Mise en œuvre de ROS2 pour le contrôle d'une voiture CoVAPSy simulée sous Webots et réelle

Culture Sciences de l'Ingénieur

La Revue 3E.I

Anthony JUTON[1] – Sergio RODRIGUEZ[2]
Jules FARNAULT[3] - Mathis GOUPILLON[3]

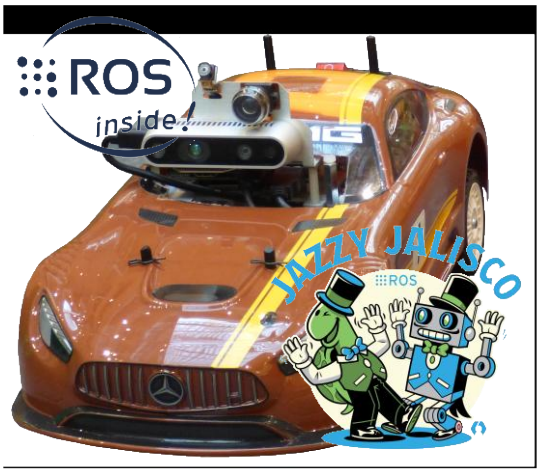
Édité le 03/02/2026

école normale supérieure paris—saclay

[1] *Professeur agrégé à l'ENS Paris Saclay, DER Sciences de l'Ingénierie Électrique et Numérique*
[2] *Maître de conférences au laboratoire SATIE, ENS Paris Saclay*
[3] *Elève normalien à l'ENS Paris Saclay, DER Sciences de l'Ingénierie Électrique et Numérique*

Cette ressource fait partie du N°118 de La Revue 3EI du premier trimestre 2026.

Cette ressource fait suite à la ressource « *ROS2 : bibliothèques et outils pour le développement logiciel en robotique* » [1]. Pour les étudiants participant à la course de voitures autonomes de Paris Saclay CoVAPSy [13], elle guide dans la mise en œuvre de ROS2 pour la conduite d'une voiture réelle ou simulée sous Webots. Pour les autres, c'est un exemple de mise en œuvre de ROS2 pour le contrôle d'un robot réel contrôlé par Raspberry Pi et simulé sous Webots. Webots fournit un autre exemple d'utilisation de ROS2 sur un robot, pour le contrôle d'un drone simulé [10].



Voiture autonome contrôlée par ROS2 ayant participé à la course de Paris-Saclay CoVAPSy

ROS2 est utilisé sur les voitures de la course CoVAPSy par plusieurs équipes pour plusieurs raisons :

- Les fournisseurs des capteurs fournissent les nœuds, écrits en C et optimisés, permettant l'acquisition des informations des capteurs (Slamtec fournit un nœud pour son LiDAR et Intel pour la caméra Realsense D435i notamment) ;
- Les fournisseurs de nano-ordinateurs embarqués (raspberry, nvidia, qualcomm) fournissent une implémentation de ROS2 fonctionnelle pour leurs cartes ;
- ROS2 est multiprocessing de par sa conception, ce qui permet d'utiliser au mieux les différents cœurs du microprocesseur du nano-ordinateur (RPI5 ou autre) ;
- Les messages ROS2 pouvant être transmis par IP, cela permet de superviser le fonctionnement de la voiture depuis un PC déporté, avec les outils de monitoring ROS2. Le nano-ordinateur n'a alors pas besoin d'une interface graphique, ce qui allège l'OS ;
- Les nœuds ROS2 peuvent être portés du simulateur vers la voiture simplement ;

- ROS2 étant très utilisé en robotique, on y trouve des nœuds permettant de mettre en œuvre des solutions avancées, comme le SLAM (Simultaneous Localization and Mapping) utilisé par l'équipe Sorbonne Université [12].

1 - Mise en œuvre de ROS2 pour le contrôle d'une voiture 1/10ème de type CoVAPSy

Cette partie présente la mise en place d'un contrôle simple de la voiture par ROS2, avec un nano-ordinateur Raspberry Pi et un LiDAR Slamtec S2. Un LiDAR (light detection and ranging) est un télémètre laser tournant, permettant d'obtenir en 2 ou 3D une cartographie des obstacles autour du véhicule.

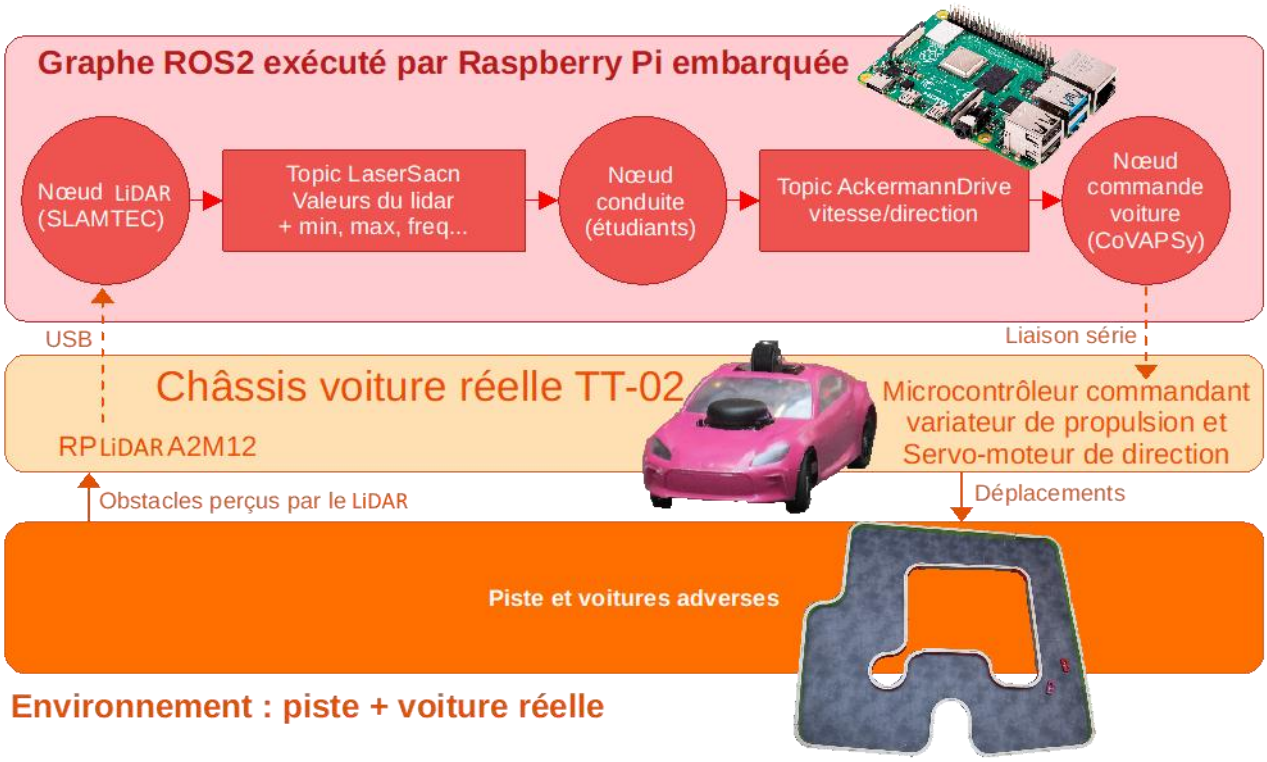


Figure 1 : Nœuds et topics ROS2 utilisés pour la conduite autonome de la voiture type CoVAPSy

1.1 - Installation ubuntu 24.04 server et ROS2 jazzy

La ressource « *ROS2 : bibliothèques et outils pour le développement logiciel en robotique* » [1] présente l'installation de ROS2 jazzy. Sur la raspberry Pi5, il est possible d'installer Ubuntu Desktop (avec la gourmande interface graphique gnome). Sinon, une version server, sans interface graphique est suffisante, ROS2 fournissant les outils pour le monitoring à distance.

Attention, un changement d'adresse de dépôt de ros2 a eu lieu (*repo.ros2.org/ubuntu* désormais), il faut peut-être modifier celle-ci dans */etc/apt/sources.list.d/ros2.list* :

```
voituremaxime@voituremaxime:~/ros2_ws$ sudo nano /etc/apt/sources.list.d/ros2.list
```

```
voituremaxime@voituremaxime: ~/ros2_ws
 GNU nano 7.2                    /etc/apt/sources.list.d/ros2.list
deb [arch=arm64 signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://repo.ros2.org/ubuntu noble main
```

La conduite de la voiture utilise des messages de type ackermannDrive, dont la définition est installable avec l'instruction :

```
sudo apt install ros-jazzy-ackermann-msgs
```

```
voituremaxime@voituremaxime:~/ros2_ws$ sudo apt install ros-jazzy-ackermann-msgs
```

1.2 - Nœud slamtec RpLidar

Pour utiliser le LiDAR RpLidar-S2 (ou le RpLidar-A2 très similaire) de Slamtec, on utilise le package sllidar_ros2 [6]. Ce package a été créé par le constructeur et est donc optimisé pour fonctionner avec tous les LiDARs slamtec RpLidar. Il permet de lire les données du capteur et de les publier dans un topic nommé */scan* sous le format *sensor_msgs/LaserScan* [7].

L'intérêt est que ce nœud a été écrit en C++ et compilé, il est plus rapide que les nœuds en python. Ainsi il permet de suivre la cadence de 1Mbits/s imposé par le RpLidar S2. Ces données peuvent ensuite être utilisées pour cartographier l'environnement et pour localiser la voiture.

Installation du package *rplidar_ros*

Pour installer le package *rplidar_ros* (via le paquet linux *sllidar_ros2*), suivre les instructions fournies par Slamtec [6].

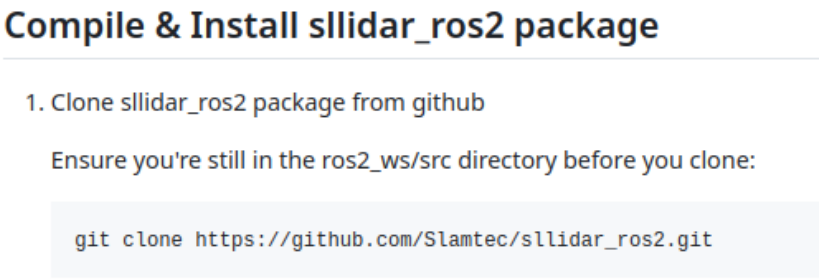


Figure 2 : extrait de la section installation du dépôt git du noeud ROS2 pour RpLidar

La section installation propose :

- De se placer dans le dossier *src* du dossier de travail : `cd ~/ros2_ws/src`
- D'y copier les fichiers source du nœud :
  `git clone -b ros2 https://github.com/Slamtec/rplidar_ros.git`
- Depuis le dossier de travail *ros2_ws*, compiler le nœud :
  `source ./install/setup.bash` puis `colcon build --symlink-install`

Quelques warnings apparaissent :

```
voituremaxime@voituremaxime:~/ros2_ws/src$ git clone -b ros2 https://github.com/Slamtec/rplidar_ros.git
Cloning into 'rplidar_ros'...
remote: Enumerating objects: 1240, done.
remote: Counting objects: 100% (627/627), done.
remote: Compressing objects: 100% (172/172), done.
remote: Total 1240 (delta 540), reused 455 (delta 455), pack-reused 613 (from 2)
Receiving objects: 100% (1240/1240), 647.66 KiB | 1.60 MiB/s, done.
Resolving deltas: 100% (823/823), done.
voituremaxime@voituremaxime:~/ros2_ws/src$ cd ..
voituremaxime@voituremaxime:~/ros2_ws$ source ./install/setup.bash
voituremaxime@voituremaxime:~/ros2_ws$ colcon build --symlink-install
Starting >>> rplidar_ros
--- stderr: rplidar_ros
/home/voituremaxime/ros2_ws/src/rplidar_ros/sdk/src/arch/linux/net_serial.cpp: In member function 'bool rp::arch::net::raw
/home/voituremaxime/ros2_ws/src/rplidar_ros/sdk/src/arch/linux/net_serial.cpp:97:74: warning: unused parameter 'flags' [-W
   97 | bool raw_serial::open(const char * portname, uint32_t baudrate, uint32_t flags)
      |                                                                  ~~~~~~~~~^~~~~
Finished <<< rplidar_ros [23.9s]

Summary: 1 package finished [24.1s]
  1 package had stderr output: rplidar_ros
```

Le package rplidar_ros2 nécessite des permissions en lecture et en écriture pour le port série. Pour lui ajouter ses permissions, on utilise la commande suivante :

```
sudo chmod 777 /dev/ttyUSB0
```

On peut également éviter ce changement de permissions nécessaire à chaque connexion sur le port série en ajoutant l'utilisateur dans le groupe DIALOUT et en redémarrant la session (ou en redémarrant le nano-ordinateur).

```
sudo usermod -aG dialout $USER
```

```
voituremaxime@voituremaxime:~/ros2_ws$ sudo usermod -aG dialout $USER
voituremaxime@voituremaxime:~/ros2_ws$ sudo reboot
```

Utilisation du package rplidar_ros2

Pour utiliser le package rplidar_ros2, on utilise, comme indiqué dans les instructions du dépôt, les commandes suivantes (en remplaçant nom du LiDAR par a2, a3 ou s2:

- `ros2 launch rplidar_ros view_rplidar_<nom du LiDAR>_launch.py`
- `ros2 launch rplidar_ros rplidar_<nom du LiDAR>_launch.py`

Les deux commandes permettent de d'exécuter le nœud du capteur et publier les données dans le topic /scan. La première ajoute l'ouverture de Rviz2 pour avoir un affichage graphique des données du capteur, ce qui fonctionne uniquement si le nano-ordinateur dispose d'un environnement graphique.

Il est possible d'avoir certains problèmes lors de l'utilisation du package. La principale erreur est un arrêt du LiDAR au bout d'une dizaine de secondes, dû au mode de scan utilisé. Il est possible de revenir à un fonctionnement plus stable en modifiant le fichier launch correspondant au LiDAR, en remplaçant la ligne suivante :

```
scan_mode = LaunchConfiguration('scan_mode', default='DenseBoost')
```

par :

```
scan_mode = LaunchConfiguration('scan_mode', default='Standard')
```

Le LiDAR utilise alors, avec robustesse, le mode 'Standard' à son prochain lancement.

On peut observer les messages publiés par le LiDAR dans une 2nde console, en utilisant les commandes suivantes :

```
ros2 topic list
ros2 topic echo /scan
```

```
voituremaxime@voituremaxime:~/ros2_ws$ ros2 launch rplidar_ros rplidar_s2_launch.py
[INFO] [launch]: All log files can be found below /home/voituremaxime/.ros/log/2026-01-22-00-15-50-856487-voituremaxime-3032
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [rplidar_node-1]: process started with pid [3036]
[rplidar_node-1] [INFO] [1769040950.994800993] [rplidar_node]: RPLidar running on ROS2 package rplidar_ros. RPLIDAR SDK Version:2.1.0
[rplidar_node-1] [INFO] [1769040951.011953632] [rplidar_node]: RPLidar S/N: AADDECF8C4E699D7B8EB99F926024717
[rplidar_node-1] [INFO] [1769040951.012007076] [rplidar_node]: Firmware Ver: 1.01
[rplidar_node-1] [INFO] [1769040951.012027836] [rplidar_node]: Hardware Rev: 18
[rplidar_node-1] [INFO] [1769040951.013766612] [rplidar_node]: RPLidar health status : 0
[rplidar_node-1] [INFO] [1769040951.013801631] [rplidar_node]: RPLidar health status : OK.
[rplidar_node-1] [INFO] [1769040951.015755463] [rplidar_node]: Start
[rplidar_node-1] [INFO] [1769040951.180077574] [rplidar_node]: current scan mode: DenseBoost, sample rate: 32 Khz, max_distance: 30.0
[rplidar_node-1] [INFO] [1769040953.301341506] [rplidar_node]: set lidar scan frequency to 10.0 Hz(600.0 Rpm)
```

```
^Cvoituremaxime@voituremaxime:~/ros2_ws$ ros2 topic list
/clicked_point
/goal_pose
/initialpose
/parameter_events
/rosout
/scan
/tf
/tf_static
voituremaxime@voituremaxime:~/ros2_ws$ ros2 topic echo /scan
header:
  stamp:
    sec: 1769041086
    nanosec: 346143364
  frame_id: laser
angle_min: -3.1241390705108643
angle_max: 3.1415927410125732
angle_increment: 0.0019344649044796824
time_increment: 3.049539191124495e-05
scan_time: 0.09877457469701767
range_min: 0.15000000596046448
range_max: 30.0
ranges:
- 0.5989999771118164
- 0.6050000190734863
- 0.6060000061988831
```

1.3 - Création du nœud de commande de la voiture

Le nœud de commande de la voiture, comme indiqué sur la Figure 1, reçoit un topic de type AckermannDrive et envoie ensuite les consignes de vitesse et direction au microcontrôleur via la liaison USB-série.

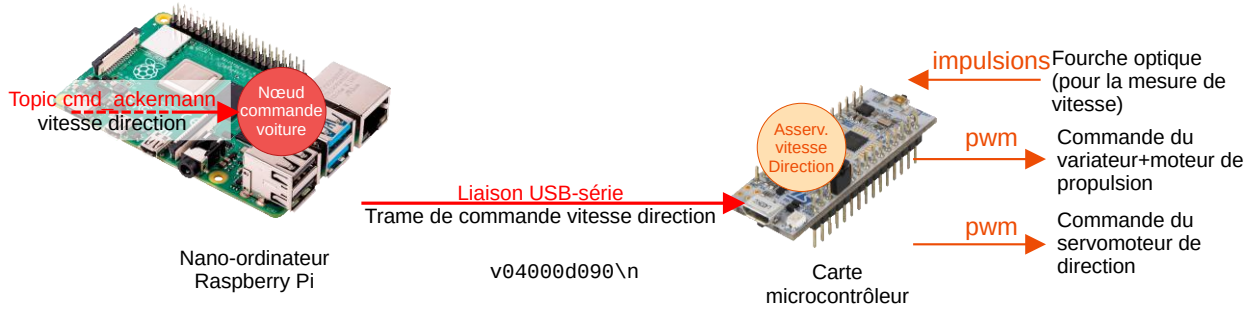


Figure 3 : Messages impliqués dans la transmission des consignes de vitesse et direction du topic ROS jusqu'aux moteurs

Ackermann fait référence à une modélisation des véhicules automobiles classiques. Le topic cmd_ackermann, de type AckermannDrive, contient 5 informations, dont seulement 2 (steering angle et speed) seront utilisées dans cette ressource :

- float32 steering_angle # consigne d'angle de direction (radians)
- float32 steering_angle_velocity # consigne de vitesse de direction (radians/s)
- float32 speed # consigne de vitesse (m/s)
- float32 acceleration # consigne d'acceleration (m/s^2)
- float32 jerk # consigne de jerk (m/s^3)

La trame envoyée au microcontrôleur dépend du code de réception implantée dans le microcontrôleur. Ici, a été choisie la forme d'une trame ASCII (plus facile à lire pour le débogage) avec le format suivant : « v12345d678\r ».

- 'v' marque le début de la trame,

- 12345 est un nombre entier sur 5 chiffres indiquant la consigne de vitesse en mm.s^{-1} avec un offset de 4000 (04000 correspond à 0 m.s^{-1}, 05000 correspond à 1 m.s^{-1} et 03000 correspond à 1 m.-1 en marche arrière).
- 'd' marque la transition entre les consignes de vitesse et de direction
- 678 est un nombre entier sur 3 chiffres indiquant la consigne de direction en degré, avec un offset de 90° (072 correspond à une consigne de -18° donc la rotation maximale dans le sens horaire, vers la droite et 108 correspond à une consigne de +18° donc la rotation dans le sens trigonométrique, vers la gauche.
- '\r' est le caracère de « retour chariot » indiquant la fin de la transmission.

La trame envoyée au repos est donc « v04000d090\r ». Le très léger logiciel *minicom* (`sudo apt install minicom`) permet de tester l'envoi des commandes par la liaison USB-série.

Création du paquet monPaquetCoVAPSyR

L'instruction suivante, comme indiqué dans le tutoriel *ros2/jazzy*, crée le paquet ROS2 *monPaquetCoVAPSyR* et le nœud *CoVAPSy_cmdR* avec l'ajout en dépendance des messages ackermann. Il faut l'exécuter depuis le dossier ros2_ws/src, où sont réunis les paquets personnels.

```
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name CoVAPSy_cmdR monPaquetCoVAPSyR --dependencies rclpy geometry_msgs ackermann_msgs
```

Codage du nœud de commande de la voiture

Coder le nœud de commande de la voiture, en remplissant le fichier *CoVAPsy_cmdR.py* situé dans le dossier *ros2_ws/src/monPaquetCoVAPSyR/monPaquetCoVAPSyR* avec le code suivant (disponible aussi en annexe). Pour faciliter l'édition des fichiers distants, il est possible d'utiliser le mode remote de VsCode (avec le plugin remote-ssh).

```python
from ackermann_msgs.msg import AckermannDrive

import rclpy
from rclpy.node import Node

import serial as s

port_serie = s.Serial(port='/dev/ttyACM0', baudrate=115200, bytesize=8, parity='N',
                      stopbits=1, timeout=None, write_timeout=None,
                      xonxoff=False, rtscts=False, dsrdtr=False)


class NoeudCommande(Node):
    def __init__(self):
        super().__init__('CoVAPSy_cmdR')
        self.__vitesse_m_s = 0.0
        self.__direction_degre = 0
        self.create_subscription(AckermannDrive, 'cmd_ackermann', self.__cmd_ackermann_callback, 1)
        self.get_logger().info('noeud cree')

    def __cmd_ackermann_callback(self, message):
        self.__vitesse_m_s = message.speed
        self.__direction_degre = message.steering_angle
        if self.__direction_degre > 25:
            self.__direction_degre = 25
        elif self.__direction_degre < -25:
            self.__direction_degre = -25
        try:
            direction = int(float(90 + self.__direction_degre))
        except:
            self.get_logger().warn('Bug direction:{},{}'.format(direction, type(direction)))
        vitesse = int(4000 + self.__vitesse_m_s*1000)   # 4000 vitesse nulle
        port_serie.write(str.encode('v{0:05}d{1:03}\r'.format(vitesse, direction)))
        self.get_logger().info('v{0:05}d{1:03}'.format(vitesse, direction))
```

```
def main(args=None):
    rclpy.init(args=args)
    noeud = NoeudCommande()
    rclpy.spin(noeud)
    rclpy.shutdown()


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

Après l'initialisation du port série, la fonction constructeur *__init__()* crée les attributs privés de l'objet, dont *self.__vitesse_m_s* et *self.__direction_degre*, crée le nœud et le fait souscrire au topic */cmd_ackermann* (le topic utilisé par le nœud de conduite pour transmettre les consignes de vitesse et direction).

A chaque réception d'un message du topic */cmd_ackermann*, la fonction *__cmd_ackermann_callback()* est appelée et les valeurs des attributs *self.__vitesse_m_s* et *self.__direction_degre* y sont mises à jour puis envoyées au moteur de propulsion (*vitesse_m_s*) et au moteur de direction (*direction_degre*).

Déclaration des fichiers ajoutés au projet

Dans *ros2_ws/src/monPaquetCoVAPSyR/setup.py*, ajouter les liens vers les noeuds nécessaires. Le fichier est aussi donné en annexe [14], il faut juste mettre en commentaire, pour l'instant, la ligne concernant le nœud de conduite

```
from setuptools import find_packages, setup

package_name = 'monPaquetCoVAPSyR'

setup(
        name=package_name,
        version='0.0.0',
        packages=find_packages(exclude=['test']),
        data_files=[
        ('share/ament_index/resource_index/packages',
                ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        ],
        install_requires=['setuptools'],
        zip_safe=True,
        maintainer='voituremaxime',
        maintainer_email='voituremaxime@todo.todo',
        description='TODO: Package description',
        license='Apache-2.0',
        tests_require=['pytest'],
        entry_points={
                'console_scripts': [
                        'CoVAPSy_cmdR = monPaquetCoVAPSyR.CoVAPSy_cmdR:main'
                        # 'CoVAPSy_conduiteR = monPaquetCoVAPSyR.CoVAPSy_conduiteR:main'
                ],
        },
)
```

Test du nœud CoVAPSy_cmdR

Une fois le nœud créé, il est possible de tester sa syntaxe :

```
colcon test-result --all --verbose
```

Le nœud testé et le paquet configuré dans *setup.py*, on construit le paquet et on lance le nœud, depuis le dossier *ros2_ws* :

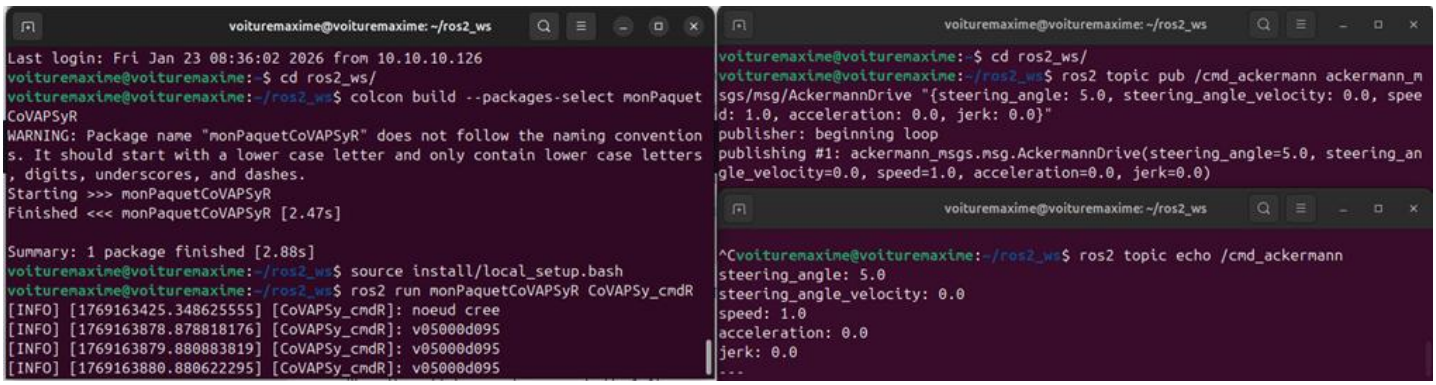
```
colcon build --packages-select monPaquetCoVAPSyR
source install/local_setup.bash
ros2 run monPaquetCoVAPSyR CoVAPSy_cmdR
```

Pour tester le bon fonctionnement, il est possible d'envoyer des messages avec consignes de vitesse et de direction sur le topic */cmd_ackermann* auquel le nœud CoVAPSy_cmd est abonné.

```
ros2 topic pub /cmd_ackermann ackermann_msgs/msg/AckermannDrive
"{steering_angle: 5.0, steering_angle_velocity: 0.0, speed: 1.0,
acceleration: 0.0, jerk: 0.0}"
```

Dans une nouvelle console, on affiche également, en guise de monitoring, les messages du topic */cmd_ackermann* avec la commande :

```
ros2 topic echo /cmd_ackermann
```



La voiture répond bien aux commandes du topic */cmd_ackermann*, le nœud de commande est fonctionnel.

1.4 - Création du nœud de conduite

Le nœud d'acquisition des données LiDAR et le nœud de commande étant fonctionnels, il reste à créer le nœud de conduite où sera codé l'algorithme de contrôle de la voiture. Ce nœud est abonné au topic de type *LaserScan* (nommé */scan*) du LiDAR et émet le topic de type *AckermannDrive* (nommé */cmd_ackermann*) destiné à commander le véhicule.

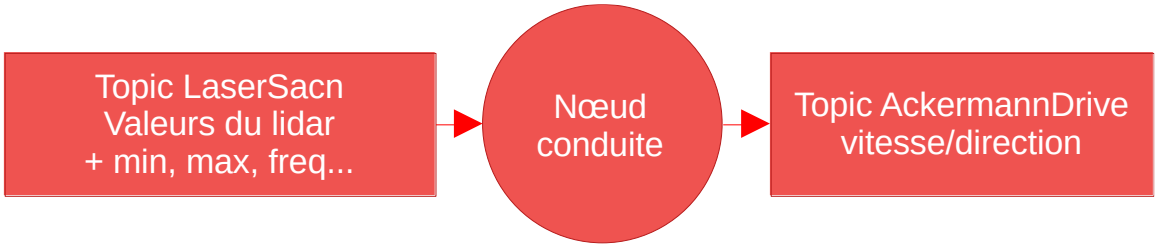


Figure 4 : Topics reçus et émis par le nœud conduite

Pour faire simple, l'algorithme de conduite est extrêmement simple : la vitesse est de 0,5 $m.s^{-1}$ et la direction est proportionnelle à la différence entre la distance à l'obstacle à gauche et la distance à l'obstacle à droite.

```
Vitesse = 0,5
Direction = tableauDesValeursLidar[indexAngle 60°] - tableauDesValeursLidar[indexAngle -60°]
```

Ajouter un fichier *CoVAPSy_conduiteR.py* au dossier *ros2_ws/src/monPaquetCoVAPSyR/monPaquetCoVAPSyR* et y copier le contenu suivant (le fichier est aussi fourni en annexe.

```
cd src/monPaquetCoVAPSyR/monPaquetCoVAPSyR/
nano CoVAPSy_conduite.py
```

```
import rclpy
from ackermann_msgs.msg import AckermannDrive
from sensor_msgs.msg import LaserScan
from rclpy.node import Node
```

```
class Noeudconduite(Node):
    def __init__(self):
        super().__init__('CoVAPSy_conduiteR')
        # ROS interface
        self.__ackermann_publisher = self.create_publisher(AckermannDrive, 'cmd_ackermann', 1)
        self.create_subscription(LaserScan, 'scan', self.__on_lidar_acquisition, 1)
        self.get_logger().info('noeud cree')

    def __on_lidar_acquisition(self, message):
        tableauLidar = list(message.ranges)
        self.get_logger().info(f'60 {tableauLidar[533]:.2f} et -60 {tableauLidar[2666]:.2f}')
        command_message = AckermannDrive()
        command_message.speed = 1.0
        try:
            command_message.steering_angle = 100 * (tableauLidar[533] - tableauLidar[2666])
        except IndexError:
            command_message.steering_angle = 0.0
        if command_message.steering_angle > 18.0:
            command_message.steering_angle = 18.0
        if command_message.steering_angle < -18.0:
            command_message.steering_angle = -18.0
        self.__ackermann_publisher.publish(command_message)
        self.get_logger().info(f'v={command_message.speed:.2f} m/s,d= {command_message.steering_angle:.2f} rad')


def main(args=None):
    rclpy.init(args=args)
    noeud = Noeudconduite()
    rclpy.spin(noeud)
    rclpy.shutdown()


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

La fonction constructeur *__init__()* crée le nœud, crée le topic */cmd_ackermann* pour publier les consignes de vitesse et direction et souscrit au topic */scan* où publie le LiDAR.

Grâce à cette souscription, quand un message est publié par le LiDAR, la fonction *__on_lidar_acquisition()* s'exécute. Les 3200 données de distance (en m) acquises sur un tour (attribut range du message) sont stockées dans un tableau. Sont utilisées dans cet exemple très simple seulement la valeur à 60° (devant à gauche, index 533 du tableauLidar) et devant à droite (index 2666 du tableauLidar). On crée ensuite un message de type AckermannDrive dont on met l'attribut vitesse à 1 et la direction proportionnelle à la différence des deux distances citées ci-dessus, ce qui est suffisant pour une conduite simple. Le message est ensuite publié.

Une fois le nœud enregistré, il faut l'ajouter au fichier *setup.py*.

```
        entry_points={
                'console_scripts': [
                        'CoVAPSy_cmdR = monPaquetCoVAPSyR.CoVAPSy_cmdR:main'
                        'CoVAPSy_conduiteR = monPaquetCoVAPSyR.CoVAPSy_conduiteR:main'
                ],
```

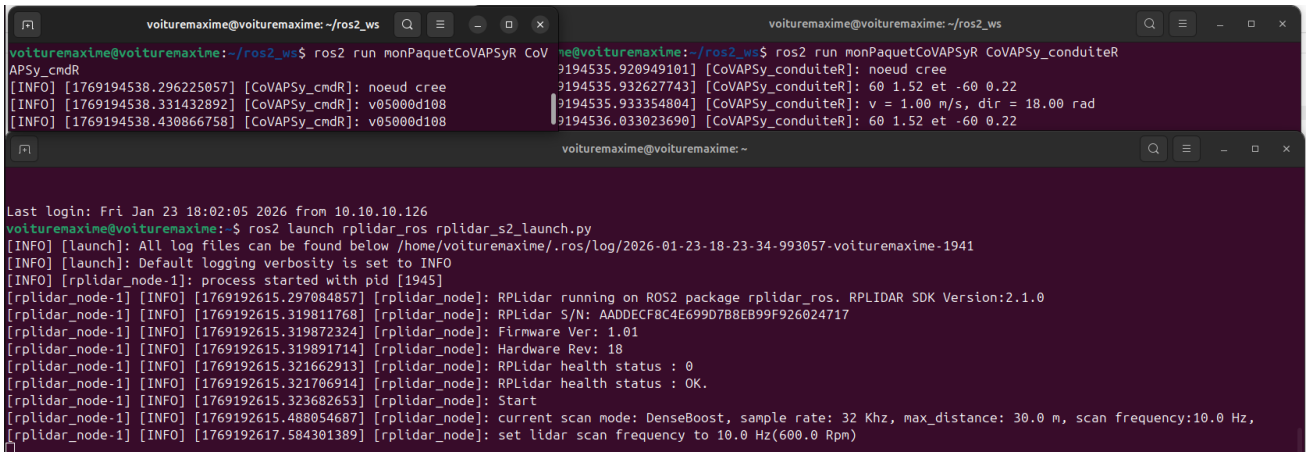
On peut alors tester et construire le nœud et le lancer pour vérifier qu'il s'exécute.

```
cd
cd ros2_ws/
colcon test-result --all --verbose
colcon build --packages-select monPaquetCoVAPSyR
source install/local_setup.bash
ros2 run monPaquetCoVAPSyR CoVAPSy_conduiteR
```

1.5 - Test de la conduite du véhicule

Les trois nœuds construits, il est possible de tester la conduite du véhicule, en ouvrant trois consoles pour lancer les trois nœuds.

```
ros2 launch rplidar_ros rplidar_s2_launch.py
ros2 run monPaquetCoVAPSyR CoVAPSy_cmdR
ros2 run monPaquetCoVAPSyR CoVAPSy_conduiteR
```



La voiture parcourt la piste ou un couloir.

1.6 - Utilisation d'un PC stationnaire pour la visualisation et le monitoring et/ou le calcul déporté

Les messages ROS2 pouvant utiliser UDP pour être transmis, il est possible de déporter le nœud de conduite sur un PC stationnaire plus puissant. Pour cela, le PC doit être sur le même réseau wifi que la voiture, avoir la même version de ROS2 et utiliser le même *ROS_DOMAIN_ID*.

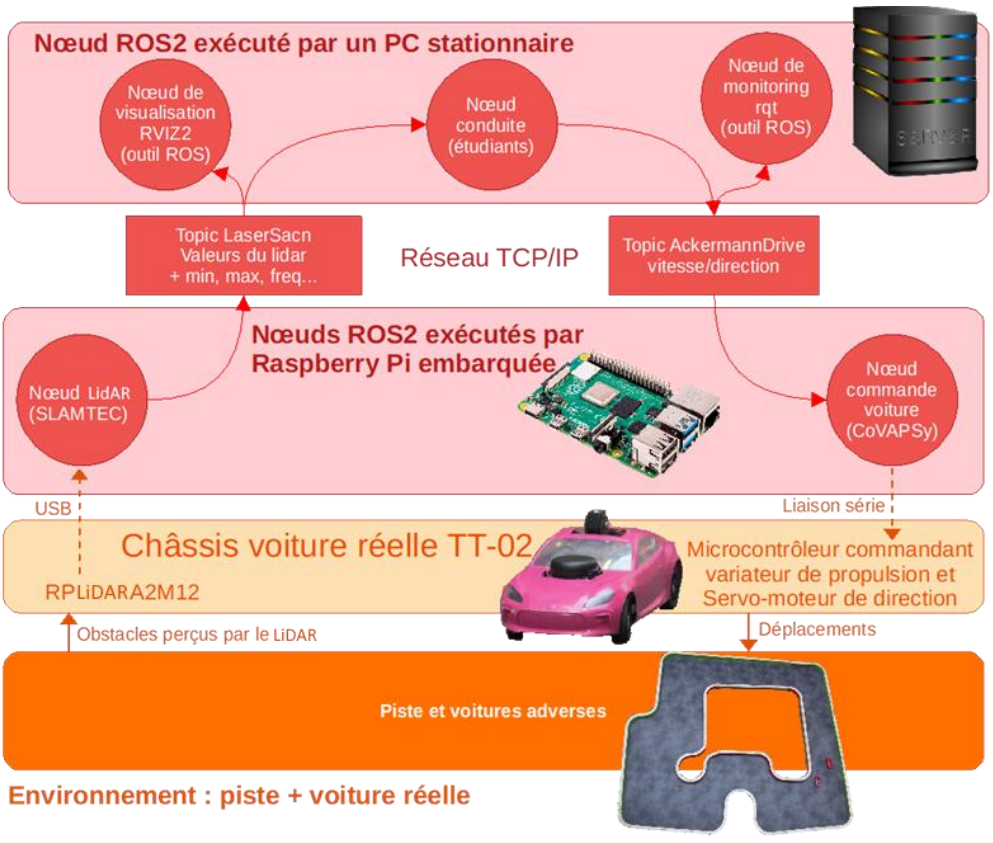
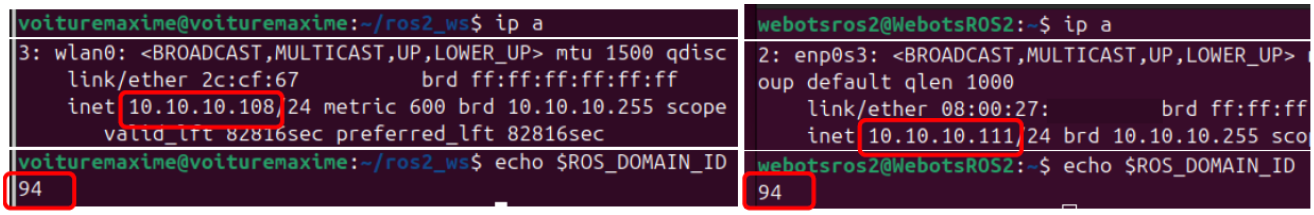


Figure 5 : Nœuds et topic ROS2 dans le cas d'un contrôle déporté et d'un monitoring déporté

Quel que soit la machine sur laquelle est exécuté le nœud de conduite, il est possible d'utiliser le PC stationnaire pour le monitoring avec les outils ROS2 : *rviz* et *rqt*. Cela a l'intérêt notamment de dispenser le nano-ordinateur d'un environnement graphique, ce qui améliore ses performances, tout en permettant de superviser le bon fonctionnement.

Commencer par vérifier que la voiture et le PC stationnaire sont dans le même réseau (ip a pour afficher l'adresse IP) et ont le même *ROS_DOMAIN_ID* (*echo $ROS_DOMAIN_ID* pour l'afficher).



Les nœuds exécutés sur la voiture, il est alors possible de lancer *rqt* sur le PC stationnaire :

Figure 6 : Graphe des nœuds ROS2 exécutés dans la voiture, affiché sur le PC stationnaire

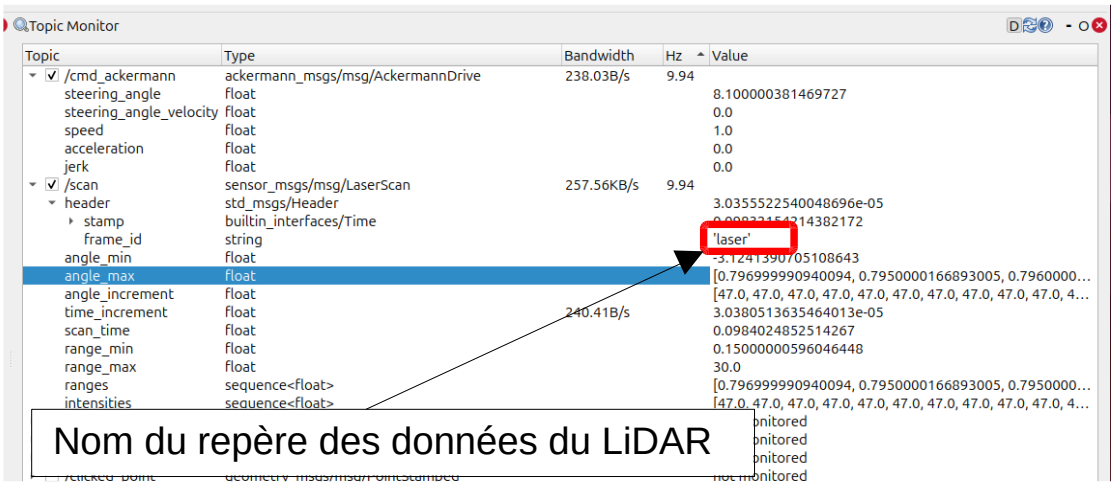


Figure 7 : Supervision des messages échangés dans la voiture sur les topics /cmd_ackermann et /scan, par rqt depuis le PC stationnaire

Pour afficher les données du LiDAR dans rviz2, il faut situer le LiDAR dans la carte. Pour un usage avancé, pour faire du SLAM par exemple, on situe le LiDAR par rapport à l'origine de la voiture et la voiture dans l'espace. Ici, juste pour afficher les données du LiDAR, on le place à l'origine. Le LiDAR est associé au repère (*frame_id* dans ROS2) *'laser'*, comme l'indique la supervision du topic */scan* ci-dessus.

La commande ROS2 pour placer le repère *laser* à l'origine du monde (!) est la suivante :

```
ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 laser world
```

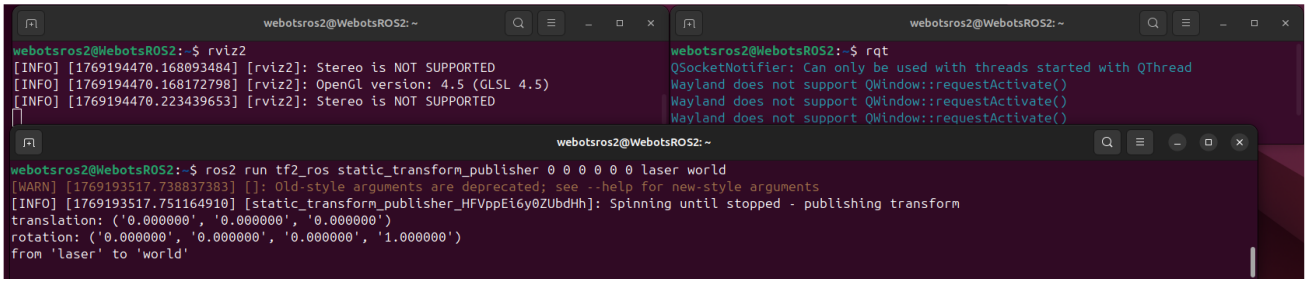


Figure 8 : Consoles exécutées sur le PC stationnaires pour la supervision du comportement de la voiture avec rqt et rviz2

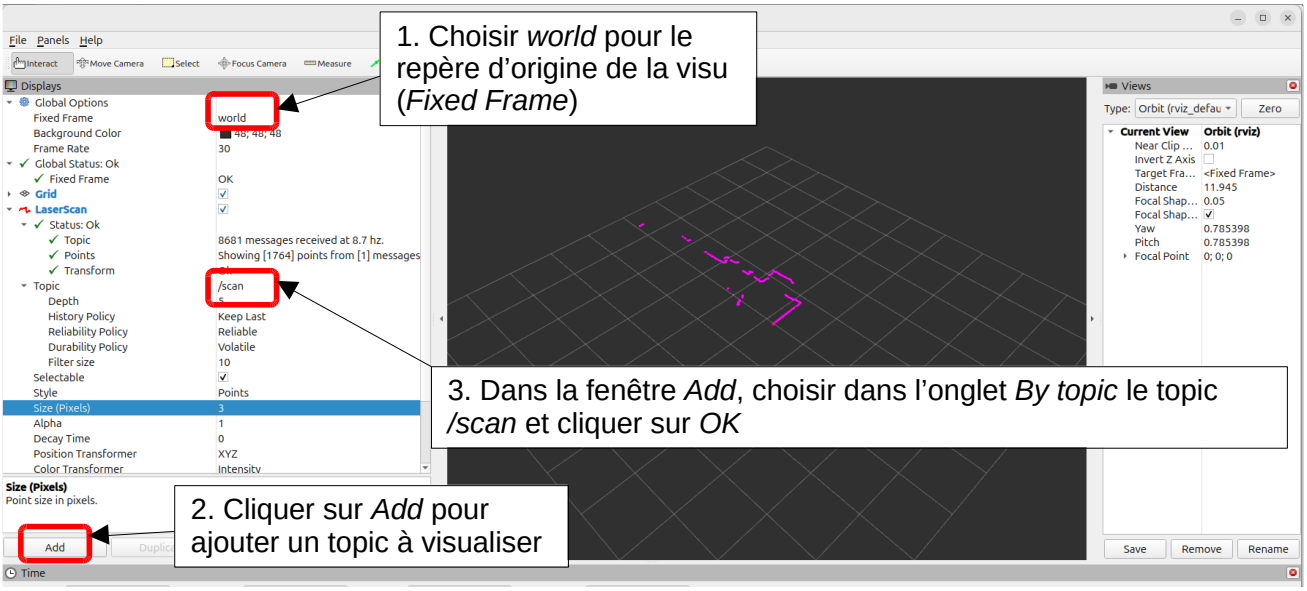


Figure 9 : Affichage des données du LiDAR de la voiture dans rviz2 depuis le PC stationnaire

2 - Mise en œuvre de ROS2 pour le contrôle d'une voiture 1/10ème de type CoVAPSy simulée sous Webots

Pour développer les algorithmes de conduite sans les contraintes matérielles de la voiture (espace pour la piste, recharge des batteries…), il est intéressant de pouvoir travailler sur un simulateur. Cela devient indispensable lorsqu'on souhaite faire de l'apprentissage par renforcement [8], la voiture nécessitant un nombre d'essais et de chocs incompatible avec la robustesse de sa mécanique. Webots est le simulateur choisi pour sa popularité en robotique, sa facilité de mise en œuvre, et sa faculté à fonctionner sur un ordinateur sans carte graphique [2].

Webots peut être utilisé sans ROS2 pour simuler la voiture [2]. Cependant, si la voiture réelle est prévue pour fontionner avec ROS2, simuler avec ROS2 est un atout important, très utilisé en robotique. Webots-ROS2 fournit un nœud rpLidar émettant un topic similaire au nœud slamtec et le topic de type AckermannDrive est facilement accepté pour commander la voiture simulée. Le nœud de conduite sera alors identique de toute part au nœud de la voiture physique. Il est alors possible de développer ce nœud sur la voiture simulée pour ensuite le copier sur la voiture réelle.

Cette partie présente pas à pas l'installation de ROS2 pour webots, l'exploitation des messages LiDAR et l'écriture du nœud de commande pour mener à la conduite basique, avec le même nœud que dans la partie précédente.

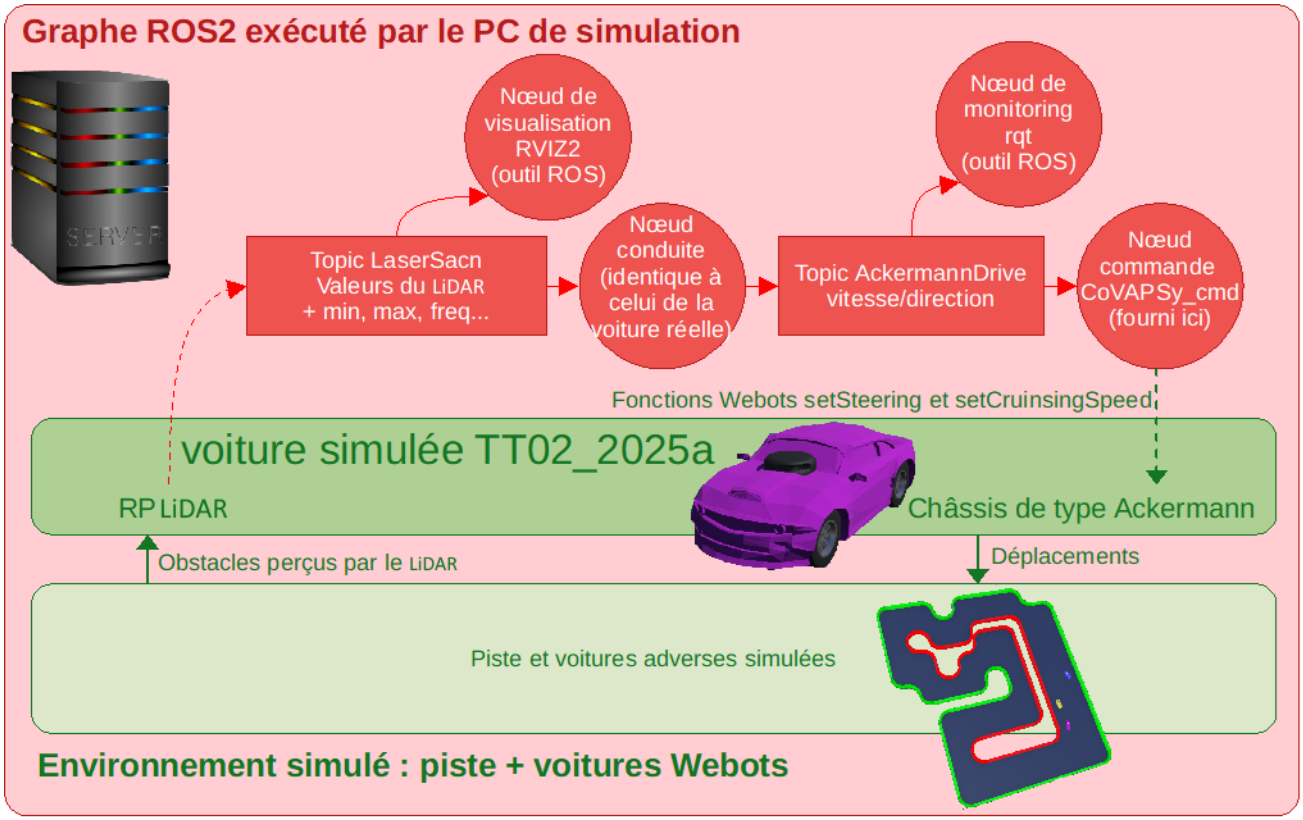


Figure 10 : Nœuds et topic ROS2 dans le cas de la conduite d'une voiture simulée sur webots

2.1 - Installation de webots R2025a

La ressource « *CoVAPSy : Mise en œuvre du simulateur Webots* » [2] permet de faire les premiers pas avec webots (attention elle a été écrite pour webots 2023a, quelques ajustements mineurs sont à prévoir pour fonctionner avec webots R2025a) et les voitures CoVAPSy.

Sur ubuntu 24 (sur une machine physique ou virtuelle), quelques paquets sont à installer avant webots :

```
sudo apt install make g++ ffmpeg libfreeimage3 libssh-dev libzip-dev
libxcb-xinerama0 libxcb-cursor0
```

Si un problème de dépendances persiste, la commande suivante résoud habituellement les soucis, avant de relancer la ligne d'installation précédente :

```
sudo apt --fix-broken install
```

Le paquet webots_2025a_amd64.deb se télécharge depuis la page d'accueil de webots et s'installe avec la commande suivante.

```
sudo dpkg -i webots_2025a_amd64.deb
```

Depuis le dépôt git de la course [9], copier le dossier *Simulateur_CoVAPSy_Webots2025a_Base.zip* (1,1 Mo), en extraire le contenu dans le dossier *Documents* par exemple. Depuis webots, ouvrir le monde (*Documents/Simulateur_CoVAPSy_Webots2025a_ROS2/worlds/Piste_CoVAPSy_2025a.wbt*) et tester le bon fonctionnement du simulateur.

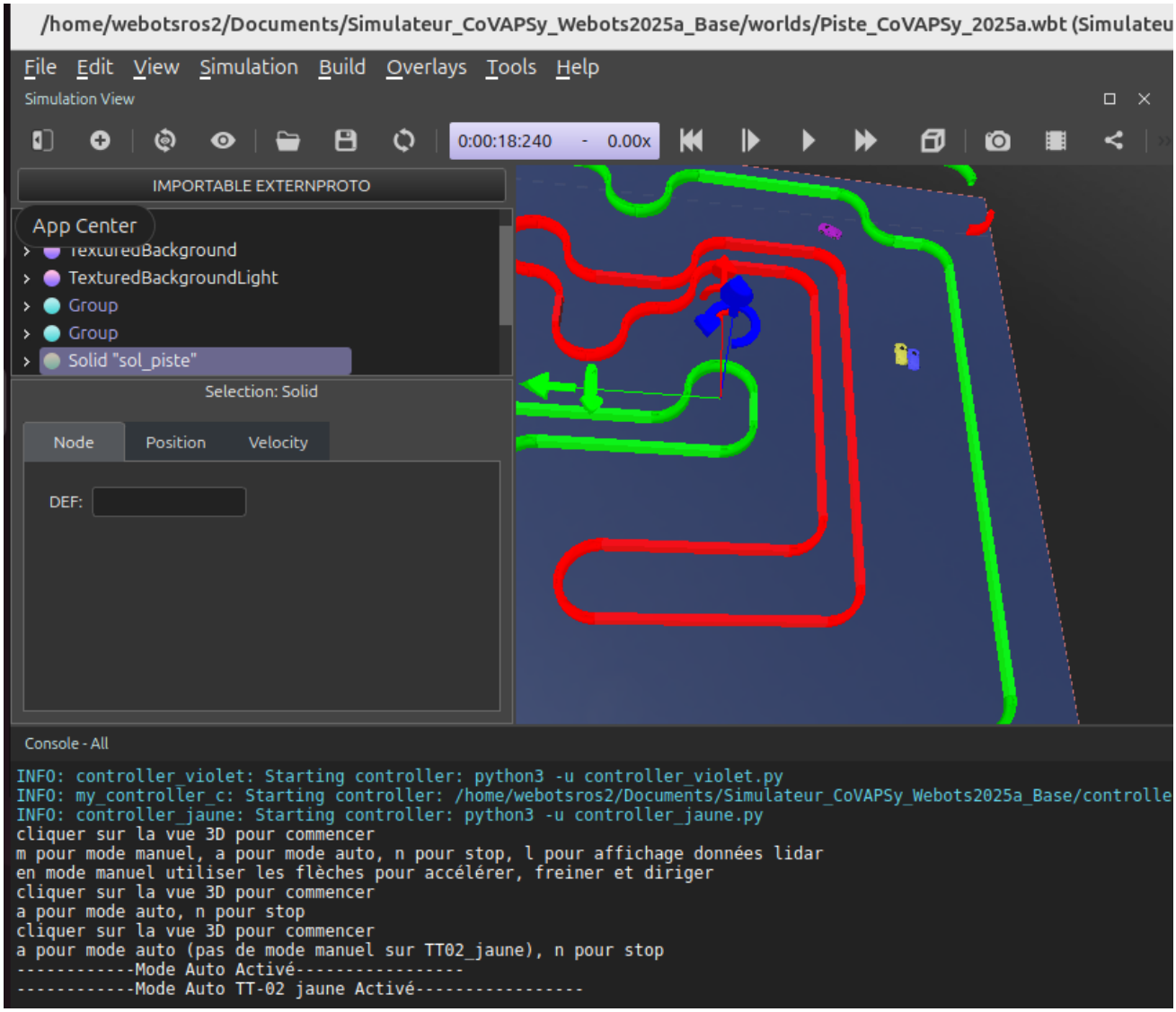


Figure 11 : Test du monde de base de la simulation CoVAPSy sous webots R2025a

2.2 - Suivi du tutoriel ros2 pour webots

Les liens [3], [4] et [5] permettent de prendre en main ROS2 pour webots.

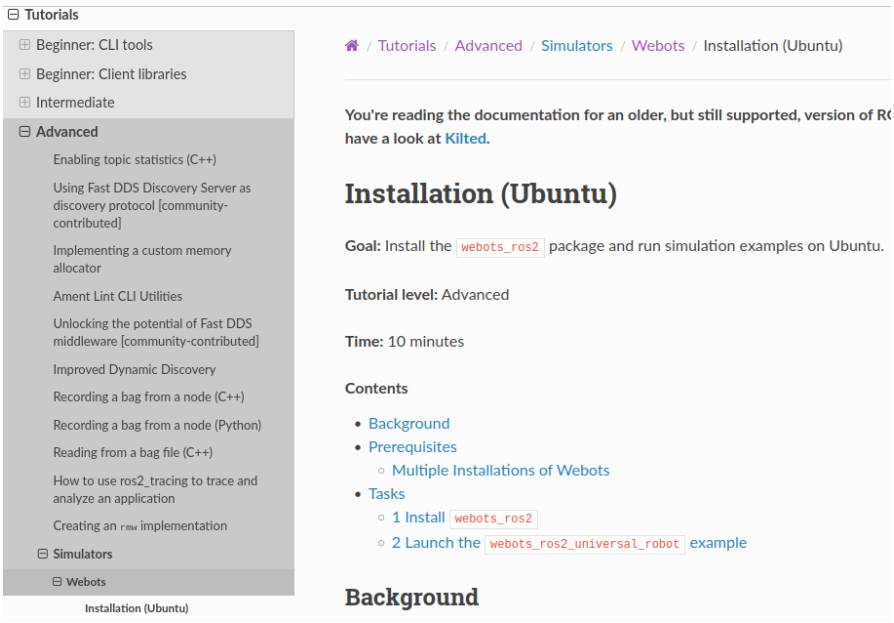


Figure 12 : Copie d'écran de la page web du tutoriel ROS2 jazzy

Le tutoriel [3], onglet *Simulators/webots/Installation (Ubuntu)* donne les indications pour l'installation du paquet ROS2 pour webots et le lancement d'un exemple :

```
sudo apt-get install ros-jazzy-webots-ros2
source /opt/ros/jazzy/setup.bash
export WEBOTS_HOME=/usr/local/webots
cd ~/ros2_ws
source install/local_setup.bash
ros2 launch webots_ros2_universal_robot multirobot_launch.py
```

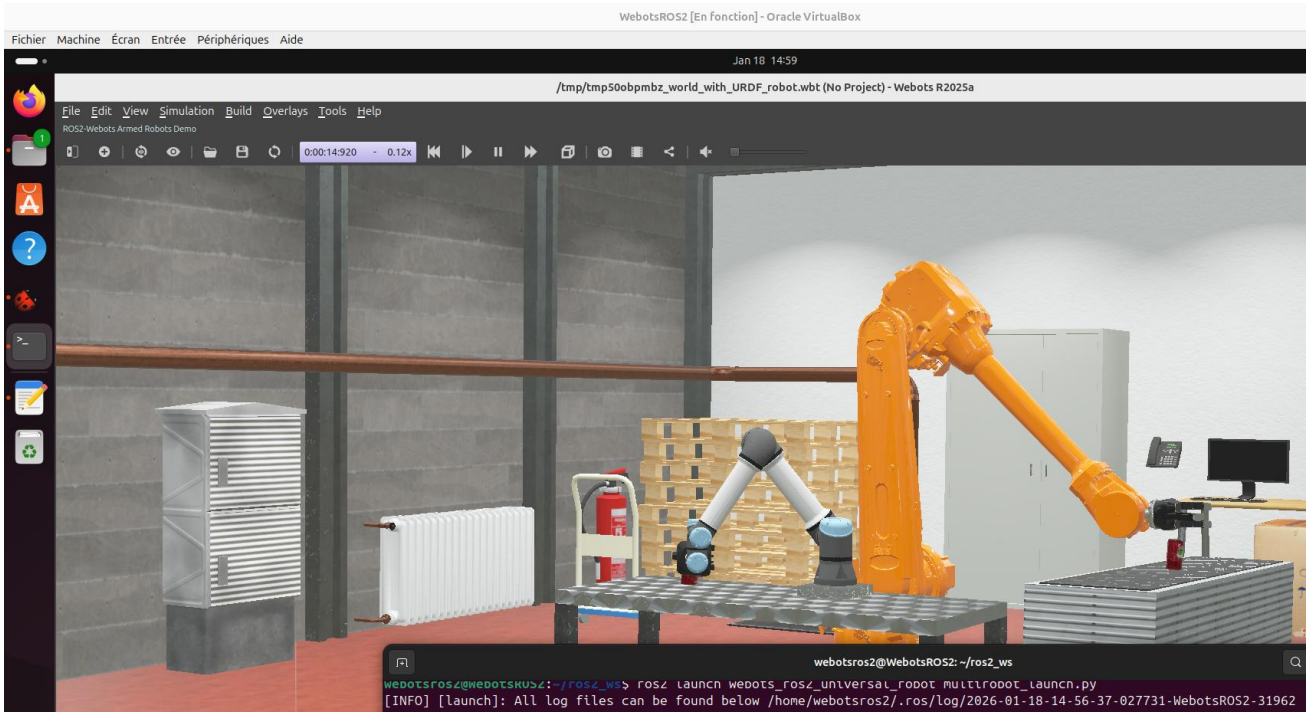


Figure 13 : Exemple ros2 pour webots

2.3 - Création du package et du nœud de commande de la voiture

ROS2 pour webots installé, il s'agit désormais de créer le package *monPaquetCoVAPSy* avec le monde associé et d'écrire le nœud de commande de la voiture. L'onglet *Setting up a robot simulation* du tutoriel enseigne cela.

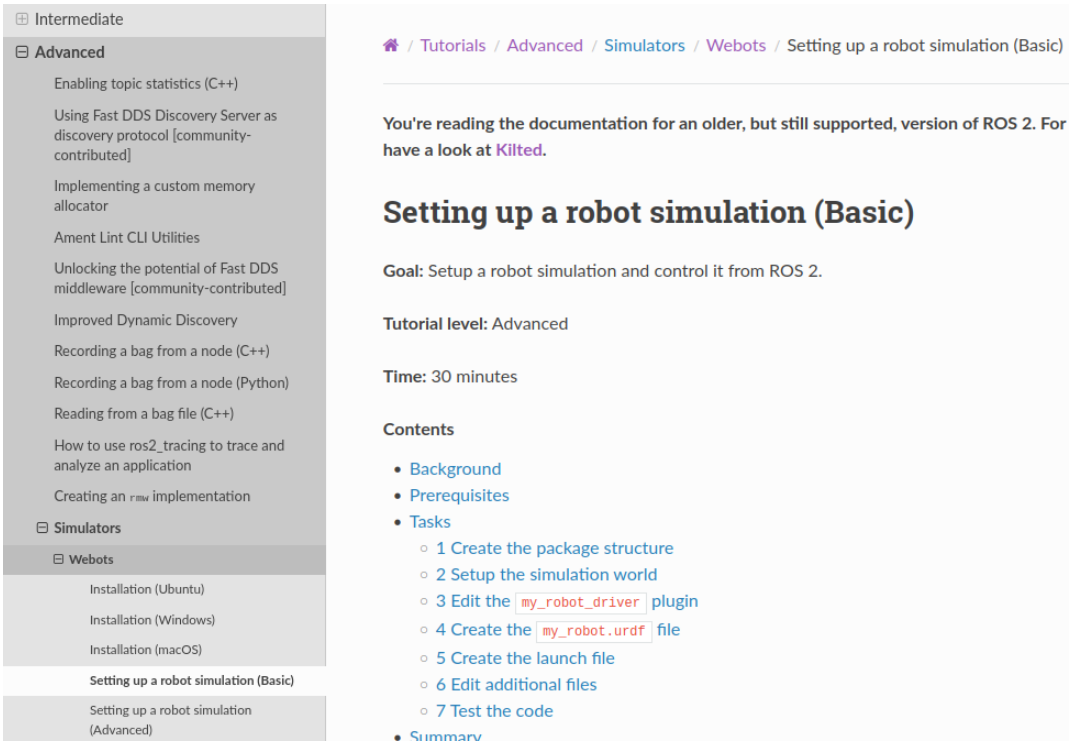


Figure 14 : Copie d'écran du tutoriel de création du package pour utiliser ros2 et un environnement webots

Création du paquet monPaquetCoVAPSy

L'instruction suivante, issue du tutoriel, avec l'ajout en dépendance des messages ackermann, crée le paquet ROS2 *monPaquetCoVAPSy* qui sera utile pour s'interfacer avec webots. Il faut l'exécuter depuis le dossier ros2_ws/src, où sont réunis les paquets personnels.

```
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name CoVAPSy_cmd monPaquetCoVAPSy --dependencies rclpy geometry_msgs webots_ros2_driver ackermann_msgs
```

Ajout du monde CoVAPSy au package

Ensuite, dans le dossier *ros2_ws/src/monPaquetCoVAPSy*, copier les dossiers *worlds*, *protos* et *controllers* du dossier *Simulateur_CoVAPSy_Webots2025a_Base_v2* disponible sur le dépôt git de la course [11] et en annexe de cette ressource.

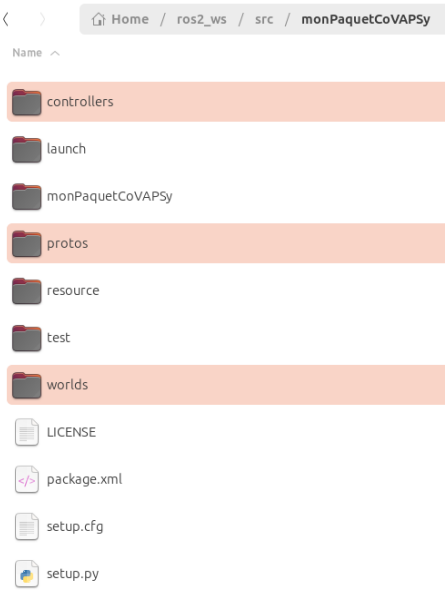


Figure 15 : Arborescence du dossier ros2_ws/src/monPaquetCoVAPSy

Lancer alors le logiciel webots et ouvrir le fichier monde suivant :

ros2_ws/src/monPaquetCoVAPSy/worlds/Piste_CoVAPSy_2025a.wbt

Dans l'arborescence du projet webots, au temps 0 et en pause, modifier le contrôleur de la voiture jaune pour un contrôleur externe (la voiture jaune n'est plus contrôllée par le programme python d'exemple) puis fermer webots. Depuis l'arborescence, supprimer également la voiture bleue, non utilisée ici. Enregistrer le monde (*File > Save World*) et fermer

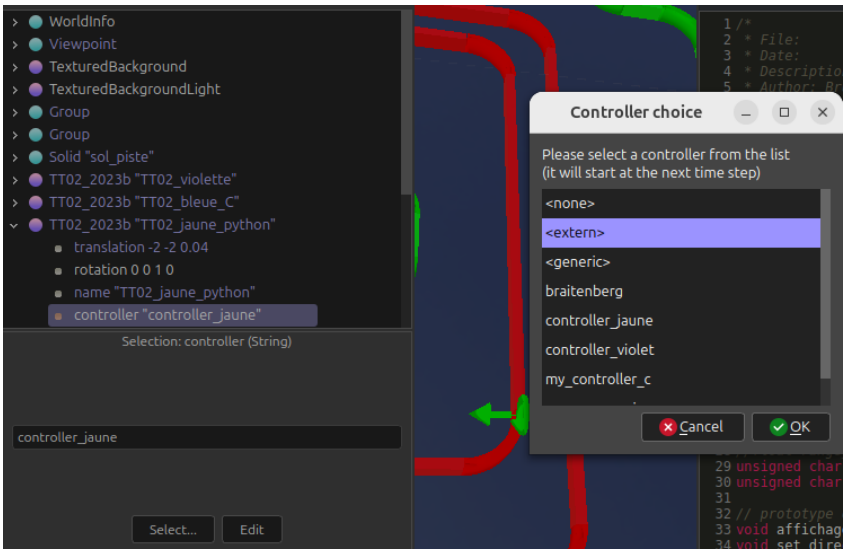


Figure 16 : Modification du type de contrôleur pour la voiture TT02_jaune_python

Création du nœud de commande de la voiture

Créer le nœud de commande de la voiture, en remplissant le fichier *CoVAPSy_cmd.py* situé dans le dossier *ros2_ws/src/monPaquetCoVAPSy/monPaquetCoVAPSy* avec le code suivant (disponible aussi en annexe) :

```
import rclpy
from ackermann_msgs.msg import AckermannDrive

class CoVAPSy_cmd:
    def init(self, webots_node, properties):
        self.__robot = webots_node.robot
        self.__vitesse_m_s = 0.0
        self.__direction_degre = 0
        # ROS interface
        rclpy.init(args=None)
        self.__node = rclpy.create_node('CoVAPSy_cmd')
        self.__node.create_subscription(AckermannDrive,'cmd_ackermann', self.__cmd_ackermann_callback, 1)
        self.__node.get_logger().info("noeud cree")
        self.__robot.setCruisingSpeed(self.__vitesse_m_s*3.6)
        self.__robot.setSteeringAngle(-self.__direction_degre*3.14/180)

    def __cmd_ackermann_callback(self, message):
        self.__vitesse_m_s = message.speed
        self.__direction_degre = message.steering_angle
        # self.__node.get_logger().info(
        #     f"[CoVAPSy_cmd] Reçu : vitesse = {self.__vitesse_m_s} m/s, direction = {self.__direction_degre}°")

    def step(self):
        rclpy.spin_once(self.__node, timeout_sec=0)
        self.__robot.setCruisingSpeed(self.__vitesse_m_s*3.6)
        self.__robot.setSteeringAngle(-self.__direction_degre*3.14/180)
```

La fonction constructeur *init()* crée les attributs privés de l'objet, dont *self.__vitesse_m_s* et *self.__direction_degre*, crée le nœud et le fait souscrire à *cmd_ackermann* (le topic utilisé par le nœud de conduite pour transmettre les consignes de vitesse et direction). Ensuite, dans l'initialisation puis dans la fonction *step()* appelée à chaque pas du simulateur, les valeurs de vitesse

et direction sont envoyées au moteur de propulsion (*setCruisingSpeed*) et au moteur de direction (*setSterringAngle*).

Lorsqu'un message du topic *cmd_ackermann* arrive, la fonction *__cmd_ackermann_callback()* est appelée et les attributs *self.__vitesse_m_s* et *self.__direction_degre* sont mis à jour avec les valeurs *speed* et *steering_angle* du topic.

Création du lien entre le nœud de commande et la voiture webots

Créer ensuite le lien entre le nœud ROS2 *CoVAPSy_cmd* et l'objet webots *TT02_jaune_python* en créant dans le dossier */ros2_ws/src/monPaquetCoVAPSy/ressource* un fichier texte *TT02_jaune_python.urdf* avec le contenu suivant (disponible aussi en annexe) :

```
<?xml version="1.0"?>
<robot name="TT02_jaune_python">
    <webots>
        <plugin type="monPaquetCoVAPSy.CoVAPSy_cmd.CoVAPSy_cmd" />
    </webots>
</robot>
```

Création du fichier de lancement

Le fichier *launch* est un fichier regroupant l'ensemble des instructions à effectuer pour démarrer un système ROS2. Créer un dossier *launch* dans */ros2_ws/src/monPaquetCoVAPSy/* et, dans ce dossier *launch*, un fichier *monPaquetCoVAPSy_launch.py*, avec le contenu suivant (aussi fourni en annexe, il faut juste mettre en commentaire les 4 lignes concernant le nœud de conduite) :

```
import os
import launch
from launch_ros.actions import Node
from launch import LaunchDescription
from ament_index_python.packages import get_package_share_directory
from webots_ros2_driver.webots_launcher import WebotsLauncher
from webots_ros2_driver.webots_controller import WebotsController

def generate_launch_description():
    package_dir = get_package_share_directory('monPaquetCoVAPSy')
    robot_description_path = os.path.join(package_dir, 'resource', 'TT02_jaune_python.urdf')


    webots = WebotsLauncher(world=os.path.join(package_dir, 'worlds', 'Piste_CoVAPSy_2025a.wbt'))

    CoVAPSy_cmd = WebotsController(
        robot_name='TT02_jaune_python', parameters=[{'robot_description': robot_description_path},]
        )

#    CoVAPSy_conduite = Node(
#        package='monPaquetCoVAPSy',
#        executable='CoVAPSy_conduite',
#    )

    return LaunchDescription([
        webots,
        CoVAPSy_cmd,
        #CoVAPSy_conduite,
        launch.actions.RegisterEventHandler(
            event_handler=launch.event_handlers.OnProcessExit(
                target_action=webots,
                on_exit=[launch.actions.EmitEvent(event=launch.events.Shutdown())],
            )
        )
    ])
```

Dans ce fichier, on retrouve dans *generate_launch_description()*, le lien entre le projet *webots* et le nœud *CoVAPSy_cmd*.

Via *LaunchDescription()* sont lancés *webots* et le nœud *CoVAPSy_cmd*. Le nœud de conduite n'est pour l'instant pas exécuté et reste en commentaire.

Déclaration des fichiers ajoutés au projet

Dans *ros2_ws/src/monPaquetCoVAPSy/setup.py*, ajouter les liens vers les fichiers *.proto* et *.stl* nécessaire au projet webots. Le fichier est donné en annexe [14], il faut juste mettre en commentaire la ligne concernant le nœud de conduite

```
from setuptools import find_packages, setup

package_name = 'monPaquetCoVAPSy'

data_files = []
data_files.append(('share/ament_index/resource_index/packages', ['resource/' + package_name]))
data_files.append(('share/' + package_name + '/launch', ['launch/monPaquetCoVAPSy_launch.py']))
data_files.append(('share/' + package_name + '/worlds', ['worlds/Piste_CoVAPSy_2025a.wbt']))
data_files.append(('share/' + package_name + '/worlds', ['worlds/ImageToStl_virage.obj']))
data_files.append(('share/' + package_name + '/resource', ['resource/TT02_jaune_python.urdf']))
data_files.append(('share/' + package_name + '/protos', ['protos/TT02_2025a.proto']))
data_files.append(('share/' + package_name + '/protos', ['protos/TT02Wheel.proto']))
data_files.append(('share/' + package_name + '/protos', ['protos/ChevroletCamaroLight.stl']))
data_files.append(('share/' + package_name + '/controllers/controller_violet',
    ['controllers/controller_violet/controller_violet.py']))
data_files.append(('share/' + package_name, ['package.xml']))


setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=data_files,
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='webotsros2',
    maintainer_email='webotsros2@toto.fr',
    description='paquet de commande de la voiture CoVAPSy simulee',
    license='Apache-2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'CoVAPSy_cmd = monPaquetCoVAPSy.CoVAPSy_cmd:main',
            #'CoVAPSy_conduite = monPaquetCoVAPSy.CoVAPSy_conduite:main'
        ],
    },
)
```

Test du nœud CoVAPSy_cmd

Le nœud et l'environnement configurés, on construit le nœud et on le lance, depuis le dossier *ros2_ws* :

```
colcon build --packages-select monPaquetCoVAPSy
source install/local_setup.bash
ros2 launch monPaquetCoVAPSy monPaquetCoVAPSy_launch.py
```

```
webotsros2@WebotsROS2:~/ros2_ws$ colcon build --packages-select monPaquetCoVAPSy

WARNING: Package name "monPaquetCoVAPSy" does not follow the naming conventions. It should sta
rt with a lower case letter and only contain lower case letters, digits, underscores, and dash
es.
WARNING: Package name "paquetROS2CoVAPSy" does not follow the naming conventions. It should st
art with a lower case letter and only contain lower case letters, digits, underscores, and das
hes.
Starting >>> monPaquetCoVAPSy
Finished <<< monPaquetCoVAPSy [0.98s]

Summary: 1 package finished [1.10s]
webotsros2@WebotsROS2:~/ros2_ws$ source install/local_setup.bash

webotsros2@WebotsROS2:~/ros2_ws$ ros2 launch monPaquetCoVAPSy monPaquetCoVAPSy_launch.py
```

La fenêtre webots doit alors se lancer. Pour tester le bon fonctionnement, il est possible d'envoyer des messages sur le topic cmd_ackermann auquel le nœud CoVAPSy_cmd est abonné. Pour cela on utilise l'outil *rqt* de ROS2.

En plus du terminal de lancement du nœud, un terminal permet de lancer rqt et un terminal permet de lancer l'affichage des messages du topic */cmd_ackermann* avec la commande :

```
ros2 topic echo /cmd_ackermann
```

Dans *rqt*, on choisit d'afficher les nœuds (Plugins > Introspection > Node Graph) et de publier des messages sur le topic /cmd_ackermann (*Plugins* > *Topics* > *Message Publisher*). *Rqt* a d'autres fonctionnalités. Ne pas hésiter à les explorer.

Dans le simulateur webots, la voiture violette qui sert de *sparring partner* à la voiture jaune (celle que l'on contrôle) peut être démarrée en cliquant dans la vue 3D et en appuyant sur la touche 'a'. On peut alors depuis *rqt* contrôller la voiture jaune pour concourir contre la voiture automatique violette.

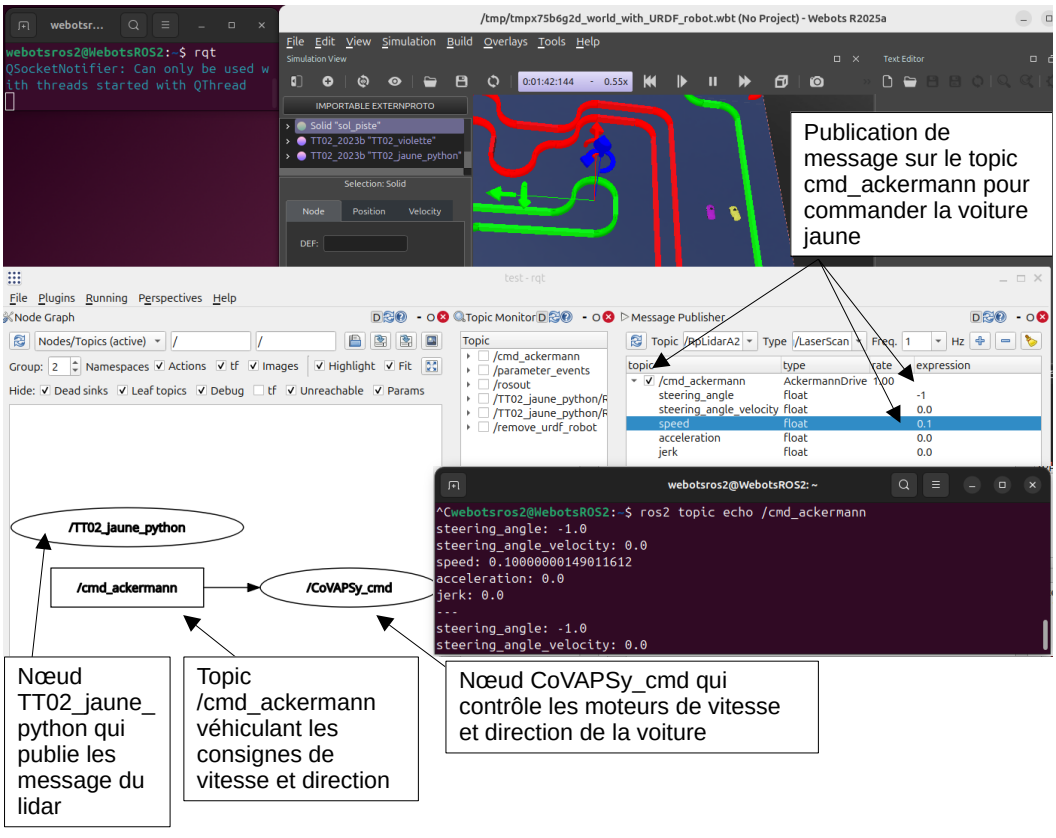
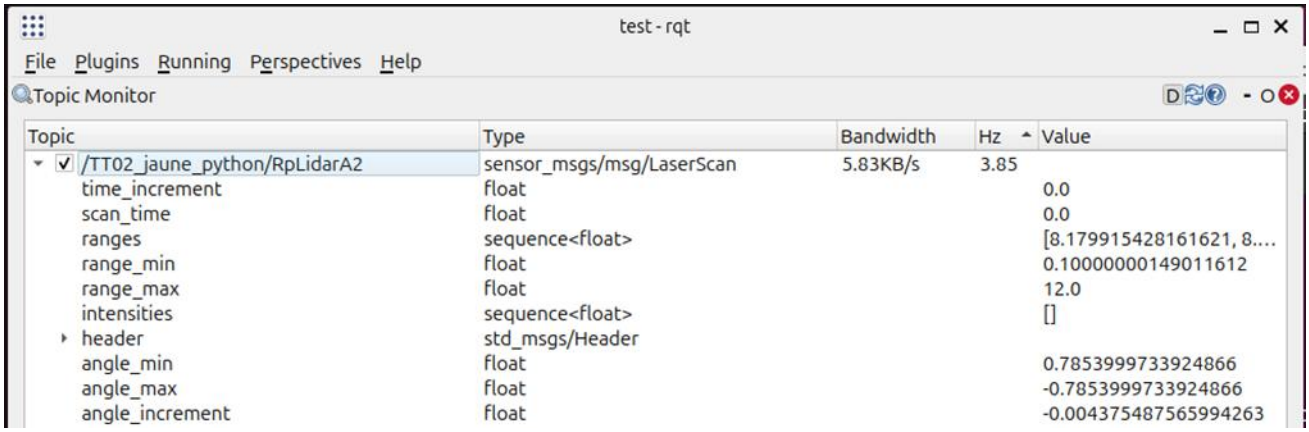


Figure 17 : Test du nœud CoVAPSy_cmd avec la publication par rqt de messages sur le topic cmd_ackermann

2.4 - Les messages RpLidarA2 publiés par la voiture simulée

Le nœud TT02_jaune_python publie des messages de type LaserScan avec les données du LiDAR de la voiture. On peut retrouver le nom du topic et les valeurs publiées dans *rqt* > *Topics* > *Topic Monitor*.



L'attribut ranges contient les distances mesurées par le LiDAR, avec le nombre de points correspondant à ce qui a été défini dans les attributs. Dans le projet de base fourni, la fréquence de rotation est de 12 Hz avec 360 points par tour. Il est possible de modifier ces paramètres pour se rapprocher du LiDAR réel (1600 points par tour annoncés pour le *A2M12* et 3200 points pour le *S2*). On se limite ici à 360 points, suffisants pour développer un nœud fonctionnel. Attention, à chaque modification, il faut reconstruire le paquet (*colcon build*...).

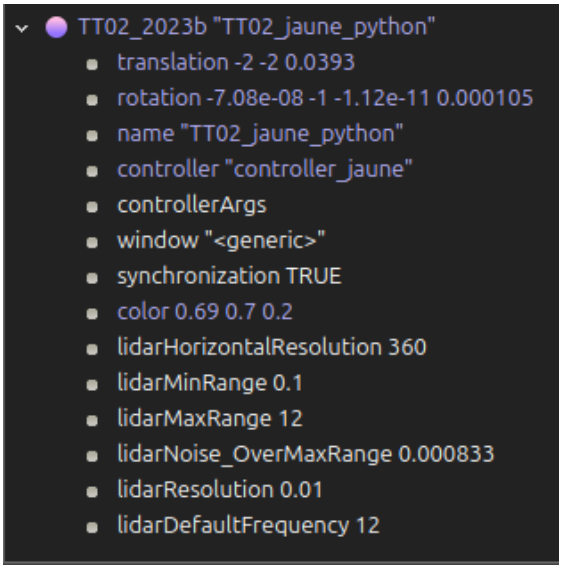


Figure 18 : Paramètres du LiDAR RpLidarA2 de la voiture TT02_jaune_python dans webots

On peut afficher aussi les messages depuis une console avec les commandes suivantes :

```
ros2 topic list
ros2 topic echo /TT02_jaune_python/RpLidarA2
```

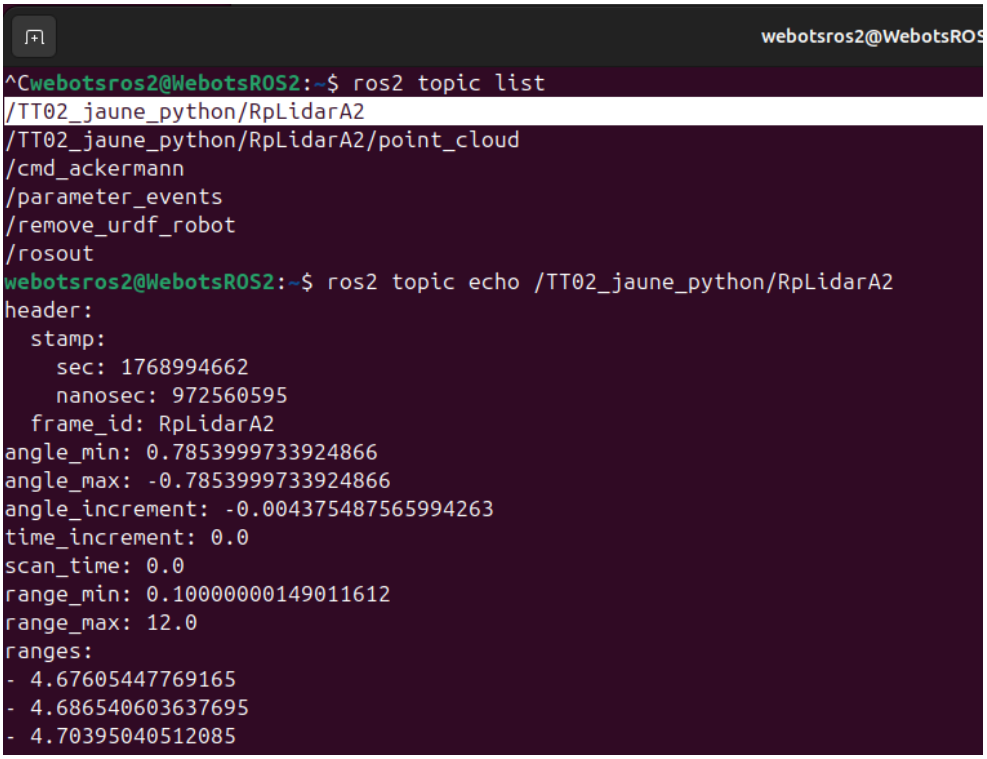


Figure 19 : Affichage des messages du topic RpLidarA2 depuis une console

Pour afficher les données du LiDAR dans rviz2, il faut situer le LiDAR dans la carte. Ici, on le place à l'origine. Le LiDAR est associé au repère (*frame_id* dans ROS2) *RpLidarA2*, comme l'indique la figure ci-dessous.

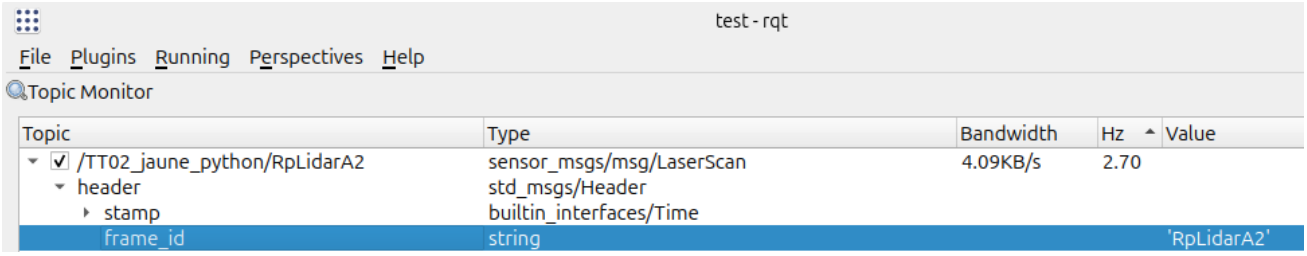
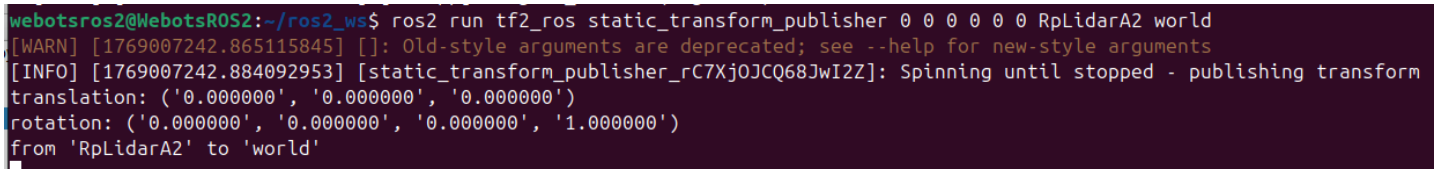


Figure 20 : Repère (frame_id) dans lequel sont données les valeurs du LiDAR

La commande ROS2 pour placer le repère *RPLidarA2* à l'origine du monde est la suivante :

```
ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 RpLidarA2 world
```



Pour afficher les données dans *rviz2*, lancer le paquet *monPaquetCoVAPSy*, lancer la transformée *tf* ci-dessus et lancer *rviz2* (on tape *rviz2* dans une console) et suivre les instructions ci-dessous :

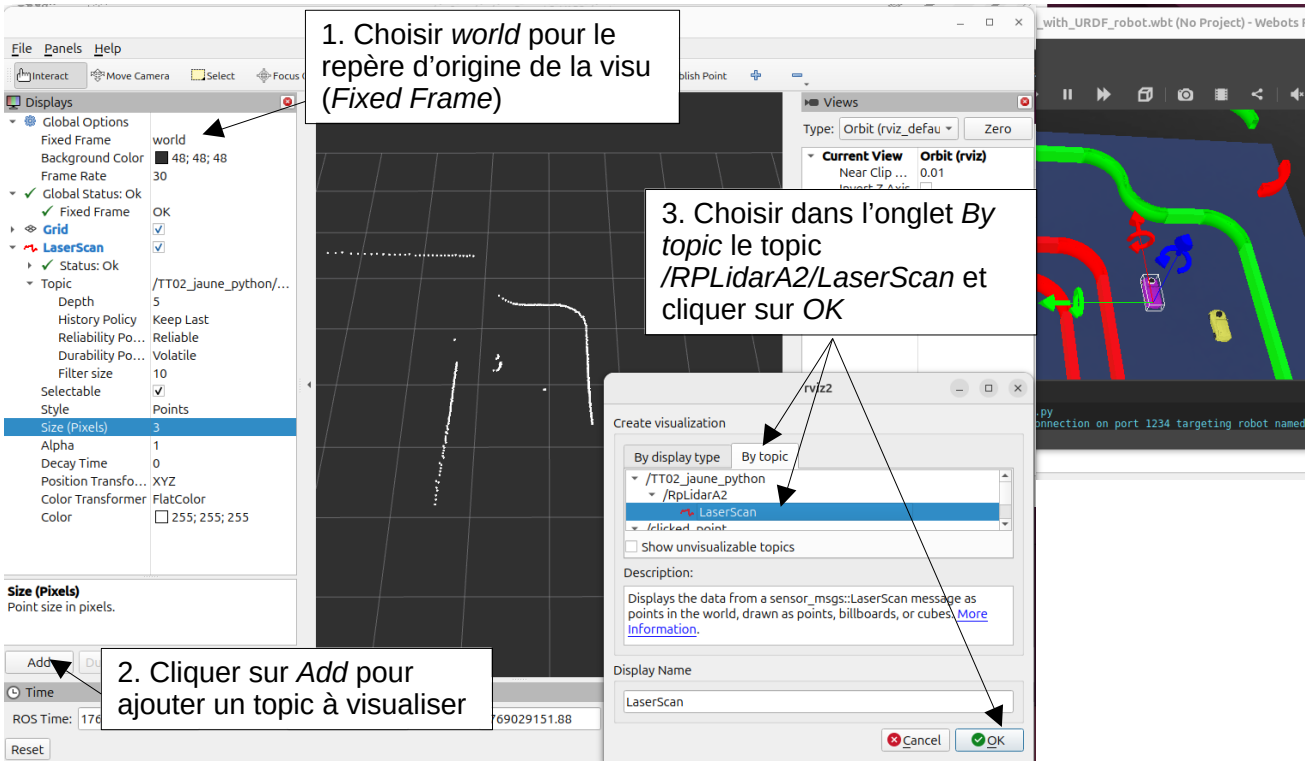


Figure 21 : Affichage des données du LiDAR de la voiture jaune simulée dans rviz2

2.5 - Création du nœud de conduite

La voiture publie les données du LiDAR dans le topic /TT02_jaune_python/RpLidarA2 et consomme (via le nœud CoVAPSy_cmd) les données du topic /cmd_ackermann pour ses consignes de vitesse et de direction.

Il est donc possible d'installer le même nœud que dans la voiture réelle qui, à partir des données du LiDAR élabore des consignes de vitesse et de direction.

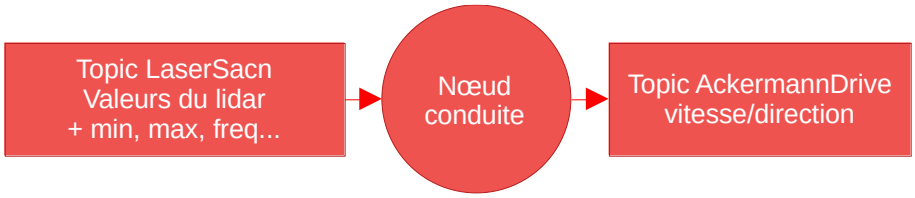


Figure 22 : Topics reçus et émis par le nœud conduite

Pour créer le nœud, créer un fichier *CoVAPSy_conduite.py* dans le dossier *ros2_ws/src/monPaquetCoVAPSy/monPaquetCoVAPSy* avec le code suivant, identique au nom des messages prêts à celui de la voiture réelle :

```
import rclpy
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDrive
from rclpy.node import Node

class CoVAPSy_conduite(Node):
    def __init__(self):
        super().__init__('CoVAPSy_conduite')

        # ROS interface
        self.__ackermann_publisher = self.create_publisher(AckermannDrive, 'cmd_ackermann',1)
        self.create_subscription(LaserScan, 'TT02_jaune_python/RpLidarA2', self.__on_lidar_acquisition,1)
        self.get_logger().info(f"[CoVAPSy_conduite] noeud cree")

    def __on_lidar_acquisition(self, message):
        tableauLidar = list(message.ranges)
        self.get_logger().info(f'60 {tableauLidar[120]:.2f} et -60 {tableauLidar[240]:.2f}')
        command_message = AckermannDrive()
```

```
        command_message.speed = 1.0
        try:
            command_message.steering_angle = 100 * (tableauLidar[120] - tableauLidar[240])
        except IndexError:
            command_message.steering_angle = 0.0
        if command_message.steering_angle > 18.0:
            command_message.steering_angle = 18.0
        if command_message.steering_angle < -18.0:
            command_message.steering_angle = -18.0
        self.__ackermann_publisher.publish(command_message)
        self.get_logger().info(f"v= {command_message.speed:.2f} m/s, dir= {command_message.steering_angle:.2f} rad")

def main(args=None):
    rclpy.init(args=args)
    controller = CoVAPSy_conduite()
    rclpy.spin(controller)
    rclpy.shutdown()

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

On ajoute alors le nœud au fichier *monPaquetCoVAPSy_launch.py* et au fichier setup.py. Il suffit d'ôter les commentaires de ces fichiers décrit dans la partie 2.3. Les fichiers sont aussi fournis en annexe de cette ressource [14].

On construit de nouveau le paquet et on lance le nœud.

```
webotsros2@WebotsROS2:~/ros2_ws$ colcon build --packages-select monPaquetCoVAPSy
WARNING: Package name "monPaquetCoVAPSy" does not follow the naming conventions. It should start with a lower case letter and only contain lower cas
e letters, digits, underscores, and dashes.
Starting >>> monPaquetCoVAPSy
Finished <<< monPaquetCoVAPSy [1.07s]

Summary: 1 package finished [1.20s]
webotsros2@WebotsROS2:~/ros2_ws$ source install/local_setup.bash
webotsros2@WebotsROS2:~/ros2_ws$ ros2 launch monPaquetCoVAPSy monPaquetCoVAPSy_launch.py
[INFO] [launch]: All log files can be found below /home/webotsros2/.ros/log/2026-01-21-22-05-45-469965-WebotsROS2-14597
[INFO] [launch]: Default logging verbosity is set to INFO
WARNING: No valid Webots directory specified in `ROS2_WEBOTS_HOME` and `WEBOTS_HOME`, fallback to default installation folder /usr/local/webots.
[INFO] [webots-1]: process started with pid [14600]
[INFO] [webots_controller_TT02_jaune_python-2]: process started with pid [14601]
[INFO] [CoVAPSy_conduite-3]: process started with pid [14602]
[webots_controller_TT02_jaune_python-2] The specified robot (at /tmp/webots/webotsros2/1234/ipc/TT02_jaune_python/extern) is not in the list of robo
ts with <extern> controllers, retrying for another 50 seconds...
[CoVAPSy_conduite-3] [INFO] [1769033145.912721664] [CoVAPSy_conduite]: [CoVAPSy_conduite] noeud cree
[webots_controller_TT02_jaune_python-2] The Webots simulation world is not yet ready, pending until loading is done...
[webots_controller_TT02_jaune_python-2] [INFO] [1769033154.606303463] [CoVAPSy_cmd]: noeud cree
[webots_controller_TT02_jaune_python-2] [INFO] [1769033154.606612488] [TT02_jaune_python]: Controller successfully connected to robot in Webots simu
lation.
[CoVAPSy_conduite-3] [INFO] [1769033156.285486404] [CoVAPSy_conduite]: 60 0.53 et -60 0.00
[CoVAPSy_conduite-3] [INFO] [1769033156.285831491] [CoVAPSy_conduite]: v = 1.00 m/s, dir = 18.00 rad
```

Il est possible bien sûr de limiter le bavardage du nœud en commentant les lignes *get_logger()…* dans les fichiers python.

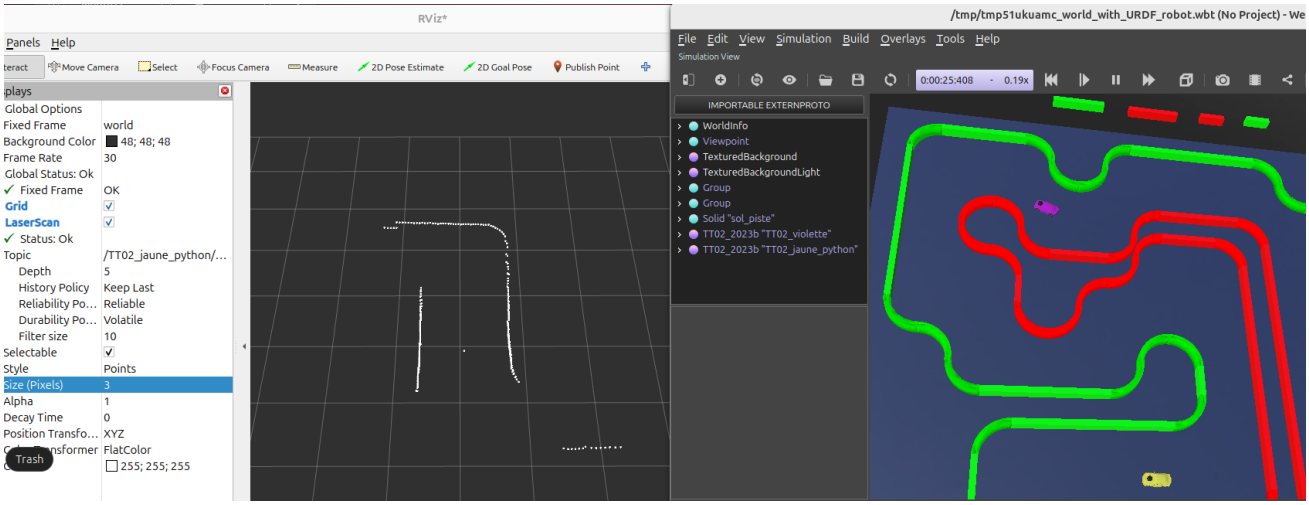


Figure 23 : Sur Webots, la Voiture jaune est contrôlée par le nœud ROS2 « CoVAPSy_conduite », la voiture violette est contrôlée par un algorithme basique pour représenter les voitures adverses

3 - Conclusion

Cette ressource permet de mettre en œuvre une conduite basique avec ROS2 sur la voiture réelle et sur le simulateur et également de manipuler quelques outils de monitoring ROS2 (*rviz2* et *rqt*). Aux étudiants de se l'approprier pour programmer une voiture performante et innovante.

Cette ressource est appelée à s'améliorer, ne pas hésiter à envoyer des commentaires (anthony.juton@ens-paris-saclay.fr).

Références :

[1]: ROS2 : bibliothèques et outils pour le développement logiciel en robotique, J. Farnault, S. Rodriguez, A. Juton, 2026, https://sti.eduscol.education.fr/si-ens-paris-saclay/ressources_pedagogiques/ros2-bibliotheques-outils-pour-developpement-logiciel-en-robotique

[2]: CoVAPSy : Mise en œuvre du simulateur Webots, T. Boulanger, E. Délégue , K. Hoarau, A. Juton, 2023, https://sti.eduscol.education.fr/si-ens-paris-saclay/ressources_pedagogiques/covapsy-mise-en-oeuvre-du-simulateur-webots

[3]: Tutoriel ROS2 pour webots https://docs.ros.org/en/jazzy/Tutorials/Advanced/Simulators/Webots/Simulation-Webots.html

[4]: Playlist *Webots ROS2 Tutorial* de la chaîne YouTube *Soft illusion* https://www.youtube.com/playlist?list=PLt69C9MnPchkP0ZXZOqmIGRTOch8o9GiQ

[5]: La documentation webots ROS2 https://github.com/cyberbotics/webots_ros2 avec notamment les types de messages envoyés par les différents capteurs. https://github.com/cyberbotics/webots_ros2/wiki/References-Nodes

[6]: Dépôt git du package « ROS2 node for SLAMTEC LiDAR » et les instructions associées : https://github.com/Slamtec/sllidar_ros2

[7]: Détails du format des messages LaserScan : https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/LaserScan.html

[8]: Apprentissage par renforcement et transfert simulation vers réalité pour la conduite de voitures autonomes, R. Bennani, K. Hoarau, A. Juton, 2024, https://sti.eduscol.education.fr/si-ens-paris-saclay/ressources_pedagogiques/apprentissage-renforcement-transfert-simulation-vers-realite-pourla-conduite-voitures-autonomes

[9]: Dépôt git de la course de voiture autonomes, dossier simulateur : https://github.com/ajuton-ens/CourseVoituresAutonomesSaclay/tree/main/Simulator

[10]: https://github.com/cyberbotics/webots_ros2/wiki/Example-Mavic-2-Pro

[11]: Dépôt git de la course de voitures autonomes : https://github.com/ajuton-ens/CourseVoituresAutonomesSaclay

[12]: Dépôt git de l'équipe de Sorbonne Université (ROS2 et SLAM) : https://github.com/SU-Bolides

[13]: Course de Voitures Autonomes Paris-Saclay (CoVAPSy) : Travaux pratiques autour des voitures autonomes, T. Boulanger, E. Délégue , K. Hoarau, A. Juton, 2023, https://sti.eduscol.education.fr/si-ens-paris-saclay/ressources_pedagogiques/covapsy-tp-autour-des-voitures-autonomes

[14]: Annexes de : Mise en œuvre de ROS2 pour le contrôle d'une voiture CoVAPSy simulée sous Webots et réelle, J. Farnault, S. Rodriguez, A. Juton, M. Goupillon, 2026, https://sti.eduscol.education.fr/si-ens-paris-saclay/ressources_pedagogiques/mise-en-oeuvre-ros2-pour-controle-voiture-autonome-1-10e

- Remarques sur l'installation de webots sur une machine virtuelle
- Configuration du wifi sur une machine ubuntu server (sans interface graphique)
- Fichiers simulateur
- Fichiers voiture