = gn2;
261     imu_msg.orientation_covariance[8] = gn2;
262
263 #ifdef IGN_PROFILER_ENABLE
264     IGN_PROFILE_END();
265     IGN_PROFILE_BEGIN("publish");
266 #endif
267     // Publish to ROS
268     pub_->publish(imu_msg);
269 #ifdef IGN_PROFILER_ENABLE
270     IGN_PROFILE_END();
271 #endif
272     // Save last time stamp
273     last_time_ = current_time;
274 }
275
276 GZ_REGISTER_MODEL_PLUGIN(SolarSensor)
277 } // namespace gazebo_plugins

```

9.4.5.2 En-têtes

Le plugin utilise une architecture d'implémentation privée qui permet d'éviter d'exposer inutilement certains détails des champs pour l'API.

Pour cette raison la définition de la classe faisant réellement le travail est inclue dans ce fichier au lieu d'être exposée dans un fichier d'en-tête.

Les entêtes commencent par celles liant Gazebo et ROS (lignes 1-4). On y trouve l'entête permettant l'accès à l'API des Nodes de ROS, un entête pour des utilitaires génériques. Puis les entêtes de conversions

sont utilisés afin de pouvoir passer des outils mathématiques de Gazebo (librairie ignition) aux messages ROS. Il faut ensuite spécifier le fichier qui contient la déclaration de la classe **SolarSensor** (ligne 5) On trouve ensuite l'entête pour déclarer les messages ROS pour les imus (ligne 6). Il y a ensuite l'entête permettant d'utiliser le C++ avec ROS (rclcpp).

9.4.5.3 Déclaration de SolarSensorPrivate

Après des en-têtes systèmes standards (string et memory), l'implémentation privée du capteur solaire est déclarée (lignes 24-64). Nommée **SolarSensorPrivate** elle stocke :

- un pointeur partagé sur le corps où se trouve la lumière **light_** (ligne 33)
- un pointeur partagé sur le corps de référence du modèle du robot **reference_link_** (ligne 36)
- un pointeur partagé sur le node ROS associé au simulateur **ros_node_** (ligne 39)
- un publisher ROS de message ROS Imu **pub_** (ligne 42)
- le nom du topic sur lequel publié le message d'IMU **topic_name_** (ligne 45)
- le nom du corps de référence du robot **frame_name_** (ligne 48)
- l'offset de pose du capteur **offset_** (ligne 51)
- la dernière date de la mise à jour du capteur **offset_** (ligne 54)
- la vitesse désirée de mise à jour du capteur **update_rate_** (ligne 57)
- le bruit Gaussien du capteur **Gaussian_** (ligne 60)
- la fonction d'appel lorsqu'une mise à jour est nécessaire **update_connection_** (ligne 63)

L'instanciation de la classe privée se fait dans le constructeur du plugin **SolarSensor** (ligne 67).

9.4.5.4 Méthode SolarSensor : :Load

La méthode **SolarSensor : :Load** s'occupe de lire le fichier SDF et d'initialiser tous les paramètres du plugin et donc de l'implémentation privée stockée dans le champ **impl_**. Les deux paramètres de la méthode sont le lien vers le modèle dans lequel est situé l'appel au plugin (**model**), et le pointeur vers les éléments SDF paramètres du plugin.

La première chose est de stocker le pointeur vers le Node ROS (ligne 80). On peut ensuite stocker la qualité de service utilisé par le Node (83). La méthode ensuite récupère la valeur de la balise **update_rate** (lignes 85- 91). La méthode ensuite récupère le nom de la lumière **light_name** (lignes 93- 99). Grâce à cette information il est alors possible grâce à la référence au monde d'obtenir la lumière correspondante (**light_**).

La méthode créée ensuite un publisher ROS pour publier un topic de type **sensor_msgs : :msg : :Imu** avec le nom **topic_name_**, et la qualité de service de type **reliable** (lignes 110-115).

La méthode lit ensuite les valeurs de décalage avec les balises **xyz_offset** (lignes 117- 128) et **rpy_offset** (lignes 130- 143) si elles existent.

Le bruit gaussien est également lu de la balise **gaussian_noise** (lignes 145- 150).

La variable **last_time_** est ensuite initialisé avec le temps simulé.

Il faut ensuite lire le nom du frame de référence **frame_name** (lignes 154-169) pour ensuite trouver le corps de référence grâce au pointeur sur le monde. La dernière opération est de connecter la mise à jour du plugin avec l'appel à la méthode **OnUpdate** de l'implémentation privée du plugin.

9.4.5.5 Méthode SolarSensorPrivate : :OnUpdate

La méthode vérifie qu'une lumière a été trouvée et retourne immédiatement si ce n'est pas le cas.

Ensuite, la méthode vérifie que le temps écoulé correspond à la fréquence de mise à jour demandée par l'utilisateur. Si ce n'est pas le cas elle retourne immédiatement (lignes 185-197). Afin de gagner du temps, la méthode retourne immédiatement si aucun autre Node n'a souscrit au topic (ligne 201).

La construction du message à publier sur le topic s'effectue des lignes 212 à 261.

Tout d'abord la direction du soleil est exprimée dans le corps du capteur. Ensuite le temps est converti du format Gazebo au format ROS.

La vitesse, l'accélération et enfin la pose de la lumière (si elle bouge) sont ensuite récupérées. Elles sont initialement exprimées dans le repère du monde. Puis ces quantités sont projetées de façon à être exprimées dans le repère du capteur (lignes 226-239).

Si des offsets sont fournis ils sont utilisés des lignes 241 à 247.

On calcule ensuite la direction de la lumière en calculant la direction du vecteur vers la lumière en coordonnées polaires (lignes 249 à 250). Le résultat est ensuite convertit en quaternion puis utiliser pour remplir le champ orientation du message à publier.

Il reste ensuite à remplir la matrice de covariance avec le bruit Gaussien pour compléter le message.

Le message est ensuite publier (ligne 268).

Introduction à ROS-2

10 Graphe d'application avec ROS-2 105

- 10.1 Turtlesim et rqt
- 10.2 Node
- 10.3 Service
- 10.4 Paramètres
- 10.5 Actions
- 10.6 rqt_console
- 10.7 Bag : fichier de données
- 10.8 Launch : démarrer une application
- 10.9 Création de messages et de services

11 Ecrire des nodes ROS-2 131

- 11.1 Topics
- 11.2 Services

10. Graphe d'application avec ROS-2

Dans ce chapitre nous allons introduire les changements de concepts liés à ROS-2. La différence essentielle du point de vue de l'utilisateur est la disparition du concept de master. Il n'y donc plus d'annuaire d'objets. Les noeuds se découvrent les uns les autres en suivant un protocole assez similaire à celui d'Internet.

On retrouve de nombreux concepts similaires à ROS-1 :

- les nodes
- les topics
- les services

Il y a toutefois des modifications dans la gestion des paramètres qui n'existe plus maintenant que comme des paramètres rattachés à un node.

D'une manière générale tous les concepts similaires à ROS-1 sont atteignables via l'utilisation de la commande *ros2*.

10.1 Turtlesim et rqt

Cette section correspond au didacticiel Introduction à Turtlesim. Turtlesim est un petit simulateur pour apprendre ROS-2. Il illustre les concepts de ROS-2 à un niveau très simple pour ensuite les utiliser sur un robot ou un simulateur plus complet.

rqt est une interface graphique pour ROS-2. Tout ce que fait rqt peut-être fait par des commandes en ligne. Mais il fournit un moyen d'utiliser les éléments ROS-2 plus facilement et plus amicale pour les utilisateurs.

The didacticiel suppose que vous avez déjà configuré votre environnement.

10.1.1 Utilisation de turtlesim

10.1.1.1 Vérification de l'installation de turtlesim

On peut vérifier que le paquet **turtlesim** est installé en utilisant la commande suivante :

```
1 ros2 pkg executables turtlesim
```

Ce qui donne :

```
1 turtlesim draw_square
2 turtlesim mimic
3 turtlesim turtle_teleop_key
4 turtlesim turtlesim_node
```

L'installation du package turtlesim avec linux s'effectue avec :

```

1 sudo apt update
2
3 sudo apt install ros-foxy-turtlesim

```

10.1.1.2 Lancer turtlesim

Pour lancer le node **turtlesim** il faut lancer la commande suivante :

```
1 ros2 run turtlesim turtlesim_node
```

Vous devriez voir la sortie suivante dans le terminal :

```

1 [INFO] [1608722845.948451886] [turtlesim]: Starting turtlesim with node name /turtlesim
2 [INFO] [1608722845.985347636] [turtlesim]: Spawning turtle [turtle1] at x=[5,544445], y=[5,544445], theta
   =[0,000000]

```

Ces informations indiquent le nom par défaut de la tortue, i.e. **turtle1**, et les coordonnées où elle apparaît.

Une fenêtre bleue avec une tortue apparaît comme celle affichée dans la figure 10.1. La tortue peut-être différente car elles sont tirées aléatoirement dans un ensemble de tortues.

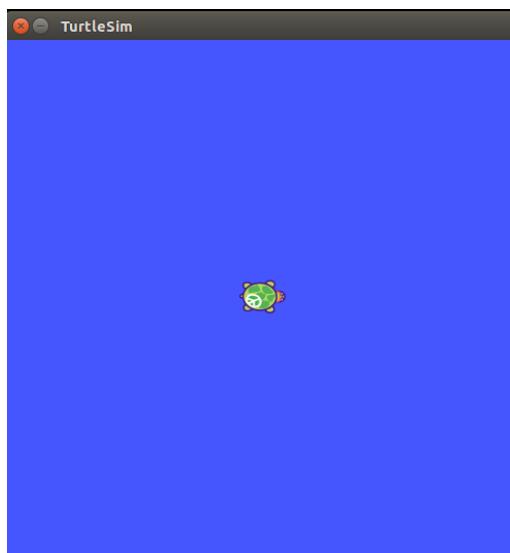


FIGURE 10.1 – Fenêtre résultat de `ros2 run turtlesim turtlesim_node`

10.1.1.3 Utiliser turtlesim

Pour démarrer un node permettant de contrôler la tortue, il faut ouvrir un nouveau terminal avec l'environnement proprement configuré :

```
1 ros2 run turtlesim turtle_teleop_key
```

Vous devriez avoir maintenant 3 fenêtres ouvertes : un terminal où fonctionne **turtlesim_node**, un autre terminal avec **turtle_teleop_key** et la fenêtre **turtlesim**. Il est préférable d'arranger les fenêtres avec de façon à voir la fenêtre graphique **turtlesim**, mais il faut que le focus de votre souris soit sur le terminal de **turtle_teleop_key** de façon à pouvoir contrôler la tortue de **turtlesim**.

Les flèches du clavier permettent de contrôler la tortue. Elle se déplace sur l'écran utilisant son crayon ("pen") pour tracer le chemin parcouru.

Note 10.1 Les flèches ne font bouger la tortue que sur une distance courte. Ce comportement est motivé par le fait que réallement on ne souhaite pas continuer à émettre une commande si, par exemple, l'opérateur perd la connection avec le robot. ■

Vous pouvez voir les nodes, les services associés, topics et actions grâce à la commande **list** :

```

1 ros2 node list
2 ros2 topic list
3 ros2 service list
4 ros2 action list

```

Ces concepts sont développés dans d'autres didacticiels. A ce stade on ne cherche à avoir qu'une vue très générale de turtlesim, la suite utilise rqt (une interface utilisateur graphique pour ROS 2) pour regarder les services.

10.1.2 Installer rqt

Il faut ouvrir un terminal pour installer **rqt** et ses plugins. Sous linux il suffit de faire :

```

1 sudo apt update
2 sudo apt install ros-foxy-rqt*

```

Pour démarrer rqt :

```
1 rqt
```

10.1.3 Utiliser rqt

La première fois où cette commande est démarre la fenêtre peut-être vide. Il suffit de sélectionner dans Plugins > Services > Service Caller de la barre de menu tout en haut.

Note 10.2 rqt peut prendre du temps à localiser tous les plugins. Si en cliquant sur **Plugins** vous ne voyez pas **Services**, il faut fermer rqt, et entrer la commande **rqt --force-discover** dans le terminal ■

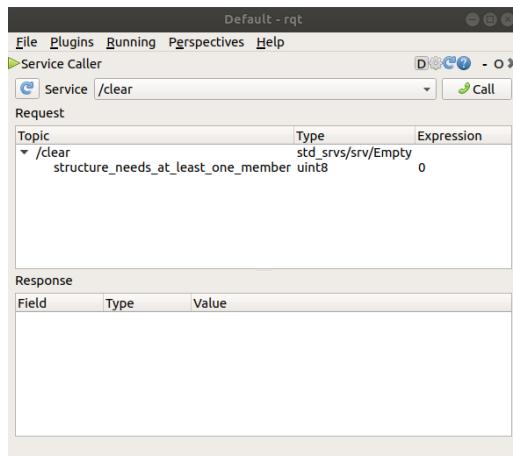


FIGURE 10.2 – Fenêtres correspondant à **rqt** avec Services

On peut utiliser le bouton **Refresh** pour être sûr d'avoir la liste de tous les services du node turtlesim.

Il faut cliquer sur la liste déroulante **Service** pour voir tous les services de turtlesim, et sélectionner le service **/spawn**.

10.1.4 Essayer le service spawn

Nous allons utiliser le service **/spawn**. Il est possible de deviner qu'à partir du nom, le service **/spawn** va créer une autre tortue dans la fenêtre graphique.

Il faut donner à la nouvelle tortue un nom unique, comme **turtle2** en double cliquant entre les quotes simples dans la colonne **Expression**. Cette expression correspond à la valeur de la variable **name** du service qui est de type **string**.

Il faut également entrer des coordonnées pour la tortue à créer, comme **x = 1.0** et **y = 1.0**.

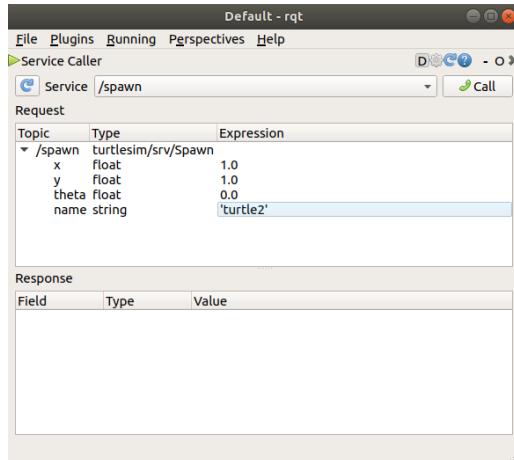


FIGURE 10.3 – Valeurs à spécifier pour l'appel au service **spawn** dans l'interface **rqt**

Note 10.3 Si vous essayez de créer une tortue avec le même nom qu'une tortue existante, par exemple le nom par défaut **turtle1**, vous obtiendrez un message d'erreur dans le terminal dans lequel est lancé **turtlesim_node** :

```
[ERROR] [turtlesim]: A turtle named [turtle1] already exists
```

Pour lancer la tortue **turtle2**, il faut appeler le service en cliquant sur le bouton **Call** en haut à droite de la fenêtre graphique **rqt**.

Vous verrez une nouvelle tortue (avec un choix aléatoire) aux coordonnées fixées en **x** et **y**.

Si vous rafraîchissez la liste de services dans **rqt**, vous verrez qu'il y a maintenant des services liés aux nouvelles tortues **/turtle2/...** en plus de celles de **/turtle1....**

10.1.5 Essayer le service **set_pen**

Essayons de changer la trace de la tortue en utilisant le service **set_pen**.

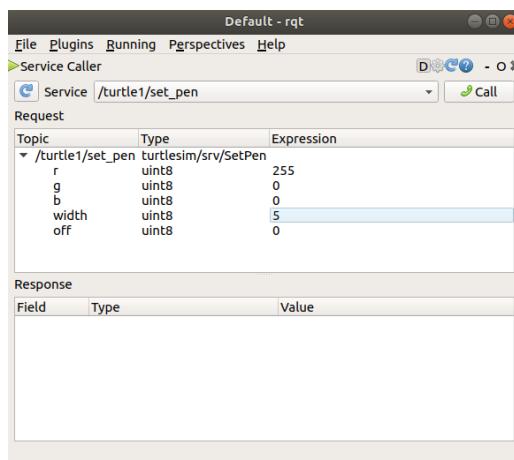


FIGURE 10.4 – Fenêtre correspondant au service **set_pen**

Les valeurs pour **r,g** et **b** sont comprises entre 0 et 255, définissent la couleur de la trace de la tortue **turtle1** et **width** sa largeur.

Pour avoir la tortue turtle1 tracer une ligne rouge très visible, il faut changer la valeur **r** à 255 et la valeur **width** à 5. Il ne faut pas oublier d'appeler le service après avoir mise à jour les valeurs.

Il faut maintenant retourner au terminal où **turtle_teleop_node** est exécuté et appuyer sur les flèches, vous devriez voir que la trace de turtle1 a changé (cf. fig. 10.5)

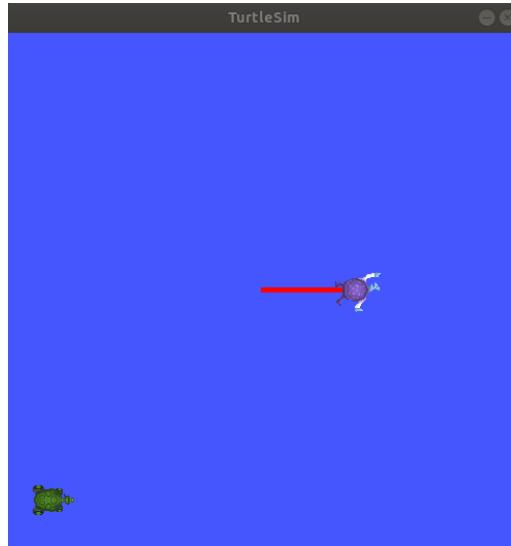


FIGURE 10.5 – Fenêtre montrant l'effet de l'appel au service **set_pen**

Vous avez probablement noté qu'il n'est pas possible de bouger turtle2. Ceci peut-être accompli en redirigeant le topic **cmd_vel** d'un node de type **turtle_teleop_key** vers la tortue turtle2.

10.1.6 Redirection

Dans un nouveau terminal, il faut lancer la commande suivante :

```
! ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=turtle2/cmd_vel
```

Il est maintenant possible de bouger la tortue turtle2 quand ce terminal a le focus de la souris, et la tortue turtle1 quand c'est l'autre terminal qui est actif.



FIGURE 10.6 – Faire bouger la deuxième tortue grâce à la redirection

10.1.7 Arrêt de turtlesim

Pour arrêter la simulation, il suffit de faire la combinaison de touches **Ctrl + C** dans le terminal de **turtlesim_node** et utiliser la touche **q** dans le terminal **teleop**.

10.2 Node

10.2.1 Définition d'un node

Comme pour ROS-1 on peut définir un node comme étant un processus associé à l'exécution d'un fichier exécutable dans un paquet ROS.

10.2.2 ros2 run

Pour lancer ce processus on peut utiliser la commande **ros2 run** :

```
1 ros2 run <package\_\_name> <executable\_\_name>
```

Par exemple pour démarrer turtlesim, il faut ouvrir un terminal et entrer la commande suivante :

```
1 ros2 run turtlesim turtlesim_node
```

La fenêtre graphique va s'ouvrir comme vu dans le didacticiel précédent.

Ici le nom du package est **turtlesim** et le nom de l'exécutable **turtlesim_node**.

On peut donc voir les exécutables dans un paquet ROS, ici **turtlesim**, avec :

```
1 ros2 pkg executables turtlesim
```

Ce qui donne :

```
1 turtlesim draw_square
2 turtlesim mimic
3 turtlesim turtle_teleop_key
4 turtlesim turtlesim_node
```

Pour obtenir la liste des nodes actifs on peut utiliser la commande **ros2 node list**.

10.2.3 ros2 node list

La commande **ros2 node list** va vous montrer les noms de tous les nodes actifs. C'est utile lorsque vous souhaitez interagir avec un node, ou si vous souhaitez savoir quels sont les nodes dans un système qui en exécutent beaucoup.

En ouvrant un autre terminal et avec turtlesim toujours actif dans une autre fenêtre, il faut entrer la commande :

```
1 ros2 node list
```

Ce qui donne :

```
1 /turtlesim
```

Dans un autre terminal, taper :

```
1 ros2 run turtlesim turtle_teleop_key
```

Ici, le système cherche le paquet **turtlesim** et l'exécutable **turtle_teleop_key**.

En retournant dans le terminal où vous avez tapé **ros2 node list**, relancez la même commande. On voit maintenant le nom des deux nodes actifs :

```
1 /turtlesim
2 /teleop_turtle
```

10.2.4 Redirection

La redirection permet de changer des propriétés du node comme le nom d'un topic, le nom d'un service à des valeurs particulières. Dans le didacticiel précédent, la redirection a été utilisé pour que **turtle_teleop_key** puisse contrôler la tortue souhaitée.

Essayons maintenant de changer le nom de notre node **/turtlesim**. Dans un nouveau terminal, il faut lancer la commande suivante :

```
1 ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

Comme on exécute une deuxième fois le node `turtlesim_node`, une autre fenêtre graphique s'ouvre. Toutefois, en retournant sur le terminal où on a exécutée la commande **ros2 node list**, on peut voir les trois noms suivants :

```
1 /my\_\_turtle
2 /turtlesim
3 /teleop\_\_turtle
```

10.2.5 ros2 node info

Maintenant que vous connaissez les noms de vos nodes, vous pouvez avoir accès à plus d'information à leur propos avec :

```
1 ros2 node info <node_name>
```

Pour obtenir des informations sur votre dernier node **my_turtle** :

```
1 ros2 node info /my_turtle
```

ce qui donne :

```
1 There are 2 nodes in the graph with the exact name "/turtlesim". You are seeing information about only one of them.
2 /my_turtle
3   Subscribers:
4     /my_turtle/cmd_vel: geometry_msgs/msg/Twist
5     /parameter_events: rcl_interfaces/msg/ParameterEvent
6     /turtle1/cmd_vel: geometry_msgs/msg/Twist
7   Publishers:
8     /my_turtle/color_sensor: turtlesim/msg/Color
9     /my_turtle/pose: turtlesim/msg/Pose
10    /parameter_events: rcl_interfaces/msg/ParameterEvent
11    /rosout: rcl_interfaces/msg/Log
12    /turtle1/color_sensor: turtlesim/msg/Color
13    /turtle1/pose: turtlesim/msg/Pose
14   Service Servers:
15     /clear: std_srvs/srv/Empty
16     /kill: turtlesim/srv/Kill
17     /my_turtle/set_pen: turtlesim/srv/SetPen
18     /my_turtle/teleport_absolute: turtlesim/srv/TeleportAbsolute
19     /my_turtle/teleport_relative: turtlesim/srv/TeleportRelative
20     /reset: std_srvs/srv/Empty
21     /spawn: turtlesim/srv/Spawn
22     /turtle1/set_pen: turtlesim/srv/SetPen
23     /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
24     /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
25     /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
26     /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
27     /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
28     /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
29     /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
30     /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
31   Service Clients:
32
33   Action Servers:
34     /my_turtle/rotate_absolute: turtlesim/action/RotateAbsolute
35     /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
36   Action Clients:
```

Tutoriel associé : Tutorial Understanding ROS Nodes : <https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Nodes.html>

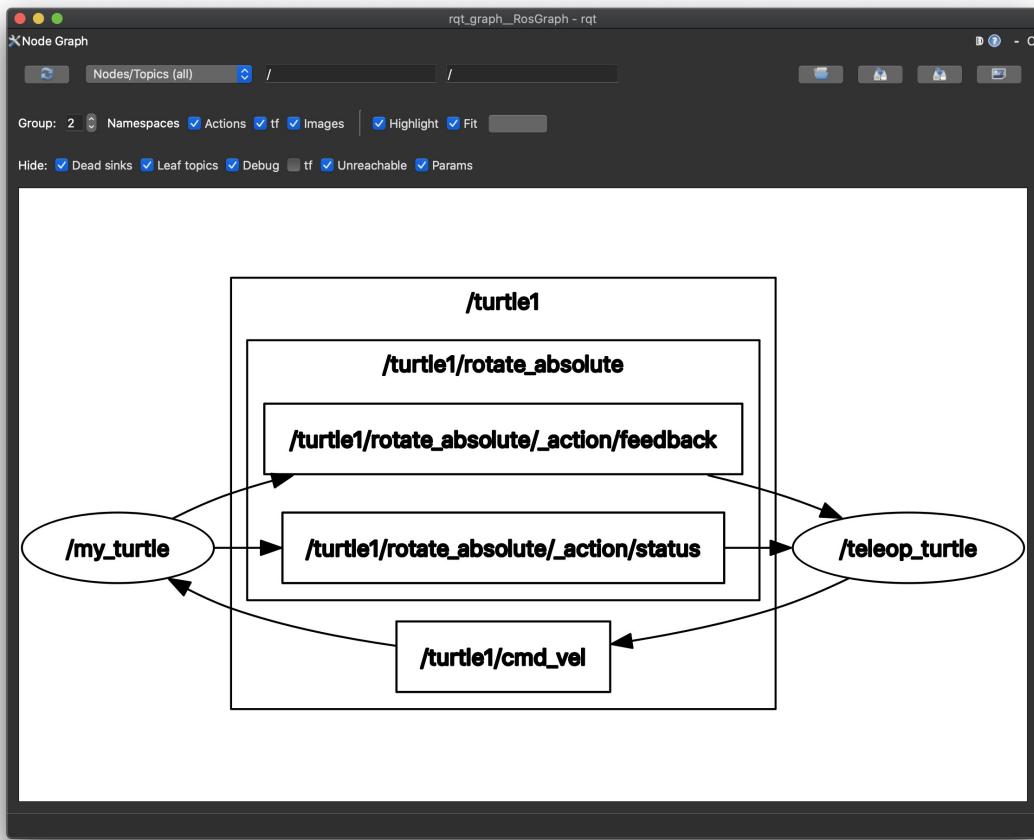


FIGURE 10.7 – Graphe de l'application **turtlesim** et **turtle_teleop_key**

10.2.6 rqt_graph

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

```
1 rqt_graph
```

Il est également possible d'ouvrir rqt_graph en ouvrant **rqt** et en sélectionnant **Plugins > Introspection > Node Graph**.

Il est possible de voir les nodes et les topics comme dans la figure fig. 10.7. On y voit deux actions à la périphérie du graphe. Lorsque vous passez votre souris sur les topics au centre, des couleurs apparaissent.

Le graphe montre comment le node **/turtlesim** et le node **teleop_turtle** communiquent ensemble à travers un topic. Le node **teleop_turtle** publie des données (les pressions sur le clavier que vous effectuez pour bouger la tortue) sur le topic **/turtle1/cmd_vel**, et le node **/turtlesim** a souscrit à ce topic pour recevoir des données.

Les mécanismes de mise en évidence de rqt_graph sont très utiles pour examiner des systèmes plus complexes avec beaucoup de nodes et des topics connectés de façons très différentes.

Pour démarrer le graphe de l'affichage des topics :

```
1 ros2 run rqt_plot rqt_plot
```

On obtient le graphe affiché dans la figure Fig.10.8.

10.2.7 Topic

Les topics sont des données publiées par des noeuds et auxquelles les noeuds souscrivent. L'exécutable permettant d'avoir des informations sur les topics est **ros2 topic**.

La liste des commandes en ROS-2 est similaire à celle de ROS-1 :

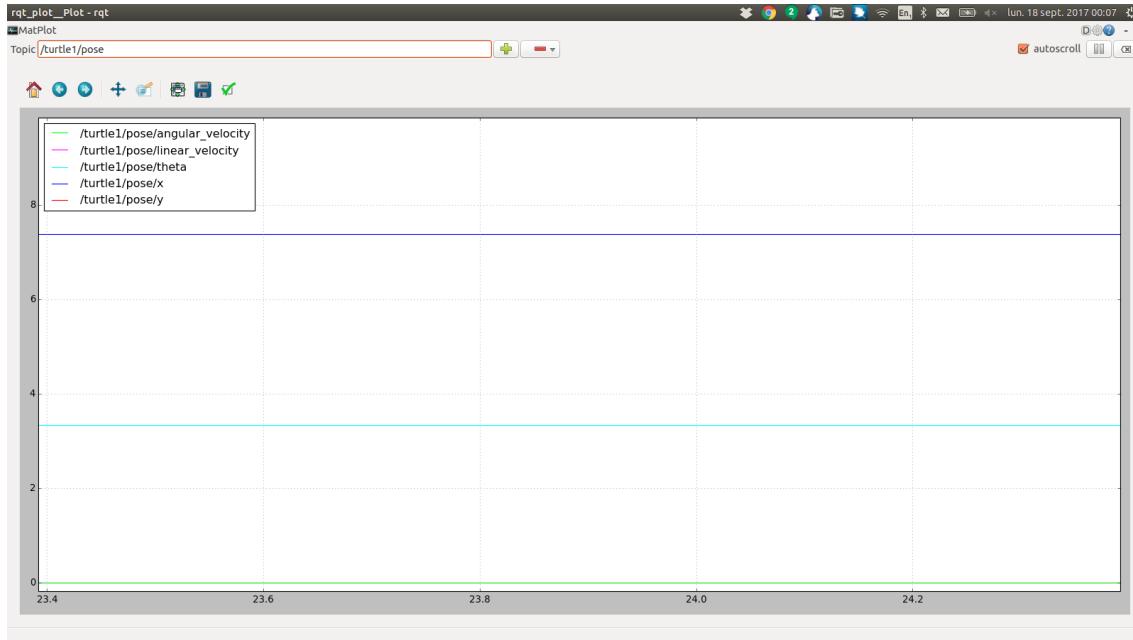


FIGURE 10.8 – Affichage de la position de la tortue dans la fenêtre

bw : Affiche la bande passante prise par le topic
echo : Affiche le contenu du topic en texte
hz : Affiche la fréquence de publication du topic
info : Fournit des informations sur un topic
list : Affiche la liste des topics actifs
pub : Publie des données sur le topic
type : Affiche le type du topic

10.2.7.1 ros2 topic list

Cette commande affiche la liste des topics.

```
1 ros2 topic list
```

Le résultat est le suivant :

```
1 /parameter_events
2 /rosout
3 /turtle1/cmd_vel
4 /turtle1/color_sensor
5 /turtle1/pose
```

La commande suivante

```
1 ros2 topic list -t
```

va retourner la même liste de topics, mais cette fois avec le type de topic ajouté et entourer de crochets.

```
1 /parameter_events [rcl_interfaces/msg/ParameterEvent]
2 /rosout [rcl_interfaces/msg/Log]
3 /turtle1/cmd_vel [geometry_msgs/msg/Twist]
4 /turtle1/color_sensor [turtlesim/msg/Color]
5 /turtle1/pose [turtlesim/msg/Pose]
```

Ces attributs, et plus particulièrement les types, montrent comment les nodes savent qu'ils échangent la même information alors que celle-ci est transmise à travers les topics.

Si vous demandez où se trouvent tous ces topics dans rqt_graph, vous pouvez décocher toutes les boîtes sous Hide.

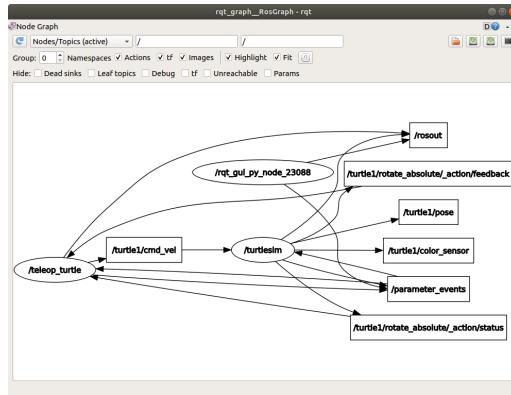


FIGURE 10.9 – Fenêtre de rqt_graph en décochant les boîtes à droite de l’option **Hide**

10.2.7.2 ros2 topic bw

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

```
1 ros2 topic bw /turtle1/pose
```

On obtient alors le résultat suivant :

```
1 39 B/s from 52 messages
2   Message size mean: 24 B min: 24 B max: 24 B
3 38 B/s from 52 messages
4   Message size mean: 24 B min: 24 B max: 24 B
```

10.2.7.3 ros2 topic echo

Pour afficher le contenu d'un topic en fonction du message transmis on peut utiliser la commande :

```
1 ros2 topic echo <topic_name>
```

Par exemple, puisque nous savons que le topic **/teleop_turtle** publie des données pour **/turtlesim** sur le topic **/turtle1/cmd_vel**, utilisons la commande **echo** pour faire de l'introspection sur ce topic :

```
1 ros2 topic echo /turtle1/cmd\_\_vel
```

Cette commande ne publiera rien au départ. C'est parce que le topic attend que **/teleop_turtle** publie quelque chose.

Il faut donc mettre le focus sur le terminal où le node **turtle_teleop_key** s'exécute et utiliser les flèches pour bouger la tortue autour. En regardant le terminal où la commande **echo** s'exécute, on peut voir les données être publiées pour chaque mouvement effectué :

```
1 linear:
2   x: 2.0
3   y: 0.0
4   z: 0.0
5 angular:
6   x: 0.0
7   y: 0.0
8   z: 0.0
9 ---
```

On peut alors retourner sur la fenêtre rqt_graph et désélectionner la boîte **Debug**.

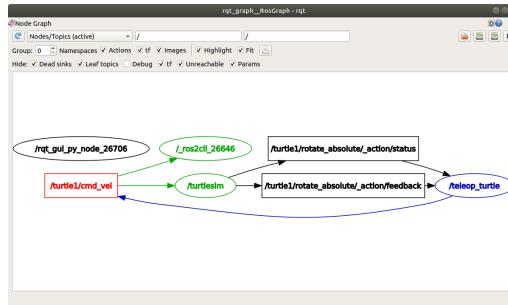
Le node **_ros2cli_26646** est le node créé par la commande **echo** (le nombre peut-être différent). On peut voir maintenant que le publisher publie des données sur le topic **/turtle1/cmd_vel**, et qu'il y a deux souscripteurs à ce topic.

10.2.7.4 ros2 topic hz

Cette commande permet d'afficher la fréquence à laquelle les données sont publiées :

```
1 ros2 topic hz /turtle1/pose
```

avec

FIGURE 10.10 – Fenêtres correspondant à **rqt_graph** avec le mode Debug

```

1 average rate: 47.815
2 min: 0.000s max: 2.621s std dev: 0.10933s window: 576
3 average rate: 49.036
4 min: 0.000s max: 2.621s std dev: 0.10374s window: 640

```

10.2.7.5 ros2 topic info

Affiche des informations sur un topic c'est à dire le type, le nombre de publishers et le nombre de subscribers. Par exemple :

```
1 ros2 topic info /turtle1/pose
```

donne :

```

1 Type: turtlesim/msg/Pose
2 Publisher count: 2
3 Subscription count: 0

```

10.2.7.6 ros2 interface show

Les nodes envoient des données par les topics en utilisant les messages. Les publishers et les subscribers doivent envoyer et recevoir les même types de messages pour communiquer.

Les types de topic que nous avons vu précédemment après avoir exécuté **ros2 topic list -t** nous permettent de connaître le type de message a utilisé sur chaque topic. Ainsi le topic **cmd_vel** a le type :

```
1 geometry_msgs/msg/Twist
```

Ceci spécifie que dans le paquet **geometry_msgs** il y a un **msg** appelé **Twist**.

Il est possible d'exécuter **ros2 interface show <msg type>** sur ce type pour le connaître en détails, spécifiquement la structure des données que le message attend.

```
1 ros2 interface show geometry_msgs/msg/Twist
```

Le résultat est le suivant :

```

1 # This expresses velocity in free space broken into its linear and angular parts.
2 Vector3 linear
3 Vector3 angular

```

Cette information nous indique le node **/turtlesim** attend un message avec deux vecteurs, **linear** et **angular** de trois éléments chacun. Cela correspond aux données vues passant du node **/teleop_turtle** au node **/turtlesim** avec la commande **echo**.

10.2.7.7 ros2 topic type

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

```
1 ros2 topic type /turtle1/pose
```

affiche le type du topic (appelé message) :

```
1 turtlesim/msg/Pose
```

10.2.7.8 Afficher la structure d'un topic (message)

Pour afficher la structure d'un message il faut utiliser *ros2 interface show* :

```
1 ros2 interface show turtlesim/msg/Pose
```

ce qui affiche :

```
1 float32 x
2 float32 y
3 float32 theta
4
5 float32 linear_velocity
6 float32 angular_velocity
```

10.2.7.9 ros2 topic pub

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

```
1 ros2 topic pub <topic_name> <msg_type> '<args>'
```

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

```
1 ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "[Linear: {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}]"
```

Cette commande demande à la tortue de tourner sur elle-même à la vitesse angulaire de 1.8 rad/s . Grâce à la compléction 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 .

10.3 Service

Les services sont une autre façon pour les noeuds de communiquer entre eux. Les services permettent aux noeuds d'envoyer des *requêtes* et de recevoir des *réponses*. Tandis que les topics permettent aux nodes de souscrire aux flux de données

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 **ros2 service** permet d'interagir facilement avec le système client/serveur de ROS appelés *services*. **ros2 service** a plusieurs commandes qui peuvent être utilisées sur les services comme le montre l'aide en ligne :

- call : Appelle le service
- find : Trouve une liste de services disponible pour un type spécifique
- list : Liste des services actifs
- type : Affiche le type d'un service

10.3.1 ros2 service list

La commande

```
1 ros2 service list
```

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

```
1 /clear
2 /kill
3 /reset
4 /spawn
5 /turtle1/set_pen
6 /turtle1/teleport_absolute
7 /turtle1/teleport_relative
8 /turtlesim/describe_parameters
9 /turtlesim/get_parameter_types
10 /turtlesim/get_parameters
11 /turtlesim/list_parameters
12 /turtlesim/set_parameters
13 /turtlesim/set_parameters_atomically
```

On y trouve 9 services fournis par le node **turtlesim** : clear, kill, reset, spawn, turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/describe_parameters, turtlesim/get_parameters, turtlesim/list_parameters, turtlesim/set_parameters, turtlesim/set_parameters_atomically. Les 6 services relatifs aux paramètres se retrouvent avec tous les nodes.

Les deux services relatifs au node **rosout** : /rosout/get_loggers and /rosout/set_logger_level n'existent plus en ROS-2.

10.3.2 ros2 service type (service)

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

```
1 ros2 service type <service>
```

Ce qui donne pour le service clear :

```
1 ros2 service type /clear
```

Le résultat est le suivant :

```
1 std_srvs/srv/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 lorsqu'une requête est effectuée et aucune donnée n'est transmise lors de la réponse).

10.3.3 ros2 service list -t

Pour voir tous les types des services actifs en même temps, on peut ajouter l'option **-show-types**, dont l'abréviation est **-t** à la commande **list** :

```
1 ros2 service list -t
```

Le résultat est le suivant :

```
1 /clear [std_srvs/srv/Empty]
2 /kill [turtlesim/srv/Kill]
3 /reset [std_srvs/srv/Empty]
4 /spawn [turtlesim/srv/Spawn]
5 ...
6 /turtle1/set_pen [turtlesim/srv/SetPen]
7 /turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
8 /turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
```

10.3.4 ros2 interface show

Pour obtenir la structure de la requête et de la réponse on peut utiliser la commande **interface**. Si on souhaite afficher cette structure pour le service **turtlesim/srv/Spawn** :

```
1 ros2 interface show turtlesim/srv/Spawn
```

le résultat est :

```
1 float32 x
2 float32 y
3 float32 theta
4 string name
5 ---
6 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. Le type de la réponse est spécifié après le séparateur définit par "—".

Voyons maintenant comment faire appel à un service.

10.3.5 ros2 service call

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

```
1 ros2 service call [service] [args]
```

Pour effacer la fenêtre graphique on va utiliser le service `/clear`. On a vu précédemment que le type est **Empty**. Pour voir sa structure on peut faire :

```
| ros2 interface show std_srvs/srv/Empty
```

Le résultat est le suivant :

```
| ---
```

Comme nous aurions pu le deviner grâce au nom, ceci signifie que le service n'a ni argument d'entrée ni argument de sortie.

On peut donc l'appeler de la manière suivante :

```
| ros2 service call /clear
```

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

```
| ros2 service type /spawn
```

Le résultat est :

```
| turtlesim/srv/Spawn
```

Ce service nous permet de lancer une autre tortue, par exemple :

```
| ros2 service call /spawn turtlesim/srv/Spawn "x:0.0, y:0.0, theta: 0.0, name: "my_turtle""
```

On trouve maintenant une deuxième tortue dans la fenêtre comme représenté dans la figure Fig.10.11.



FIGURE 10.11 – Fenêtre résultat de ros2 run call /spawn turtlesim/srv/Spawn "x :0.0, y :0.0, theta : 0.0, name : 'my_turtle'"

10.4 Paramètres

ROS-2 gère les paramètres qui permet de stocker des données. Contrairement à ROS-1 les paramètres ne sont pas centralisés mais répartis sur chacun des nodes.

Les mécanismes de synchronisation des paramètres est complètement différent des topics qui implémentent 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. Les paramètres utilisent le langage YAML pour la syntaxe de ces paramètres. Dans des cas simples, YAML paraît très naturel.

Les commandes suivantes permettent de gérer les paramètres :

- delete : Efface un paramètre
- describe : Montre les informations décrivant le paramètre
- dump : Sauve les paramètres dans un fichier
- get : Affiche la valeur d'un paramètre
- list : Donne la liste des paramètres
- set : Spécifie la valeur d'un paramètre

La suite des commandes suppose que les deux nodes de l'exemple turtlesim sont lancés :

```
1 ros2 run turtlesim turtlesim_node
2 ros2 run turtlesim turtle_teleop_key
```

10.4.1 ros2 param list

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

```
1 ros2 param list
```

On obtient alors la liste suivante :

```
1 /teleop_turtle:
2   scale_angular
3   scale_linear
4   use_sim_time
5 /turtlesim:
6   background_b
7   background_g
8   background_r
9   use_sim_time
```

Chaque paramètre est dans le namespace d'un node. On trouve donc deux espaces de noms : `teleop_turtle` et `/turtlesim`.

Les 3 premiers paramètres de `turtlesim` 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 :

```
1 ros2 param set /turtlesim background_r 150
```

Le résultat est représenté dans la figure 10.12.

On peut voir la valeur des autres paramètres. Par exemple, pour avoir la valeur du canal vert de la couleur de fond on peut utiliser :

```
1 ros2 param set /background_g
```

10.4.2 ros2 param dump

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

```
1 ros2 param dump /turtlesim
```

Le résultat est le fichier `turtlesim.yaml` :

```
1 turtlesim:
2   ros__parameters:
3     background_b: 255
4     background_g: 86
5     background_r: 150
6     use_sim_time: false
```

Il est ensuite possible de recharger ces paramètres.



FIGURE 10.12 – Résultat de l'appel **ros2 param set /turtlesim background_r 150**

10.4.3 ros2 param load

La structure générale de la commande est :

```
ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
```

Pour charger le fichier :

```
ros2 run turtlesim turtle_sim_node --ros-args --params-file params.yaml
```

10.5 Actions

Cette section correspond au didacticiel Introduction aux actions

10.5.1 Introduction

Les actions sont l'un des types de communications de ROS-2 et sont utilisées pour des tâches relativement longues. Elles sont constituées de trois parties : un but, un retour d'état, et un résultat.

Les actions sont construites sur les topics et les services. Leurs fonctionnalités sont similaires aux services, excepté que les actions peuvent être annulées. Elles fournissent également un retour constant sur l'état à l'opposé des services qui ne retournent qu'une seule réponse.

Les actions utilisent un modèle client-serveur, similaire au modèle publisher-subscriber. Un node "action client" envoie un but à un node serveur d'action "action server" qui accueille la réception du but et retourne un flux de retour d'état et un résultat.

10.5.2 Préparation

Il faut démarrer les deux nodes de turtlesim : **/turtlesim** et **/teleop_turtle**. Pour cela il faut démarrer un nouveau terminal et démarrer :

```
ros2 run turtlesim turtle_sim_node
```

```
ros2 run turtlesim turtle_teleop_key
```

10.5.3 Utilisation des actions

Quand le node **/teleop_turtle** est lancé, il est possible de voir le message suivant :

```
1 Use arrow keys to move the turtle.
2 Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
```

La deuxième ligne correspond aux actions. (La première ligne correspond au topic "cmd_vel" traité dans les didacticiels précédents).

On peut noter que les lettres **G|B|V|C|D|E|R|T** forment une boîte autour de la lettre **F** sur un clavier US QWERTY. Chaque touche autour de **F** correspond à un orientation pour la tortue de la fenêtre graphique. Par exemple, la lettre **E** fait tourner la tortue vers le coin en haut à gauche.

Il est intéressant de regarder le terminal où le node **/turtlesim** est exécuté. A chaque fois que l'on presse une des touches, un goal est envoyé au serveur d'action (action server) qui fait partir du node **/turtlesim**. Le but est de faire tourner la tortue dans une direction particulière. Un message affichant le résultat du goal s'affiche une fois que la tortue est effectuée :

```
1 [INFO] [turtlesim]: Rotation goal completed successfully
```

La touche **F** annule le goal en cours d'exécution.

On peut par exemple appuyer sur **C** puis presser la touche **F** avant que la tortue puisse terminer sa rotation. Dans le terminal où le node **/turtlesim** s'exécute, on peut voir le message :

```
1 [INFO] [turtlesim]: Rotation goal canceled
```

Non seulement il est possible pour le client d'annuler un goal (par une entrée dans teleop) mais le serveur (le node **/turtlesim**) peut également. Quand le service choisit de stopper le goal, on dit qu'il "avorte" le goal.

On peut par exemple appuyer sur **D**, puis la touche **G** avant que la première rotation ne soit terminée. Dans le terminal où le node **/turtlesim** s'exécute on peut voir le message :

```
1 [WARN] [turtlesim]: Rotation goal received before a previous goal finished. Aborting previous goal
```

Le serveur d'action (action server) choisit d'avorter le premier but car il en a reçu un nouveau. Il aurait pu choisir une autre stratégie comme rejeter le nouveau but ou exécuter le second but après que le premier soit fini. On ne peut pas supposer que chaque serveur d'action va choisir d'annuler le but courant quand il en reçoit un nouveau.

10.5.4 ros2 node info

Pour voir les actions du node **/turtlesim**, il faut ouvrir un nouveau terminal et lancer la commande :

```
1 ros2 node info /turtlesim
```

Cette commande retourne la liste des subscribers, publishers, services, serveurs d'action et clients d'action pour le node **/turtlesim** :

```
1 /turtlesim
2   Subscribers:
3     /parameter_events: rcl_interfaces/msg/ParameterEvent
4     /turtle1/cmd_vel: geometry_msgs/msg/Twist
5   Publishers:
6     /parameter_events: rcl_interfaces/msg/ParameterEvent
7     /rosout: rcl_interfaces/msg/Log
8     /turtle1/color_sensor: turtlesim/msg/Color
9     /turtle1/pose: turtlesim/msg/Pose
10  Service Servers:
11    /clear: std_srvs/srv/Empty
12    /kill: turtlesim/srv/Kill
13    /reset: std_srvs/srv/Empty
14    /spawn: turtlesim/srv/Spawn
15    /turtle1/set_pen: turtlesim/srv/SetPen
16    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
17    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
18    /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
19    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
20    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
21    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
22    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
23    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
24  Service Clients:
25
26  Action Servers:
27    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
28  Action Clients:
```

On peut noter que l'action **/turtle1/rotate_absolute** pour **/turtlesim** est dans la liste **Action Servers**. Ce qui signifie que **/turtlesim** répond et fournit le retour d'état pour l'action **/turtle1/rotate_absolute**.

Le node **/teleop_turtle** a le nom **/turtle1/rotate_absolute** dans la liste **Action Clients** ce qui signifie qu'il envoie des buts pour cette action.

```
1 ros2 node info /teleop_turtle
```

ce qui retourne :

```
1 /teleop_turtle
2   Subscribers:
3     /parameter_events: rcl_interfaces/msg/ParameterEvent
4   Publishers:
5     /parameter_events: rcl_interfaces/msg/ParameterEvent
6     /rosout: rcl_interfaces/msg/Log
7     /turtle1/cmd_vel: geometry_msgs/msg/Twist
8   Service Servers:
9     /teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
10    /teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
11    /teleop_turtle/get_parameters: rcl_interfaces/srv/GetParameters
12    /teleop_turtle/list_parameters: rcl_interfaces/srv/ListParameters
13    /teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
14    /teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
15   Service Clients:
16
17   Action Servers:
18
19   Action Clients:
20     /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

10.5.5 ros2 action list

Pour afficher toutes les actions dans un graphe ROS, il faut exécuter la commande :

```
1 ros2 action list
```

ce qui retourne :

```
1 /turtle1/rotate_absolute
```

C'est la seule action dans le graphe ROS disponible pour le moment. Elle contrôle la rotation de la tortue comme nous l'avons vu précédemment. Il est également possible de savoir qu'il y a un seul action client (qui fait partie de **/teleop_turtle**) et un serveur d'action (qui fait partie de **/turtlesim**) pour cette action en utilisant la commande **ros2 node info <node_name>**.

10.5.5.1 ros2 action list -t

Les actions ont des types, de la même manière que les topics et les services. Pour trouver le type de **/turtle1/rotate_absolute** il faut lancer la commande :

```
1 ros2 action list -t
```

ce qui retourne :

```
1 /turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

Dans les crochets à la droite du nom de chaque action (dans ce cas **/turtle1/rotate_absolute**) est le type de l'action : **/turtlesim/action/RotateAbsolute**. Cette information est nécessaire quand il faut exécuter une action à partir de la ligne de commande ou à partir de code.

10.5.6 ros2 action info

Il est possible d'introspecter avec plus de détails l'action **/turtle1/rotate_absolute** avec la commande :

```
1 ros2 action info /turtle1/rotate_absolute
```

ce qui retourne :

```
1 Action: /turtle1/rotate_absolute
2 Action clients: 1
3   /teleop_turtle
4 Action servers: 1
5   /turtlesim
```

Ceci nous donne de façon concise les informations obtenues avec la commande **ros2 node info** sur chacun des nodes : Le node **/teleop_turtle** un action client et le node **/turtlesim** est un serveur d'action de l'action **/turtle1/rotate_absolute**.

10.5.7 ros2 interface show

Afin d'exécuter un but d'action (action goal) il faut connaître la structure d'une action. Rappelons qu'il est possible d'identifier le type de l'action **/turtle1/rotate_absolute** lorsque l'on exécute la commande **ros2 action list -t**. Il faut pour cela entrer la commande suivante avec le type de l'action dans le terminal :

```
1 ros2 interface show turtlesim/action/RotateAbsolute
```

Ce qui retourne :

```
1 # The desired heading in radians
2 float32 theta
3 ---
4 # The angular displacement in radians to the starting position
5 float32 delta
6 ---
7 # The remaining rotation in radians
8 float32 remaining
```

La première section de ce message au-dessus de — est la structure (type de données et nom) de la requête du but. La partie suivante est la structure du résultat. La dernière partie est la structure du retour d'état.

10.5.8 ros2 action send_goal

Il est possible d'envoyer un but d'action (action goal) à partir la ligne de commande avec la syntaxe suivante :

```
1 ros2 action send_goal <action_name> <action_type> <values>
```

avec **<values>** doit être au format YAML.

En gardant un oeil sur la fenêtre turtlesim, entrer la commande suivante dans un terminal :

```
1 ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}"
```

La tortue doit se mettre à tourner, et on doit voir le message suivant dans le terminal :

```
1 Waiting for an action server to become available...
2 Sending goal:
3   theta: 1.57
4
5 Goal accepted with ID: f8db8f44410849eaa93d3feb747dd444
6
7 Result:
8   delta: -1.568000316619873
9
10 Goal finished with status: SUCCEEDED
```

Tous les buts ont un identifiant unique dans le message de retour. Il est possible également de voir le résultat, un champ avec le nom **delta**, qui est le déplacement à la position de départ.

Pour voir le retour de ce but, il suffit d'ajouter **--feedback** à la commande **ros2 action send_goal**.

```
1 ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: -1.57}" --feedback
```

Le terminal va retourner le message :

```
1 Sending goal:
2   theta: -1.57
3
4 Goal accepted with ID: e6092c831f994afda92f0086f220da27
5
6 Feedback:
7   remaining: -3.1268222332000732
8
9 Feedback:
10  remaining: -3.1108222007751465...
11
12
13
14 Result:
15  delta: 3.1200008392333984
16
17 Goal finished with status: SUCCEEDED
```

Le retour d'état s'effectue en donnant l'angle en radians qu'il reste à effectuer jusqu'à ce que le but soit atteint.

10.5.9 Résumé

Les actions sont comme des services qui permettent d'exécuter des tâches longues, fournissent un retour régulier, et sont annulables.

Un système robotique utilisent les actions pour de nombreux comportements : la navigation, l'exécution de contrôleurs, la planification de mouvements. Un but d'action peut dire à un robot d'aller vers une position. Tant que le robot navigue vers la position finale, il peut envoyer le long du chemin sa position (i.e. un retour) et finalement un message final lorsqu'il a atteint sa destination.

Turtlesim a un serveur d'action auquel un client d'action peut envoyer des buts pour orienter les tortues. Dans ce didacticiel l' action **/turtle1/rotate_absolute** est analysée pour avoir une meilleure idée de ce que sont les actions et comment elles fonctionnent.

10.6 rqt_console

rqt est une interface d'affichage qui ne fait pas uniquement de la 3D et 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**.

10.6.1 rqt

Démarrer *rqt* se fait simplement avec :

```
| rqt
```

La première fois où cette commande est démarrer la fenêtre peut-être vide. Il suffit de sélectionner dans Plugins > Logging > Console pour obtenir l'équivalent de :

```
| ros2 run rqt_console rqt_console
```

10.6.2 rqt_console

```
| ros2 run rqt_console rqt_console
```

La commande permet de lancer le système d'affichage des messages d'alertes et d'erreur. Cela crée la fenêtre affichée dans la figure Fig. 10.13. Pour illustrer cet exemple on peut lancer le noeud **turtlesim** avec :

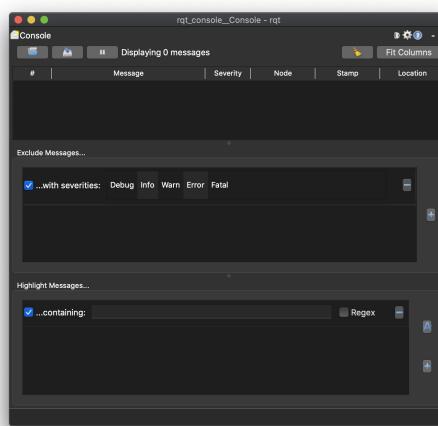


FIGURE 10.13 – Fenêtres correspondant à **rqt_console**

```
| ros2 run 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** :

```
ros2 topic pub -r 1 /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.10.14.

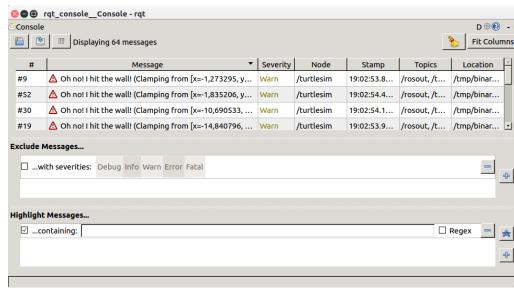


FIGURE 10.14 – Fenêtres correspondant à **rqt_console_console** lorsque la tortue s'écrase dans les murs

10.6.3 Niveaux d'enregistrement

Les niveaux d'enregistrements (logger levels) sont ordonnés par sévérité : Fatal, Error, Warn, Info, Debug. Fatal : le message indique que le système va s'arrêter pour se protéger de tout préjudice.

Error : le message indique que le système ne va pas être endommager mais qu'il ne va pas fonctionner normalement.

Warn : le message indique une activité inhabituelle ou un résultat non idéal qui peut représenter un problème plus profond mais n'impact la fonctionnalité immédiatement.

Info : le message indique des mises à jour d'évènements et de statut qui permettent de visualiser que le système fonctionne correctement.

Debug : le message fournit des informations qui détaillent le fonctionnement pas à pas du système.

Le niveau par défaut est **Info**. Seuls les messages d'un niveau de sévérité égal ou supérieur s'afficheront.

Normalement seuls les messages **Debug** seront cachés car ils sont les seuls à avoir un niveau de sévérité inférieur à **Info**. Par exemple, en fixant le niveau par défaut à **Warn**, on ne voit que les messages de sévérité **Warn, Error et Fatal**.

10.6.3.1 Fixé le niveau d'enregistrement par défaut

Il est possible de spécifier le niveau par défaut d'enregistrement au démarrage du node en utilisant les arguments d'entrée. Les messages de warning de sévérité **Info** ne s'affichent plus dans le terminal. C'est parce que ces messages de sévérité ont un niveau inférieur au niveau par défaut spécifié i.e. **Info**.

```
ros2 run turtlesim turtlesim_node --ros-args --log-level WARN
```

Cependant le mécanisme permettant de changer à la volée les niveaux de logs ne sont pas encore complètement supportés.

Tutoriel associé : Tutorial Using rqt console et roslaunch <https://docs.ros.org/en/foxy/Tutorials/Rqt-Console/Using-Rqt-Console.html> ;

10.7 Bag : fichier de données

La commande ros2 bag fournit les sous commandes suivantes :

info : Affiche des informations sur le fichier de données à la console

play : Rejoue les données enregistrées dans le fichier de données .bag

record : Enregistre des données ROS dans un fichier de données au format .bag

Il est possible d'enregistrer et de rejouer des flux de données transmis par les topics. Cela s'effectue en utilisant la commande **ros2 bag**. 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 générés sur les topics).

```
1 ros2 bag 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`.

```
1 ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose
```

Pour rejouer un fichier de données il suffit de faire :

```
1 ros2 bag play subset.bag
```

10.8 Launch : démarrer une application

10.8.1 Introduction

ros2 launch est une commande qui permet de lancer plusieurs noeuds. **ros2 launch** exécute un fichier python pour lancer plusieurs noeuds. Le fichier python doit se trouver dans un répertoire `launch`.

Dans ROS-2 les fichiers XMLs de fichier de launch ne sont plus proposés. Effectivement utiliser un fichier python permet d'avoir un meilleur contrôle sur la synchronisation du lancement des nodes de calcul.

10.8.2 Un exemple

Cet exemple se base sur l'utilisation de turtlesim. On crée tout d'abord un répertoire `launch` avec :

```
1 mkdir launch
```

Il faut créer un fichier `turtlesim_mimic_launch.py` en entrant la commande suivant :

```
1 touch launch/turtlesim_mimic_launch.py
```

Il suffit ensuite de copier coller le code python suivant :

```
1 from launch import LaunchDescription
2 from launch_ros.actions import Node
3
4 def generate_launch_description():
5     return LaunchDescription([
6         Node(
7             package='turtlesim',
8             namespace='turtlesim1',
9             executable='turtlesim_node',
10            name='sim'
11        ),
12        Node(
13            package='turtlesim',
14            namespace='turtlesim2',
15            executable='turtlesim_node',
16            name='sim'
17        ),
18        Node(
19            package='turtlesim',
20            executable='mimic',
21            name='mimic',
22            remappings=[
23                ('/input/pose', '/turtlesim1/turtle1/pose'),
24                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel')
25            ]
26        )
27    ])
```

Les deux premières lignes importent les modules python relatifs à la fonctionnalité de launch.

```

1 from launch import LaunchDescription
2 from launch_ros.actions import Node

```

La description de la structure de launch est définie dans :

```

1 def generate_launch_description():
2     return LaunchDescription([
3         ])
4

```

Dans la structure *LaunchDepscrition* on trouve 3 nodes tous faisant parti du paquet *turtlesim*. Le but étant de lancer deux fenêtres turtlesim et un node qui mime les mouvements d'une tortue vers une autre.

Les deux premières parties lance les deux fenêtres turtlesim.

```

1 Node(
2     package='turtlesim',
3     namespace='turtlesim1',
4     executable='turtlesim_node',
5     name='sim'
6 ),
7 Node(
8     package='turtlesim',
9     namespace='turtlesim2',
10    executable='turtlesim_node',
11    name='sim'
12 )

```

On peut noter que la seule différence entre les deux nodes correspond à leur valeurs d'espace de noms (namespace). Un espace de nom unique permet au système de démarrer les deux fenêtres sans conflit de noms pour les topics ou les nodes.

Les deux tortues recoivent des commandes à travers le même topic et publient leurs posent sur le même topic. Sans l'utilisation d'un namespace unique, il n'y aurait moyen de distinguer les messages pour une tortue ou l'autre.

Le node final vient également du paquet *turtlesim*, mais avec un exécutable différent *mimic*.

On peut le vérifier avec *rqt_graph* qui donne la figure Fig.10.15. On trouve également deux topics nommés */turtlesim1/turtle1/pose* et */turtlesim2/turtle1/cmd_vel*. Si on exécute :

```

1 ros2 topic list

```

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

Tutoriel associé : Tutorial Creating a launch file <https://index.ros.org/doc/ros2/Tutorials/Launch-Files/Creating-Launch-Files>

10.8.3 Préparer son package

La suite correspond au tutoriel Launching/monitoring multiple nodes with Launch <https://index.ros.org/doc/ros2/Tutorials/Launch-system/>

10.8.3.1 Paquet python

Pour les paquets python, la structure du paquet doit ressembler à :

```

1 src/
2   my_package/
3     launch/
4       setup.py
5       setup.cfg
6       package.xml

```

Pour que colcon puisse trouver les fichiers launch, il faut informer les outils de paramétrage de python qu'il y a des fichiers de launch dans le champ *data_files* de setup.

Donc pour le fichier *setup.py* il faut mettre :

```

1 import os
2 from glob import glob
3 from setuptools import setup
4
5 package_name = 'my_package'
6
7 setup(

```

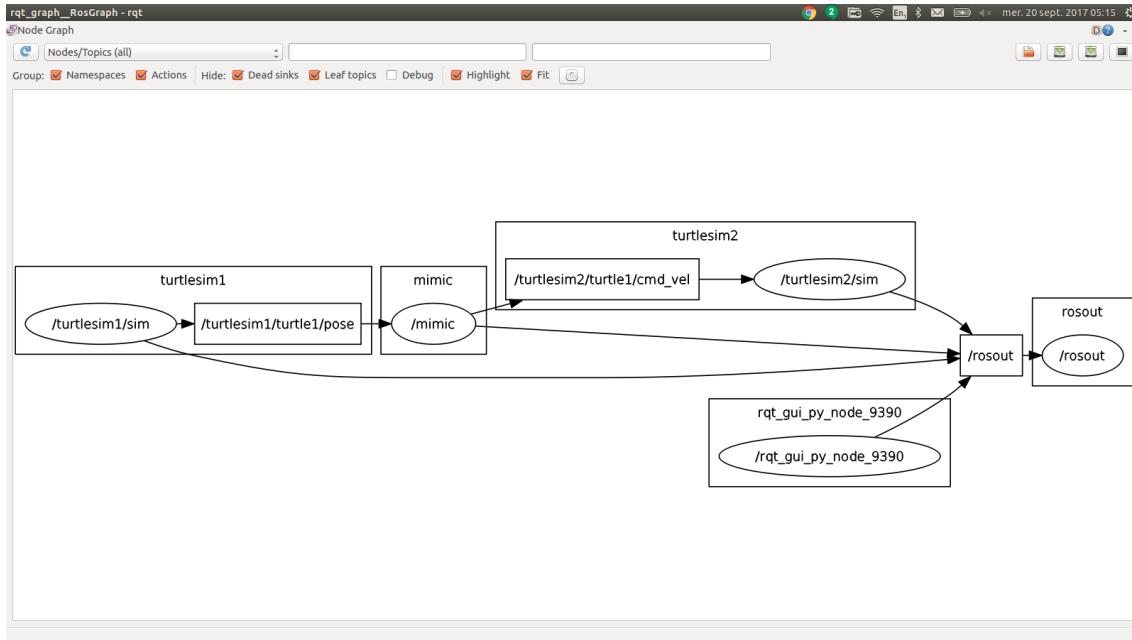


FIGURE 10.15 – Graphe correspondant au fichier turtlemimic.launch

```

8     # Other parameters ...
9     data_files=[
10         # ... Other data files
11         # Mettre tous les fichiers de launch. La ligne la plus importante est ici !
12         (os.path.join('share', package_name), glob('launch/*.launch.py'))
13     ]
14 )

```

10.8.3.2 Paquet C++

Si votre paquet est exclusivement un paquet C++, il suffit d'ajuster le fichier CMakeLists.txt en ajoutant :

```

1 # Install launch files.
2 install(DIRECTORY
3     launch
4     DESTINATION share/${PROJECT_NAME}/
5 )

```

à la fin du fichier mais avant `ament_package()`.

10.8.3.3 Ecrire le fichier de launch

Dans le répertoire launch, il faut créer le fichier launch avec le suffixe `.launch.py`. Par exemple le nom de fichier peut-être `my_script.launch.py`.

Le suffixe `.launch.py` n'est pas spécifiquement obligatoire. Une autre option populaire est d'utiliser `_launch.py` utilisé dans l'exemple précédent. Si vous souhaitez changer le suffixe, il faut s'assurer de changer l'argument `glob()` dans le fichier `setup.py`.

10.9 Création de messages et de services

10.9.1 Introduction

C'est une bonne pratique en général de réutiliser des interfaces qui existent déjà. Mais il peut s'avérer nécessaire de créer vos propres messages ou services.

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

- **msg** : msg files sont de simples fichiers textes qui donnent les champs d'un message ROS. Ils sont utilisés pour générer le code source des messages dans des langages 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 référence 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 chaîne de caractères, et deux autres messages :

```
1 Header header
2 string child_frame_id
3 geometry_msgs/PoseWithCovariance pose
4 geometry_msgs/TwistWithCovariance twist
```

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

```
1 int64 A
2 int64 B
3 ---
4 int64 Sum
```

Pour la suite nous allons créer un package ROS-2 dans le workspace `dev_ws` avec la commande suivante :

```
1 cd ~/dev_ws/src
2 ros2 pkg create --build-type ament_cmake tutorial_interfaces
```

`tutorial_interfaces` est le nom du nouveau paquet. Il s'agit d'un paquet CMake, car il n'y a pour l'instant pas moyen de générer un fichier `.msg` ou `.srv` à partir d'un paquet Python pur. Il est cependant possible de créer une interface personnalisée dans un paquet CMake puis de l'utiliser dans un paquet Python.

10.9.2 Création d'un msg

10.9.2.1 Format du fichier

Créons maintenant un msg dans le paquet `tutorial_interfaces`

```
1 mkdir msg
2 echo "int64 num" > msg/Num.msg
```

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

```
1 string first_name
2 string last_name
3 uint8 age
4 uint32 score
```

10.9.3 Création d'un srv

10.9.3.1 Format du fichier

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

```
1 mkdir srv
```

Dans ce répertoire `tutorial_interfaces/srv` nous allons maintenant créer un fichier `AddThreeInts.srv` avec la requête et la réponse suivantes :

```
1 int64 a
2 int64 b
3 int64 c
4 ---
5 int64 sum
```

Notons que la requête et la réponse sont séparées par les trois tirets.

10.9.4 CMakeList.txt

Afin de transformer les fichiers `Num.msg` et `AddThreeInts.srv` en fichier C++, Python et autres langages, il faut modifier le fichier `CMakeLists.txt` du paquet.

Il suffit de rajouter les lignes suivantes :

```
1 find_package(rosidl_default_generators REQUIRED)
2
3 rosidl_generate_interfaces(${PROJECT_NAME}
4   "msg/Num.msg"
5   "srv/AddThreeInts.srv"
6 )
```

10.9.5 package.xml

Il faut maintenant modifier le fichier `package.xml` pour que les trois lignes suivant aient été décommentées :

```
1 <build_depend>rosidl_default_generators</build_depend>
2 <exec_depend>rosidl_default_runtime</exec_depend>
3 <member_of_group>rosidl_interface_packages</member_of_group>
```

10.9.6 Génération des codes

Afin de générer les codes dans les différents langages que l'on souhaite il faut compiler le paquet en utilisant la commande suivante dans le répertoire `/dev_ws` :

```
1 colcon build --packages-select tutorial_interfaces
```

Les interfaces sont maintenant découvrables pour les autres paquets ROS-2.

10.9.7 Vérifier la création du fichier msg et srv

Dans un nouveau terminal il faut sourcer le fichier de configuration `setup.bash` de l'espace `/dev_ws`.

```
1 . install/setup.bash
```

Il est maintenant possible d'afficher le message du paquet `tutorial_interfaces` en utilisant la commande suivante :

```
1 ros2 interface show tutorial_interfaces/msg/Num
```

retourne :

```
1 int64 num
```

et la commande suivante :

```
1 ros2 interface show tutorial_interfaces/srv/AddThreeInts
```

retourne :

```
1 int64 a
2 int64 b
3 int64 c
4 ---
5 int64 sum
```

11. Ecrire des nodes ROS-2

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*.

11.1 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.

11.1.1 Crédation d'un paquet

Il faut d'abord ouvrir un terminal en s'assurant que l'environnement ROS-2 est bien sourcé de telle sorte que les commandes ROS-2 fonctionnent.

Aller ensuite dans le répertoire `dev_ws` créer précédemment. Il faut également se souvenir que les paquets doivent être créés dans le répertoire `src` et pas à la racine du répertoire.

11.1.1.1 Crédation d'un paquet C++

Il faut donc aller dans `dev_ws/src` et lancer la commande de création de paquet :

```
ros2 pkg create --build-type ament_cmake cpp_pubsub
```

Il faut naviguer jusqu'à `dev_ws/src/cpp_pubsub/src`. C'est dans ce répertoire que les fichiers sources des exécutables de tous les paquets CMake se trouvent.

11.1.1.2 Crédation d'un paquet Python

En allant dans le répertoire dans `dev_ws/src` et lancer la commande de création de paquet :

```
ros2 pkg create --build-type ament_python py_pubsub
```

Le terminal doit retourner un message confirmant la création du paquet `py_pubsub` et de tous les fichiers et répertoires nécessaires.

11.1.2 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.

11.1.2.1 C++

11.1.2.1.1 Obtenir le code

Il faut tout d'abord créer un répertoire src dans le paquet `cpp_pubsub` :

```
1 mkdir src
```

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

Il faut ensuite créer le fichier `publisher_member_function.cpp` en faisant :

```
1 wget -O publisher_member_function.cpp https://raw.githubusercontent.com/ros2/examples/master/rclcpp/topics/minimal_publisher/member_function.cpp
```

11.1.2.1.2 Le code lui-même

Le code est le suivant :

```
1 #include <chrono>
2 #include <functional>
3 #include <memory>
4 #include <string>
5
6 #include "rclcpp/rclcpp.hpp"
7 #include "std_msgs/msg/string.hpp"
8
9 using namespace std::chrono_literals;
10
11 /* This example creates a subclass of Node and uses std::bind() to register a
12 * member function as a callback from the timer. */
13
14 class MinimalPublisher : public rclcpp::Node
15 {
16 public:
17     MinimalPublisher()
18     : Node("minimal_publisher"), count_(0)
19     {
20         publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
21         timer_ = this->create_wall_timer(
22             500ms, std::bind(&MinimalPublisher::timer_callback, this));
23     }
24
25 private:
26     void timer_callback()
27     {
28         auto message = std_msgs::msg::String();
29         message.data = "Hello, world! " + std::to_string(count_++);
30         RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
31         publisher_->publish(message);
32     }
33     rclcpp::TimerBase::SharedPtr timer_;
34     rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
35     size_t count_;
36 };
37
38 int main(int argc, char * argv[])
39 {
40     rclcpp::init(argc, argv);
41     rclcpp::spin(std::make_shared<MinimalPublisher>());
42     rclcpp::shutdown();
43     return 0;
44 }
```

11.1.2.1.3 Explications

L'entête ros.h

```
6 #include "rclcpp/rclcpp.hpp"
```

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

L'entête "std_msgs/msg/string.hpp" :

```
7 #include "std_msgs/msg/string.hpp"
```

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

```
40 rclcpp::init(argc, argv);
```

initialise ROS. Contrairement à ROS-1 le nom n'est plus spécifié à ce niveau. On démarre ensuite un mécanisme de synchronisation en attente de la fin de la réalisation de la classe **MinimalPublisher**.

```
41 rclcpp::spin(std::make_shared<MinimalPublisher>());
```

Une fois que le processus de la classe est terminé, on indique au système que le Node est arrêté.

```
42 rclcpp::shutdown();
```

Le membre **publisher_** déclare l'émetteur à la ligne 34 :

```
34 rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
```

il est ensuite instancié à la ligne 20 :

```
20 publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
```

Le type du publisher est défini par le paramètre de template **std_msgs::msg::String**, le nom du topic est **topic**, et la taille de la file d'attente est définie à 10.

Le timer **timer_** déclaré à la ligne 33 :

```
33 rclcpp::TimerBase::SharedPtr timer_;
```

est instancié avec les lignes 21 et 22 :

```
21 timer_ = this->create_wall_timer(
22 500ms, std::bind(&MinimalPublisher::timer_callback, this));
```

Le timer créé appelle la méthode **MinimalPublisher::timer_callback** toutes les 500 ms.

La classe **MinimalPublisher** hérite de la classe **rclcpp::Node**. Chaque référence à **this** dans le code fait référence à la classe **Node**. La classe est instanciée avec le constructeur de la façon suivante :

```
176 public:
177 MinimalPublisher()
178 : Node("minimal_publisher"), count_(0)
```

Cela permet de nommer le Node **minimal_publisher** et d'initialiser le compteur **count_** à zéro.

La méthode **timer_callback** est celle où les données des messages sont spécifiées et où les messages sont publiés. La macro **RCLCPP_INFO** permet que chaque message publié soit affiché à la console.

11.1.2.1.4 Ajouter les dépendances dans le package.xml

Il faut aller dans le répertoire au-dessus du répertoire **src**, c'est à dire dans **dev_ws/src/cpp_pubsub** où les fichiers **CMakeLists.txt** et **package.xml** se trouvent.

Il faut ouvrir le fichier **package.xml** dans votre éditeur de texte préféré.

Il faut remplir les champs **<description>**, **<maintainer>**, et **<license>** :

```
1 <description>Examples of minimal publisher/subscriber using rclcpp</description>
2 <maintainer email="you@email.com">Your Name</maintainer>
3 <license>Apache License 2.0</license>
```

Il faut ensuite ajouter deux nouvelles lignes après la dépendance buildtool d'**ament_cmake** :

```
1 <depend>rclcpp</depend>
2 <depend>std_msgs</depend>
```

Ceci déclare la dépendance aux paquets **rclcpp** et **std_msgs**.

Il faut ensuite s'assurer d'avoir bien sauvegarder le fichier.

11.1.2.1.5 CMakeLists.txt

Il faut ensuite ouvrir le fichier `CMakeLists.txt`. En-dessous de la ligne qui déclare la dépendence à `ament_cmake` `find_package(ament_cmake REQUIRED)`, il faut ajouter :

```
1 find_package(rclcpp REQUIRED)
2 find_package(std_msgs REQUIRED)
```

Ensuite, pour obtenir les exécutables et le nommer `talker` pour pouvoir le démarrer en utilisant `ros2 run` :

```
1 add_executable(talker src/publisher_member_function.cpp)
2 ament_target_dependencies(talker rclcpp std_msgs)
```

Il faut ensuite ajouter la section d'installation pour que la commande `ros2 run` le trouve :

```
1 install(TARGETS
2   talker
3   DESTINATION lib/${PROJECT_NAME})
```

Il est ensuite possible de nettoyer le fichier `CMakeLists.txt` en enlevant les sections et commentaires inutiles de façon à ce qu'il ressemble à cela :

```
1 cmake_minimum_required(VERSION 3.5)
2 project(cpp_pubsub)
3
4 # Default to C++14
5 if(NOT CMAKE_CXX_STANDARD)
6   set(CMAKE_CXX_STANDARD 14)
7 endif()
8
9 if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
10  add_compile_options(-Wall -Wextra -Wpedantic)
11 endif()
12
13 find_package(ament_cmake REQUIRED)
14 find_package(rclcpp REQUIRED)
15 find_package(std_msgs REQUIRED)
16
17 add_executable(talker src/publisher_member_function.cpp)
18 ament_target_dependencies(talker rclcpp std_msgs)
19
20 install(TARGETS
21   talker
22   DESTINATION lib/${PROJECT_NAME})
23
24 ament_package()
```

Il est maintenant possible de construire le paquet, de sourcer l'environnement et de le lancer, mais il faut le node `subscriber` pour voir l'ensemble du système fonctionner.

11.1.2.2 Python

11.1.2.2.1 Récupérer le fichier python

Pour le code python, contrairement à ROS-1 le code python se trouve dans un répertoire qui porte le même nom que le paquet. Il faut aller dans le répertoire `py_pubsub` dans le paquet `py_pubsub` qui se trouve dans le répertoire `dev_ws/src/py_pubsub/py_pubsub`.

Il est possible de télécharger le script python du publisher `publisher_member_function.py` dans le répertoire `py_pubsub` et de le rendre executable :

```
1 wget https://raw.githubusercontent.com/ros2/examples/master/rclpy/topics/minimal_publisher/
examples_rclpy_minimal_publisher/publisher_member_function.py
```

Le fichier se trouve maintenant à côté du fichier `__init__.py`.

11.1.2.2.2 Le fichier python

```

1 import rclpy
2 from rclpy.node import Node
3
4 from std_msgs.msg import String
5
6
7 class MinimalPublisher(Node):
8
9     def __init__(self):
10         super().__init__('minimal_publisher')
11         self.publisher_ = self.create_publisher(String, 'topic', 10)
12         timer_period = 0.5 # seconds
13         self.timer = self.create_timer(timer_period, self.timer_callback)
14         self.i = 0
15
16     def timer_callback(self):
17         msg = String()
18         msg.data = 'Hello World: %d' % self.i
19         self.publisher_.publish(msg)
20         self.get_logger().info('Publishing: "%s"' % msg.data)
21         self.i += 1
22
23
24 def main(args=None):
25     rclpy.init(args=args)
26
27     minimal_publisher = MinimalPublisher()
28
29     rclpy.spin(minimal_publisher)
30
31     # Destroy the node explicitly
32     # (optional - otherwise it will be done automatically
33     # when the garbage collector destroys the node object)
34     minimal_publisher.destroy_node()
35     rclpy.shutdown()
36
37
38 if __name__ == '__main__':
39     main()

```

11.1.2.3 Explications

La première ligne importe le module python pour interagir avec la partie système de ROS-2. La deuxième ligne importe la classe Node dont va dériver la classe utilisée dans ce script python.

```

1 import rclpy
2 from rclpy.node import Node

```

La ligne suivante importe le module python qui permet d'accéder au type `String` dont le node a besoin pour construire le message à envoyer au topic.

Ces lignes représentent les dépendances du Node. Il ne faut pas oublier que ces dépendances doivent être ajoutées au fichier `package.xml`.

```

7 class MinimalPublisher(Node):

```

La classe `MinimalPublisher` hérite de la classe `Node` avec cette ligne.

```

9 def __init__(self):
10     super().__init__('minimal_publisher')
11     self.publisher_ = self.create_publisher(String, 'topic', 10)
12     timer_period = 0.5 # seconds
13     self.timer = self.create_timer(timer_period, self.timer_callback)
14     self.i = 0

```

Dans la suite, le constructeur spécifie le nom du Node à `minimal_publisher`. Puis l'appel à `create_publisher` déclare que le Node va publier des messages de type `String` (importé par le module `std_msgs.msg`) sur le topic `topic` avec une taille de file d'attente de 10. La taille de la file d'attente est un champ de QoS (Qualité de Service) qui spécifie le nombre de messages qui sont stockés si un subscriber ne les reçoit pas assez vite.

Ensuite un timer est créé avec un fonction de rappel toutes les 500 ms. `self.i` est un compteur utilisé dans la fonction de rappel.

La méthode `timer_callback` crée un message avec la valeur du compteur ajoutée, et le publie sur la console avec l'appel à `get_logger().info`.

```

16 def timer_callback(self):
17     msg = String()
18     msg.data = 'Hello World: %d' % self.i
19     self.publisher_.publish(msg)
20     self.get_logger().info('Publishing: "%s"' % msg.data)
21     self.i += 1

```

Finalement la fonction `main` est définie :

```

9 def main(args=None):
10     rclpy.init(args=args)
11
12     minimal_publisher = MinimalPublisher()
13
14     rclpy.spin(minimal_publisher)
15
16     # Destroy the node explicitly
17     # (optional - otherwise it will be done automatically
18     # when the garbage collector destroys the node object)
19     minimal_publisher.destroy_node()
20     rclpy.shutdown()

```

La librairie `rclpy` est initialisée, le Node est créé, puis une boucle d’attente d’événements est appelée avec `spin`. Ceci permet au Node d’attendre les appels du timer.

11.1.2.2.4 Ajouter les dépendances

Il faut aller dans le répertoire `dev_ws/src/py_pubsub` où les fichiers `setup.py`, `setup.cfg`, et `package.xml` ont été créés. Il faut ensuite ouvrir le fichier `package.xml` avec un éditeur de texte, et remplir les balises : `<description>`, `<maintainer>` et `<license>`.

```

1 <description>Examples of minimal publisher/subscriber using rclpy</description>
2 <maintainer email="you@email.com">Your Name</maintainer>
3 <license>Apache License 2.0</license>

```

Après la dépendance à l’outil de construction `ament_python` il faut ajouter les dépendances correspondant aux `imports` de python.

```

1 <exec_depend>rclpy</exec_depend>
2 <exec_depend>std_msgs</exec_depend>

```

Ceci déclare le besoin des paquets `rclpy` et `std_msgs` quand le code est exécuté.

11.1.2.2.5 Ajouter un point d’entrée

Il faut ouvrir le fichier `setup.py`. Il faut ensuite que les champs `maintainer`, `maintainer_email`, `description` et `license` correspondent aux champs du fichier `package.xml` :

```

1 maintainer='YourName',
2 maintainer_email='you@email.com',
3 description='Examples of minimal publisher/subscriber using rclpy',
4 license='Apache License 2.0',

```

Il faut ajouter la ligne suivante dans les crochets de `console_scripts` du champ `entry_points` :

```

1 entry_points={
2     'console_scripts': [
3         'talker = py_pubsub.publisher_member_function:main',
4     ],
5 },

```

11.1.2.2.6 Vérification de `setup.cfg`

Le contenu de `setup.cfg` doit être correctement peuplé automatiquement de la façon suivante :

```

1 [develop]
2 script-dir=$base/lib/py_pubsub
3 [install]
4 install-scripts=$base/lib/py_pubsub

```

Ceci dit simplement que setuptools met l'exécutable dans `lib`, parce que `ros2 run` les cherchera ici.

11.1.3 Souscripteur

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

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

11.1.3.1 C++

11.1.3.1.1 Obtenir le code

Il faut retourner dans le répertoire `dev_ws/src/cpp_pubsub/src` pour créer le souscripteur. Pour obtenir le code il suffit d'utiliser la commande suivante :

```

1 wget -O subscriber_member_function.cpp https://raw.githubusercontent.com/ros2/examples/master/rclcpp/topics/
      minimal_subscriber/member_function.cpp
2 \end{lstlisting}
3
4
5 \paragraph{Le code lui-même}
6 \begin{lstlisting}[language=C++]
7 #include <memory>
8
9 #include "rclcpp/rclcpp.hpp"
10 #include "std_msgs/msg/string.hpp"
11 using std::placeholders::_1;
12
13 class MinimalSubscriber : public rclcpp::Node
14 {
15     public:
16         MinimalSubscriber()
17             : Node("minimal_subscriber")
18         {
19             subscription_ = this->create_subscription<std_msgs::msg::String>(
20                 "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
21         }
22
23     private:
24         void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
25         {
26             RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
27         }
28         rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
29     };
30
31 int main(int argc, char * argv[])
32 {
33     rclcpp::init(argc, argv);
34     rclcpp::spin(std::make_shared<MinimalSubscriber>());
35     rclcpp::shutdown();
36     return 0;
37 }

```

11.1.3.1.2 Explications

Le code du souscripteur est presque identique à celui de l'émetteur. Le node est maintenant appellé `minimal_subscriber` et le constructeur du node utilise la méthode `create_subscription` pour exécuter la méthode de rappel.

Il n'y a pas de minuteur car la méthode de rappel est appelée dès qu'un message arrive.

```

7 public:
8     MinimalSubscriber()
9         : Node("minimal_subscriber")
10    {
11        subscription_ = this->create_subscription<std_msgs::msg::String>(
12            "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));

```

```
13 }
```

Rappelons qu'il faut que le nom du topic et que le type de message correspondent à la fois dans le code de l'émetteur et du souscripteur pour qu'ils puissent communiquer.

La méthode `topic_callback` reçoit la chaîne de caractères par le message de données via le topic, et l'affiche simplement dans la console grâce à la macro `RCLCPP_INFO`.

La seule déclaration de champ est celle de la souscription.

```
48 private:
49     void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
50     {
51         RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
52     }
53     rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
```

La fonction `main` est exactement la même, à part qu'il démarre le Node `MinimalSubscriber`. Pour le node émetteur l'appel à la méthode `spin` démarre le minuteur, mais le souscripteur cela signifie simplement se préparer à recevoir des messages lorsqu'ils arrivent.

Puisque ce node a les mêmes dépendances il n'y a rien à ajouter à `package.xml`.

11.1.3.1.3 CMakeLists.txt

Il faut réouvrir le fichier `CMakeLists.txt`, et ajouter l'exécutable et la cible après l'entrée de l'émetteur.

```
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)

install(TARGETS
    talker
    listener
    DESTINATION lib/${PROJECT_NAME})
```

11.1.3.2 Python

Il faut retourner dans le répertoire `dev_ws/src/py_pubsub/py_pubsub` pour créer le node émetteur.

11.1.3.2.1 Obtenir le code

```
1 wget https://raw.githubusercontent.com/ros2/examples/master/rclpy/topics/minimal_subscriber/
examples_rclpy_minimal_subscriber/subscriber_member_function.py
```

On a donc maintenant les fichiers suivants :

```
1 __init__.py publisher_member_function.py subscriber_member_function.py
```

11.1.3.2.2 Le code lui-même

Ouvrir le fichier `subscriber_member_function.py` donne le code suivant :

```
1 import rclpy
2 from rclpy.node import Node
3
4 from std_msgs.msg import String
5
6
7 class MinimalSubscriber(Node):
8
9     def __init__(self):
10         super().__init__('minimal_subscriber')
11         self.subscription = self.create_subscription(
12             String,
13             'topic',
14             self.listener_callback,
15             10)
16         self.subscription # prevent unused variable warning
17
18     def listener_callback(self, msg):
19         self.get_logger().info('I heard: "%s"', msg.data)
```

```

20
21
22 def main(args=None):
23     rclpy.init(args=args)
24
25     minimal_subscriber = MinimalSubscriber()
26
27     rclpy.spin(minimal_subscriber)
28
29     # Destroy the node explicitly
30     # (optional - otherwise it will be done automatically
31     # when the garbage collector destroys the node object)
32     minimal_subscriber.destroy_node()
33     rclpy.shutdown()
34
35 if __name__ == '__main__':
36     main()

```

11.1.3.2.3 Explications

Le code du souscripteur est quasiment identique à celui de l'émetteur. Le constructeur crée un souscripteur avec les mêmes arguments que l'émetteur. Rappelons qu'il faut que le nom du topic et que le type de message correspondent à la fois dans le code de l'émetteur et du souscripteur pour qu'ils puissent communiquer.

```

15 self.subscription = self.create_subscription(
16     String,
17     'topic',
18     self.listener_callback,
19     10)

```

Le constructeur du souscripteur et la fonction de rappel ne contiennent pas de timer parce que cela n'est pas utile. La fonction de rappel est appelée dès qu'un message est reçu.

La fonction de rappel simplement affiche un message sur la console, avec la donnée qu'elle a reçue. Rappelons que l'émetteur définit `msg.data = 'Hello World : %d' % self.i`

```

1 def listener_callback(self, msg):
2     self.get_logger().info('I heard: "%s"' % msg.data)

```

La définition de la fonction `main` est presque exactement la même, replaçant la création et l'appel à l'émetteur par le souscripteur.

```

1 minimal_subscriber = MinimalSubscriber()
2
3 rclpy.spin(minimal_subscriber)

```

Comme le node a les mêmes dépendances que l'émetteur, il n'y a rien à rajouter au fichier `package.xml`. Le fichier `setup.cfg` peut rester également non touché.

11.1.3.2.4 Rajouter un point d'entrée

Il faut réouvrir le fichier `setup.py` et ajouter le point d'entrée pour le souscripteur après le point d'entrée de l'émetteur. Le champ `entry_points` qui doit maintenant ressembler à

```

1 entry_points={
2     'console_scripts': [
3         'talker = py_pubsub.publisher_member_function:main',
4         'listener = py_pubsub.subscriber_member_function:main',
5     ],
6 }

```

Il faut s'assurer que le fichier est sauvé, et que le système pub/sub doit être prêt à l'utilisation.

11.1.4 Lancer les nodes

11.1.4.1 Dépendances

Cette partie n'est valable que sous Linux.

Les paquets `rclpy`, `std_msgs` sont normalement déjà installés sur le système ROS-2. C'est une bonne pratique de lancer `rosdep` à la racine de l'espace de travail `dev_ws` pour vérifier les dépendances avant de construire le paquet :

```
1 rosdep install -i --from-path src --rosdistro <distro> -y
```

11.1.4.2 Compiler

11.1.4.2.1 C++

Il suffit ensuite de lancer la commande suivante dans un terminal :

```
1 colcon build --packages-select cpp_pubsub
```

11.1.4.2.2 Python

Il suffit ensuite de lancer la commande suivante dans un terminal :

```
1 colcon build --packages-select py_pubsub
```

11.1.4.3 Sourcer l'environnement

Afin de pouvoir lancer les nodes il suffit de sourcer le fichier suivant :

```
1 source ./dev_ws/install/setup.bash
```

11.1.4.4 Lancer les nodes

11.1.4.4.1 C++

Dans un terminal il faut lancer la commande suivante :

```
1 ros2 run cpp_pubsub talker
```

Le terminal doit afficher les messages suivant toutes les 0.5 s :

```
1 [INFO] [minimal_publisher]: Publishing: "Hello World: 0"
2 [INFO] [minimal_publisher]: Publishing: "Hello World: 1"
3 [INFO] [minimal_publisher]: Publishing: "Hello World: 2"
4 [INFO] [minimal_publisher]: Publishing: "Hello World: 3"
5 [INFO] [minimal_publisher]: Publishing: "Hello World: 4"
6 ...
```

Dans un autre terminal il faut lancer la commande :

```
1 ros2 run cpp_pubsub listener
```

Le souscripteur va afficher les messages suivants :

```
1 [INFO] [minimal_subscriber]: I heard: "Hello World: 10"
2 [INFO] [minimal_subscriber]: I heard: "Hello World: 11"
3 [INFO] [minimal_subscriber]: I heard: "Hello World: 12"
4 [INFO] [minimal_subscriber]: I heard: "Hello World: 13"
5 [INFO] [minimal_subscriber]: I heard: "Hello World: 14"
```

Pour arrêter les nodes il suffit de taper `Ctrl+C`.

11.1.4.4.2 Python

Dans un terminal il faut lancer la commande suivante :

```
1 ros2 run py_pubsub talker
```

Le terminal doit afficher les messages suivant toutes les 0.5 s :

```
1 [INFO] [minimal_publisher]: Publishing: "Hello World: 0"
2 [INFO] [minimal_publisher]: Publishing: "Hello World: 1"
3 [INFO] [minimal_publisher]: Publishing: "Hello World: 2"
4 [INFO] [minimal_publisher]: Publishing: "Hello World: 3"
5 [INFO] [minimal_publisher]: Publishing: "Hello World: 4"
6 ...
```

Dans un autre terminal il faut lancer la commande :

```
1 ros2 run py_pubsub listener
```

Le souscripteur va afficher les messages suivants :

```
1 [INFO] [minimal_subscriber]: I heard: "Hello World: 10"
2 [INFO] [minimal_subscriber]: I heard: "Hello World: 11"
3 [INFO] [minimal_subscriber]: I heard: "Hello World: 12"
4 [INFO] [minimal_subscriber]: I heard: "Hello World: 13"
5 [INFO] [minimal_subscriber]: I heard: "Hello World: 14"
```

Pour arrêter les nodes il suffit de taper **Ctrl+C**.

11.2 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**.

Il faut d'abord ouvrir un terminal en s'assurant que l'environnement ROS-2 est bien sourcé de telle sorte que les commandes ROS-2 fonctionnent.

Aller ensuite dans le répertoire **dev_ws** créer précédemment. Il faut également se souvenir que les paquets doivent être créés dans le répertoire **src** et pas à la racine du répertoire.

11.2.1 Création d'un paquet

Il faut d'abord ouvrir un terminal en s'assurant que l'environnement ROS-2 est bien sourcé de telle sorte que les commandes ROS-2 fonctionnent.

Aller ensuite dans le répertoire **dev_ws** créer précédemment. Il faut également se souvenir que les paquets doivent être créés dans le répertoire **src** et pas à la racine du répertoire.

11.2.1.1 Création d'un paquet C++

Il faut donc aller dans **dev_ws/src** et lancer la commande de création de paquet :

```
1 ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp example_interfaces
```

11.2.1.2 Création d'un paquet python

Il faut donc aller dans **dev_ws/src** et lancer la commande de création de paquet :

```
1 ros2 pkg create --build-type ament_python py_srvcli --dependencies rclpy example_interfaces
```

11.2.1.3 Explications

L'argument **--dependencies** ajoute automatiquement les dépendances correspondantes dans le fichier **package.xml**. **example_interfaces** est le paquet qui inclut le fichier **.srv** nécessaire pour construire les requêtes et les réponses.

```
int64 a
int64 b
---
int64 sum
```

Les deux premières lignes sont les paramètres de la requête, et en dessous la réponse.

11.2.1.4 Mise à jour de **package.xml**

Parce que la commande précédent indique les dépendances il n'y a pas besoin de les ajouter dans le fichier **package.xml**. Il faut cependant s'assurer d'ajouter la description, l'email du mainteneur et son nom, et la license.

11.2.1.4.1 C++

```
<description>C++ client server tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

11.2.1.4.2 Python

```
<description>Python client server tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

11.2.2 Serveur

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.

- La première étape est l'initialisation de la librairie client ROS-2.
- La deuxième étape est de créer le Node `add_two_ints_server`.
- La troisième étape est de créer le service.
- La quatrième étape est d'attendre les requêtes des clients.

11.2.2.1 C++

11.2.2.1.1 Le code lui même

Il faut aller dans le répertoire `dev_ws/src/cpp_srvcli/src` et créer le fichier `add_two_ints_server.cpp`. Il suffit ensuite de copier coller dans le fichier le code suivant :

```
1 #include "rclcpp/rclcpp.hpp"
2 #include "example_interfaces/srv/add_two_ints.hpp"
3
4 #include <memory>
5
6 void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
7           std::shared_ptr<example_interfaces::srv::AddTwoInts::Response> response)
8 {
9     response->sum = request->a + request->b;
10    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
11                 request->a, request->b);
12    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)response->sum);
13 }
14
15 int main(int argc, char **argv)
16 {
17     rclcpp::init(argc, argv);
18
19     std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");
20
21     rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
22         node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);
23
24     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");
25
26     rclcpp::spin(node);
27     rclcpp::shutdown();
28 }
```

Les deux premières lignes d'`include` correspondent aux dépendances du paquet.

La méthode `add` ajoute deux entiers à partir de la requête et fournie la somme en retour, et notify la console de son statut en utilisant les logs.

```
1 void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
2           std::shared_ptr<example_interfaces::srv::AddTwoInts::Response> response)
3 {
4     response->sum = request->a + request->b;
5     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
6                 request->a, request->b);
7     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)response->sum);
8 }
```

La fonction `main` accomplit les lignes suivantes :

- Initialisation de la librairie client de ROS-2 :

```
17     rclcpp::init(argc, argv);
```

- Créer un node nommé `add_two_ints_server` :

```
19     std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");
```

- Crée un service nommé `add_two_ints` pour ce node et le publie à travers le réseau. La méthode de rappel est `add` :

```
21 rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
22 node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);
```

- Affiche un message de log lorsqu'il est prêt :

```
24 RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");
```

- Lance le système d'attente de requête :

```
26 rclcpp::spin(node);
```

11.2.2.1.2 Ajout de l'exécutable

La macro `add_executable` génère un exécutable que l'on peut exécuter avec `ros2 run`. Il faut ajouter le bloc suivant à `CMakeLists.txt` pour créer un exécutable nommé `server` :

```
1 add_executable(server src/add_two_ints_server.cpp)
2 ament_target_dependencies(server
3 rclcpp example_interfaces)
```

Pour que `ros2 run` trouve l'exécutable, il faut ajouter les lignes suivantes à la fin du fichier, juste avant `ament_package()` :

```
1 install(TARGETS
2   server
3   DESTINATION lib/${PROJECT_NAME})
```

11.2.2 Python

11.2.2.1 Obtenir le code

Dans le répertoire `dev_ws/src/py_srvcli/py_srvcli`, il faut créer un nouveau fichier nommé `service_member_function.py` en copiant le code suivant :

```
1 from example_interfaces.srv import AddTwoInts
2
3 import rclpy
4 from rclpy.node import Node
5
6
7 class MinimalService(Node):
8
9     def __init__(self):
10         super().__init__('minimal_service')
11         self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_two_ints_callback)
12
13     def add_two_ints_callback(self, request, response):
14         response.sum = request.a + request.b
15         self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b))
16
17         return response
18
19
20     def main(args=None):
21         rclpy.init(args=args)
22
23         minimal_service = MinimalService()
24
25         rclpy.spin(minimal_service)
26
27         rclpy.shutdown()
28
29
30 if __name__ == '__main__':
31     main()
```

11.2.2.2.2 Explications

Le premier `import` permet d'importer les types du module `AddTwoInts` du paquet `example_interfaces`. Le deuxième `import` importe la librairie client Python ROS-2 et spécifiquement la classe `Node`.

```
1 from example_interfaces.srv import AddTwoInts
2
3 import rclpy
4 from rclpy.node import Node
```

Le constructeur de la classe `MinimalService` initialise le node avec le nom `minimal_service`. Ensuite il crée un service et définit le type, nom et méthode de rappel.

```
1 def __init__(self):
2     super().__init__('minimal_service')
3     self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_two_ints_callback)
```

La définition de la méthode de rappel reçoit la requête et les données, effectue la somme et la retourne dans une structure de réponse.

```
1 def add_two_ints_callback(self, request, response):
2     response.sum = request.a + request.b
3     self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b))
4
5     return response
```

Finalement la classe principale initialise la librairie client ROS-2, instancie la classe `MinimalService` pour créer le service et lance le node pour gérer les événements.

11.2.2.3 Ajouter un point d'entrée

Pour permettre la commande `ros2 run` de lancer le node, il faut ajouter le point d'entrée au fichier `setup.py` (localisé dans le répertoire `dev_ws/src/py_srvcli`).

Il faut ajouter la ligne suivante entre les crochets '`'console_scripts'`' :

```
1 'service = py_srvcli.service_member_function:main',
```

11.2.3 Client

11.2.3.1 C++

11.2.3.1.1 Le code

Dans le répertoire `dev_ws/src/cpp_srvcli/src` il faut créer un fichier appellé `add_two_ints_client.cpp` et copier la suite dedans :

```
1 #include "rclcpp/rclcpp.hpp"
2 #include "example_interfaces/srv/add_two_ints.hpp"
3
4 #include <chrono>
5 #include <cstdlib>
6 #include <memory>
7
8 using namespace std::chrono_literals;
9
10 int main(int argc, char **argv)
11 {
12     rclcpp::init(argc, argv);
13
14     if (argc != 3) {
15         RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client X Y");
16         return 1;
17     }
18
19     std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
20     rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
21         node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");
22
23     auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
24     request->a = atol(argv[1]);
25     request->b = atol(argv[2]);
26 }
```

```

27 while (!client->wait_for_service(1s)) {
28     if (!rclcpp::ok()) {
29         RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service. Exiting.");
30         return 0;
31     }
32     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
33 }
34
35 auto result = client->async_send_request(request);
36 // Wait for the result.
37 if (rclcpp::spin_until_future_complete(node, result) ==
38     rclcpp::FutureReturnCode::SUCCESS)
39 {
40     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
41 } else {
42     RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_two_ints");
43 }
44
45 rclcpp::shutdown();
46 return 0;
47 }
```

11.2.3.1.2 Explications

De façon similaire au node de service, les lignes de code suivantes crée le node et crée le client pour ce node :

```

1 std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
2 rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
3 node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");
```

Ensuite, la requête est créée. Sa structure est définie par le fichier `.srv` mentionné précédemment :

```

24 auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
25 request->a = atol(argv[1]);
26 request->b = atol(argv[2]);
```

La boucle `while` donne au client 1 seconde pour chercher les nodes de service dans le réseau. S'il n'en trouve pas alors il va continuer à attendre.

```
32 RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
```

Alors le client envoie sa requête, et le node est en attente jusqu'à ce qu'il reçoive sa réponse ou échoue.

11.2.3.1.3 Ajouter un exécutable

Il faut retourner dans `CMakeLists.txt` pour ajouter l'exécutable et la cible pour le nouveau node. Après avoir enlever des sections et des commenaires inutilement générés automatiquement, le fichier `CMakeLists.txt` devrait ressembler à ceci :

```

1 cmake_minimum_required(VERSION 3.5)
2 project(cpp_srvcli)
3
4 find_package(ament_cmake REQUIRED)
5 find_package(rclcpp REQUIRED)
6 find_package(example_interfaces REQUIRED)
7
8 add_executable(server src/add_two_ints_server.cpp)
9 ament_target_dependencies(server
10   rclcpp example_interfaces)
11
12 add_executable(client src/add_two_ints_client.cpp)
13 ament_target_dependencies(client
14   rclcpp example_interfaces)
15
16 install(TARGETS
17   server
18   client
19   DESTINATION lib/${PROJECT_NAME})
20
21 ament_package()
```

11.2.3.2 Python

11.2.3.2.1 Le code lui-même

Dans le répertoire `dev_ws/src/py_srvcli/py_srvcli`, il faut créer un nouveau fichier nommé `client_member_function.py` et y copier le fichier suivant :

```

1 import sys
2
3 from example_interfaces.srv import AddTwoInts
4 import rclpy
5 from rclpy.node import Node
6
7
8 class MinimalClientAsync(Node):
9
10     def __init__(self):
11         super().__init__('minimal_client_async')
12         self.cli = self.create_client(AddTwoInts, 'add_two_ints')
13         while not self.cli.wait_for_service(timeout_sec=1.0):
14             self.get_logger().info('service not available, waiting again...')
15         self.req = AddTwoInts.Request()
16
17     def send_request(self):
18         self.req.a = int(sys.argv[1])
19         self.req.b = int(sys.argv[2])
20         self.future = self.cli.call_async(self.req)
21
22
23 def main(args=None):
24     rclpy.init(args=args)
25
26     minimal_client = MinimalClientAsync()
27     minimal_client.send_request()
28
29     while rclpy.ok():
30         rclpy.spin_once(minimal_client)
31         if minimal_client.future.done():
32             try:
33                 response = minimal_client.future.result()
34             except Exception as e:
35                 minimal_client.get_logger().info(
36                     'Service call failed %r' % (e,))
37             else:
38                 minimal_client.get_logger().info(
39                     'Result of add_two_ints: for %d + %d = %d' %
40                     (minimal_client.req.a, minimal_client.req.b, response.sum))
41             break
42
43     minimal_client.destroy_node()
44     rclpy.shutdown()
45
46
47 if __name__ == '__main__':
48     main()
```

11.2.3.2.2 Explications du code

Le seul `import` différent pour le client est `import sys`. Le code du node client utilise `sys.argv` pour avoir accès à la ligne de commande afin d'extraire les arguments pour la requête.

La définition du constructeur crée un client avec le même type et nom que le node de service. Le type et nom doivent être les mêmes pour le client et le service pour être capable de communiquer.

La boucle `while` dans le constructeur vérifie si un service avec le même nom et le même type sont disponibles chaque seconde.

En dessous du constructeur se trouve la définition de la requête, suivi de `main`.

11.2.3.2.3 Ajouter un point d'entrée

Comme le node de service, vous devez aussi ajouter un point d'entrée pour être capable d'exécuter le node client.

Le champ `entry_points` du fichier `setup.py` qui doit ressembler à ceci :

```
entry_points={
    'console_scripts': [
```

```
        'service = py_srvcl.service_member_function:main',
        'client = py_srvcl.client_member_function:main',
    ],
},
```

11.2.4 Lancer les nodes

11.2.4.1 Dépendances

Cette partie n'est valable que sous Linux.

Les paquets `rclpy`, `std_msgs` sont normalement déjà installés sur le système ROS-2. C'est une bonne pratique de lancer `rosdep` à la racine de l'espace de travail `dev_ws` pour vérifier les dépendances avant de construire le paquet :

```
rosdep install -i --from-path src --rosdistro <distro> -v
```

11.2.4.2 Compiler

11.2.4.2.1 C++

Il suffit ensuite de lancer la commande suivante dans un terminal :

```
colcon build --packages-select cpp srvcli
```

11.2.4.2.2 Python

Il suffit ensuite de lancer la commande suivante dans un terminal :

```
colcon build --packages-select py_srvcli
```

11.2.4.3 Sourcer l'environnement

Afin de pouvoir lancer les nodes il suffit de sourcer le fichier suivant :

```
source /dev_ws/install/setup.bash
```

11.2.4.4 Lancer les nodes

113441_CU

Dans un terminal il faut lancer la commande suivante :

21

Le terminal doit retourner le message suivant et ensuite attendre :

[INFO] [yamlmap]: Ready to add two into

Il faut ouvrir un autre terminal,源源不断地从配置文件夹中读取配置文件。在目录 `dev_ws` 中。之后，启动客户端节点，后面跟着两个空格分隔的数字：

www.scribd.com/doc/1313123/23

Si vous choisissez 2 ou 3 par exemple, le client doit recevoir une réponse comme :

Summary

En retournant dans le terminal où le node de service fonctionne, les messages de logs indiquent les requêtes et les données reçues, et la réponse renvoyée.

```
[INFO] [rclcpp]: Incoming request  
a: 2 b: 3  
[INFO] [rclcpp]: sending back response: [5]
```

Les nodes sont arrêtés avec **Ctrl+C**

11.2.4.4.2 Python

Dans un terminal il faut lancer la commande suivante :

```
1 ros2 run py_srvcli service
```

Le node attend la requête d'un client.

Il faut alors ouvrir un autre terminal sourcer les fichiers de configuration dans le répertoire de travail `dev_ws`. Il faut démarrer le node client suivi de deux entiers séparés par un espace :

```
1 ros2 run py_srvcli client 2 3
```

Si vous choisissez 2 ou 3 par exemple, le client va recevoir une réponse qui ressemble à :

```
1 [INFO] [minimal_client_async]: Result of add_two_ints: for 2 + 3 = 5
```

Il faut retourner au terminal où le node de service a été lancé. Celui affiche un message correspondant à la requête reçue :

```
1 [INFO] [minimal_service]: Incoming request  
2 a: 2 b: 3
```

Il faut utiliser `Ctrl+C` pour arrêter les nodes.



Tutoriaux sur TIAGo

12	TIAGo	151
12.1	Installation	
13	Contrôle	153
13.1	Téléopérer la base mobile avec le clavier	
13.2	Faire bouger la base à travers des commandes en vitesse	
13.3	Contrôleur des trajectoires articulaires	
13.4	Bouger des articulations individuellement	
13.5	Contrôle de la tête	
13.6	Rejouer des mouvements du haut du corps prédéfinis	
14	TIAGo Navigation	163
14.1	Créer une carte avec gmapping	
14.2	Localisation et planification de mouvement	
15	Planification de mouvements	169
15.1	Planifier dans l'espace de configuration	
15.2	Planifier dans l'espace cartésien	
15.3	Planification avec Octomap	
15.4	Démonstration d'une prise et d'un replacement d'un objet	

12. TIAGO

12.1 Installation

12.1.1 Utilisation de la machine VMWare

L'installation des tutoriaux est faite dans l'espace de travail **tiago_public_ws**. Afin de prendre en compte les particularités de Noetic, la compilation de toutes les repositories est suivie de l'installation des paquets dans le répertoire **install** de l'espace de travail **tiago_public_ws**.

Le fichier **.bashrc** a été modifié de façon à refléter cette modification.

13. Contrôle

13.1 Téléopérer la base mobile avec le clavier

La version originale de ce tutorial est disponible en anglais [ici](#).

13.1.1 But

Le but de ce tutorial est de montrer que l'on peut faire bouger un robot de la même façon que les tortues dans leur interface graphique (voir la section 3.1.3 pour ROS-1 et la section ?? pour ROS-2).

Lancer deux consoles et assurez vous que votre environnement est bien configuré (cf Section 12.1).

13.1.2 Lancer la simulation

Cette simulation lance un robot TIAGo version Titanium (c'est avec une main noire) et un environnement de bureau avec des gens :

```
1 roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=titanium world:=simple_office_with_people
```

Pour Noetic avec la VMWare il vaut mieux utiliser le TIAGo version Steel avec la commande suivante :

```
1 roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel world:=simple_office_with_people
```

L'environnement chargé par la simulation est représenté dans la figure Fig.13.1.

13.1.3 Faire bouger le robot

Pour faire bouger le robot d'avant en arrière avec les touches, il faut lancer la commande suivante :

```
1 rosrun key_teleop key_teleop.py
```

La commande s'effectue grâce aux touches du clavier avec les flèches (haut, bas, gauche,droite).

13.2 Faire bouger la base à travers des commandes en vitesse

Ce tutorial est disponible en Anglais [ici](#).

13.2.1 But

Le but de cet tutorial est de faire bouger le robot TIAGo en envoyant des commandes sur un topic.

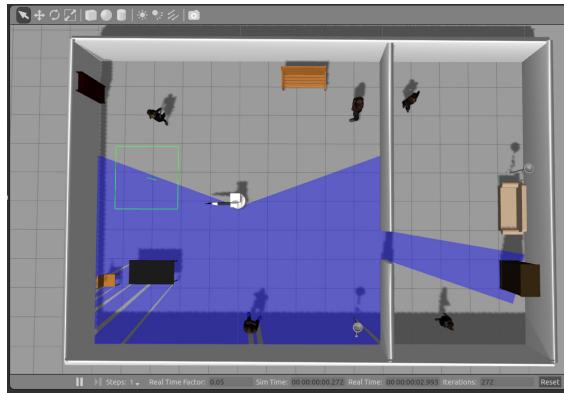


FIGURE 13.1 – TIAGo dans un bureau avec des humains

13.2.2 Lancer la simulation

La simulation proposée ici lance le robot TIAGo en version steel et sans environnement :

```
1 roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel world:=empty
```

L'environnement chargé par la simulation est représenté dans la figure Fig.13.2.

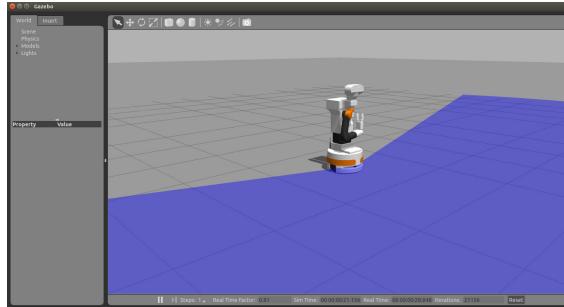


FIGURE 13.2 – Gazebo environnement vide

13.2.3 Faire bouger le robot en avant et en arrière

Pour cela on va utiliser une commande en vitesse publiée sur un topic.

```
1 rostopic pub /mobile_base_controller/cmd_vel geometry_msgs/Twist "linear:
2   x: 0.5
3   y: 0.0
4   z: 0.0
5 angular:
6   x: 0.0
7   y: 0.0
8   z: 0.0" -r 3
```

ou il est possible également de l'écrire dans la forme suivante :

```
1 rostopic pub /mobile_base_controller/cmd_vel geometry_msgs/Twist -r 3 -- '[0.5,0.0,0.0]' '[0.0, 0.0, 0.0]'
```

L'option `-r 3` permet de répéter la commande toutes les 3 secondes. Les commandes sont limitées dans le temps pour éviter les accidents.

13.3 Contrôleur des trajectoires articulaires

Ce tutoriel est disponible en Anglais [ici](#).

13.3.1 But du tutoriel

Le but est d'utiliser le contrôleur joint_trajectory_controller de ros_control pour bouger le bras de TIAGO. L'action peut-être utilisée pour exprimer la trajectoire du bras par plusieurs points intermédiaires.

Il y a deux interfaces pour envoyer des trajectoires au robot : l'interface des actions, et les topics. Les deux utilisent le message **trajectory_msgs/jointTrajectory** pour spécifier les trajectoires.

Pour le TIAGO les interfaces suivantes sont disponibles : **torso control, head control, arm control**.

13.3.2 Contrôleur du torse

Le joint prismatic du torse peut être contrôlé avec les interfaces de trajectoires articulaires suivantes :

- /torso_controller/command (trajectory_msgs/JointTrajectory)
Interface de type topic permettant de bouger le torse
- /torso_controller/follow_joint_trajectory (control_msgs/FollowJointTrajectoryAction)
Interface de type action pour bouger le torse

13.3.3 Contrôleur de la tête

Les 2 joints de la tête peuvent être contrôlés en définissant des trajectoires et en utilisant les interfaces suivantes :

- /head_controller/command (trajectory_msgs/JointTrajectory)
Interface de type topic pour bouger la tête.
- /head_controller/follow_joint_trajectory (control_msgs/FollowJointTrajectoryAction)
Interface de type action pour bouger la tête.

13.3.4 Contrôleur du bras

Les trajectoires des 7 joints du bras peuvent être contrôlées par les interfaces suivantes :

- /arm_controller/command (trajectory_msgs/JointTrajectory)
Interface de type topic pour bouger le bras.
- /arm_controller/follow_joint_trajectory (control_msgs/FollowJointTrajectoryAction)
Interface de type action pour bouger le bras.

13.3.5 Contrôleur du grippage Hey5

Pour la version titanium du robot TIAGO le contrôle de l'interface de la main est aussi disponible par les interfaces suivantes :

- /hand_controller/command (trajectory_msgs/JointTrajectory)
Interface de type topic pour bouger les 3 moteurs de la main Hey5.
- /hand_controller/follow_joint_trajectory (control_msgs/FollowJointTrajectoryAction)
Interface de type action pour bouger les 3 moteurs de la main Hey5.

13.3.6 Contrôleur du grippage

Pour la version steel du robot TIAGO le contrôle de l'interface du gripper est disponible par les interfaces suivantes :

- /parallel_gripper_controller/command (trajectory_msgs/JointTrajectory)
Interface de type topic pour bouger le joint virtuel du gripper parallèle, i.e. pour contrôler la séparation des doigts du grippage.
- /parallel_gripper_controller/follow_joint_trajectory (control_msgs/FollowJointTrajectoryAction)
Interface de type action pour contrôler la séparation de doigts du grippage.
- /gripper_control/command (trajectory_msgs/JointTrajectory)
Interface de type topic pour bouger les deux moteurs du gripper, i.e. pour chaque doigt individuellement.

13.3.7 Lancer la simulation

Cette simulation lance un robot TIAGO version Steel avec un monde vide :

```
roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel world:=empty
```

13.3.8 Lancer le node

Si les commandes des moteurs est spécifiée dans l'espace articulaire et plus particulièrement dans cet ordre : [arm_1_joint,arm_2_joint,arm_3_joint,arm_4_joint,arm_5_joint,arm_6_joint,arm_7_joint], considérons deux points dans cet espace : [0.2,0.0,-1.5,1.94,-1.57,-0.5,0.0], [2.5,0.2,-2.1,1.9,1.0,-0.5,0.0].

La commande suivante effectue un mouvement entre les deux positions spécifiées :

```
1 rosrun tiago_trajectory_controller run_traj_control
```

Le code donnée ci-dessous implémente la séquence suivante :

- Créer un action client pour contrôler le bras.
- Attendre que le serveur de l'action permettant le contrôle du bras soit en marche.
- Générer une trajectoire simple avec deux points intermédiaires (dont le but)
- Envoyer la commande pour démarrer la trajectoire.
- Donner la durée de l'exécution de la trajectoire.

Il faut faire attention à la vitesse spécifiée lors du passage au premier point intermédiaire si elle est à zéro cela risque de provoquer une discontinuité.

```
1 // C++ standard headers
2 #include <exception>
3 #include <string>
4
5 // Boost headers
6 #include <boost/shared_ptr.hpp>
7
8 // ROS headers
9 #include <ros/ros.h>
10 #include <actionlib/client/simple_action_client.h>
11 #include <control_msgs/FollowJointTrajectoryAction.h>
12 #include <ros/topic.h>
13
14
15 // Our Action interface type for moving TIAGo's head, provided as a typedef for convenience
16 typedef actionlib::SimpleActionClient<control_msgs::FollowJointTrajectoryAction> arm_control_client;
17 typedef boost::shared_ptr< arm_control_client> arm_control_client_Ptr;
18
19
20 // Create a ROS action client to move TIAGo's arm
21 void createArmClient(arm_control_client_Ptr& actionClient)
22 {
23     ROS_INFO("Creating action client to arm controller ...");
24
25     actionClient.reset( new arm_control_client("/arm_controller/follow_joint_trajectory") );
26
27     int iterations = 0, max_iterations = 3;
28     // Wait for arm controller action server to come up
29     while( !actionClient->waitForServer(ros::Duration(2.0)) && ros::ok() && iterations < max_iterations )
30     {
31         ROS_DEBUG("Waiting for the arm_controller_action server to come up");
32         ++iterations;
33     }
34
35     if ( iterations == max_iterations )
36         throw std::runtime_error("Error in createArmClient: arm controller action server not available");
37 }
38
39
40 // Generates a simple trajectory with two waypoints to move TIAGo's arm
41 void waypoints_arm_goal(control_msgs::FollowJointTrajectoryGoal& goal)
42 {
43     // The joint names, which apply to all waypoints
44     goal.trajectory.joint_names.push_back("arm_1_joint");
45     goal.trajectory.joint_names.push_back("arm_2_joint");
46     goal.trajectory.joint_names.push_back("arm_3_joint");
47     goal.trajectory.joint_names.push_back("arm_4_joint");
48     goal.trajectory.joint_names.push_back("arm_5_joint");
49     goal.trajectory.joint_names.push_back("arm_6_joint");
50     goal.trajectory.joint_names.push_back("arm_7_joint");
51
52     // Two waypoints in this goal trajectory
53     goal.trajectory.points.resize(2);
54
55     // First trajectory point
56     // Positions
57     int index = 0;
58     goal.trajectory.points[index].positions.resize(7);
59     goal.trajectory.points[index].positions[0] = 0.2;
60     goal.trajectory.points[index].positions[1] = 0.0;
61     goal.trajectory.points[index].positions[2] = -1.5;
```

```

62     goal.trajectory.points[index].positions[3] = 1.94;
63     goal.trajectory.points[index].positions[4] = -1.57;
64     goal.trajectory.points[index].positions[5] = -0.5;
65     goal.trajectory.points[index].positions[6] = 0.0;
66     // Velocities
67     goal.trajectory.points[index].velocities.resize(7);
68     for (int j = 0; j < 7; ++j)
69     {
70         goal.trajectory.points[index].velocities[j] = 1.0;
71     }
72     // To be reached 2 second after starting along the trajectory
73     goal.trajectory.points[index].time_from_start = ros::Duration(2.0);
74
75     // Second trajectory point
76     // Positions
77     index += 1;
78     goal.trajectory.points[index].positions.resize(7);
79     goal.trajectory.points[index].positions[0] = 2.5;
80     goal.trajectory.points[index].positions[1] = 0.2;
81     goal.trajectory.points[index].positions[2] = -2.1;
82     goal.trajectory.points[index].positions[3] = 1.9;
83     goal.trajectory.points[index].positions[4] = 1.0;
84     goal.trajectory.points[index].positions[5] = -0.5;
85     goal.trajectory.points[index].positions[6] = 0.0;
86     // Velocities
87     goal.trajectory.points[index].velocities.resize(7);
88     for (int j = 0; j < 7; ++j)
89     {
90         goal.trajectory.points[index].velocities[j] = 0.0;
91     }
92     // To be reached 4 seconds after starting along the trajectory
93     goal.trajectory.points[index].time_from_start = ros::Duration(4.0);
94 }
95
96
97 // Entry point
98 int main(int argc, char** argv)
99 {
100     // Init the ROS node
101     ros::init(argc, argv, "run_traj_control");
102
103     ROS_INFO("Starting run_traj_control application ...");
104
105     // Precondition: Valid clock
106     ros::NodeHandle nh;
107     if (!ros::Time::waitForValid(ros::WallDuration(10.0))) // NOTE: Important when using simulated clock
108     {
109         ROS_FATAL("Timed-out waiting for valid time.");
110         return EXIT_FAILURE;
111     }
112
113     // Create an arm controller action client to move the TIAGo's arm
114     arm_control_client_Ptr ArmClient;
115     createArmClient(ArmClient);
116
117     // Generates the goal for the TIAGo's arm
118     control_msgs::FollowJointTrajectoryGoal arm_goal;
119     waypoints_arm_goal(arm_goal);
120
121     // Sends the command to start the given trajectory 1s from now
122     arm_goal.trajectory.header.stamp = ros::Time::now() + ros::Duration(1.0);
123     ArmClient->sendGoal(arm_goal);
124
125     // Wait for trajectory execution
126     while(!(ArmClient->getState().isDone()) && ros::ok())
127     {
128         ros::Duration(4).sleep(); // sleep for four seconds
129     }
130
131     return EXIT_SUCCESS;
132 }
```

13.4 Bouger des articulations individuellement

13.4.1 Lancer la simulation

Cette simulation lance un robot TIAGo version Titanium (notamment avec une main) et un environnement de bureau avec des gens :

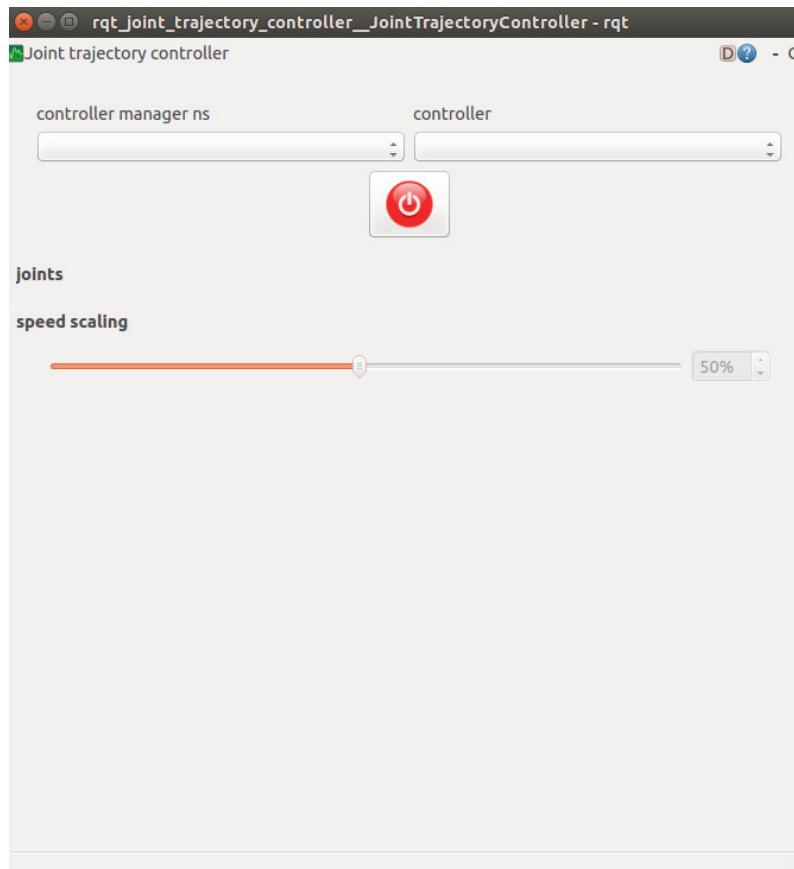
```
1 roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel world:=simple_office_with_people
```

13.4.2 Bouger les articulations individuellement

Dans une autre console, lancer la commande suivante :

```
rosrun rqt_joint_trajectory_controller rqt_joint_trajectory_controller
```

et la fenêtre suivante de la Figu.13.4.2 va apparaître. Il faut d'abord sélectionner dans la liste à gauche le



controller_manager pour ensuite pouvoir sélectionner les différents groupes de contrôleurs qui seront dans la liste à droite (donnée aussi dans le paragraphe 13.3.2). Par exemple, si on sélectionne **arm_controller**, les curseurs de la figure Fig.13.4.2 vont apparaître dans la fenêtre. Il suffit de cliquer sur le bouton rouge pour qu'il devienne vert. Cela signifie que l'interface graphique peut envoyer des commandes sur l'interface topic de **joint_trajectory_controller** correspondant au groupe sélectionné d'articulations.

On peut noter en bas de l'interface, un facteur d'échelle qui sera appliqué aux articulations.

13.4.3 Bouger les articulations à partir de la ligne de commande

Il est possible de baisser la tête de TIAGO avec la commande suivante :

```
rosrun play_motion move_joint head_2_joint -0.6 2.0
```

cela va afficher la ligne suivante :

```
Moving joint head_2_joint to position -0.6 in 2.0s
```

13.5 Contrôle de la tête

Ce tutorial est disponible en Anglais ici.

13.5.1 But du tutoriel

Le but de ce tutorial est de montrer un exemple de contrôle de la tête en utilisant l'image de la caméra.



13.5.2 Lancer la simulation

```
rosrun tiago_gazebo tiago_gazebo.launch robot:=steel public_sim:=true world:=look_to_point
```

Gazebo montre alors TIAGO en face d'une table avec des objets dans la figure Fig.13.5.2.

13.5.3 Lancer l'exemple d'un client d'action en C++

Un exemple de node C++ utilisant un client d'action est disponible dans `tiago_tutorials/look_to_point/src/look_to_point.cpp`. De façon à exécuter le node il faut utiliser la commande suivante :

```
rosrun look_to_point look_to_point
```

Le node va souscrire aux topics suivants :

- /xtion/rgb/camera_info
- /xtion/rgb/image_raw

Le premier topic contient les paramètres intrinsèques de la caméra et le second l'image rgb de la caméra RGBD de la tête de TIAGO.

L'application va afficher l'image rgb que l'on voit dans la figure Fig.13.5.3.

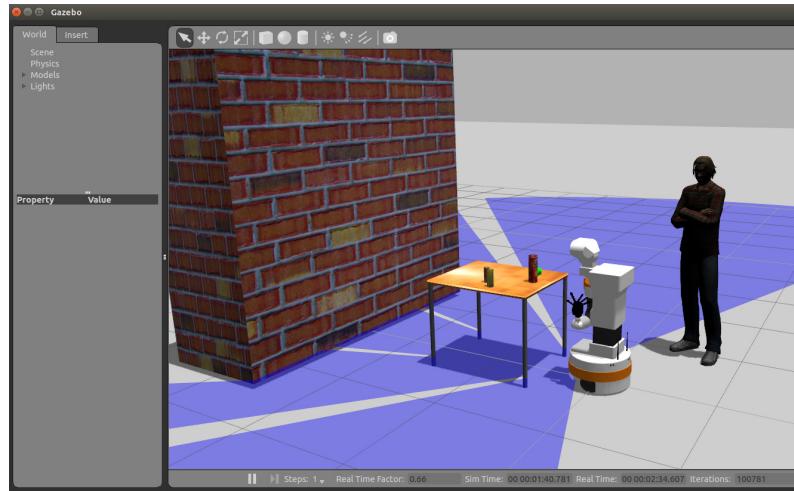
Afin de générer un mouvement il suffit de cliquer sur un pixel de l'image.

13.6 Rejouer des mouvements du haut du corps prédéfinis

13.6.1 Lancer la simulation

La simulation est lancée avec la commande suivante :

```
rosrun tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel world:=empty
```



Quand la simulation est démarée, le serveur d'action **play_motion** est automatiquement démarré et les mouvements définis dans **tiago_bringup/config/tiago_motions_titanium.yaml** ont été chargés dans le serveur de paramètres.

De façon à voir le nom des mouvements chargés dans le serveur de paramètres, on peut utiliser l'instruction suivante :

```
1 rosparam list | grep "play_motion/motions" | grep "meta/name" | cut -d ' ' -f 4
```

On peut notamment y trouver le mouvement **wave**. Les informations permettant de faire ce mouvement sont stockés dans les paramètres suivants :

```
1 /play_motion/motions/wave/joints
2 /play_motion/motions/wave/points
```

Dans ce cas le paramètre **joints** spécifie quelles sont les articulations qui sont utilisées dans le mouvement et dans quel ordre :

```
1 $ rosparam get /play_motion/motions/wave/joints
2 [arm_1_joint, arm_2_joint, arm_3_joint, arm_4_joint, arm_5_joint, arm_6_joint, arm_7_joint]
```

La trajectoire articulaire est donnée sous forme de séquence temporelle de points. La définition exacte est donnée à cette adresse http://wiki.ros.org/joint_trajectory_controller.

En voici un exemple :

```

1 $ rosparam get /play_motion/motions/wave/points
2 - positions: [0.06337464909724033, -0.679638896132783, -3.1087325315620733, 2.0882339360702575,
3   -1.1201172410014792, -0.031008601325809293, -2.0744261217334135]
4   time_from_start: 0.0
5 - positions: [0.06335930908588873, -0.7354151774072313, -2.939624246421942, 1.8341256735249563,
6   -1.1201355028397157, -0.031008601325809293, -2.0744261217334135]
7   time_from_start: 1.0
8 - positions: [0.06335930908588873, -0.7231278283145929, -2.9385504456273295, 2.2121050027803877,
9   -1.1201355028397157, -0.031008601325809293, -2.0744261217334135]
10  time_from_start: 2.0
11 - positions: [0.06335930908588873, -0.7354151774072313, -2.939624246421942, 1.8341256735249563,
12   -1.1201355028397157, -0.031008601325809293, -2.0744261217334135]
13  time_from_start: 3.0

```

Chaque ligne donne la position à laquelle chaque articulation doit être à un moment donné par rapport au démarrage du mouvement.

13.6.2 Jouer un mouvement via une interface graphique

Comme **play_motion** est un serveur d'action, il est possible d'utiliser l'interface graphique des clients d'action. Pour Melodic la commande est :

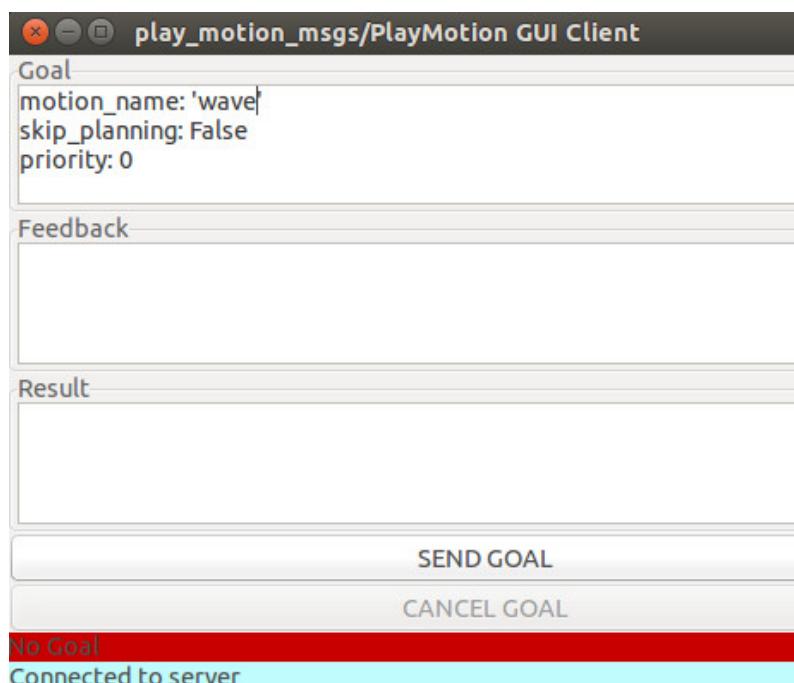
```
1 rosrun actionlib axclient.py /play_motion
```

Pour Noetic :

```
1 rosrun actionlib_tools axclient.py /play_motion
```

L'interface est représentée dans la figure Fig.13.6.2. Il faut simplement mettre le nom du mouvement dans la chaîne de caractères à côté de **motion_name**. Le champ **skip_planning** est défaut à False. Il effectue une planification de mouvements de l'état courant à la première pause pour s'assurer qu'il n'y aura pas collision. C'est de la responsabilité de l'utilisateur de s'assurer que le reste du mouvement n'est pas en collision.

En appuyant sur **SEND GOAL** la requête est envoyée au server d'action play_motion et le mouvement s'exécute.



13.6.3 Jouer un mouvement en ligne de commande

Il est possible de le faire avec le client C++ :

```
| rosrun play_motion run_motion wave
```

Il ne faut pas oublier le nom du mouvement en argument du node.

Ou le client python :

```
| rosrun play_motion run_motion_python_node.py home
```

14. TIAGo Navigation

14.1 Créer une carte avec gmapping

Ce tutorial est disponible en Anglais [ici](#).

14.1.1 But

Ce tutorial montre comment créer une carte de l'environnement avec la simulation publique de TIAGo en utilisant gmapping. Cette carte est requise pour utiliser ensuite le système de localisation AMCL. Celui-ci compare les scans lasers avec la carte pour fournir une estimation fiable de la position du robot dans la carte.

14.1.2 Execution

Dans une première console il faut lancer la simulation suivante :

```
rosrun tiago_2dnav_gazebo tiago_mapping.launch public_sim:=true
```

La simulation fournie est disponible dans la figure Fig.14.1

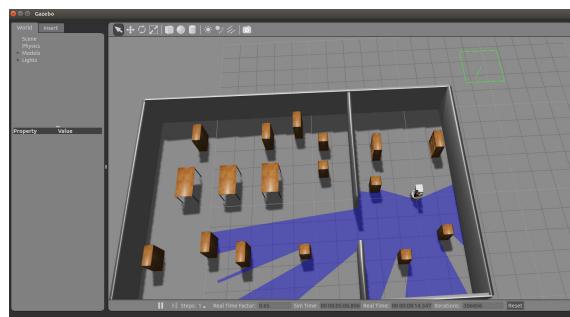


FIGURE 14.1 – TIAGo dans une bibliothèque (Simulation Gazebo).

Rviz apparaît également de façon à pouvoir visualiser la construction de la carte, comme il est possible de voir dans la figure.14.2.

Dans un second terminal, il faut lancer le node pour téléopérer le robot par le clavier.

```
rosrun key_teleop key_teleop.py
```

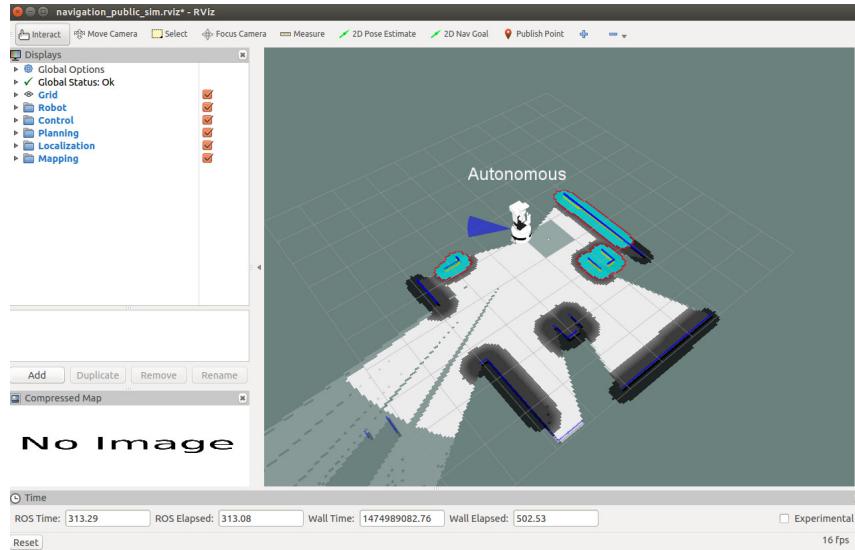


FIGURE 14.2 – Visualisation de la carte construite par tiago dans la simulation Gazebo.

En utilisant les flèches du clavier dans cette console il est possible de guider le robot TIAGo dans le monde. La carte construite apparaît dans rviz. Quand la carte est complétement construite, on trouve l'exemple représenté dans la figure.14.3.

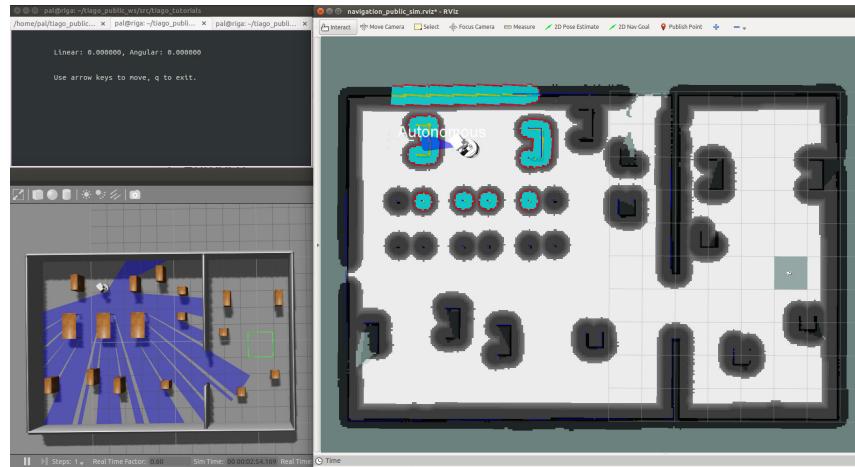


FIGURE 14.3 – Visualisation de la carte complète construite par tiago dans la simulation Gazebo.

En pressant la touche 'q' dans la console key_teleop et sauve la carte de la façon suivante :

```
rosservice call /pal_map_manager/save_map "directory: ''"
```

L'appel au service sauve la carte dans le répertoire suivant :

```
~/pal/tiago_maps/config
```

Maintenant TIAGO est prêt à effectuer une localisation autonome et de la planification de mouvements en utilisant la carte. La section suivante montre comment il est possible d'effectuer une navigation autonome.

14.2 Localisation et planification de mouvement

Ce tutorial est disponible en Anglais ici.

14.2.1 But

Ce tutorial montre comme il est possible de faire naviguer TIAGO automatiquement lorsque l'on fournit une carte construite à partir de scans laser, et en prenant en compte le laser et la caméra RGBD pour éviter les obstacles.

Dans une première console il est possible de lancer la simulation de TIAGO avec tous les noeuds requis pour lancer la navigation autonome.

```
roslaunch tiago_2dnav_gazebo tiago_navigation.launch public_sim:=true lost:=true
```

La carte créée dans la section 14.1 est affichée dans rviz. L'information suivante est affichée sur la carte :

- **Nuage de particules** : un nuage de petites flèches rouges représentant les particules du filtre d'amcl.
- **Carte globale de coûts** : régions autour des obstacles qui sont utilisées par la planification globale de façon à calculer les chemins pour naviguer d'un point de la carte à un autre sans s'approcher trop près des obstacles statiques enregistrés durant la cartographie. Plus de détails sont fournis ici : http://wiki.ros.org/costmap_2d.
- **Carte locale des coûts** : similaire à la carte globale, mais plus petite elle bouge avec le robot. Elle est utilisée pour prendre en compte les nouveaux objets qui n'étaient pas dans la carte originale. Cette carte est utilisée par des planificateurs locaux pour éviter les obstacles, à la fois statique et dynamique, tout en essayant de suivre le chemin global calculé par le planificateur global. Plus de détails sont fournis ici : http://wiki.ros.org/costmap_2d.
- **Scan laser** : les lignes en bleu représentent les points mesurés avec le laser de la base mobile. Ce scan est utilisé pour ajouter/retirer des obstacles dans les cartes de coûts locales et globales.
- **RGBD scan** : les lignes de couleur magenta représentent la projection sur le sol des points 3D reconstruits par la caméra RGBD de la tête. Ce scan artificiel est utilisé pour ajouter/retirer des obstacles dans les cartes de coûts locales et globales. Ce scan est utile pour obtenir des informations 3D de l'environnement, détecter des obstacles qui sont plus haut ou plus bas que le scan laser.

14.2.2 Localization

Comme on peut le voir dans la figure 14.4, le robot est mal localisé dans sa carte. On peut voir les particules qui indiquent les positions où le robot croit être à partir des mesures qu'il fait actuellement.

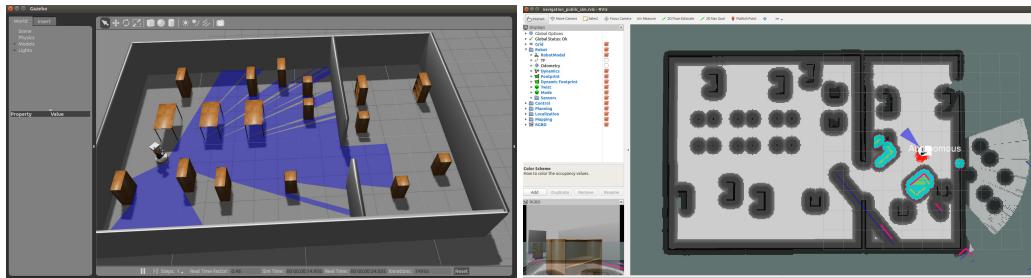


FIGURE 14.4 – Initial pose de TIAGO dans une bibliothèque simulée (à gauche) Localisation du robot dans sa carte (droite)

Pour pouvoir localiser le robot il faut lancer l'instruction suivante dans un second terminal :

```
rosservice call /global_localization "{'"
```

Ceci impose au système amcl de localisation probabiliste de répartir les particules à travers toute la carte comme on peut le voir dans la figure 14.5.

Ensuite une bonne façon de faire converger le filtre particulaire vers la bonne pose est de faire bouger le robot. Une façon sûre de faire cela est de faire tourner le robot sur lui-même. Pour cela il est possible d'utiliser les flèches droite et gauche de **key_teleop** pour le faire :

```
rosrun key_teleop key_teleop.py
```

Le résultat est illustré dans la figure 14.6, où les particules invalides disparaissent au fur et à mesure que les scans se retrouvent dans la carte et que la localisation converge éventuellement vers la pose correcte.

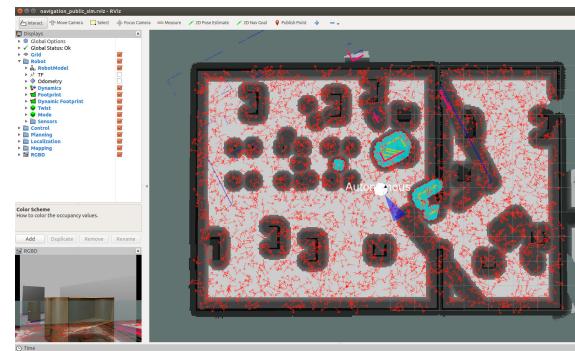


FIGURE 14.5 –

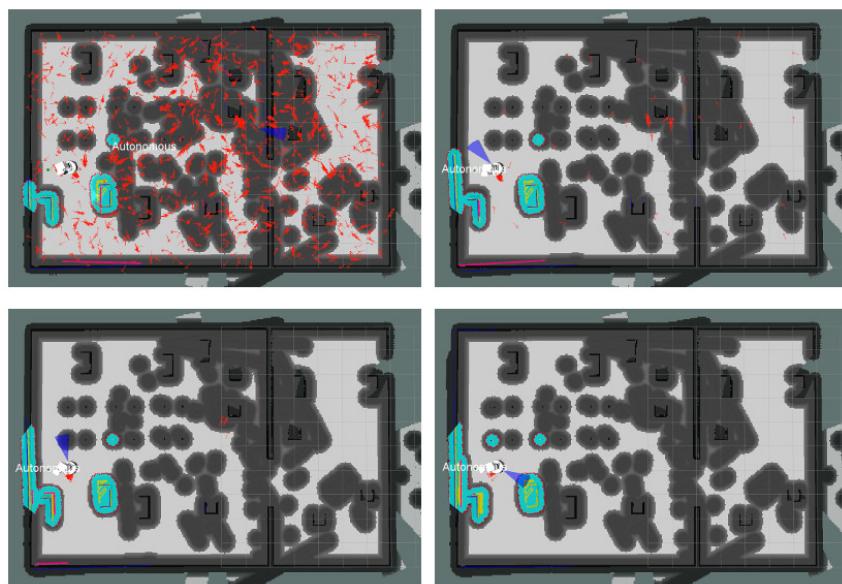


FIGURE 14.6 –

Il est maintenant préférable d'effacer les cartes de coûts car elles contiennent des informations erronées à cause de la mauvaise localisation du robot.

```
rosservice call /move_base/clear_costmaps "[]"
```

La carte de coût ne contient maintenant que les informations statiques et est prête pour la navigation.

14.2.3 Navigation autonome with rviz

Premièrement il faut s'assurer que toutes les nodes **key_teleop** soient tuées. Sinon le robot ne bougera de façon autonome car le node **key_teleop** a une priorité plus élevée.

De façon à envoyer le robot vers une autre position dans la carte il suffit d'utiliser le bouton 2D Nav Goal dans rviz comme indiqué dans la figure Fig.14.8 à gauche.

En cliquant et en maintenant le bouton gauche enfoncé sur le point dans la carte, auquel on veut envoyer le robot, une flèche verte va apparaître. En amenant la souris autour le point de la souris va changer. Dans la figure Fig.14.8 à droite, la flèche représente la position de la cible et l'orientation que le robot doit atteindre.

Quand le bouton est relâché le planificateur global calcule un chemin qui apparaît sous forme d'une ligne bleue qui démarre de la position courante du robot et qui finit avec une flèche rouge représentant la position désirée et l'orientation.

Le robot commence à bouger en suivant la trajectoire, et ensuite va automatiquement changer cette

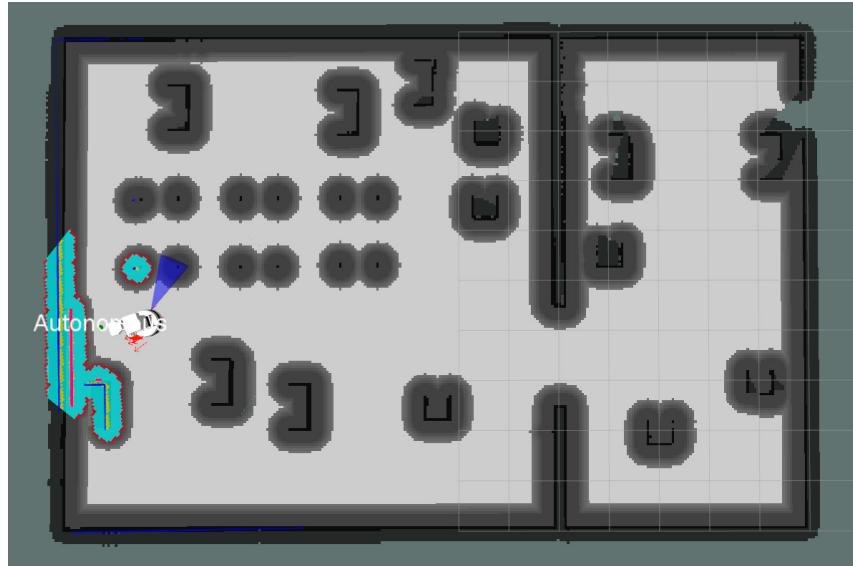


FIGURE 14.7 – Cartes de coûts sans les informations erronées.

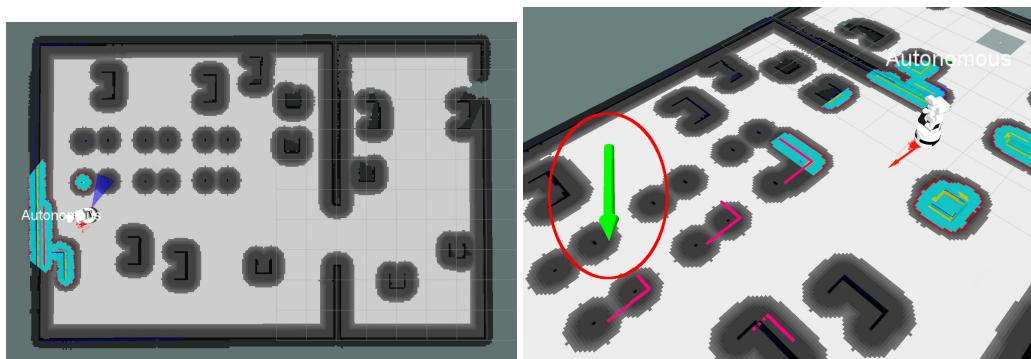


FIGURE 14.8 – Le bouton 2D Nav Goal (à gauche) Le but que TIAGo doit atteindre exprimé par une flèche en position et orientation (à gauche)

trajectoire s'il se trouver trop près d'obstacles non prévus. La figure Fig.14.10 montre différentes vues de rviz et Gazebo durant la navigation vers le but sélectionné.

Finalement le robot s'arrête quand le but est atteint dans la zone de tolérance définie par l'utilisateur.

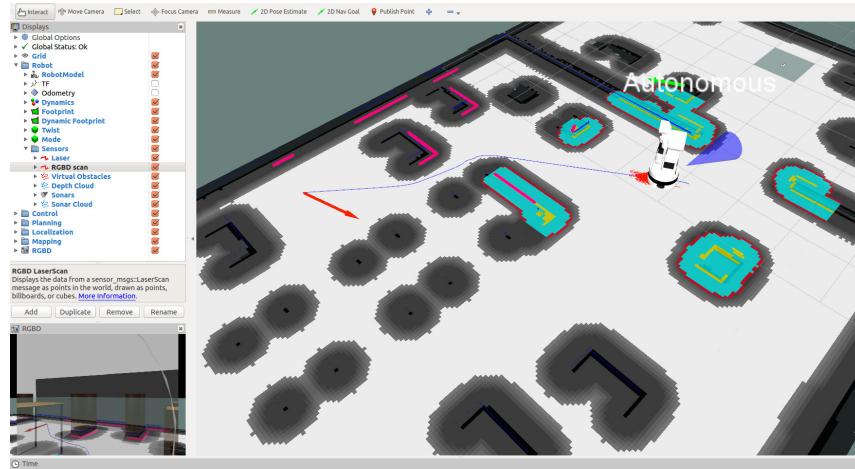


FIGURE 14.9 – Chemin planifié dans une carte de coûts.

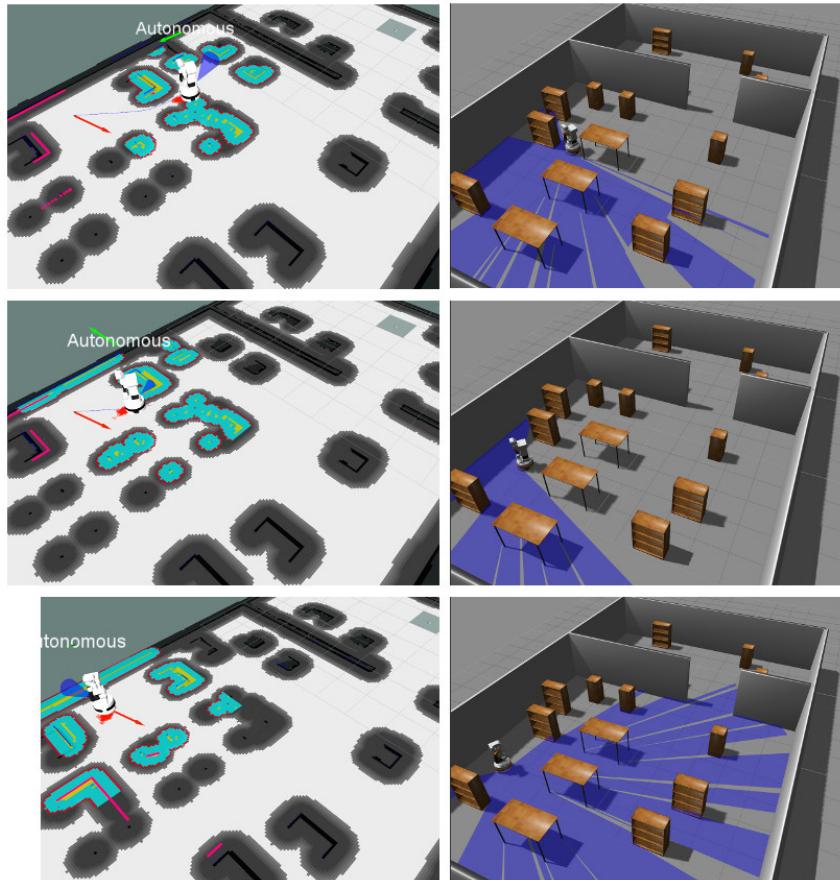


FIGURE 14.10 – Navigation

15. Planification de mouvements

15.1 Planifier dans l'espace de configuration

Ce tutorial est disponible ici en Anglais

15.1.1 But

Ce tutoriel montre comment on peut utiliser MoveIt! pour amener le groupe d'articulation torse-bras vers la configuration articulaire souhaitée tout en s'assurant que les limites des joints et l'auto-collision sont respectés. L'exemple est donné en C++.

15.1.2 Lancer la simulation

Dans la première console on démarre la simulation de la façon suivante :

```
1 rosrun tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel
```

Gazebo apparaît avec TIAGO comme illustré dans la figure Fig.15.3.

15.1.3 Lancer les nodes

Ce didacticiel lance un exemple qui amène le robot TIAGO vers la configuration du groupe torse-bras :

```
1 torso_joint: 0
2 arm_1_joint: 2.7
3 arm_2_joint: 0.2
4 arm_3_joint: -2.1
5 arm_4_joint: 2.0
6 arm_5_joint: 1.0
7 arm_6_joint: -0.8
8 arm_7_joint: 0
```

On peut noter que si on essaie d'atteindre cette cible en bougeant chaque articulation individuellement, il est très probable que le robot se retrouverait en auto-collision. Le node qui va s'occuper de trouver un plan, i.e. une séquence de mouvements, pour atteindre cette configuration articulaire est **plan_arm_torso_fk** disponible dans **tiago_moveit_tutorial**.

Il peut être appelé de la façon suivante :

```
1 rosrun tiago_moveit_tutorial plan_arm_torso_fk 0 2.7 0.2 -2.1 2.0 1.0 -0.8 0
```

Un exemple de plan exécuté par le node est illustré dans la figure Fig.15.2.

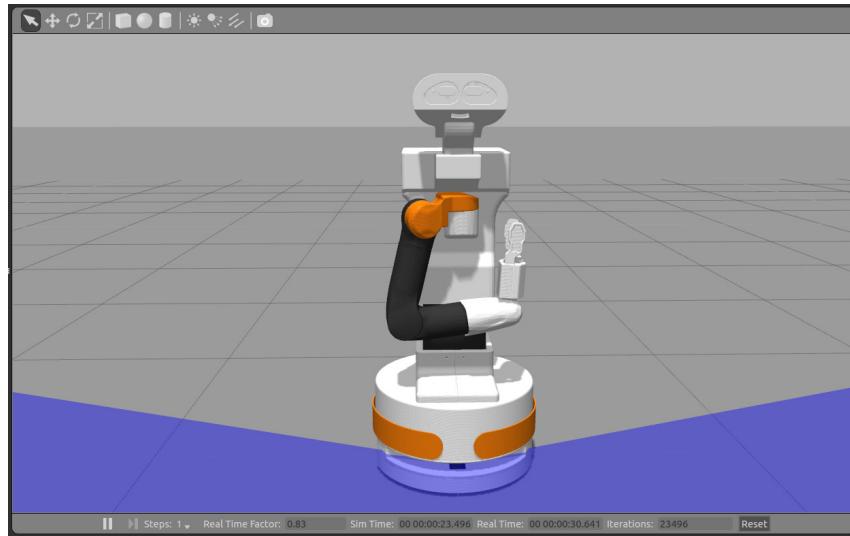


FIGURE 15.1 – TIAGo est prêt pour le planning quand son bras est bien rangé

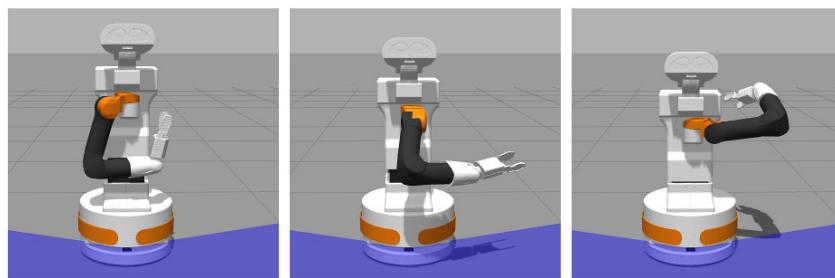


FIGURE 15.2 – Planification de mouvement entre deux configurations articulaires

15.1.4 Inspection du code

Le code pour implémenter un node capable de planifier dans l'espace articulaire est donné dans le code ci-dessous. Les parties clefs du code sont :

- Choisir un groupe de joints
- Choisir un planificateur
- Spécifier l'état initial et l'état final
- Donner du temps pour trouver un plan
- Exécuter le plan trouvé.

```

1 // ROS headers
2 #include <ros/ros.h>
3
4 // MoveIt! headers
5 #include <moveit/move_group_interface/move_group_interface.h>
6
7 // Std C++ headers
8 #include <string>
9 #include <vector>
10 #include <map>
11
12 int main(int argc, char** argv)
13 {
14     ros::init(argc, argv, "plan_arm_torso_fk");
15
16     if ( argc < 9 )
17     {
18         ROS_INFO(" ");
19         ROS_INFO("\tUsage:");
20         ROS_INFO(" ");

```

```

21     ROS_INFO("\trostrun tiago_moveit_tutorial plan_arm_torso_fk torso_lift arm_1 arm_2 arm_3 arm_4 arm_5 arm_6
22         arm_7");
23     ROS_INFO(" ");
24     ROS_INFO("\twhere the list of arguments are the target values for the given joints");
25     ROS_INFO(" ");
26     return EXIT_FAILURE;
27 }
28
29 std::map<std::string, double> target_position;
30
31 target_position["torso_lift_joint"] = atof(argv[1]);
32 target_position["arm_1_joint"] = atof(argv[2]);
33 target_position["arm_2_joint"] = atof(argv[3]);
34 target_position["arm_3_joint"] = atof(argv[4]);
35 target_position["arm_4_joint"] = atof(argv[5]);
36 target_position["arm_5_joint"] = atof(argv[6]);
37 target_position["arm_6_joint"] = atof(argv[7]);
38 target_position["arm_7_joint"] = atof(argv[8]);
39
40 ros::NodeHandle nh;
41 ros::AsyncSpinner spinner(1);
42 spinner.start();
43
44 std::vector<std::string> torso_arm_joint_names;
45 //select group of joints
46 moveit::planning_interface::MoveGroup group_arm_torso("arm_torso");
47 //choose your preferred planner
48 group_arm_torso.setPlannerId("SBLkConfigDefault");
49
50 torso_arm_joint_names = group_arm_torso.getJoints();
51
52 group_arm_torso.setStartStateToCurrentState();
53 group_arm_torso.setMaxVelocityScalingFactor(1.0);
54
55 for (unsigned int i = 0; i < torso_arm_joint_names.size(); ++i)
56     if ( target_position.count(torso_arm_joint_names[i]) > 0 )
57     {
58         ROS_INFO_STREAM("\t" << torso_arm_joint_names[i] << " goal position: " << target_position[
59             torso_arm_joint_names[i]]);
60         group_arm_torso.setJointValueTarget(torso_arm_joint_names[i], target_position[torso_arm_joint_names[i]]);
61     }
62
63 moveit::planning_interface::MoveGroup::Plan my_plan;
64 group_arm_torso.setPlanningTime(5.0);
65 bool success = group_arm_torso.plan(my_plan);
66
67 if ( !success )
68     throw std::runtime_error("No plan found");
69
70 ROS_INFO_STREAM("Plan found in " << my_plan.planning_time_ << " seconds");
71
72 // Execute the plan
73 ros::Time start = ros::Time::now();
74
75 group_arm_torso.move();
76
77 ROS_INFO_STREAM("Motion duration: " << (ros::Time::now() - start).toSec());
78
79 spinner.stop();
80
81 return EXIT_SUCCESS;
82 }
```

Notons qu'un plan est trouvé et exécuté avec la ligne de code suivante :

```
73 group_arm_torso.move();
```

Les commandes requises sont envoyées aux contrôleurs du bras et du torse à travers les interfaces d'action :

```
1 /arm_controller/follow_joint_trajectory/goal
2 /torso_controller/follow_joint_trajectory/goal
```

15.2 Planifier dans l'espace cartésien

Ce tutorial est disponible ici en Anglais

15.2.1 But

Ce tutoriel montre comment on peut utiliser MoveIt! pour amener le group d'articulation torse-bras vers la position souhaitée dans l'espace Cartésien tout en s'assurant que les limites des joints et l'auto-collision sont respectés. L'exemple est donné en C++.

15.2.2 Lancer la simulation

Dans la première console on démarre la simulation de la façon suivante :

```
roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel
```

Gazebo apparaît avec TIAGo comme illustré dans la figure Fig.15.3.

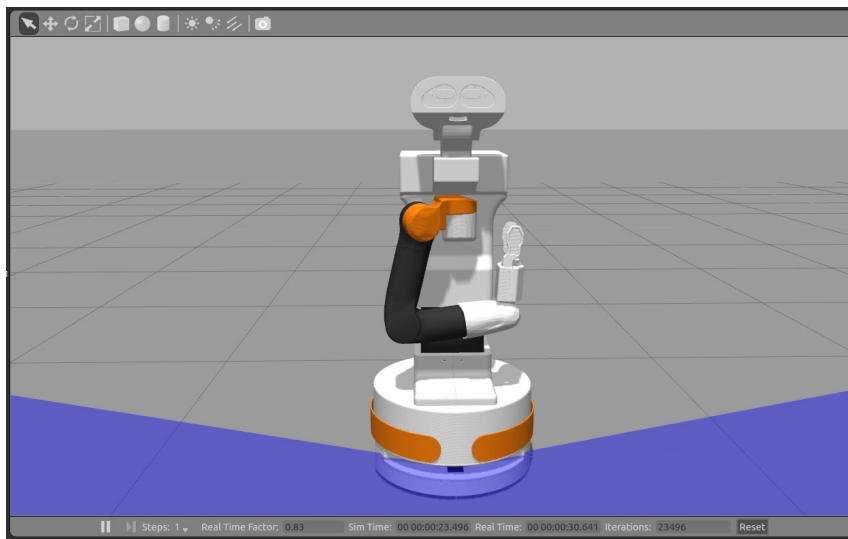


FIGURE 15.3 – TIAGo est prêt pour le planning quand son bras est bien rangé

15.2.3 Lancer les nodes

Ce didacticiel lance un exemple qui va amener le référentiel **arm_tool_link** vers la position cartésienne suivante dans le référentiel **/base_footprint**

```
1 x: 0.4
2 y: -0.3
3 z: 0.26
4 Roll: -0.011
5 Pitch: 1.57
6 Yaw: 0.037
```

qui correspond à la pose montrée dans **rviz** dans la figure Fig.15.4

De façon à atteindre cette position Cartésienne, il est possible d'utiliser le node **plan_arm_torso_ik** inclus dans le paquet **tiago_moveit_tutorial** et peut-être appellé de la façon suivante :

```
rosrun tiago_moveit_tutorial plan_arm_torso_ik 0.4 -0.3 0.26 -0.011 1.57 0.037
```

Un exemple de plan exécuté par le node est illustré dans la figure Fig.15.5. La pose finale de **/arm_tool_link** est celle désirée comme on peut le voir dans la figure Fig.15.6.

15.2.4 Inspection du code

Le code qui implémente le node capable de planifier un plan dans l'espace Cartésien est donné ci-dessous. Notons que les parties clefs de ce code sont :

- Choisir un groupe de joints
- Choisir un planificateur et définir le référentiel de référence (**base_footprint** dans ce cas)
- Spécifier la pose désirée du référentiel **arm_tool_link** encodé comme un **geometry_msgs::PoseStamped**

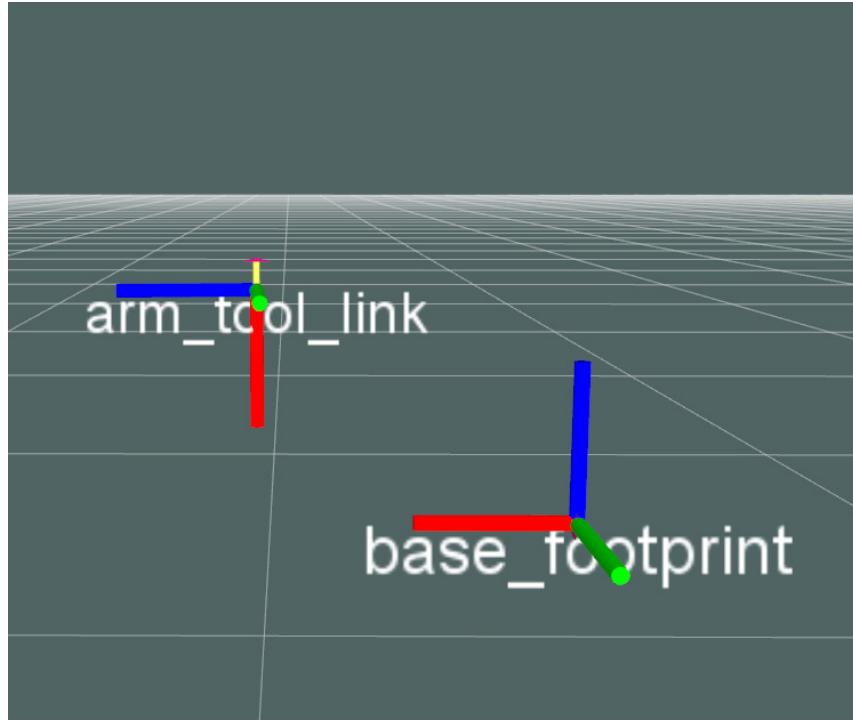


FIGURE 15.4 – Le référentiel **arm_tool_link** exprimé par rapport au référentiel **base_footprint**

- Donner du temps pour trouver un plan
- Exécuter le plan trouvé.

```

1 // ROS headers
2 #include <ros/ros.h>
3
4 // MoveIt! headers
5 #include <moveit/move_group_interface/move_group_interface.h>
6
7 // Std C++ headers
8 #include <string>
9 #include <vector>
10 #include <map>
11
12 int main(int argc, char** argv)
13 {
14     ros::init(argc, argv, "plan_arm_torso_ik");
15
16     if ( argc < 7 )
17     {
18         ROS_INFO(" ");
19         ROS_INFO("\tUsage:");
20         ROS_INFO(" ");
21         ROS_INFO("\trosrun tiago_moveit_tutorial plan_arm_torso_ik  x y z  r p y");
22         ROS_INFO(" ");
23         ROS_INFO("\twhere the list of arguments specify the target pose of /arm_tool_link expressed in /base_footprint
24             ");
25         ROS_INFO(" ");
26         return EXIT_FAILURE;
27     }
28
29     geometry_msgs::PoseStamped goal_pose;
30     goal_pose.header.frame_id = "base_footprint";
31     goal_pose.pose.position.x = atof(argv[1]);
32     goal_pose.pose.position.y = atof(argv[2]);
33     goal_pose.pose.position.z = atof(argv[3]);
34     goal_pose.pose.orientation = tf::createQuaternionMsgFromRollPitchYaw(atof(argv[4]), atof(argv[5]), atof(argv[6])
35         );
36
37     ros::NodeHandle nh;
38     ros::AsyncSpinner spinner(1);
39     spinner.start();

```

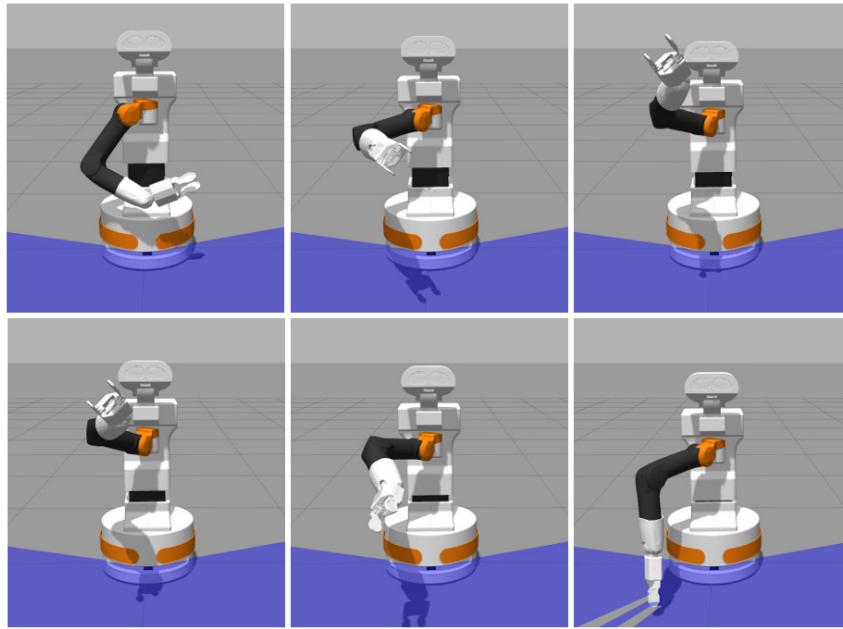


FIGURE 15.5 – La séquence de mouvements pour amener le référentiel **arm_tool_link** vers la position Cartésienne souhaitée.

```

39 std::vector<std::string> torso_arm_joint_names;
40 //select group of joints
41 moveit::planning_interface::MoveGroup group_arm_torso("arm_torso");
42 //choose your preferred planner
43 group_arm_torso.setPlannerId("SBLkConfigDefault");
44 group_arm_torso.setPoseReferenceFrame("base_footprint");
45 group_arm_torso.setPoseTarget(goal_pose);
46
47 ROS_INFO_STREAM("Planning to move " <<
48     group_arm_torso.getEndEffectorLink() << " to a target pose expressed in " <<
49     group_arm_torso.getPlanningFrame());
50
51 group_arm_torso.setStartStateToCurrentState();
52 group_arm_torso.setMaxVelocityScalingFactor(1.0);
53
54
55 moveit::planning_interface::MoveGroup::Plan my_plan;
56 //set maximum time to find a plan
57 group_arm_torso.setPlanningTime(5.0);
58 bool success = group_arm_torso.plan(my_plan);
59
60 if ( !success )
61     throw std::runtime_error("No plan found");
62
63 ROS_INFO_STREAM("Plan found in " << my_plan.planning_time_ << " seconds");
64
65 // Execute the plan
66 ros::Time start = ros::Time::now();
67
68 group_arm_torso.move();
69
70 ROS_INFO_STREAM("Motion duration: " << (ros::Time::now() - start).toSec());
71
72 spinner.stop();
73
74 return EXIT_SUCCESS;
75 }
```

Notons qu'un plan est trouvé et exécuté avec la ligne de code suivante :

```
68 group_arm_torso.move();
```

Les commandes requises sont envoyées aux contrôleurs du bras et du torse à travers les interfaces d'action :

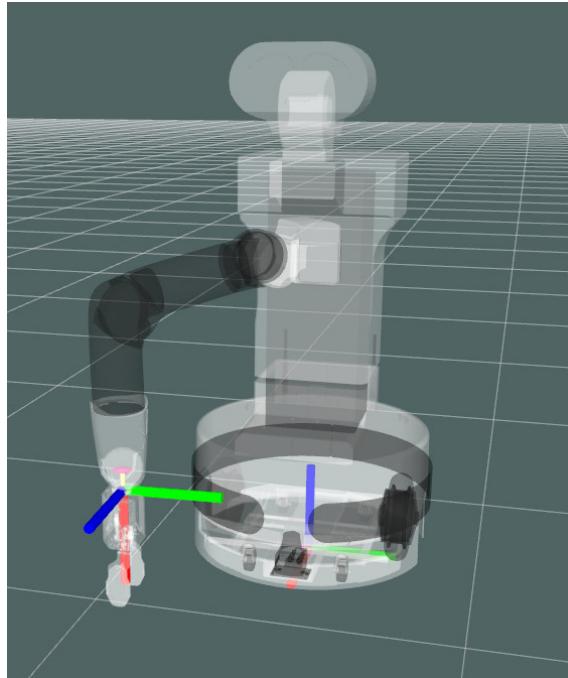


FIGURE 15.6 – La configuration finale du robot TIAGO à la fin du plan.

```

1 /arm_controller/follow_joint_trajectory/goal
2 /torso_controller/follow_joint_trajectory/goal

```

15.3 Planification avec Octomap

15.3.1 Goal

Le but de ce didacticiel est de fournir un exemple d'utilisation d'Octomap dans MoveIt! pour planifier des mouvements pour TIAGO. Le système de carte d'occupation de MoveIt! utilise une structure appelée Octomap pour maintenir une représentation de l'environnement autour du robot. L'environnement simulé comprend une table et quelques objets. Le robot localise l'environnement avec sa caméra et construit l'Octomap. MoveIt! est ensuite utilisé pour planifier une trajectoire articulaire afin d'atteindre une position Cartésienne. MoveIt! utilise Octomap dans la librairie de détection de collision.

15.3.2 Lancer la simulation

Dans une console il faut lancer la simulation qui démarre le robot TIAGO en face d'une table avec des objets dessus :

```

1 rosrun tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel world:=tutorial_office gzpose:="-x 1.40 -
y -2.79 -z -0.003 -R 0.0 -P 0.0 -Y 0.0" use_moveit_camera:=true

```

Gazebo va ensuite apparaître avec TIAGO comme illustré dans la figure Fig.15.7. Quand la simulation est lancée, MoveIt! est automatiquement lancé. Le paramètre **camera :=true** va permettre à MoveIt! d'utiliser la caméra de profondeur comme entrée. La configuration d'Octomap définie dans **tiago_moveit_config/launch/tiago_moveit_sensor_manager.launch.xml**, spécifie le référentiel dans lequel la représentation est stockée. Le référentiel utilisé dans cette démonstration est **odom**. Les détails à propos de la configuration d'Octomap dans MoveIt! est disponible ici : http://docs.ros.org/indigo/api/moveit_tutorials/html/doc/pr2_tutorials/planning/src/doc/perception_configuration.html.

Il faut attendre que TIAGO range son bras avant de lancer les prochaines étapes.

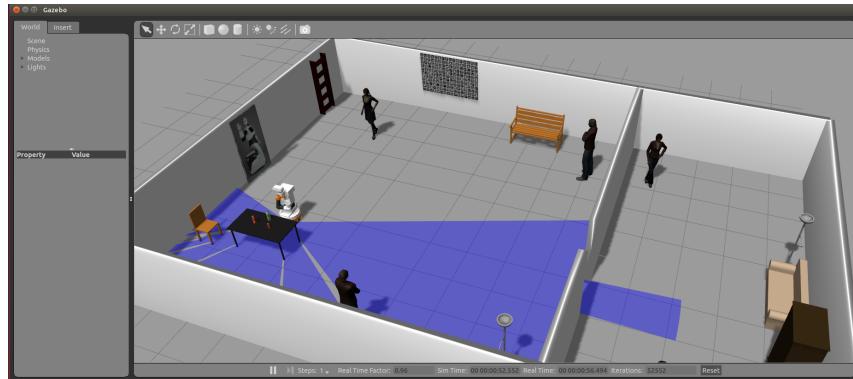


FIGURE 15.7 – Simulation de TIAGO dans Gazebo

15.3.3 Lancer les nodes

Dans un deuxième terminal il faut le node qui extrait des points clefs :

```
rosrun tiago_moveit_tutorial octomap_tiago.launch
```

cela lance également Rviz de façon à aider la visualisation des différentes étapes de la démonstration (voir la figure Fig.15.8).

Dans une troisième console, il faut lancer un node qui va limiter la fréquence à 2 Hz avec laquelle les nuages de points 3D sont publiés. Le node publie également sur le topic auquel MoveIt! s'attend.

```
rosrun topic_tools throttle messages /xtion/depth_registered/points 2 /throttle_filtering_points/filtered_points
```

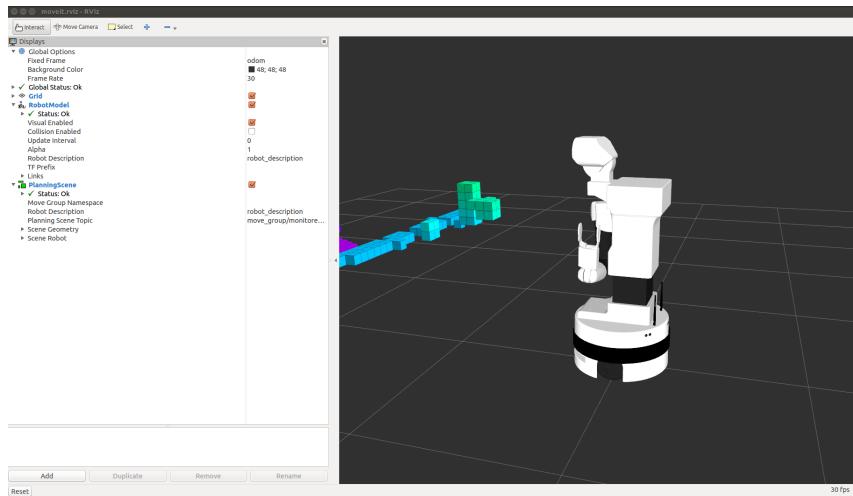


FIGURE 15.8 – Affichage dans Rviz de la carte Octomap reconstruite par TIAGO

15.3.4 Démarrer la démonstration

Dans une troisième console le mouvement **head_look_around** doit être démarré pour construire une représentation de l'environnement. Quand la simulation est lancée, le serveur d'action **play_motion** est automatiquement lancé (voir la section 13.6). Avant de démarrer le mouvement **head_look_around**, le fichier **tiago_moveit_tutorial/config/tiago_octomap_motions.yaml** doit être chargé dans le serveur de paramètres ros.

```
rosparam load `rospack find tiago_moveit_tutorial`/config/tiago_octomap_motions.yaml
```

Afin de lancer le mouvement, il faut utiliser la ligne de commande suivante :

```
1 rosrun play_motion run_motion head_look_around
```

Quand le mouvement du node **head_look_around** est terminé, Rviz montre l'octomap construit qui représente l'environnement comme dans la figure Fig.15.9

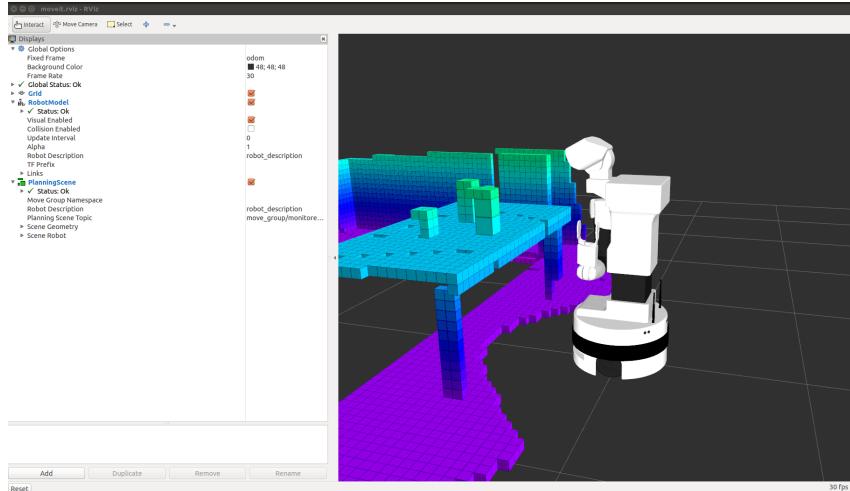


FIGURE 15.9 – Représentation de l'octomap dans Rviz

Puis, dans la même console, il faut que le robot TIAGo se place dans une pose qui permette de démarrer la planification de mouvement :

```
1 rosrun play_motion run_motion unfold_arm
```

Il faut attendre que TIAGo déploie son bras. Il est alors possible de procéder aux étapes suivantes.

Quand le node de mouvement **unfold_arm** est terminé, il est possible d'amener le référentiel de l'outil du robot TIAGo à 3 configurations cartésienne par rapport au référentiel **base_footprint**.

Les 3 configurations spatiales cartésiennes avec le référentiel **base_footprint** sont :

```
1 x: 0.508
2 y: -0.408
3 z: 0.832
4 Roll: 0.005
5 Pitch: -0.052
6 Yaw: 0.039
```

```
1 x: 0.706
2 y: 0.357
3 z: 0.839
4 Roll: 0.007
5 Pitch: -0.041
6 Yaw: 0.040
```

```
1 x: 0.5
2 y: 0.35
3 z: 0.38
4 Roll: 0.003
5 Pitch: -0.030
6 Yaw: 0.039
```

Comme dans le tutorial de la section 15.2 il est possible d'utiliser le node **plan_arm_torso_ik** du paquet **tiago_moveit_tutorial**. Il faut cependant attendre que chaque node soit fini avant de lancer le suivant.

```
1 rosrun tiago_moveit_tutorial plan_arm_torso_ik 0.508 -0.408 0.832 0.005, -0.052, 0.039
```

```
1 rosrun tiago_moveit_tutorial plan_arm_torso_ik 0.706 0.357 0.839 0.007, -0.041, 0.040
```

```
1 rosrun tiago_moveit_tutorial plan_arm_torso_ik 0.5 0.35 0.38 0.003, -0.030, 0.039
```

Il arrive que MoveIt! affiche des erreurs internes. Cela peut-être l'échec d'avoir trouver un plan ou l'impossibilité d'exécuter la trajectoire planifiée. Dans ce cas, il faut relancer la ligne de commande qui a généré l'erreur.

Les images dans la figure Fig.15.10 représentent les 3 configurations des commandes précédentes.



FIGURE 15.10 – Les 3 configurations du robot TIAGO trouvées pour les 3 configurations Cartésiennes demandés

15.3.5 Le plugin Rviz de MoveIt!

MoveIt! vient avec un plugin pour le visualiseur Rviz (voir les détails ici). Il s'agit d'une façon graphique de définir un nouveau but. Ce plugin permet mettre en place des scènes dans lequel le robot va travailler, générer les plans, visualiser le résultat et interagir directement avec un robot visualisé.

L'usage du plugin de MoveIt! avec Octomap dans nos simulations s'effectue de la façon suivante :

```
rosrun moveit_rviz.launch
```

Rviz va apparaître avec TIAGO et les marqueurs interactifs. La figure Fig.15.11 montre le plugin MoveIt! avec Octomap après les mouvements **head_look_around** et **unfold_arm**.

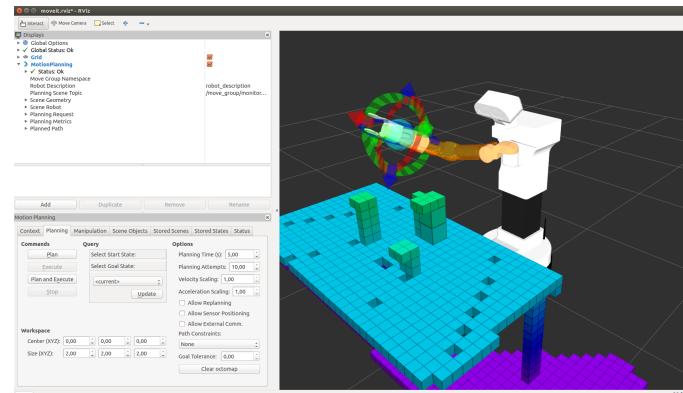


FIGURE 15.11 – TIAGO après les mouvements **head_look_around** et **unfold_arm**

15.4 Démonstration d'une prise et d'un remplacement d'un objet

Ce didacticiel est disponible en Anglais [ici](#)

15.4.1 But

Le but de ce didacticiel est montrer une exemple de manipulation avec TIAGO. L'environnement simulé comprend une table et une boîte sur laquelle un marqueur ArUco. Le robot localise l'objet grâce à sa caméra RGB et fournit la pose 3D.

Alors, MoveIt! est utilisé de façon à planifier une trajectoire pour attraper un objet, le soulever, et finalement une trajectoire est planifiée pour remettre l'objet dans sa position initiale.

15.4.2 Lancer la simulation

Dans le premier terminal il faut lancer la simulation :

```
rosrun tiago_pick_demo pick_simulation.launch
```

Gazebo se lance avec TIAGo en face de la table avec l'objet qui a le marqueur ArUco sur le dessus.

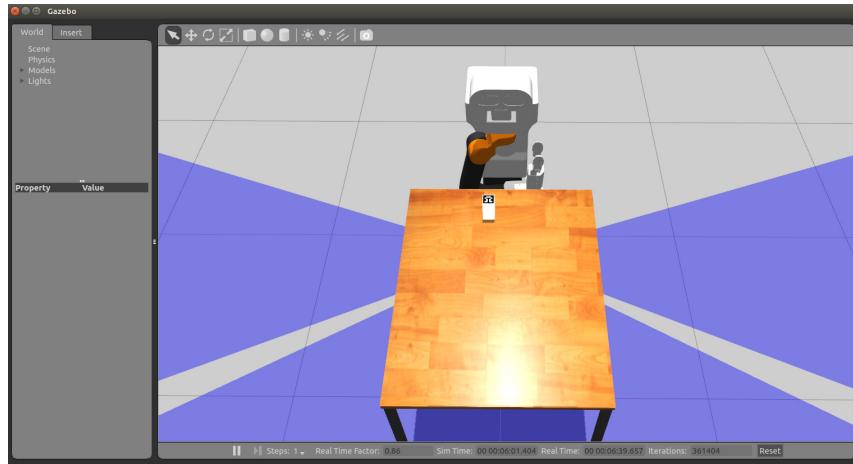


FIGURE 15.12 – Gazebo avec le robot TIAGo en face de la table et l'objet ArUco

Il faut ensuite attendre que TIAGo ait rangé son bras pour procéder aux étapes suivantes.

15.4.3 Lancer les nodes

Dans le second terminal, il faut lancer l'instruction suivante :

```
rosrun tiago_pick_demo pick_demo.launch
```

ce qui lance les nodes suivants :

- **/aruco_single** le node qui détecte le marqueur ArUco
- **/pick_and_place_server** le node en charge de définir la scène de planification, de faire la requête pour planifier les mouvements avec MoveIt!et de les exécuter.
- **/pick_client** : le node qui prépare le robot pour la détection d'objet et les opérations de pick et place : le robot lève le bras vers une pose sans collision, et ensuite baisse la tête pour regarder la table. Il attend alors que l'objet avec un marqueur soit détecté et que la pose soit reconstruite de façon à envoyer le but pour le serveur d'action **/pick_and_place_server**.
- **/rviz** : de façon à visualiser toutes les étapes impliquées dans la démonstration.

Rviz se lance pour aider l'utilisateur à visualiser les différentes étapes du didacticiel comme on peut le voir dans la figure Fig.15.13.

15.4.4 Démarrer la démonstration

Finalement, dans le troisième terminal, le service suivant sera appellé pour démarrer l'exécution de la démonstration :

```
rosservice call /pick_gui
```

Le service utilise le node **/pick_client** pour faire bouger TIAGo vers une pause permettant de détecter l'objet et de l'attraper. Le torse de TIAGo va se lever et la tête va descendre pour regarder la table. A ce point, le marqueur ArUco sera détecté et la pose estimée est affichée dans rviz.

Une fois que le marqueur est détecté la géométrie de l'objet sera reconstruit grâce à la dimension de la boîte connue à l'avance. Le modèle de l'objet sera ajouté à la scène de planification de MoveIt!ainsi qu'une grosse boîte en dessous de l'objet de façon à représenter la table.

Une fois que la scène est initialisée MoveIt!planifie plusieurs prises et sélectionne celle qui est la plus appropriée. Les différentes prises calculées sont montrées dans la figure Fig.15.15 sous la forme de flèches rouges qui représentent les poses du référentiel **arm_tool_link** qui permettent de faire une prise.

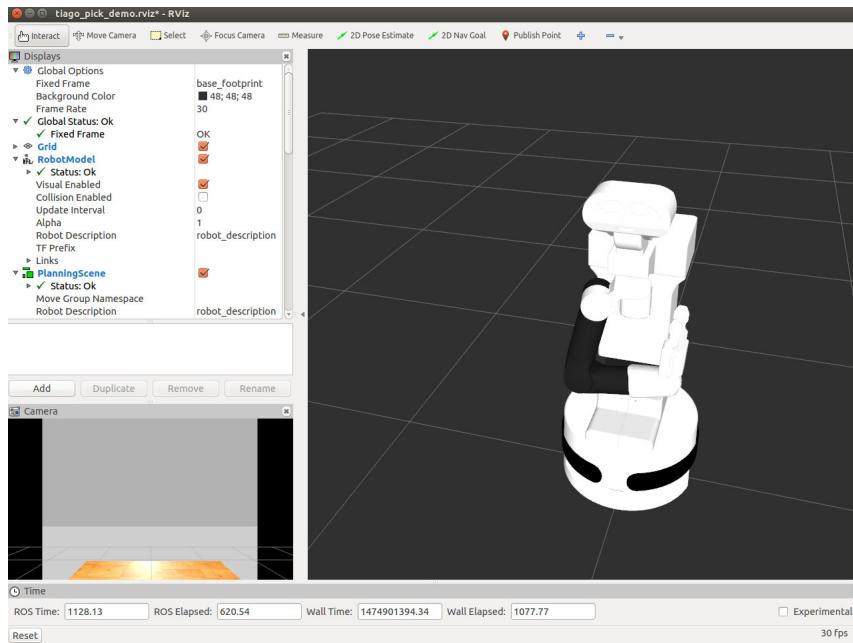


FIGURE 15.13 – Rviz visualisation de la table

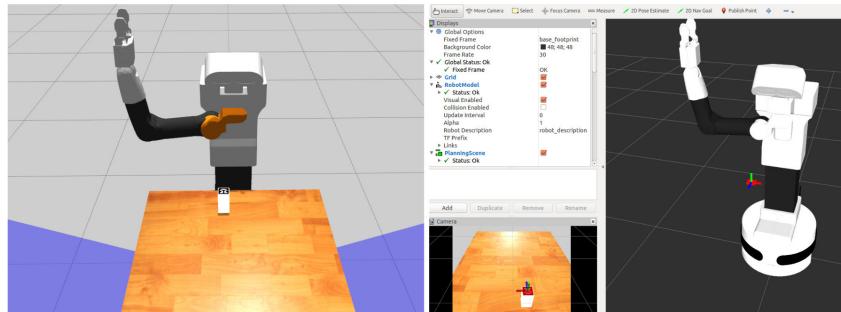


FIGURE 15.14 – Phase de détection

Un exemple d'exécution est démontrée dans la vidéo disponible ici.

Notons que chaque exécution du plan sélectionné peut être différente à cause des algorithmes de planification utilisant des marches aléatoires. Un exemple d'un tel plan est donné dans la figure Fig.15.16 Le robot commence d'abord par exécuter la trajectoire de prise planifiée contrôlant l'articulation du torse et les 7 articulations du bras.

Une fois que l'objet est pris par le robot, le robot le soulève. Ensuite une trajectoire pour reposer l'objet est planifiée pour reposer l'objet à un endroit sensiblement identique à celui où l'objet a été pris. La figure Fig.15.17 illustre une trajectoire planifiée par MoveIt!.

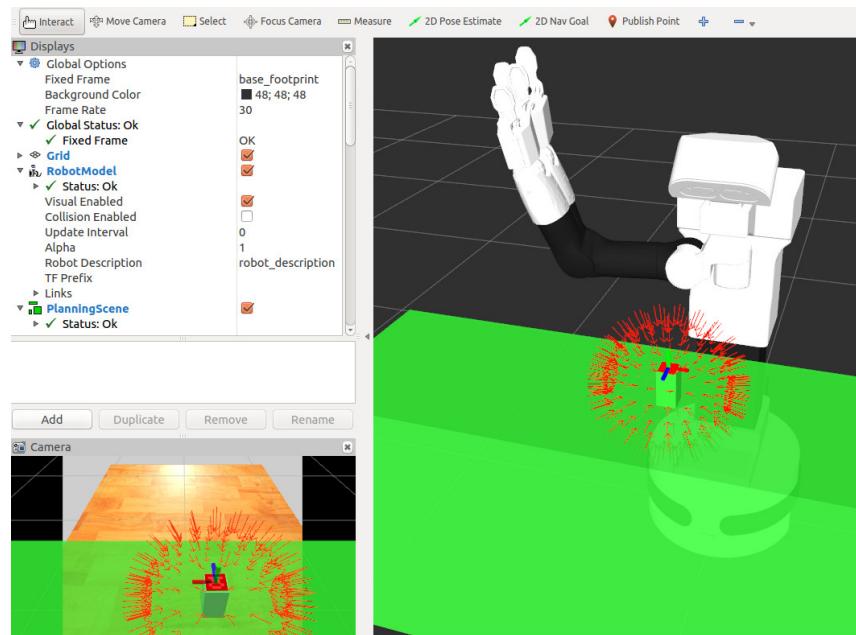


FIGURE 15.15 – Rviz affiche la scène avec laquelle le mouvement est planifié et les affordances possibles.

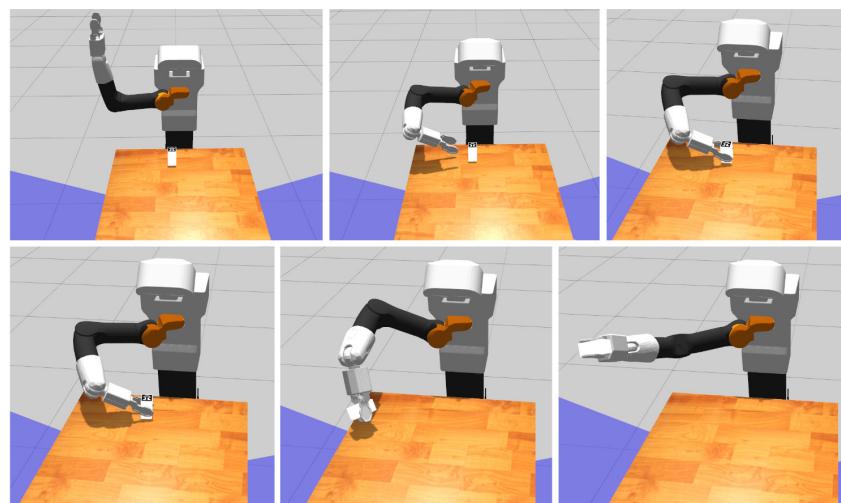


FIGURE 15.16 – Exemple du mouvement de prise par le robot TIAGo planifié par MoveIt!

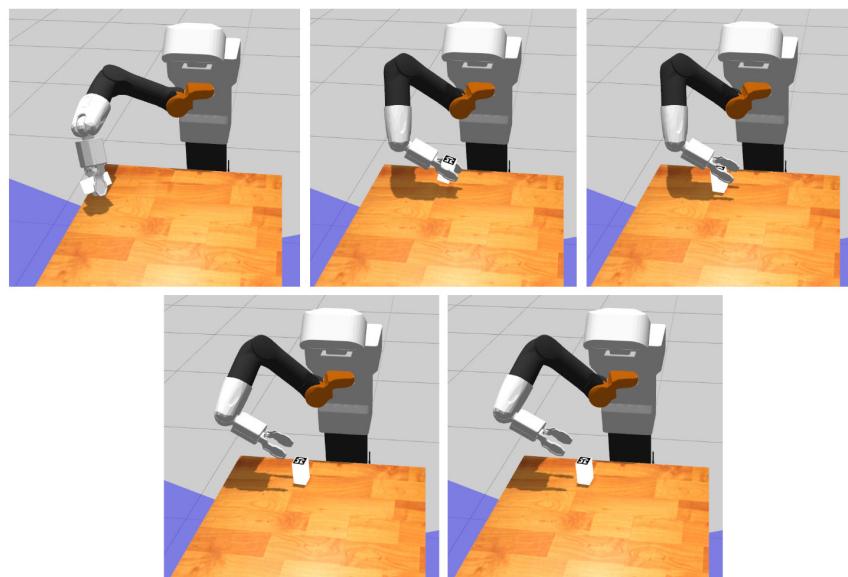


FIGURE 15.17 – Exemple du mouvement pour reposer l'objet par le robot TIAGo planifié par MoveIt!

Navigation

16	Navigation : Nav2	185
16.1	Introduction	
16.2	Concepts de Navigation	
16.3	Serveurs de navigation	
16.4	Estimation d'état	
16.5	Représentation de l'environnement	
16.6	REP-105	
17	Plugins pour la navigation	193
17.1	Introduction	
17.2	Ecrire un nouveau plugin Costmap2D	
17.3	Exporter et construire le plugin GradientLayer	
17.4	Utiliser le plugin dans Costmap2D	
17.5	Lancer le plugin GradientLayer	
18	Plugin de planification	199
18.1	Introduction	
18.2	Création d'un nouveau plugin de planification de mouvements	
18.3	Exporter le plugin de planification	
18.4	Passer le nom du plugin par le fichier de paramètres	
18.5	Lancer le plugin StraightLine	
19	Ecrire un nouveau plugin de contrôleur	203
19.1	Introduction	
19.2	Créer un nouveau plugin contrôleur	
19.3	Exporter le plugin du contrôleur	

16. Navigation : Nav2

16.1 Introduction

Les éléments suivants ont pour source <https://navigation.ros.org/> et en sont pour une large partie la traduction.

Le projet Nav2 est le successeur spirituel de la pile de Navigation ROS. Ce projet cherche à trouver un chemin sûr pour un robot mobile afin d'aller d'un point A à un point B. Il peut également utiliser dans d'autres applications qui implique de la navigation comme suivre des points dynamiques. Ceci implique de la planification dynamique, de calculer des vitesses pour les moteurs, d'éviter des obstacles, et des comportements de recouvrements. Il est possible d'avoir plus d'informations ici : <https://navigation.ros.org/about/index.html#about>.

Nav2 utilise les arbres de comportement (BT pour Behavior Trees en Anglais) pour appeler les serveurs modulaires afin de réaliser une action. Une action peut consister à calculer un chemin, une commande en effort, d'effectuer une action de réparation, ou tout autre tâche liée à la navigation. Chacun est un node séparé qui communique avec l'arbre de comportement (AC-BT) à travers un serveur d'action ROS. Le diagramme 16.1 donne une bonne vue globale de la structure de Nav2.

Note 16.1 Il est possible d'avoir des plugins pour les contrôleurs, les planificateurs, et les comportements de récupération avec les plugins de comportement (AC-BT) correspondants. Cela permet de créer des comportements de navigation contextualisés. Une comparaison entre ce projet et la navigation ROS-1 est disponible ici. ▶

Les entrées attendues de Nav2 sont des transformations TF conformément à la REP-105, une carte source si la couche Static Costmap est utilisée, un fichier BT XML, et toutes informations pertinentes issues de capteurs. Il fournira alors des commandes en vitesse valide pour tout robot holonomique ou non holonomiques. Le système supporte tous les types majeurs de robots : holonomique, à roues différentielles, à pattes, et les bases ackermann (voiture). Ils ne sont supportés qu'à travers des formes circulaires et des formes fixées arbitrairement pour de la détection de collision dans SE(2).

Les outils sont les suivants :

- Charge, fournit et sauve des cartes (Map Server)
- Localise le robot dans une carte (AMCL)
- Planifie un chemin du point A au point B autour des obstacles (Nav2 Planner)
- Contrôle le robot pour qu'il suive le chemin (Nav2 Controller)
- Lisse les chemins planifiés pour les rendre continus et faisables (Nav2 Smoother)

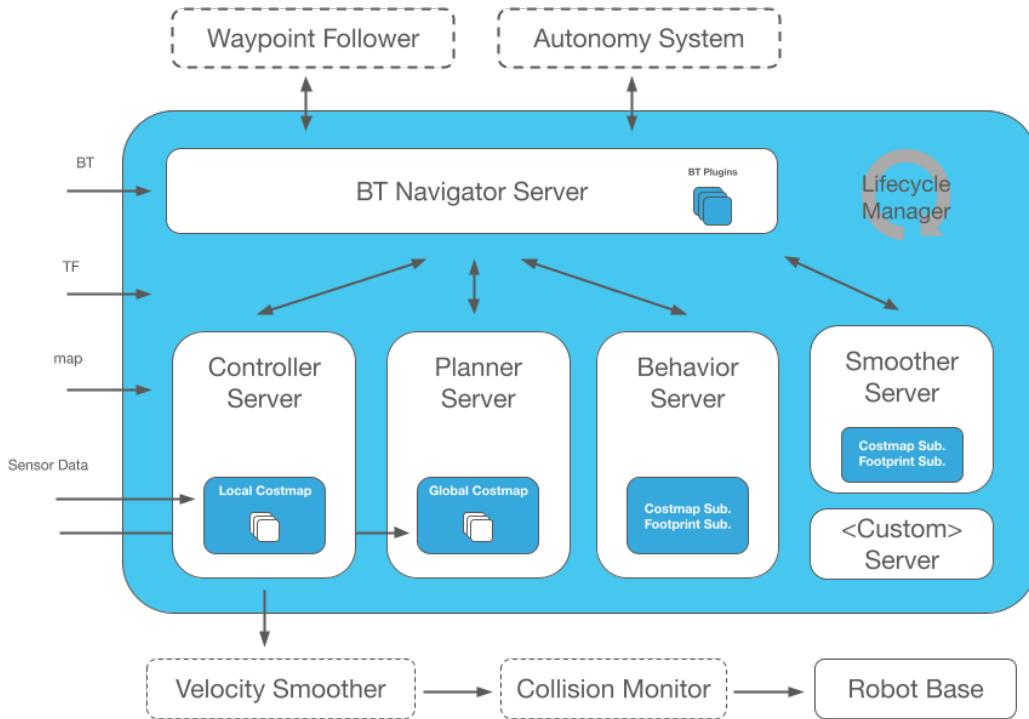


FIGURE 16.1 – Architecture globale de la pile de navigation Nav2

- Convertit les données capteurs dans une carte coût représentant le monde (Nav2 Costmap 2D)
- Construit des comportements de robot complexes en utilisant des arbres de comportements (Nav2 Behavior Trees and BT Navigator)
- Suit des successions de points intermédiaires (Nav2 Waypoint Follower)
- Gère le cycle de vie et les sécurités (chien de garde) des serveurs (Nav2 Lifecycle Manager)
- Des plugins pour permettre de faire votre propre algorithme et comportement (Nav2 Core)

Nav2 fournit également un ensemble de plugins de démarrage. NavFn calcule le chemin le plus court d'une pose vers une pose cible en utilisant A* ou l'algorithme de Dijkstra. DWB utilise l'algorithme DWA pour calculer une commande en effort pour suivre un chemin, avec plusieurs plugins pour évaluer les trajectoires. Il y a des comportements de réparation qui inclusent : attente, tourner sur place, effacer les cartes de coût, et revenir. Un ensemble de plugins BT sont également fournis. Ils appellent les serveurs pré-cités et calcule les informations nécessaires. Finalement un ensemble de plugins rviz pour interagir avec la pile de navigation et contrôler le cycle de vie. Une liste de tous les plugins reportée par les utilisateurs est disponible ici.

16.2 Concepts de Navigation

Cette section est issue du lien disponible ici. Elle est résumée ici puisque de nombreux concepts généraux sont disponibles dans les autres parties du document. Seules les informations spécifiques à la pile de Navigation sont reportées.

16.2.1 Serveurs d'action

Les serveurs d'actions sont utilisés dans cette pile pour communiquer avec l'arbre de comportement AC-BT de navigation à travers le message d'action **NavigateToPose**. Ils sont aussi utilisés par le AC-BT de navigation pour communiquer avec les autres, plus petits, serveurs d'actions pour calculer les plans, les commandes d'efforts et de récupérations. Chacun a ses propres types d'actions (**.action**) dans le package **nav2_msgs** pour interagir avec les serveurs.

16.2.2 Les nodes avec cycles de vie

Les nodes avec cycles de vie (Lifecycle ou Managed nodes en Anglais) sont nouveaux dans ROS-2. Ces nodes contiennent des transitions de machines à états pour démarrer et arrêter des serveurs ROS-2. Ceci permet d'obtenir des comportements déterministiques au démarrage et à l'arrêt. Cela permet également aux utilisateurs de structurer leur programmes d'une manière raisonnable pour des utilisations commerciales et la résolution de problèmes.

Quand un node démarre, il est dans un état non-configuré **unconfigured**, ne traitant que le constructeur du Node qui ne doit pas contenir aucun mécanisme de mise en place ROS du réseau, ni de lecture de paramètres. À travers le système de launch, ou par le gestionnaire de cycle de vie, les nodes doivent être passés à inactif **inactive** par configuration. Logiquement, il est ensuite possible d'activer le node en faisant la transition à travers l'étape d'activation.

Cet état permet au node de traiter les informations et d'être pleinement configuré pour fonctionner. L'étape de configuration, déclenchée par la méthode **on_configure()**, doit s'occuper de spécifier tous les paramètres, les interfaces réseaux ROS, et pour les systèmes de sécurité toutes les allocations de mémoire. L'étape d'activation, démarre l'appel à la méthode **on_activate()** active les interfaces réseaux ROS et spécifie tous les états nécessaires dans le programme pour démarrer le traitement des informations.

Pour arrêter, les transitions nécessaires sont la désactivation, le nettoyage (par exemple de la mémoire), l'arrêt des processus, pour terminer dans l'état **finalized**. Les interfaces réseaux sont désactivées et le traitement des informations est arrêté, la mémoire est désallouée, et le node termine proprement.

L'approche node avec cycle de vie est utilisée fréquemment à travers le projet et tous les serveurs l'utilise. C'est une bonne pratique recommandée pour tous les systèmes ROS d'utiliser des nodes de cycle de vie quand cela est possible.

Nav2 utilise une encapsulation de LifeCycleNodes : **nav2_utilLifeCycleNode**. Cette encapsulation englobe l'ensemble de la complexité de LifeCycleNode pour des applications typiques. Elle inclue également une connection **bond** pour le gestionnaire de cycle de vie pour s'assurer que lorsqu'un serveur démarre, il reste actif. Lorsqu'un serveur crache, il informe le gestionnaire de cycle de vie et permet au système de se mettre dans un état évitant un échec critique.

16.2.3 Arbre de Comportement - Behavior Trees

Les arbres de comportement sont devenus très populaires dans les tâches robotique communes. Ils correspondent à une structure en arbre de tâches à compléter. Cela permet de créer un cadre plus compréhensible et passant bien à l'échelle pour définir des applications avec plusieurs états et plusieurs étapes. C'est une alternative aux machines d'états finis (FSM) qui peuvent avoir des douzaines d'états et des centaines de transitions. Un exemple est un robot jouant au football. Embarquer la logique d'un jeu de football dans une FSM est difficile et générateur d'erreur. De plus, des choix de modèles pour tirer dans un but de la droite, de la gauche ou du centre peut-être particulièrement compliqué à comprendre. Avec un BT, des primitives de base comme "tirer", "marche", "aller à la balle" peuvent être créées et réutilisées pour plusieurs comportements. Plus d'informations sont disponibles dans ce livre. Il est fortement recommandé de lire les 3 premiers chapitres pour avoir une bonne compréhension de la nomenclature et du l'enchaînement. Cela ne devrait prendre que 30 minutes.

Pour ce projet, la librairie BehaviorTree CPP V3 est utilisée. Des nodes à travers des plugins peuvent être construits dans un arbre dans le **BTNavigator**. Les plugins de node sont chargés dans le BT, et quand le fichier XML de cet arbre est lu, les noms enregistrés sont associés. Il est alors possible d'explorer l'arbre de comportement pour naviguer.

Une des raisons pour laquelle cette librairie est utilisée est sa capacité à charger des sous-arbres. Cela veut dire qu'un arbre de comportement Nav2 peut-être chargé dans un autre BT de haut niveau pour utiliser ce projet comme un plugin de node. Un exemple pourrait être dans un jeu de footbal, où l'arbre de navigation Nav2 serait le node "go to ball" avec une détection de balle comme une partie d'une tâche plus large. De plus un plugin **NavigateToPoseAction** pour les BT est fourni afin que la pile Nav2 puisse être appelée par une application client à travers l'interaction d'action habituelle.

16.3 Serveurs de navigation

Les planificateurs et les contrôleurs sont au coeur d'une tâche de navigation. Les récupérations sont utilisées pour sortir le robot d'une situation difficile et tenter de gérer plusieurs types de problèmes afin de

rendre le système résistant aux fautes. Les lisseurs peuvent être utilisées pour améliorer la qualité du chemin planifié. Dans cette section les concepts généraux associés et leur utilisation dans ce projet sont analysés.

16.3.1 Planificateurs, Contrôleurs, Lisseurs et serveurs de récupération

Quatres types de serveurs d'actions importants existent dans ce projet. Ce sont le planificateur, le serveur de comportements, le lisseur et le contrôleur.

Ces serveurs d'action sont utilisés pour stocker une carte de plugins d'algorithmes pour réaliser diverses tâches. Ils stockent également la représentation de l'environnement utilisés par les plugins d'algorithmes pour calculer leurs sorties.

Le planificateur, le lisseur et le contrôleur sont configurés à l'exécution avec les noms (alias) et les types d'algorithmes à utiliser. Ces types sont les noms de pluginlib qui ont été enregistrés et les noms sont les alias pour les tâches. Un exemple pourrait être le contrôleur DWB utilisé avec le nom **FollowPath**, qui suit un chemin de référence. Dans ce cas, alors tous les paramètres de DWB seront placés dans l'espace de nom e.g. **FollowPath.<param>**.

Ces 3 serveurs exposent alors une interface d'action correspond à leur tâche. Quand le BT arrive au noeud BT correspondant, il va appeler le serveur d'action pour effectuer la tâche. La fonction de traitement (ou callback function) du serveur d'action appelle ensuite l'algorithme désigné par son nom (i.e. **FollowPath**) qui le lie avec un algorithme spécifique. Cela permet à l'utilisateur d'abstraire l'algorithme utilisé dans l'arbre de comportement à des classes d'algorithmes. Par exemple, on peut avoir N plugins de contrôleurs pour suivre des chemins, se connecter à un chargeur, éviter des obstacles dynamiques, ou s'interfacer avec un outil. Avoir tous ces plugins dans le même serveur permet à l'utilisateur d'utiliser un seul objet pour la représentation de l'environnement qui est coûteuse à dupliquer.

Pour le serveur de comportements, chacun des comportements contient son propre nom, toutefois chaque plugin expose également son propre serveur d'action. Ceci est réalisé car la large variété des actions de comportement qui peuvent être créées ne peuvent avoir une interface simple à partager. Le serveur de comportement contient également un souscripteur de carte de coût à une carte locale de coût. Il reçoit ainsi des mises à jour en temps-réel du serveur de contrôleurs pour calculer ses tâches. Ceci permet d'éviter d'avoir de multiples instances de la carte de coût qui est chère à dupliquer.

Alternativement comme les nodes BT sont des plugins triviaux appelant une action, de nouveaux nodes BT peuvent être créés pour appeler des serveurs d'actions avec d'autres types d'actions. Il est recommandé d'utiliser les serveurs fournis le plus possible. Si à cause des interfaces des plugins et des interfaces, un nouveau serveur est nécessaire, il peut être intégrer dans le cadre du projet. Le nouveau serveur doit utiliser le nouveau type et le plugin d'interface, de façon similaire aux serveurs fournis. Un nouveau plugin de node BT doit être créé pour appeler le nouveau serveur d'actions. Cependant il n'y a pas besoin de fork ou de modifier la repository de Nav2 en utilisant de façon intensive les serveurs et les plugins.

16.3.2 Planificateurs

La tâche d'un planificateur est de calculer un plan pour minimiser une fonction objectif. Le chemin peut-être aussi connu comme une route, suivant la nomenclature et de l'algorithme sélectionné. Deux exemples canoniques sont le calcul d'un plan pour atteindre un but (i.e. d'une position courante à un but) ou pour effectuer une couverture complète d'un espace. Le planificateur a accès à une représentation globale de l'environnement et les données capteurs.

Les planificateurs sont écrits pour :

- Calcul de plus court chemin
- Calcul d'un chemin de couverture
- Calcul des chemins le long de chemins connus et éparses

La tâche générale dans Nav2 du planificateur est de calculer un chemin valide, et potentiellement optimal, de la pose courante vers la pose ciblée. Plusieurs classes de plans et de routes existent et sont supportés.

16.3.3 Contrôleurs

Les contrôleurs, également connus comme des planificateurs locaux dans ROS-1, sont les mécanismes qui permettent de suivre un chemin désiré calculé de façon global, ou de réaliser une tâche locale. Les contrôleurs vont avoir accès à une représentation locale de l'environnement pour essayer de calculer des commandes en effort faisables pour la base. Plusieurs contrôleurs vont considérer une fenêtre de temps limité sur laquelle ils calculent un chemin local faisable à chaque itération. Les contrôleurs sont écrits pour :

- Suivre un chemin
- S'interfacer avec une station de chargement en utilisant des détecteurs dans le répère odométrique
- Rentrer dans un ascenseur
- S'interfacer avec un outil

La tâche générale dans Nav2 pour un contrôleur est de calculer une commande en effort valide pour suivre un plan général. Toutefois, plusieurs classes de contrôleurs et planificateurs locaux existent. Dans le cadre de ce projet, tous les algorithmes de contrôleurs peuvent être des plugins dans le serveur de planification.

16.3.4 Comportements

Les comportements de récupération sont un pilier des systèmes tolérants aux fautes. Le but des comportements de récupérations sont de gérer des situations inconnues et des conditions d'échecs de façon autonome. Les exemples peuvent inclure des erreurs dans le système de perception dont le résultat dans la représentation de l'environnement peut-être de faux obstacles. Effacer la carte de coût peut-être déclenché afin de permettre au robot de bouger.

Un autre exemple de blocage est un robot bloqué par des obstacles dynamiques ou un mauvais contrôle. Revenir en arrière ou tourner sur place, si cela est possible, permet au robot de s'éloigner d'une mauvaise position pour aller dans un espace libre où le robot peut naviguer correctement.

Finalement, dans le cas d'un échec plus important, un comportement de récupération peut-être d'appeler un opérateur pour demander de l'aide. Ceci peut-être fait via un email, un SMS, Slack, Matrix, etc...

Il est important de noter que le serveur de comportements peut gérer tout type de comportement pour partager l'accès à des ressources chères comme des cartes de coût ou des buffers TFs, pas uniquement des comportements de récupération. Chacun peut avoir leurs propres API.

16.3.5 Lisseurs

Les critères d'optimalité des planificateurs sont souvent insuffisant pour réaliser le chemin dans la réalité. Il est donc nécessaire d'affiner le chemin. Les serveur d'action pour le lissage ont été introduits pour cette raison. Ils sont destinés à réduire la discontinuité des chemins et des rotations, mais également à accroître la distance aux obstacles et des zones à haut risques car les lisseurs ont accès à une représentation global de l'environnement.

L'utilisation d'un lisseur séparé par rapport à un lisseur inclus dans un planificateur est avantageux quand on le combine avec d'autres planificateurs et avec d'autres lisseurs. Ceci est également intéressant si le lissage est réalisé par un contrôle spécifique sur une partie spécifique du chemin.

La tâche générale dans Nav2 d'un lisseur est de recevoir un chemin et de retourner une version améliorée. Il est possible d'utiliser différents chemins d'entrées, des critères d'améliorations, et les méthodes pour les acquérir.

16.3.6 Suiveur de point intermédiaires

Le suivi de points intermédiaires est une fonctionnalité de base d'un système de navigation. Il dit à notre système comment utiliser la navigation pour atteindre de multiples destinations.

Le serveur **nav2_waypoint_follower** contient un programme de suivi d'un point intermédiaire avec une interface de plugin pour les exécuteurs de tâches spécifiques. C'est utile si le robot doit aller à un endroit spécifique et d'effectuer une tâche comme prendre une image, prendre une boîte, ou attendre une entrée de l'utilisateur.

Il existe deux écoles de pensée pour la gestion de flotte de robots : (un robot stupide, un système de gestion de flotte intelligent) ou (un robot intelligent, un système de gestion de flotte stupide).

Dans la première école de pensée, le serveur **nav2_waypoint_follower** est amplement suffisant pour créer une solution sur le robot prête pour un produit. Comme le système autonome / gestionnaire prend en compte la pose du robot, son niveau de batterie, la tâche courante, et plus quand il assigne des tâches, l'application sur le robot doit simplement s'occuper de la tâche à faire et non de toute la complexité du système pour réaliser la tâche. Dans cette situation, il est préférable de penser à la requête de suivi de point intermédiaire comme une unité de travail (i.e. prendre un objet dans un entrepôt, une boucle de patrouille, une aile) de faire une tâche et puis de retourner au gestionnaire pour la prochaine tâche, et de demander une recharge. Dans cette école de pensée, l'application de suivi de points est juste une étape au-dessus de la navigation et sous l'application de système autonome.

Dans la second école de pensée, le serveur **nav2_waypoint_follower** est un exemple d'application intéressante. On peut le voir comme une preuve de concept. Mais il est nécessaire d'avoir son propre système de suivi de points intermédiaires pour que le robot puisse transporter plus de poids afin de faire une solution robuste. Dans ce cas, l'utilisateur est fortement invité à utiliser le paquet **nav2_behavior_tree** pour créer une application utilisant un arbre de comportement (BT) utilisant la navigation pour réaliser la tâche. Ceci peut inclure des sous-arbres comme vérifier la charge de la batterie pour éventuellement faire revenir le robot à sa station de charge, ou effectuer plus d'une unité de tâche de travail dans une tâche plus complexe. Il est prévu de faire un paquet **nav2_bt_waypoint_follower** qui devrait permettre de faire ces applications plus aisément. Dans cette école de pensée, l'application de suivi de points intermédiaires est lié plus intimement au système autonome, et dans beaucoup de cas, est l'autonomie du système.

L'un n'est pas meilleur que l'autre, et le choix dépend très fortement des tâches à effectuer par les robots, du type d'environnement, et des ressources de calcul distribué disponibles. Souvent cette distinction est très claire suivant le cas de figure.

16.4 Estimation d'état

Dans le projet de navigation, il y a 2 transformations majeures qui doivent être fournies selon les standards de la communauté. La transformation de **map** à **odom** est fournie par le système de positionnement (localisation, mapping, SLAM). La transformée de **odom** à **base_link** est elle fournie par un système d'odométrie.

Note 16.2 Il n'y aucune obligation à utiliser un LIDAR sur le robot pour utiliser le système de navigation. Il n'y a aucune obligation d'utiliser un système d'évitement de collision basé LIDAR. Il est possible d'utiliser la vision par ordinateur ou des capteurs fournissant la profondeur pour l'évitement de collision. La seule requête est suivre les standards décrits dans le paragraphe suivant avec le choix d'implémentation.

■

16.4.1 Standards

REP 105 définit les repères et les conventions requises pour la navigation et le large écosystème de ROS. Les conventions doivent être tout le temps suivies pour utiliser les projets disponibles concernant l'odométrie, la localisation et les projets de SLAM fournis par la communauté.

Pour résumer, REP-105 spécifie qu'il faut au minimum que l'arbre TF doit contenir le chemin complet : **map -> odom -> base_link -> [sensor°frames]** pour le robot. TF2 est la librairie de transformations variant dans le temps qui est utilisée dans ROS-2 pour obtenir et représenter les transformations synchronisées dans le temps. C'est le travail du système de positionnement global (GPS, SLAM, Motion Capture) de fournir au minimum la transformation **odom -> base_link**. Le reste de la transformation relative à **base_link** doit être statique et définie dans l'URDF du robot.

16.4.2 Localisation globale et SLAM

Le travail du système de positionnement global (GPS, SLAM, Motion Capture) est de fournir au minimum la transformation **map -> odom**. Le paquet **amcl** du projet est une technique Adaptative de Localisation Monte-Carlo basée sur une filtre particulière pour se localiser dans une carte statique. Une boîte à outils de SLAM est l'algorithme de SLAM par défaut pour positionner le robot et construire une carte statique.

Ces méthodes peuvent également produire d'autres sorties des topics pour fournir la position du robot, des cartes, et d'autres métadonnées, mais elles doivent fournir cette transformation pour être valide. Des méthodes de positionnement multiples peuvent être fusionnées ensemble en utilisant la localisation du robot discutée précédemment.

16.4.3 Odométrie

Le rôle du système d'odométrie est de fournir la transformation **odom -> base_link**. L'odométrie peut venir de plusieurs sources notamment LIDAR, RADAR, les encodeurs des roues, l'odométrie visuelle (VIO), et les IMUs. Le but de l'odométrie est de fournir un repère local lisse et continu basé sur le mouvement du robot. Le système de positionnement global va mettre à jour la transformation relative au repère global pour prendre en compte le décalage de l'odométrie.

Le paquet Robot Localization est typiquement utilisé pour cette fusion. Il va prendre **N** capteurs de différents types et fournir une odométrie continue et lisse à l'arbre des transformations (TF) et sur un topic.

Une application mobile typique peuvent avoir l'odométrie à partir des encodeurs des roues, des IMUs, et la vision fusionnée de cette manière.

La sortie lisse peut-être utilisée comme estimation pour faire des mouvements précis et mettre à jour la position du robot précisément entre deux mises à jour de la position global du robot.

16.5 Représentation de l'environnement

La représentation de l'environnement par le robot est l'objet qui est utilisé pour combiner l'information venant de plusieurs sources dans un seul espace. Cet espace est alors utilisé par les contrôleurs, les planificateurs, et les mécanismes de récupération pour calculer les tâches efficacement et de façon sûre.

16.5.1 Les cartes de coût et les couches

La représentation de l'environnement utilisée à l'heure actuelle est une carte de coût. Une carte de coût est une grille régulière de cellules contenant un coût représentant les parties inconnues, libres, occupées ou des coûts gonflés. La carte de coût est utilisée pour calculer un plan global ou échantillonné pour calculer des commandes en couple locales.

Plusieurs couches de carte de coût sont implementées comme des plugins pluginlig pour stocker les informations. Ceci inclut des informations des LIDARs, des RADAR, des sonars, de la profondeur, des images, etc... Il peut être nécessaire de traiter les données capteurs avant de les insérer dans la couche du coût, mais cela dépend de la façon dont est exploitée cette carte de coût.

Les couches de cartes de coût peuvent être créées pour détecter et suivre les obstacles dans la scène pour les évitements de collision en utilisant la caméra, et les capteurs de profondeur. De plus, les couches peuvent être créées pour changer algorithmiquement la carte de coût globale en utilisant des règles ou des heuristiques. Finalement, elles doivent être utilisées pour stocker des données en ligne dans les représentations 2D ou 3D pour marquer les obstacles binairement.

16.5.2 Filtres de cartes de coût

Il est possible d'annoter une carte dans un fichier de façon à avoir une action spécifique qui soit exécuté en fonction de la position dans la carte annotée. Les exemples d'utilisation des annotations sont des zones d'exclusion pour éviter de planifier des mouvements à l'intérieur, ou limiter la vitesse. Ces cartes annotées sont appelées "masques de filtre". Ces zones peuvent être, ou ne pas être, différentes de la carte principale en taille, pose, ou échelle. Le but principal de ces filtres de masque est de fournir la possibilité de marquer des zones avec des caractéristiques supplémentaires ou des changements de comportements.

Les filtres de cartes de coût sont une approche basée sur des couches de cartes de coût. Cela permet de réaliser des changements de comportements basé sur des informations spatiales annotées dans des masques/filtres. Les filtres de carte de coût sont implantés par des plugins de carte de coût. Ces plugins sont appelés des filtres car ils filtrent une carte de coût par des annotations spatiales spécifiées dans un masque/filtre. De façon à filtrer une carte de coût et pour changer le comportement du robot dans les zones annotées, les plugins des filtres lisent les informations qui viennent du masque/filtre. Cette donnée est transformée dans une carte de caractéristique (feature map) dans un espace filtre. Une fois cette carte de caractéristique transformée avec une carte de coût, toute données de capteurs peut mettre à jour la carte de coût sous-jacente et changer le comportement du robot en fonction de l'endroit où il se trouve. Par exemple, les fonctionnalités suivantes peuvent être réalisées grâce à l'utilisation de filtres de cartes de coût :

- Définition de zones où les robots ne vont jamais entrer ou seront en sécurité.
- Des zones de restriction de vitesses en limitant la vitesse maximal du robot.
- Spécification de routes préférées pour des robots évoluant dans des environnements industriels ou des entrepôts.

16.6 REP-105

Cette REP spécifie les conventions de noms à utiliser pour les repères des plateformes mobiles à utiliser pour ROS.

16.6.1 Spécifications

Les noms des repères sont :

- **base_link** : Il s'agit du repère attaché rigidement à la base du robot mobile. Il peut-être attaché à n'importe quel partie de la base qui fournit un point de référence évident. La REP 103 spécifie une direction de préférence. (x : devant, y : gauche, z : haut)
- **odom** : Ce repère de coordonnées est fixe dans le monde. La pose d'une base mobile dans le repère **odom** peut dériver sans limite. Cette dérive rend le repère **odom** inutile comme référence globale à long-terme. Toutefois, la pose d'un robot dans le repère **odom** est continue, ce qui signifie que l'évolution de la base mobile évolue d'une façon continue sans saut ou discontinuité.
- **map** : Ce repère de coordonnées est fixe dans le monde et son axe Z pointe vers le haut. La pose d'une plateforme mobile est exprimée dans le repère **frame** et ne doit pas dériver à travers le temps. Le repère **map** n'est pas continu, ce qui signifie que la pose d'une plateforme mobile dans le repère **map** peut changer abruptement à tout moment.

16.6.2 Relation entre repères



17. Plugins pour la navigation

17.1 Introduction

Ce tutoriel est disponible ici. Il décrit comment écrire son propre plugin de type Costmap2D.

Il est possible d'avoir une introduction (en Anglais) qui décrit les concepts liés aux couches de Costmap2D et les principes de bases des plugins.

17.1.1 Pré requis

La suite suppose que ROS-2, Gazebo et les paquets Turtlebot 3 sont installés et construits localement. Il faut également que le projet Navigation 2 ait été installé.

17.2 Ecrire un nouveau plugin Costmap2D

A des fins de démonstration, cet exemple va créer un plugin de carte de coût avec des gradients de coûts qui se répètent dans la carte de coût. Le code annoté est disponible dans le dépôt `navigation2_tutorials`, et plus précisément dans le paquet `nav2_gradient_costmap_plugin`. Il est recommandé de s'y référer lorsque vous faites votre propre plugin.

Le plugin de la classe `nav2_gradient_costmap_plugin` : `:GradientLayer` est hérité de la classe `nav2_costmap_2d` : `:Layer` :

```
1 namespace nav2_gradient_costmap_plugin
2 {
3
4     class GradientLayer : public nav2_costmap_2d::Layer
```

La classe de base fournit l'ensemble des méthodes virtuelles de l'API pour travailler avec des couches de cartes de coûts dans un plugin. Ces méthodes sont appelées à l'exécution par `LayeredCostmap`. La liste des méthodes, leurs descriptions et la nécessité d'avoir ces méthodes dans le code d'un plugin sont présentés dans la table 17.1.

Dans l'exemple qui nous intéresse ces méthodes ont la fonctionnalité suivante :

1. `GradientLayer` : `:onInitialize()` contient les déclarations des paramètres ROS avec leurs valeurs par défaut :

```
1 declareParameter("enabled", rclcpp::ParameterValue(true));
2 node_->get_parameter(name_ + "." + "enabled", enabled_);
```

et initialise l'indicateur de recalcul des limites :

Virtual method	Method description	Requires override
onInitialize()	Cette méthode est appelée à la fin de l'initialisation d'un plugin. Elle est généralement constituée de déclarations de paramètres ROS. C'est ici que toute initialisation requise doit prendre place.	No
updateBounds()	Cette méthode est appelée pour demander au plugin quelle partie de la couche de la carte de coût doit être mise à jour. Il y a 3 paramètres d'entrées à la méthode : la position du robot en 2d et son orientation. et 4 paramètres de sortie : les pointeurs vers les limites de la fenêtre. Ces limites sont utilisées pour des raisons de performances : pour mettre à jour la surface à l'intérieur de la fenêtre où de nouvelles informations sont disponibles, évitant ainsi de faire des mises à jour sur l'ensemble de la couche de la carte de coût.	Yes
updateCosts()	Cette méthode est appelée à chaque fois que l'évaluation de la carte de coût est nécessaire. La mise à jour ne s'effectue que dans la région de la fenêtre des limites. Cette méthode a 4 paramètres d'entrée : les limites de la fenêtre, et un paramètre de sortie : la référence vers la carte de coût master_grid . La classe Layer fournie au plugin une carte de coût interne nommée costmap_ pour les mises à jour. master_grid doit être mise à jour avec les valeurs dans la fenêtre des limites en utilisant une des méthodes de mise à jour suivantes : updateWithAddition() , updateWithMax() , updateWithOverWrite() ou updateWithTrueOverwrite() .	Yes
matchSize()	La méthode est appelée à chaque fois qu'une taille de carte a changé.	No
onFootprintChanged()	La méthode est appelée à chaque fois que le plan a changé.	No
reset()	La méthode est appelée à chaque fois que la carte doit être réinitialiser.	Yes

Tableau 17.1 – Les méthodes virtuelles du plugin costmap_2d

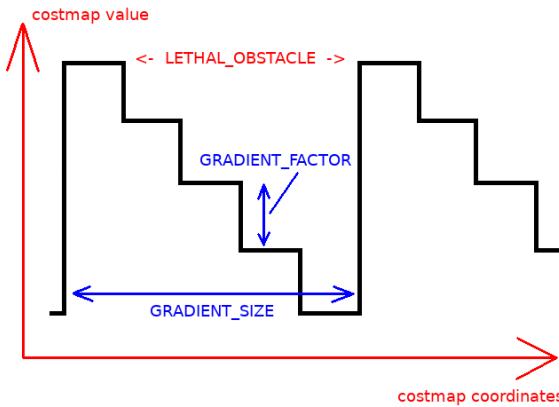


FIGURE 17.1 – Gradient explanations

```
1 need_recalculation_ = false;
```

2. **GradientLayer : :updateBounds()** re-initialise les limites de la fenêtre si **need_recalculation_** est égale à **true**, et sinon effectue une comparaison avec les dernières sauvegardées et prend les plus petites limites. La méthode initiale est une méthode pure virtuelle.
3. **GradientLayer : :updateCosts()** dans cette méthode, le gradient est écrit directement dans la carte de coût résultante **master_grid** sans mélanger les couches précédentes. Ceci est égale à travailler avec le pointeur **costmap_** interne et ensuite appelé la méthode **updateWithTrueOverwrite()**. Voici l'algorithme qui fait le gradient pour la carte de coût générale :

```
1 int gradient_index;
2 for (int j = min_j; j < max_j; j++) {
3     // Reset gradient_index each time when reaching the end of re-calculated window
4     // by OY axis.
5     gradient_index = 0;
6     for (int i = min_i; i < max_i; i++) {
7         int index = master_grid.getIndex(i, j);
8         // setting the gradient cost
9         unsigned char cost = (LETHAL_OBSTACLE - gradient_index*GRADIENT_FACTOR)%255;
10        if (gradient_index <= GRADIENT_SIZE) {
11            gradient_index++;
12        } else {
13            gradient_index = 0;
14        }
15        master_array[index] = cost;
16    }
17}
18}
```

où la variable **GRADIENT_SIZE** est la taille de chaque période du gradient dans les cellules de la carte, **GRADIENT_FACTOR** décrémente la valeur de la carte de coût à chaque étape. Ces paramètres sont définis dans le fichier d'en-tête du plugin.

4. **GradientLayer : :onFootprintChanged()** simplement réinitialise la valeur **need_recalculation_**
5. La méthode **GradientLayer : :reset()** ne sert à rien : elle n'est pas utilisée dans cet exemple.

17.3 Exporter et construire le plugin GradientLayer

Le plugin est chargé à l'exécution à travers sa classe mère et est ensuite appellée par le plugin manipulant les modules (pour les cartes de coût 2D par **LayeredCostmap**). Pluginlib ouvre un plugin donné à l'exécution et fournit des méthodes des classes exportées qui peuvent être appelées. Le mécanisme d'exportation des classes spécifie à pluginlib qu'elle classe de base utilisée durant ces appels. Ceci permet d'étendre une application par des plugins sans connaître le code source de l'application ou la recompiler.

Dans l'exemple la classe du plugin **nav2_gradient_costmap_plugin : GradientLayer** doit être dynamiquement chargée grâce à la classe de base **nav2_costmap_2d : Layer**. Pour cela le plugin doit être enregistré de la façon suivante :

- La classe du plugin doit être enregistrée avec un type de base de la classe chargée. Pour cela, il y a une spéciale macro **PLUGINLIB_EXPORT_CLASS**. qui doit être ajoutée à chaque fichier source qui compose la librairie du plugin :

```
1 #include "pluginlib/class_list_macros.hpp"
2 PLUGINLIB_EXPORT_CLASS(nav2_gradient_costmap_plugin::GradientLayer, nav2_costmap_2d::Layer)
```

Cette partie est généralement placée à la fin du fichier source où la classe du plugin est écrite (dans notre exemple **gradient_layer.cpp**). C'est une bonne pratique de placer ces lignes à la fin du fichier mais techniquement, on peut le placer également au démarrage.

- Les informations du plugin doivent être stockées dans un fichier décrivant le plugin. Pour cela on utilise un fichier XML séparé (dans notre exemple **gradient_plugins.xml** dans le paquet du plugin). Le fichier contient les informations de :

- **path** : Chemin et nom de la librairie où se trouve le plugin.
- **name** : Le type du plugin référencé dans le paramètre **plugin_types**.
- **type** : La classe du plugin avec le namespace tel que donné dans le code source.
- **basic_class_type** : La classe de base à partir de laquelle la classe du plugin a été dérivé.
- **description** : La description du plugin sous forme textuelle.

```
1 <library path="nav2_gradient_costmap_plugin_core">
2   <class name="nav2_gradient_costmap_plugin::GradientLayer" type="nav2_gradient_costmap_plugin::GradientLayer"
3     base_class_type="nav2_costmap_2d::Layer">
4     <description>This is an example plugin which puts repeating costs gradients to costmap</description>
5   </class>
6 </library>
```

L'export du plugin s'effectue en incluant la macro **pluginlib_export_plugin_description_file()** dans le fichier **CMakeLists.txt**. Cette fonction installe le fichier de description du plugin dans le répertoire **share** et initialise les index d'ament pour que le fichier XML de description du plugin soit découvrable comme un plugin du type sélectionné :

```
1 pluginlib_export_plugin_description_file(nav2_costmap_2d gradient_layer.xml)
```

Le fichier de description du plugin doit être également ajouté au fichier **package.xml**. **costmap_2d** est le paquet de définition de l'interface, pour notre cas **Layer**, et requiert un chemin vers le fichier XML :

```
1 <export>
2   <costmap_2d plugin="${prefix}/gradient_layer.xml" />
3   ...
4 </export>
```

Une fois que tout est prêt, il faut mettre le package du plugin dans le répertoire **src** d'un espace de travail ROS-2, construire le package (**colcon build --packages-select nav2_gradient_costmap_plugin --symlink-install**) et sourcer le fichier **setup.bash** quand cela est nécessaire.

Maintenant le plugin est prêt à être utilisé.

17.4 Utiliser le plugin dans Costmap2D

La prochaine étape est de d'indiquer à Costmap2D qu'il y a un nouveau plugin. Pour cela il faut qu'il soit ajouté aux listes **plugin_names** et **plugin_types** du fichier **nav2_params.yaml**, optionnellement pour **local_costmap/global_costmap** de façon à être utilisable à l'exécution des serveurs Controller et Planner. La liste **plugin_names** contient les noms des plugins. Ces noms peuvent être ce que l'on souhaite. **plugin_types** contient les types des plugins listés dans **plugin_names**. Ces types doivent correspondre au champ **name** des classes de plugin spécifiée dans le fichier de description XML.

Par exemple :

```
1 --- a/nav2 Bringup/bringup/params/nav2_params.yaml
2 +++ b/nav2 Bringup/bringup/params/nav2_params.yaml
3 @@ -124,8 +124,8 @@
4   local_costmap:
5     width: 3
6     height: 3
7     resolution: 0.05
8   - plugin_names: ["obstacle_layer", "voxel_layer", "inflation_layer"] # For Eloquent and earlier
9   - plugin_types: ["nav2_costmap_2d::ObstacleLayer", "nav2_costmap_2d::VoxelLayer", "nav2_costmap_2d::InflationLayer"] # For Eloquent and earlier
10  + plugin_names: ["obstacle_layer", "voxel_layer", "gradient_layer"] # For Eloquent and earlier
11  + plugin_types: ["nav2_costmap_2d::ObstacleLayer", "nav2_costmap_2d::VoxelLayer", "nav2_gradient_costmap_plugin::GradientLayer"] # For Eloquent and earlier
```

```

11 -     plugins: ["obstacle_layer", "voxel_layer", "inflation_layer"] # For Foxy and later
12 +     plugins: ["obstacle_layer", "voxel_layer", "gradient_layer"] # For Foxy and later
13     robot_radius: 0.22
14     inflation_layer:
15         cost_scaling_factor: 3.0
16 @@ -171,8 +171,8 @@
17     robot_base_frame: base_link
18     global_frame: map
19     use_sim_time: True
20 -     plugin_names: ["static_layer", "obstacle_layer", "voxel_layer", "inflation_layer"] # For Eloquent and
earlier
21 -     plugin_types: ["nav2_costmap_2d::StaticLayer", "nav2_costmap_2d::ObstacleLayer", "nav2_costmap_2d::
Voxellayer", "nav2_costmap_2d::InflationLayer"] # For Eloquent and earlier
22 +     plugin_names: ["static_layer", "obstacle_layer", "voxel_layer", "gradient_layer"] # For Eloquent and
earlier
23 +     plugin_types: ["nav2_costmap_2d::StaticLayer", "nav2_costmap_2d::ObstacleLayer", "nav2_costmap_2d::
Voxellayer", "nav2_gradient_costmap_plugin/GradientLayer"] # For Eloquent and earlier
24 -     plugins: ["static_layer", "obstacle_layer", "voxel_layer", "inflation_layer"] # For Foxy and later
25 +     plugins: ["static_layer", "obstacle_layer", "voxel_layer", "gradient_layer"] # For Foxy and later
26     robot_radius: 0.22
27     resolution: 0.05
28     obstacle_layer:

```

Le fichier YAML peut aussi contenir la liste des paramètres (s'il y en a) pour chaque plugin, identifié leur noms.

NOTE : Il peut y avoir plusieurs plugins chargés à partir du même type. Pour cela la liste **plugin_names** doit contenir plusieurs noms de plugins différents tandis que **plugin_types** conserver la même valeur.

```

1 plugin_names: ["obstacle_layer", "gradient_layer_1", "gradient_layer_2"] # For Eloquent and earlier
2 plugin_types: ["nav2_costmap_2d::ObstacleLayer", "nav2_gradient_costmap_plugin/GradientLayer", "
nav2_gradient_costmap_plugin/GradientLayer"] # For Eloquent and earlier
3 plugins: ["obstacle_layer", "gradient_layer_1", "gradient_layer_2"] # For Foxy and later

```

Dans ce cas chaque plugin sera géré par son propre arbre de paramètres dans un fichier YAML comme :

```

1 gradient_layer_1:
2     plugin: nav2_gradient_costmap_plugin/GradientLayer # For Foxy and later
3     enabled: True
4 ...
5 gradient_layer_2:
6     plugin: nav2_gradient_costmap_plugin/GradientLayer # For Foxy and later
7     enabled: False
8 ...

```

17.5 Lancer le plugin GradientLayer

Il faut lancer la simulation Turtlebot3 qui permet d'avoir Nav2. La commande est :

```

1 $ ros2 launch nav2_bringup tb3_simulation_launch.py

```

Il faut ensuite aller sur RViz et cliquer sur le bouton "2D Pose Estimate" tout en haut et pointer la position sur la carte. Le robot sera localisé sur une carte et le résultat est représenté dans la figure Fig. 17.2. On peut voir la carte de coût avec le gradient. On peut voir très clairement la mise à jour dynamique de la carte de coût effectué par la méthode **GradientLayer** : **:updateCosts()** avec ses limites et le chemin global courbé par le gradient.

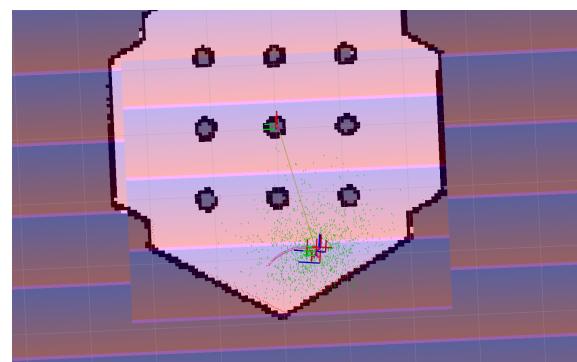


FIGURE 17.2 – Rendu du gradient pour la carte de coût

18. Plugin de planification

18.1 Introduction

Le but de ce didacticiel est de montrer comment il est possible d'écrire son propre planificateur. Il requiert d'avoir installer ROS-2, Nav2, Gazebo et turtlebot3.

18.2 Création d'un nouveau plugin de planification de mouvements

Nous allons créer un simple planificateur de ligne droite. Le code annoté est accessible sur la repository navigation2_tutorials dans le paquet **nav2_straightline_planner**. Ce paquet peut-être considéré comme la référence pour écrire un plugin de planification.

Le plugin exemple que nous considérons ici hérite de la classe de base **nav2_core : GlobalPlanner**. La classe de base fournit 5 méthodes pures virtuelles pour implémenter un plugin de planification. Le plugin sera utilisée par le serveur de planificateurs pour calculer des trajectoires. On peut voir dans la table 19.1 une description détaillée des différentes méthodes virtuelles.

Dans ce didacticiel, les méthodes **StraightLine : configure()** et **StraightLine : createPlan()** sont utilisées pour planifier des lignes droites.

Dans les planificateurs, la méthode **configure()** doit initialiser les variables membres à partir des paramètres ROS et effectuer toutes les initialisations requises.

```
1 node_ = parent;
2 tf_ = tf;
3 name_ = name;
4 costmap_ = costmap_ros->getCostmap();
5 global_frame_ = costmap_ros->getGlobalFrameID();
6
7 // Parameter initialization
8 nav2_util::declare_parameter_if_not_declared(node_, name_ + ".interpolation_resolution", rclcpp::ParameterValue
9 (0.1));
node_->get_parameter(name_ + ".interpolation_resolution", interpolation_resolution_);
```

Ici, la chaîne de caractères construite avec `name_ + ".interpolation_resolution"` permet d'extraire le paramètre **interpolation_resolution** qui est spécifié pour le planificateur. Navigation2 permet de charger plusieurs plugins et de garder une organisation où chaque plugin est associé à un identifiant ou nom. Maintenant si nous souhaitons retrouver les paramètres pour chaque spécifique plugin, nous utilisons `<map-ped_name_of_plugin>.name_of_parameter` comme cela est fait dans l'extrait de code. Par exemple, notre exemple de planificateur est identifié par le nom "GridBased" et pour retrouver le paramètre **interpolation_resolution** qui est spécifique à "GridBased", la chaîne `GridBased.interpolation_resolution` est

Méthodes virtuelles	Description de la méthode	Ecriture par dessus requise
configure()	La méthode est appelée quand le serveur de planification entre dans l'état <i>on_configure</i> . Idéalement cette méthode doit effectuer la déclaration des paramètres ROS et l'initialisation des variables membres de la classe. Cette méthode prend 4 paramètres en entrée : un pointeur partagé du node parent, le nom du planificateur, le pointeur vers le buffer tf et une pointeur partagé vers la carte de coût.	Oui
activate()	La méthode est appelée quand le serveur de planification entre dans l'état <i>on_activate</i> . Idéalement cette méthode doit implémenter les opérations qui sont nécessaires à effectuer avant que le planificateur n'aille dans l'état <i>activate</i> .	Oui
deactivate()	La méthode est appelée quand le serveur de contrôleurs entre dans l'état <i>on_deactivate</i> . Idéalement cette méthode doit implémenter les opérations qui sont nécessaires avant que le planificateur n'aille dans un état inactif.	Yes
cleanup()	La méthode est appelée quand le serveur de planification va dans l'état <i>on_cleanup</i> . Idéalement cette méthode doit nettoyer les ressources qui sont créées pour le contrôleur.	Yes
cretePlan()	Cette méthode est appelée quand le serveur de planification demande un plan global pour spécifier la pose de départ et de fin. Cette méthode retourne un message de type <i>nav_msgs</i> : <i>:msg</i> : <i>:Path</i> qui contient tout le plan. Cette méthode prend 2 variables d'entrées : la pose de départ et la pose d'arrivée.	Yes

Tableau 18.1 – Liste des méthodes utilisée dans la classe des plugins de planification

utilisée. Il est possible de l'interpréter de la façon suivante : **GridBased** est utilisé comme un namespace pour les paramètres spécifiques au plugin. Ceci est décrit plus en détails dans la suite concernant le fichier de paramètres.

Dans la méthode **createPlan()**, nous devons créer un chemin de la pose de départ à la pose d'arrivée. La méthode **StraightLine** : **:createPlan()** est appellé en utilisant la pose de départ et la pose d'arrivée pour résoudre le problème général de planification. Une fois la planification réussie le chemin est converti en un message **nav_msgs** : **:msg** : **:Path** et retourne au serveur de planification. Ci-dessous suit l'implémentation de la méthode annotée :

```

1 nav_msgs::msg::Path global_path;
2
3 // Checking if the goal and start state is in the global frame
4 if (start.header.frame_id != global_frame_) {
5     RCLCPP_ERROR(
6         node_>get_logger(), "Planner will only except start position from %s frame",
7         global_frame_.c_str());
8     return global_path;
9 }
10
11 if (goal.header.frame_id != global_frame_) {
12     RCLCPP_INFO(
13         node_>get_logger(), "Planner will only except goal position from %s frame",
14         global_frame_.c_str());
15     return global_path;
16 }
17
18 global_path.poses.clear();
19 global_path.header.stamp = node_->now();
20 global_path.header.frame_id = global_frame_;
21 // calculating the number of loops for current value of interpolation_resolution_
22 int total_number_of_loop = std::hypot(
23     goal.pose.position.x - start.pose.position.x,
24     goal.pose.position.y - start.pose.position.y) /
25     interpolation_resolution_;
26 double x_increment = (goal.pose.position.x - start.pose.position.x) / total_number_of_loop;
27 double y_increment = (goal.pose.position.y - start.pose.position.y) / total_number_of_loop;

```

```

28 for (int i = 0; i < total_number_of_loop; ++i) {
29     geometry_msgs::msg::PoseStamped pose;
30     pose.pose.position.x = start.pose.position.x + x_increment * i;
31     pose.pose.position.y = start.pose.position.y + y_increment * i;
32     pose.pose.position.z = 0.0;
33     pose.pose.orientation.x = 0.0;
34     pose.pose.orientation.y = 0.0;
35     pose.pose.orientation.z = 0.0;
36     pose.pose.orientation.w = 1.0;
37     pose.header.stamp = node_->now();
38     pose.header.frame_id = global_frame_;
39     global_path.poses.push_back(pose);
40 }
41
42 global_path.poses.push_back(goal);
43
44 return global_path;
45

```

Le reste des méthodes ne sont pas utilisées mais il est nécessaire de les réécrire. Elles sont donc vides.

18.3 Exporter le plugin de planification

Maintenant que nous avons notre planificateur personnalisé, il faut exporter le plugin de planification pour qu'il soit visible par le serveur de planification. Les plugins sont chargés à l'exécution et s'ils ne sont pas visibles le serveur ne pourra le charger. Dans ROS-2, exporter et charger les plugins sont gérés par **pluginlib**.

Dans ce didacticiel **nav2_straightline_planner** : **:StraightLine** est chargé dynamiquement avec la class de base **nav2_core** : **:GlobalPlanner**.

- Pour exporter le planificateur, nous devons fournir les deux lignes :

```

1 #include "pluginlib/class_list_macros.hpp"
2 PLUGINLIB_EXPORT_CLASS(nav2_straightline_planner::StraightLine, nav2_core::GlobalPlanner)
3

```

C'est la macro nommée **PLUGINLIB_EXPORT_CLASS** qui fait tout le travail d'export.

Par convention, les lignes sont placées à la fin du fichier mais techniquement on peut placer ces lignes également au début du fichier.

- La prochaine étape est de créer le fichier de description à la racine du répertoire du paquet. Par exemple, on trouve le fichier **global_planner_plugin.xml** dans le paquet du didacticiel. Le fichier contient les informations suivantes :

- **library path** : Le nom de la librairie et son emplacement.
- **class name** : Le nom de la classe.
- **class type** : Type de la classe.
- **base classe** : Nom de la classe de base.
- **description** : Description du plugin.

```

1 <library path="nav2_straightline_planner_plugin">
2   <class name="nav2_straightline_planner/StraightLine" type="nav2_straightline_planner::StraightLine">
3     <base_class_type="nav2_core::GlobalPlanner">
4       <description>This is an example plugin which produces straight path.</description>
5     </base_class_type>
6   </class>
7 </library>

```

- La prochaine étape est d'exporter le plugin en utilisant **CMakeLists.txt** en utilisant la fonction `cmake pluginlib_export_plugin_description_file()`. Cette fonction installe un fichier de description du plugin dans le répertoire **share** et rend le plugin accessible au système d'ament.

```

1 pluginlib_export_plugin_description_file(nav2_core_global_planner_plugin.xml)
2

```

- Le fichier de description doit être aussi ajouter au fichier **package.xml**

```

1 <export>
2   <build_type>ament_cmake</build_type>
3   <nav2_core plugin="${prefix}/global_planner_plugin.xml" />
4 </export>

```

- Il faut ensuite compiler pour que le fichier soit enregistré. Il est ensuite possible d'utiliser le plugin.

18.4 Passer le nom du plugin par le fichier de paramètres

Pour permettre l'utilisation du plugin, il faut modifier le fichier **nav2_params.yaml**. La suite décrit les paragraphes à remplacer.

Note 18.1 Depuis Galactic, les champs **plugin_names** et **plugin_types** ont été remplacé par un vecteur de chaînes de caractères nommé **plugins**. Les types sont maintenant définis dans l'espace de nom **plugin_name** dans le champ **filed** (e.g. **plugin : MyPlugin : :Plugin**). Les commentaires dans les blocs du code servent de guides. ■

```

1 planner_server:
2   ros_parameters:
3     planner_plugin_types: ["nav2_navfn_planner/NavfnPlanner"] # For Eloquent and earlier
4     planner_plugin_ids: ["GridBased"] # For Eloquent and earlier
5     plugins: ["GridBased"] # For Foxy and later
6     use_sim_time: True
7     GridBased:
8       plugin: nav2_navfn_planner/NavfnPlanner # For Foxy and later
9       tolerance: 2.0
10      use_astar: false
11      allow_unknown: true

```

avec

```

1 planner_server:
2   ros_parameters:
3     planner_plugin_types: ["nav2_straightline_planner/StraightLine"] # For Eloquent and
4       earlier
5     planner_plugin_ids: ["GridBased"] # For Eloquent and earlier
6     plugins: ["GridBased"] # For Foxy and later
7     use_sim_time: True
8     GridBased:
9       plugin: nav2_straightline_planner/StraightLine # For Foxy and later
10      interpolation_resolution: 0.1

```

Dans le fragment de code précédent, on peut observer que le planificateur **nav2_straightline_planner/StraightLine** a comme identifiant **GridBased**. On peut constater que pour passer des paramètres spécifiques au plugin on utilise **<plugin_id>. <plugin_specific_parameter>**.

18.5 Lancer le plugin StraightLine

Il est possible de lancer la simulation de Turtlebot3 avec un fichier de paramètres spécifiques :

```
1 ros2 launch nav2_bringup tb3_simulation_launch.py params_file:=/chemin/vers/votre_fichier_params.yaml
```

On peut ensuite utiliser rviz et cliquer sur le bouton "2D Pose Estimate" en haut de l'interface. Il faut indiquer une position sur la carte. Le robot va se localiser sur la carte et ensuite il faut cliquer sur "Navigation2 goal" puis indiquer un but sur la carte. Une fois que le planificateur fournit le plan, le robot démarre son mouvement vers le but.

19. Ecrire un nouveau plugin de contrôleur

19.1 Introduction

Ce didacticiel montre comment créer son propre contrôleur.

Dans ce didacticiel nous allons implémenter un algorithme de suivi de chemin basé sur le papier [4]. Il est recommandé de le parcourir.

Note 19.1 Ce didacticiel est basé sur une version simplifiée précédente du contrôleur de poursuite pure régulée maintenant dans la pile Nav2. Il est possible de trouver les codes sources correspondant au didacticiel dont le serveur github est ici. ■

19.2 Créer un nouveau plugin contrôleur

Ce didacticiel implémente un contrôleur de poursuite. Le code annoté de ce didacticiel peut être à nouveau trouvé dans le dépôt navigation_tutorials et plus spécifiquement dans le répertoire **nav2_pure_pursuit_controller**. Ce paquet peut être considéré comme une référence pour écrire son propre contrôleur.

La classe **nav2_pure_pursuit_controller** : **PurePursuitController** de l'exemple hérite de la classe de base **nav2_core** : **Controller**. La classe de base fournit un ensemble de méthodes virtuelles pour implémenter un plugin de contrôleur. Ces méthodes sont appellées à l'exécution par le serveur de contrôleur pour calculer des commandes en vitesse. La liste des méthodes, et leurs descriptions, et leur nécessité sont présentées dans la table 19.1

Dans ce didacticiel, les méthodes suivantes vont être utilisées : **PurePursuitController** : **:configure**, **PurePursuitController** : **:setPlan** et **PurePursuitController** : **:computeVelocityCommands**.

Dans les contrôleurs, la méthode **configure()** doit initialiser les membres de la classe à partir des paramètres ROS et effectuer les initialisations requises.

```
1 void PurePursuitController::configure(
2     const rclcpp_lifecycle::LifecycleNode::WeakPtr & parent,
3     std::string name, const std::shared_ptr<tf2_ros::Buffer> & tf,
4     const std::shared_ptr<nav2_costmap_2d::Costmap2DROS> & costmap_ros)
5 {
6     node_ = parent;
7     auto node = node_.lock();
8
9     costmap_ros_ = costmap_ros;
10    tf_ = tf;
11    plugin_name_ = name;
12    logger_ = node->get_logger();
```

Méthodes virtuelles	Description de la méthode	Ecriture par dessus requise
configure()	La méthode est appelée quand le serveur de contrôleurs entre dans l'état <i>on_configure</i> . Idéalement cette méthode doit effectuer la déclaration des paramètres ROS et l'initialisation des variables membres de la classe. Cette méthode prends 4 paramètres en entrée : un pointeur partagé du node parent, le nom du contrôleur, le pointeur vers le buffer tf et une pointeur partagé vers la carte de coût.	Oui
activate()	La méthode est appelée quand le serveur de contrôleurs entre dans l'état <i>on_activate</i> . Idéalement cette méthode doit implémenter les opérations qui sont nécessaires à effectuer avant que le contrôleur n'aille dans l'état <i>activate</i> .	Oui
deactivate()	La méthode est appelée quand le serveur de planification entre dans l'état <i>on_deactivate</i> . Idéalement cette méthode doit implémenter le opérations qui sont nécessaires avant que le planificateur n'aille dans un état inactif.	Yes
cleanup()	La méthode est appelée quand le serveur de planification va dans l'état <i>on_cleanup</i> . Idéalement cette méthode doit nettoyer les ressources qui sont créées pour le planificateur.	Yes
setPlan()	Cette méthode est appelée quand le plan global est mis à jour. Idéalement cette méthode effectue des opérations qui transforme le plan et le stocke.	Yes
computeVelocityCommands()	Cette méthode est appelée quand une nouvelle commande en vitesse est demandée par le serveur du contrôleur pour que le robot suive le plan global. Cette méthode retourne un message geometry_msgs : :msg : :TwistStamped qui représente la commande en vitesse pour que le robot conduise. Cette méthode passe 2 références de paramètres qui correspondent à la pose courante du robot et sa vitesse courante	Yes

Tableau 19.1 – Liste des méthodes utilisée dans la classe des plugins de planification

```

13     clock_ = node->get_clock();
14
15     declare_parameter_if_not_declared(
16         node, plugin_name_ + ".desired_linear_vel", rclcpp::ParameterValue(
17             0.2));
18     declare_parameter_if_not_declared(
19         node, plugin_name_ + ".lookahead_dist",
20         rclcpp::ParameterValue(0.4));
21     declare_parameter_if_not_declared(
22         node, plugin_name_ + ".max_angular_vel", rclcpp::ParameterValue(
23             1.0));
24     declare_parameter_if_not_declared(
25         node, plugin_name_ + ".transform_tolerance", rclcpp::ParameterValue(
26             0.1));
27
28     node->get_parameter(plugin_name_ + ".desired_linear_vel", desired_linear_vel_);
29     node->get_parameter(plugin_name_ + ".lookahead_dist", lookahead_dist_);
30     node->get_parameter(plugin_name_ + ".max_angular_vel", max_angular_vel_);
31     double transform_tolerance;
32     node->get_parameter(plugin_name_ + ".transform_tolerance", transform_tolerance);
33     transform_tolerance_ = rclcpp::Duration::from_seconds(transform_tolerance);
34 }
```

Ici, la chaîne de caractères `plugin_name_ + ".desired_linear_vel"` permet de lire le paramètre ROS `desired_linear_vel` qui est spécifique à ce contrôleur. Nav2 permet de charger plusieurs plugins, afin de les identifier chacun d'eux à un identifiant ou un nom. Donc afin de retrouver les paramètres pour un plugin particulier il est possible d'utiliser la convention suivante `<nom_du_plugin>.<nom_du_paramètre>` comme cela est fait dans le fragment de code précédent. Par exemple, le contrôleur de ce didacticiel est nommé **FollowPath**. Afin de retrouver le paramètre `desired_linear_vel` spécifique à **FollowPath** on utilise **FollowPath.desired_linear_vel**. On peut également voir **FollowPath** comme un espace de noms pour des paramètres spécifiques à **FollowPath**. Certains de ces concepts sont discutés lorsque le fichier de paramètres est discuté.

Les arguments qui sont lus sont stockés dans les variables membres de la classe pour qu'ils soient utilisés plus tard si nécessaire.

La méthode `setPlan()` reçoit le chemin global mis à jour que le robot doit suivre. Dans cet exemple, le chemin global reçu est transformé dans le repère du robot et le stocke pour des usages ultérieurs.

```

1 void PurePursuitController::setPlan(const nav_msgs::msg::Path & path)
2 {
3     // Transform global path into the robot's frame
4     global_plan_ = transformGlobalPlan(path);
5 }
```

Le calcul des vitesses désirées s'effectue dans la méthode `computeVelocityCommands()`. Elle est utilisée pour calculer la commande en vitesse désirée en considérant la vitesse et la pose courantes. Dans le cas d'une poursuite pure, l'algorithme calcule la commande en vitesse pour que le robot essaie de suivre le chemin global aussi précisément que possible. Cet algorithme suppose une vitesse linéaire constante and calcule la vitesse angulaire basée sur la courbure du chemin global.

```

1 geometry_msgs::msg::TwistStamped PurePursuitController::computeVelocityCommands(
2     const geometry_msgs::msg::PoseStamped & pose,
3     const geometry_msgs::msg::Twist & velocity)
4 {
5     // Find the first pose which is at a distance greater than the specified lookahead distance
6     auto goal_pose = std::find_if(
7         global_plan_.poses.begin(), global_plan_.poses.end(),
8         [&](const auto & global_plan_pose) {
9             return hypot(
10                 global_plan_pose.pose.position.x,
11                 global_plan_pose.pose.position.y) >= lookahead_dist_;
12         })->pose;
13
14     double linear_vel, angular_vel;
15
16     // If the goal pose is in front of the robot then compute the velocity using the pure pursuit algorithm
17     // else rotate with the max angular velocity until the goal pose is in front of the robot
18     if (goal_pose.position.x > 0) {
19
20         auto curvature = 2.0 * goal_pose.position.y /
21             (goal_pose.position.x * goal_pose.position.x + goal_pose.position.y * goal_pose.position.y);
22         linear_vel = desired_linear_vel_;
23         angular_vel = desired_linear_vel_ * curvature;
24     } else {
25         linear_vel = 0.0;
26         angular_vel = max_angular_vel_;
```

```

27 }
28
29 // Create and publish a TwistStamped message with the desired velocity
30 geometry_msgs::msg::TwistStamped cmd_vel;
31 cmd_vel.header.frame_id = pose.header.frame_id;
32 cmd_vel.header.stamp = clock_.now();
33 cmd_vel.twist.linear.x = linear_vel;
34 cmd_vel.twist.angular.z = max(
35     -1.0 * abs(max_angular_vel_), min(
36         angular_vel, abs(
37             max_angular_vel_)));
38
39 return cmd_vel;
40 }
```

Les autres méthodes ne sont pas utilisées, mais il est nécessaire de les surcharger. Elles sont donc vides.

19.3 Exporter le plugin du contrôleur

Une fois le contrôleur prêt, il faut que celui-ci soit exporté pour qu'il soit visible par le serveur de contrôleur. Les plugins sont chargés à l'exécution, et s'ils ne sont pas visibles, le serveur de contrôleur ne sera pas capable de les charger. Avec ROS-2, les plugins exportés et chargés sont gérés par **pluginlib**.

Dans ce didacticiel, la classe **nav2_pure_pursuit_controller** : **PurePursuitController** est chargée dynamiquement comme étant de la classe de base **nav2_core** : **Controller**.

1. Pour export le contrôleur, nous avons besoin de fournir les deux lignes suivantes :

```

1 #include "pluginlib/class_list_macros.hpp"
2 PLUGINLIB_EXPORT_CLASS(nav2_pure_pursuit_controller::PurePursuitController, nav2_core::Controller)
3 }
```

pluginlib est nécessaire pour exporter la classe de plugin. pluginlib fournit la macro **PLUGIN-LIB_EXPORT_CLASS** qui effectue toutes les tâches d'export.

Il est conseillé de mettre ces lignes à la fin du fichier source, mais techniquement il est également possible de les mettre au début.

2. La prochaine étape est de créer le fichier de description du plugin dans la racine du répertoire du paquet. Par exemple dans ce didacticiel il s'agit du fichier **pure_pursuit_controller_plugin.xml**. Ce fichier contient les informations suivantes :

- **library path** : Le nom de la librairie du plugin et sa place sur le disque.
- **class name** : Nom de la classe.
- **class type** : Type de la classe.
- **base class** : Nom de la classe de base.
- **description** : Description du plugin.

```

1 <library path="nav2_pure_pursuit_controller">
2   <class type="nav2_pure_pursuit_controller::PurePursuitController" base_class_type="nav2_core::Controller">
3     <description>
4       This is pure pursuit controller
5     </description>
6   </class>
7 </library>
8 
```

3. L'étape suivante est d'exporter le plugin en utilisant le fichier **CMakeLists.txt** en utilisant la fonction CMake en utilisant la fonction **pluginlib_export_plugin_description_file()**. Cette fonction installe le fichier de description du plugin dans le répertoire **share** et positionne les index d'ameute pour le rendre découvrable.

```

1 pluginlib_export_plugin_description_file(nav2_core_pure_pursuit_controller_plugin.xml)
2 
```

4. Le fichier de description du plugin doit être ajouté à **package.xml**

```

1 <export>
2   <build_type>ament_cmake</build_type>
3   <nav2_core plugin="${prefix}/pure_pursuit_controller_plugin.xml" />
4 </export>
5 
```

5. Il faut ensuite compiler et le plugin va être enregistré.

19.3.1 Passer le nom du plugin à travers le fichier de paramètres

Pour activer le plugin, il faut modifier le fichier **nav2_params.yaml** de la façon suivante :

```
1 controller_server:  
2   ros__parameters:  
3     controller_plugins: ["FollowPath"]  
4  
5     FollowPath:  
6       plugin: "nav2_pure_pursuit_controller::PurePursuitController"  
7       debug_trajectory_details: True  
8       desired_linear_vel: 0.2  
9       lookahead_dist: 0.4  
10      max_angular_vel: 1.0  
11      transform_tolerance: 1.0
```

Dans l'exemple précédent on peut observer que le contrôleur **nav2_pure_pursuit_controller/PurePursuitController** a l'identifiant **FollowPath**. Pour passer des paramètres spécifiques au plugin on peut installer : **<plugin_id>. <plugin_specific_parameter>**.

19.3.2 Lancer le plugin de contrôleur en poursuite pure

Il faut lancer la simulation Turtlebot3 avec Nav2. La commande pour lancer cette partie est :

```
1 ros2 launch nav2_bringup tb3_simulation_launch.py params_file:=/path/to/your_params_file.yaml
```

Il faut ensuite aller dans rviz et cliquer sur le bouton "2D Pose Estimate" sur la barre du haut et pointer sur la position du robot dans la carte. Le robot va se localiser. Il faut ensuite cliquer le bouton "Nav2 goal" et cliquer sur l'endroit où l'on souhaite que le robot navigue. Ensuite le contrôleur va forcer le robot à suivre le chemin global.

Appendices

20	Mots clefs pour les fichiers launch	211
20.1	<launch>	
21	Rappels sur le bash	213
21.1	Lien symbolique	
21.2	Gestion des variables d'environnement	
21.3	Fichier .bashrc	
22	Utiliser docker pour ROS	215
22.1	Préparer votre ordinateur	
22.2	Utiliser l'image docker	
22.3	Installer les images docker de ROS	
22.4	Construire l'image docker	
22.5	Mac	
23	Mémo	219
	Bibliography	223
	Books	
	Articles	
	Chapitre de livres	
	Autres	

20. Mots clefs pour les fichiers launch

20.1 <launch>

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

20.1.1 Attributs

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

20.1.2 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 éléments partageant un espace de nom ou un renommage.

21. Rappels sur le bash

21.1 Lien symbolique

21.1.1 Voir les liens symboliques

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

```
| ls
```

La sortie est la suivante :

```
1 catkin_ws          Music
2 catkin_ws_prenom_nom Pictures
3 Desktop            Public
4 Documents           Templates
5 Downloads           Videos
6 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
```

21.1.2 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
```

21.1.3 Enlever un lien symbolique

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

```
| rm catkin_ws
```

21.2 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.

21.3 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
```

22. Utiliser docker pour ROS

22.1 Préparer votre ordinateur

Pour préparer votre ordinateur en distanciel, voici la liste des choses à faire :

- Installer docker pour votre machine
- Récupérer l'image docker pour les TPs :

```
| docker pull aipprimecaoccitanie/ros-introduction-robmob:latest
```

- Lancer l'image :

```
| docker run -p 6080:6080 --name ros-intro-vnc -it aipprimecaoccitanie/ros-introduction-robmob:latest
```

- Dans votre navigateur :

```
| http://localhost:6080/vnc.html?host=localhost&port=6080
```

Vous devriez voir apparaître l'image représentée dans la figure Fig.22.1.

Si le navigateur indique l'erreur `timeout`: il faut augmenter la valeur `timeout` en allant dans le menu de la roue crantée. Cliquer directement sur connect sans rentrer de mot de passe. Vous devriez voir ensuite l'image représentée dans la figure Fig.22.2

22.2 Utiliser l'image docker

Démarrer le container spécifique à l'AIP s'effectue avec la commande suivante :

```
| docker run -p 6080:6080 --name ros-intro-vnc -it aip-primeca-occitanie/ros-intro-robmob:latest
```

Le nom de l'image est spécifié en dernier et cette commande a créé le container **ros-intro-vnc**.

22.2.1 Lancer un bash

Il est ensuite possible de démarrer un executable bash dans une autre console en tapant :

```
| docker exec -it ros-intro-vnc bash
```

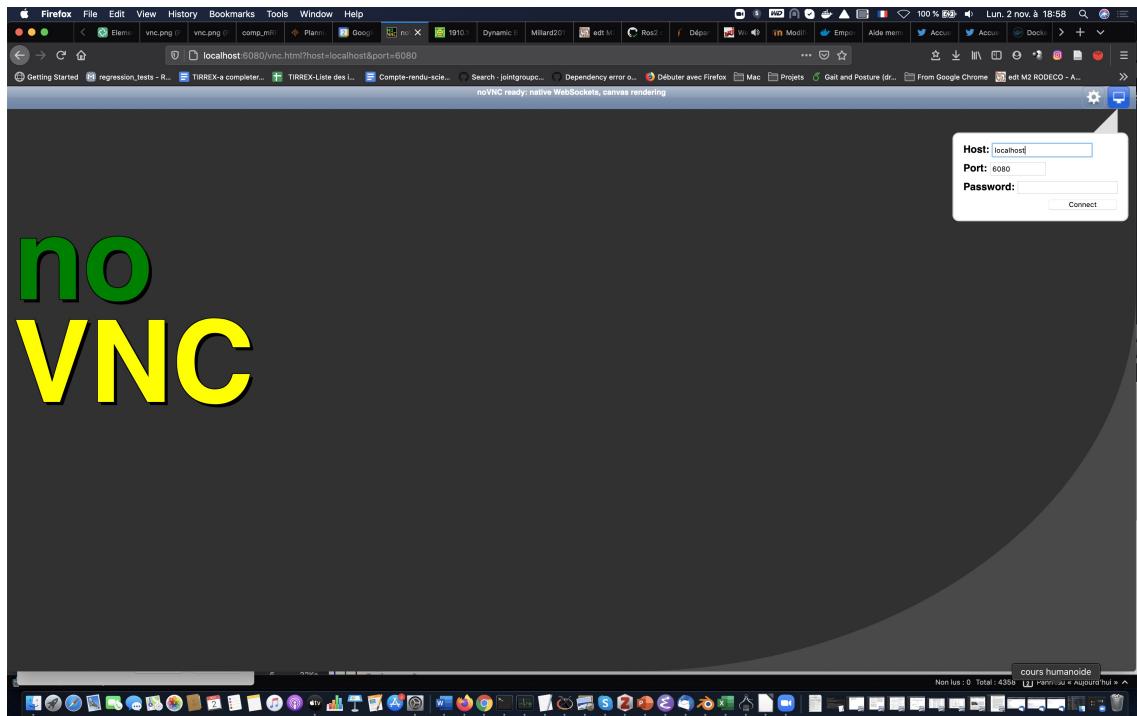


FIGURE 22.1 – Lancer le client VNC

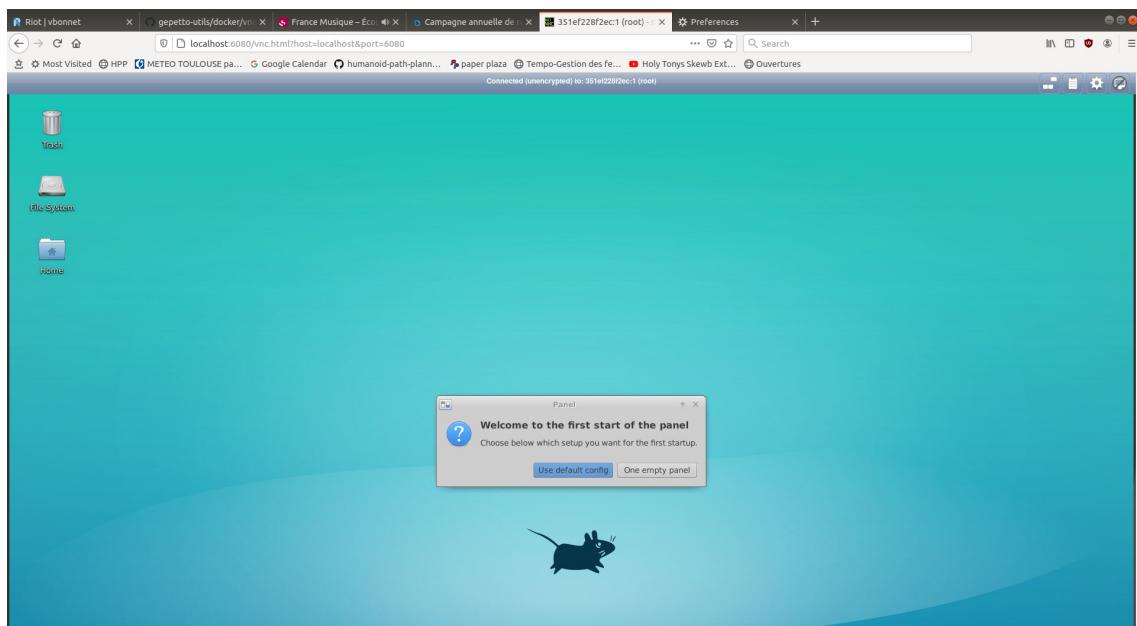


FIGURE 22.2 – Le gestionnaire de fenêtre XFCE

22.2.2 Stopper, redémarrer une image

Pour avoir la liste des containers :

```
1 docker container ps
```

La sortie typique est :

1 CONTAINER ID	IMAGE NAMES	COMMAND	CREATED	STATUS	PORTS
2 3d5bcf2a273c	cc925c2a2ebd	" <code>/ros_entrypoint.sh ...</code> "	10 hours ago	Up 9 hours	0.0.0.0:6080->6080/tcp ros-intro-vnc

L'identifiant du container est indiqué dans la première colonne. Ici il s'agit de 3d5bcf2a273c.

Pour stopper le container

```
1 docker container stop
```

Pour démarrer le container :

```
1 docker start -ai 3d5bcf2a273c
```

22.2.3 Copier un fichier de et vers un docker

Pour copier le fichier test.dat qui se trouve dans le répertoire /tmp vers le répertoire /root du container ros-intro-vnc

```
1 docker cp /tmp/test.dat ros-intro-vnc:/root/
```

Pour copier le fichier test.dat du répertoire root du container ros-intro-vnc vers le répertoire /tmp

```
1 docker cp ros-intro-vnc:/root/test.dat /tmp
```

22.3 Installer les images docker de ROS

Les images docker ROS sont disponibles ici à l'adresse suivante : https://hub.docker.com/_/ros

Par exemple pour installer l'image docker de melodic pour un robot il faut taper :

```
1 docker pull osrf/ros:melodic-desktop-full
```

En général le nom des images docker est nom_distribution_ros-type_install_ros-version_os avec :

- nom_distribution : Le nom de la distribution ROS
- type_install : le type d'installation ROS. Elle peuvent être soit de type destkop, desktop-full ou robot. Les deux premières correspondant à des installations sur des machines de développements, tandis que la dernière correspondant au robot.
- version_os : la version du système d'exploitation. Par exemple ici bionic correspondant au nom de la distribution Ubuntu 18

Pour démarrer une image docker de ROS classique :

```
1 docker run --name ros-melodic -it osrf/ros:melodic-desktop-full
```

22.3.1 Mise à jour de l'image

Une fois dans l'image vous pouvez faire

```
1 apt update
2 apt install
```

22.3.2 Erreurs

Il arrive parfois que vous ayez l'erreur suivante :

```
1 W: GPG error: http://security.ubuntu.com/ubuntu bionic-security InRelease: At least one invalid signature was
encountered.
2 E: The repository 'http://security.ubuntu.com/ubuntu bionic-security InRelease' is not signed.
```

Il se peut simplement que docker prenne trop de place sur le disque typiquement jusqu'à 50 Go. Pour le vérifier :

```
1 docker system df
```

Pour résoudre le problème :

```
1 docker system prune
```

Cette commande retire les containers, réseaux et images non utilisées.

22.4 Construire l'image docker

Vous pouvez utiliser la repository <https://github.com/aip-primeca-occitanie/ros-introduction-robmob>

```
1 cd ros-vnc  
2 docker build .
```

Pour créer un tag :

```
1 cd ros-vnc  
2 docker build . -t aip-primeca-occitanie/ros-introduction-robmob
```

22.5 Mac

22.5.1 Interface X11 sous Mac

La première étape est d'installer le serveur XQuartz sur votre ordinateur. Il faut ensuite autoriser les connections des clients réseaux dans la configuration XQuartz.

Dans le terminal du Mac vous devez autoriser le terminal du docker à accéder au terminal X :

```
1 xhost + 127.0.0.1
```

Il faut ensuite rediriger la sortie du bash dans le docker vers le host du docker :

```
1 export DISPLAY=host.docker.internal:0
```

23. Mémo

ROS Mémo

Outils en ligne de commande pour le système de fichiers

<code>rospack</code>	Un outil pour inspecter les packages/stacks
<code>roscd</code>	Change de répertoire vers le paquet ou la stack.
<code>rosls</code>	Liste les paquets et les stacks.
<code>catkin_create_pkg</code>	Créer un nouveau paquet ROS.
<code>rosdep</code>	Installe le système des dépendances des paquets ROS.
<code>catkin_make</code>	Construit les paquets ROS.
<code>rosws</code>	Commande pour manipuler les workspaces.

Usage :
\$ rospack find [package]
\$ roscl [package[/subdir]]
\$ rosld [package[/subdir]]
\$ catkin_create_pkg [package name]
\$ catkin_make [package]
\$ rosdep install [package]
\$ rosdtf or rosldt [file]

Outils en ligne de commande les plus utilisés

`roscore`

Une collection de **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

`rosmg/rossrv`

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

Usage :

\$ rosmg show

Affiche les champs dans le message

\$ rosmg md5

Affiche les valeurs md5 des messages

\$ rosmg package

Liste tous les messages d'un paquet

\$ rosmg packages

Liste tous les packages avec un message

Exemples :

Affiche le msg Pose

\$ rosmg show Pose

Liste les messages du paquet nav_msgs

\$ rosmg package nav_msgs

Liste les packages ayant des messages

\$ rosmg 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
```

roslauch

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

Exemples :

Lancer un fichier d'un package

```
roslauch package filename.launch
```

Lancer sur un port différent :

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

Lancer uniquement les nodes locaux :

```
roslauch -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'appeler 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'URL 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 désérialisation et la resérialisation des messages.

rosbag record génère 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>
```

Exemples :

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

- [Bro16] Bernard BROGLIATO. *Nonsmooth Mechanics*. Third edition. Springer International Publishing Switzerland, 2016 (cf. page 84).

Articles

- [Sta+09] O. STASSE et al. « Strategies for humanoid robots to dynamically walk over large obstacles ». In : *IEEE Transactions on Robotics* 4 (2009), pages 960-967 (cf. page 84).
- [Lee+18] J. LEE et al. « DART : Dynamic Animation and Robotics Toolkit ». In : *Journal of Open Source Software* 3.22 (2018) (cf. page 84).
- [Set+18] A. SETH et al. « OpenSim : Simulating musculoskeletal dynamics and neuromuscular control to study human and animal movement ». In : *PLOS Computational Biology* (2018) (cf. page 84).

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 63).

Autres

- [Goo16] GOOGLE. 2016. URL : <https://developers.google.com/project-tango/> (cf. page 12).

