CoVAPSy : Premiers programmes python sur la voiture réelle

Culture Sciences de l'Ingénieur
La Revue 3E.I
école normale supérieure paris—saclay

Thomas BOULANGER[1] – Eve DÉLÈGUE[2]– Kévin HOARAU[3]
Anthony JUTON[4]

Édité le
xx/xx/2024

[1] *Élève en année de recherche pré-doctorale à l'étranger, ENS Paris-Saclay - DER Nikola Tesla,*
[2] *Élève en année de recherche en intelligence artificielle, ENS Paris-Saclay - DER Nikola Tesla,*
[3] *Élève en M2 Formation à l'Enseignement Supérieur, ENS Paris-Saclay - DER Nikola Tesla,*
[4] *Professeur agrégé de physique appliquée au DER Nikola Tesla, ENS Paris-Saclay*

Cette ressource fait partie du N°111 de La Revue 3EI de janvier 2024.

Faisant suite à la ressource d'introduction « Course Voitures Autonomes Paris Saclay (CoVAPSy) : Travaux pratiques autour des voitures autonomes » [1] et parallèle à la ressource de simulation « CoVaPSy : Mise en œuvre du Simulateur Webots »[2], l'objectif de cette ressource est de mener le lecteur au démarrage, à la configuration et à la programmation des premiers pas d'une voiture autonome réelle, la plus simple possible (CoVAPSy RPIonly utilise juste un nano-ordinateur raspberry Pi et un lidar).

Pour se faire, plusieurs signaux vont devoir être récupérés des capteurs et d'autres devront être générés afin d'activer les actionneurs. Les fonctions et tableaux utilisés pour cela sont les mêmes que sur le simulateur webots présenté dans la ressource « CoVaPSy : Mise en œuvre du Simulateur Webots »[2].

Les programmes de test sont disponibles sur le dépôt github :
https://github.com/ajuton-ens/CourseVoituresAutonomesSaclay/tree/main/Bibliotheques_logicielles/programmes_python_base_lidar_propulsion_direction_conduite

1 - Réalisation de la voiture

Une fois la voiture montée en suivant les instructions Tamiya, il faut lui ajouter l'électronique pour pouvoir la programmer.

1.1 - Montage mécanique de l'électronique

Pour fixer le nano-ordinateur raspberry Pi, le lidar et le convertisseur DC/DC, le plus simple est de réaliser une plaque découpée / percée au laser ou à la scie sauteuse / perceuse. Voici un exemple dont le plan est fourni en stp et dxf à l'adresse suivante : https://github.com/ajuton-ens/CourseVoituresAutonomesSaclay/tree/main/Materiel/Pieces_mecaniques_stp_dxf

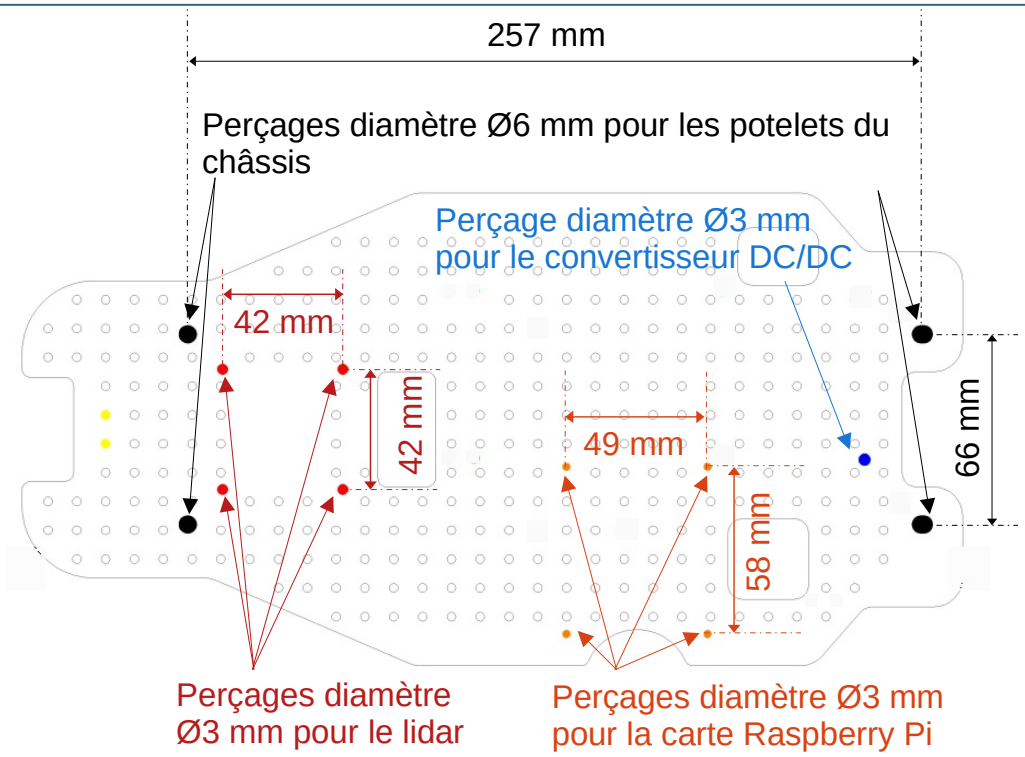


Figure 1 : Découpe et perçage pour le support du lidar et de l'électronique

La découpe de la carrosserie pour laisser passer le lidar peut se faire avec une mini-fraiseuse de type Dremel, ou avec une machine de découpe laser

Figure 2 : Découpe du passage du lidar au laser

1.2 - Câblage de l'électronique

Le câblage de l'électronique se fait par soudure pour la puissance et par soudure ou câbles de prototypage pour les signaux.

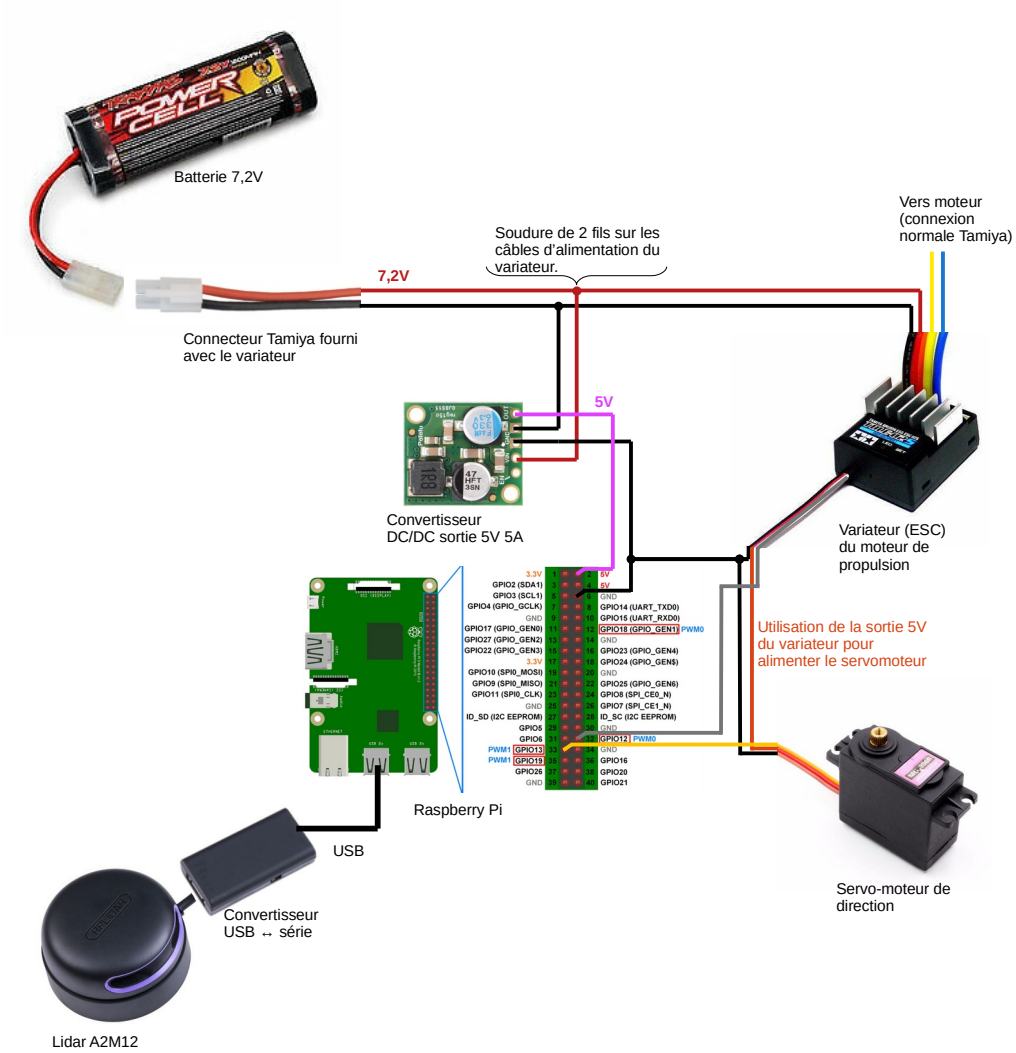


Figure 3 : Câblage des composants de la voiture CoVAPSY_RPIonly

2 - Installation de la carte Raspberry Pi

L'installation de Linux sur la carte Raspberry Pi 4 est expliquée dans le document *Installation_RaspberryPI4_minimum.pdf*.

La suite de cette ressource suppose que, comme expliqué dans le guide d'installation cité ci-dessus, les bibliothèques sont installées et que l'on se connecte via VNC au nano-ordinateur de la voiture.

L'environnement de développement utilisé est l'interpréteur python Thonny, fourni avec Raspberry OS.

3 - Mise en œuvre des entrées/sorties

3.1 - Réception des données du Lidar

La réception des données du Lidar se fait par une transmission série puis, après une adaptation série vers USB, par liaison USB.

Attention : La version A2M8 (noir et rouge) du RPLidar utilise une communication à 115200 baud et la version A2M12 (noir et violet) une communication à 256000 baud. Il faut donc adapter ce paramètre de la fonction RPLidar() au lidar utilisé.

Ce premier code (raz_lidar.py) permet de vérifier la communication avec le lidar, grâce à la fonction lidar.get_info() et de remettre le lidar à 0, avec une déconnexion propre si la communication a été coupée brutalement.

```
from rplidar import RPLidar
import time

lidar = RPLidar("/dev/ttyUSB0",baudrate=115200)
lidar.disconnect()
time.sleep(1)
lidar.connect()
try :
    print (lidar.get_info())
except :
    print("la communication ne s'est pas établie correctement")
lidar.start_motor()
time.sleep(1)
lidar.stop_motor()
lidar.stop()
time.sleep(1)
lidar.disconnect()
```

Le code suivant (test_lidar_v2.py) permet de stocker les données captées par le Lidar dans un tableau python de 360 cases, l'indice indiquant l'angle du lidar (0 pour l'avant de la voiture puis dans le sens trigonométrique) et la valeur indiquant la distance en mm.

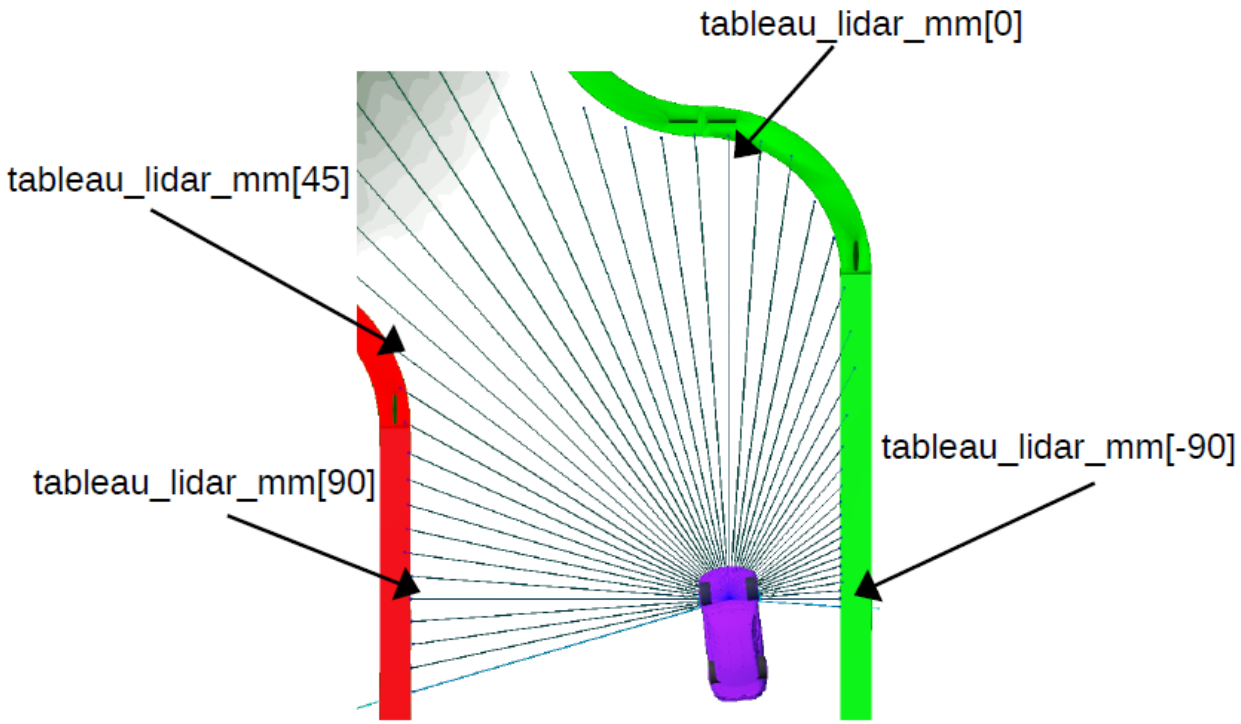


Figure 4 : Affichage sur le simulateur d'un rayon de mesure sur 4 du lidar

Le lidar capte tout autour de lui, mais les mesures vers l'arrière (entre 100 et 260° environ) sont perturbées par l'habitacle de la voiture

```
from rplidar import RPLidar
import numpy as np
import time
import matplotlib.pyplot as plt

#connexion et démarrage du lidar
lidar = RPLidar("/dev/ttyUSB0",baudrate=115200)
lidar.connect()
```

```
print (lidar.get_info())
lidar.start_motor()
time.sleep(1)

tableau_lidar_mm = [0]*360 #création d'un tableau de 360 zéros

try :
    for scan in lidar.iter_scans(scan_type='express') :
    #Le tableau se remplissant continuement, la boucle est infinie
        #affichage du nb de points récupérés lors du tour, pour les tests
        print("nb pts : " + str(len(scan)))
        #rangement des données dans le tableau
        for i in range(len(scan)) :
            angle = min(359,max(0,359-int(scan[i][1]))) #scan[i][1]:angle
            tableau_lidar_mm[angle]=scan[i][2]    #scan[i][2]:distance

except KeyboardInterrupt: #récupération du CTRL+C
    print("fin des acquisitions")

#arrêt et déconnexion du lidar
lidar.stop_motor()
lidar.stop()
time.sleep(1)
lidar.disconnect()

#######################################################
#affichage des données acquises sur l'environnement
#pour les tests
#######################################################

teta = [0]*360 #création d'un tableau de 360 zéros

for i in range(360) :
    teta[i]=i*np.pi/180

fig = plt.figure()
ax = plt.subplot(111, projection='polar')
line = ax.scatter(teta, tableau_lidar_mm, s=5)
line.set_array(tableau_lidar_mm)
ax.set_rmax(8000)
ax.grid(True)
plt.show()
```

Une fois l'acquisition lancée, Thonny affiche le nombre de points acquis par scan (moins que 360, ce qui signifie qu'il n'y a pas un point à chaque degré). CTRL+C permet de stopper l'acquisition et d'afficher le résultat sous forme d'un graphique Matplotlib.

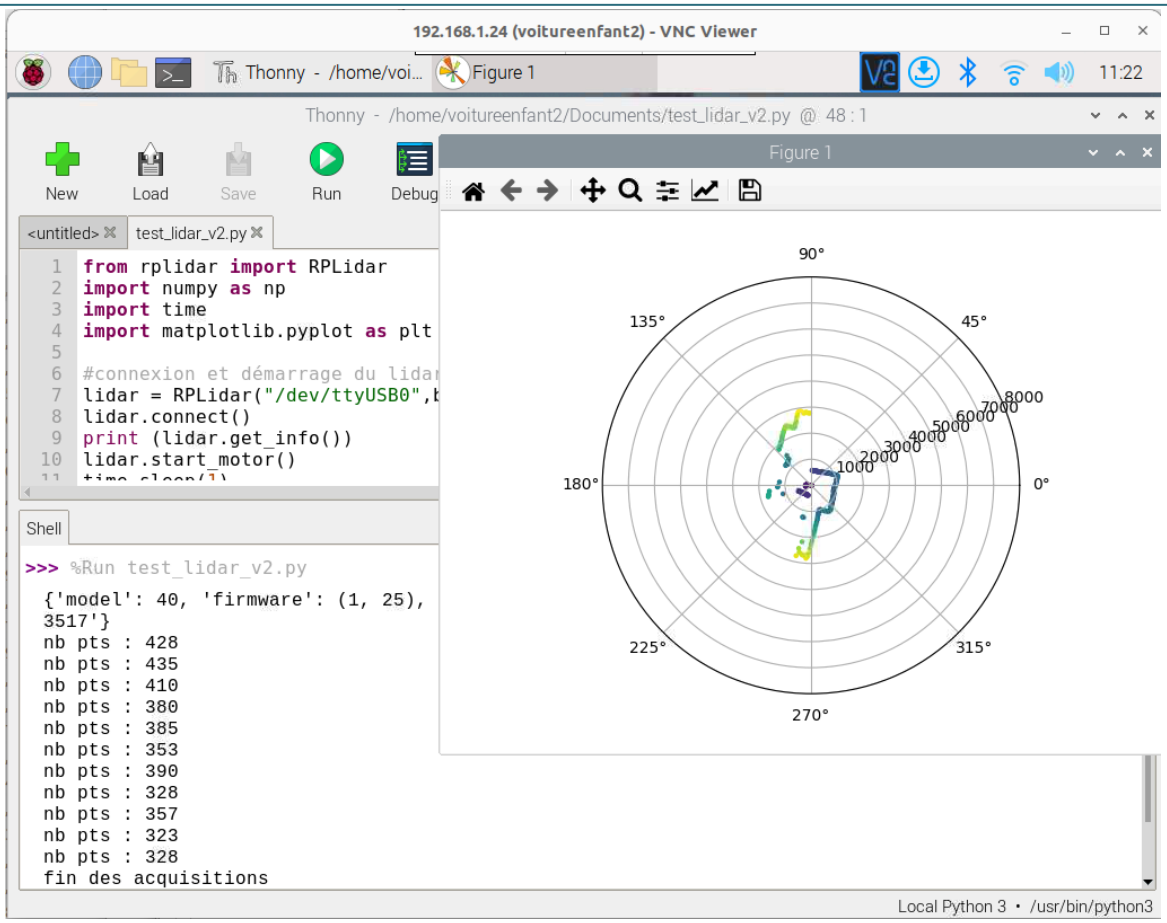
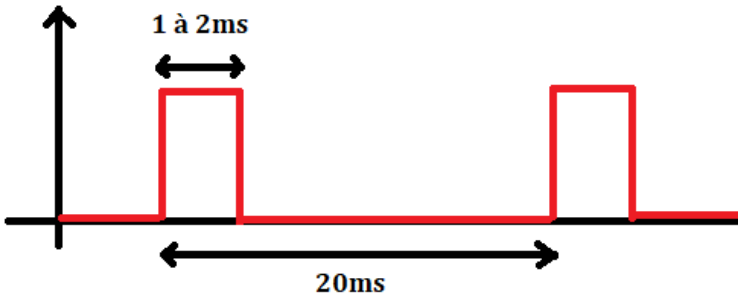


Figure 5 : Fenêtre VNC de la raspberry Pi après un arrêt de l'acquisition lidar et affichage des données

3.2 - Génération d'un signal de contrôle pour le servomoteur

Le servomoteur permet de contrôler la direction du véhicule. Le contrôle du servomoteur se fait par l'émission d'un signal PWM (pulse width modulation) à 50Hz avec un rapport cyclique variant entre 5% et 10%. Ce signal a la forme suivante :



- Un rapport cyclique de 5% sera équivalent à un état haut pendant 1ms, le servomoteur se positionnera alors tout à gauche (ou à droite suivant les modèles).
- Un rapport cyclique de 10% sera équivalent à un état haut pendant 2ms, le servomoteur se positionnera alors tout à droite (ou à gauche suivant les modèles).
- Un rapport cyclique de 7.5% sera équivalent à un état haut pendant 1.5ms, le servomoteur se positionnera alors au centre.

Attention : Le servomoteur étant relativement fragile, il est conseillé de rechercher progressivement les butées depuis la position centrale, d'autant plus que, suivant la marque de servomoteur installé, le sens de rotation est inversé...

Les sorties pwm de la Raspberry Pi n'ont parfois pas exactement la fréquence souhaitée, ce qui n'est pas gênant pour la période du signal mais demande d'ajuster la valeur centrale et les butées.

Voici donc un programme (test_pwm_direction.py) permettant de trouver les butées et le sens de rotation du servo-moteur pour compléter ensuite les paramètres de la fonction set_direction_degre().

```
from rpi_hardware_pwm import HardwarePWM
import time

#paramètres de départ, avec des butées très proche du centre
direction = 1 #1 pour angle_pwm_min a gauche, -1 pour angle_pwm_min à droite
angle_pwm_min = 6.7   #min
angle_pwm_max = 8.3   #max
angle_pwm_centre= 7.5

angle_degre_max = +18 #vers la gauche
angle_degre=0

pwm_dir = HardwarePWM(pwm_channel=1,hz=50)
pwm_dir.start(angle_pwm_centre)

def set_direction_degre(angle_degre) :
    global angle_pwm_min
    global angle_pwm_max
    global angle_pwm_centre
    angle_pwm = angle_pwm_centre + direction * (angle_pwm_max - angle_pwm_min) * angle_degre /(2 *
angle_degre_max )
    if angle_pwm > angle_pwm_max :
        angle_pwm = angle_pwm_max
    if angle_pwm < angle_pwm_min :
        angle_pwm = angle_pwm_min
    pwm_dir.change_duty_cycle(angle_pwm)

print("réglage des butées, Q pour quitter")
print("valeur numérique pour tester un angle de direction")
print("I pour inverser droite et gauche")
print("g pour diminuer la butée gauche et G pour l'augmenter")
print("d pour diminuer la butée droite et D pour l'augmenter")

while True :
    a = input("g, G, d, D ?")
    try :
        angle_degre=int(a)
        set_direction_degre(angle_degre)
    except :
        if a == "I" :
            direction = -direction
            print("nouvelle direction : " + str(direction))
        elif a == "g" :
            if direction == 1 :
                angle_pwm_max -=0.1
                print("nouvelle butée gauche : " + str(angle_pwm_max))
            else :
                angle_pwm_min +=0.1
                print("nouvelle butée gauche : " + str(angle_pwm_min))
            angle_pwm_centre = (angle_pwm_max+angle_pwm_min)/2
            set_direction_degre(18)
        elif a == "G" :
            if direction == 1 :
                angle_pwm_max +=0.1
                print("nouvelle butée gauche : " + str(angle_pwm_max))
```

```
            else :
                angle_pwm_min -=0.1
                print("nouvelle butée gauche : " + str(angle_pwm_min))
            angle_pwm_centre = (angle_pwm_max+angle_pwm_min)/2
            set_direction_degre(18)
        elif a == "d" :
            if direction == -1 :
                angle_pwm_max -=0.1
                print("nouvelle butée droite : " + str(angle_pwm_max))
            else :
                angle_pwm_min +=0.1
                print("nouvelle butée droite : " + str(angle_pwm_min))
            angle_pwm_centre = (angle_pwm_max+angle_pwm_min)/2
            set_direction_degre(-18)
        elif a == "D" :
            if direction == -1 :
                angle_pwm_max +=0.1
                print("nouvelle butée droite : " + str(angle_pwm_max))
            else :
                angle_pwm_min -=0.1
                print("nouvelle butée droite : " + str(angle_pwm_min))
            angle_pwm_centre = (angle_pwm_max+angle_pwm_min)/2
            set_direction_degre(-18)
        else :
            break

print("nouvelles valeurs")
print("direction : "        + str(direction))
print("angle_pwm_min : "    + str(angle_pwm_min))
print("angle_pwm_max : "    + str(angle_pwm_max))
print("angle_pwm_centre : " + str(angle_pwm_centre))
```

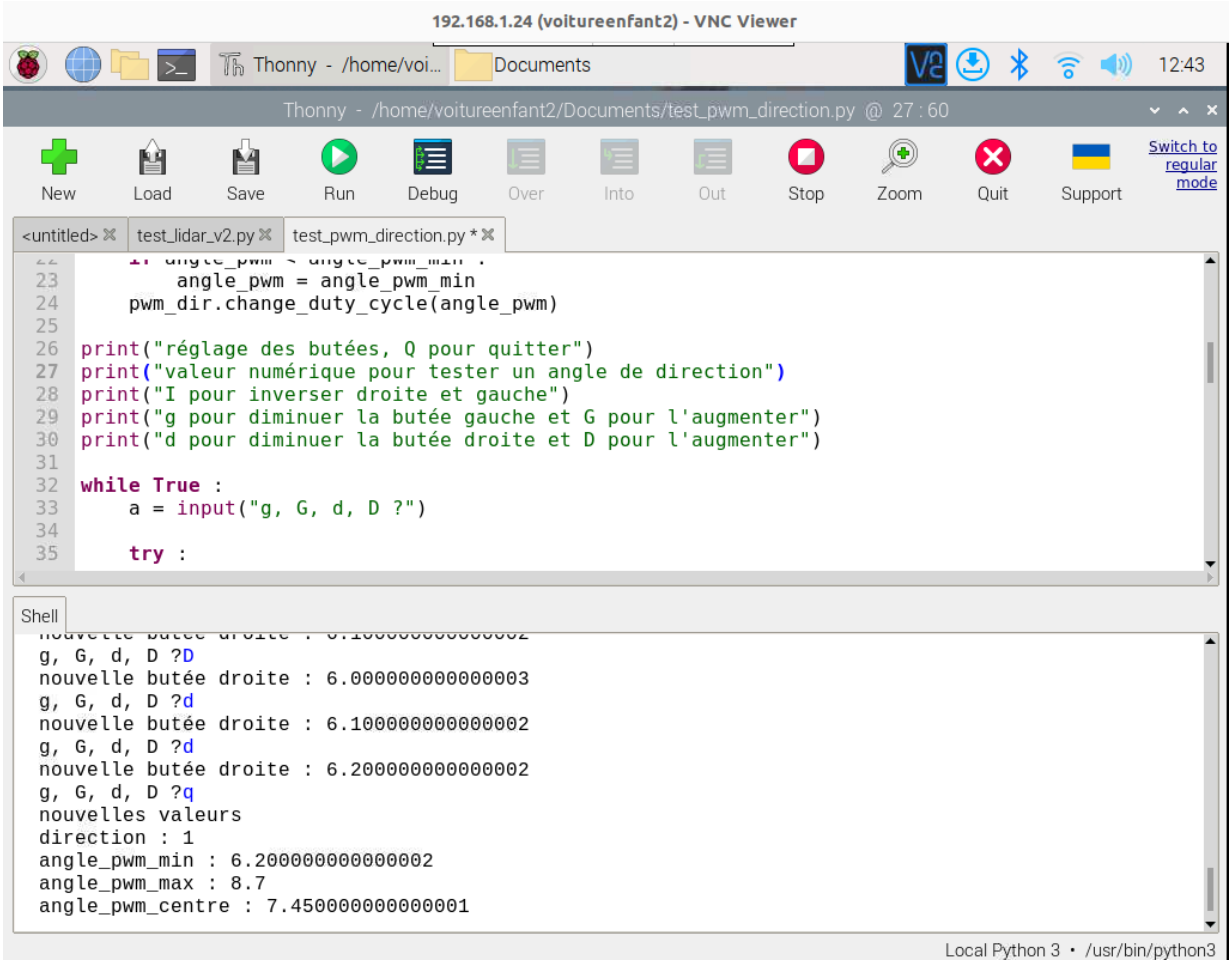


Figure 6 : Fenêtre VNC de la raspberry pi après un étalonnage des butées de la fonction set_direction

3.3 - Génération d'un signal de contrôle pour le variateur moteur

Le variateur de vitesse permet de contrôler la vitesse et la direction du moteur. Il se contrôle avec le même type de signal que le servomoteur de direction :

- Un rapport cyclique de 10% est équivalent à un état haut pendant 2 ms, le moteur sera à pleine vitesse en marche avant. Pour certains variateur, c'est un rapport cyclique à 5 % qui correspond à la pleine vitesse.
- Un rapport cyclique de 7.5% est équivalent à un état haut pendant 1.5 ms, le moteur est à l'arrêt.

Dans les faits, il y a un point mort autour de 7,5 % et la vitesse maximale est atteinte bien avant les 10 %. Il est donc important de trouver les butées pour avoir une commande de vitesse proche de la vitesse réelle, même si l'absence de mesure de la vitesse réelle empêche une bonne précision.

La marche arrière de la plupart des variateurs de voitures radiocommandés se fait en 2 temps :

- d'abord un freinage, rapport cyclique minimum (1 ms → 5%) pendant 0,3 seconde,
- un passage par le point mort (1,5 ms → 7,5%) pendant 0,3 seconde,
- la marche arrière à vitesse réglable (entre 1,2 et 1,4 ms → 6 à 7%)

Le code suivant (test_pwm_propulsion.py) permet de tester les fonctions set_vitesse_m_s() et recule() de la voiture et d'étalonner les butées, à l'image du code similaire test_pwm_direction.py

```
from rpi_hardware_pwm import HardwarePWM
import time

#paramètres de la fonction vitesse_m_s, à étalonner
direction_prop = -1 # -1 pour les variateurs inversés
pwm_stop_prop = 8.17
point_mort_prop = 0.13
delta_pwm_max_prop = 1.5 #pwm à laquelle on atteint la vitesse maximale

vitesse_max_m_s_hard = 8 #vitesse que peut atteindre la voiture
vitesse_max_m_s_soft = 2 #vitesse maximale que l'on souhaite atteindre

pwm_prop = HardwarePWM(pwm_channel=0, hz=50)
pwm_prop.start(pwm_stop_prop)

def set_vitesse_m_s(vitesse_m_s):
    if vitesse_m_s > vitesse_max_m_s_soft :
        vitesse_m_s = vitesse_max_m_s_soft
    elif vitesse_m_s < -vitesse_max_m_s_hard :
        vitesse_m_s = -vitesse_max_m_s_hard
    if vitesse_m_s == 0 :
        pwm_prop.change_duty_cycle(pwm_stop_prop)
    elif vitesse_m_s > 0 :
        vitesse = vitesse_m_s * (delta_pwm_max_prop)/vitesse_max_m_s_hard
        pwm_prop.change_duty_cycle(pwm_stop_prop + direction_prop*(point_mort_prop + vitesse ))
    elif vitesse_m_s < 0 :
        vitesse = vitesse_m_s * (delta_pwm_max_prop)/vitesse_max_m_s_hard
        pwm_prop.change_duty_cycle(pwm_stop_prop - direction_prop*(point_mort_prop - vitesse ))

def recule():
    set_vitesse_m_s(-vitesse_max_m_s_hard)
    time.sleep(0.2)
    set_vitesse_m_s(0)
    time.sleep(0.2)
    set_vitesse_m_s(-1)

print("réglage des butées, Q pour quitter")
print("valeur numérique pour tester une vitesse en mm/s")
print("R pour reculer")
print("I pour inverser droite et gauche")
print("p pour diminuer delta_pwm_max_prop et P pour l'augmenter")
print("z pour diminuer le point zéro 1,5 ms et Z pour l'augmenter")
print("m pour diminuer le point mort et M pour l'augmenter")


while True :
```

```
    a = input("vitesse en mm/s, R, I, p, P, z, Z, m, M")
    try :
        vitesse_mm_s=int(a)
        set_vitesse_m_s(vitesse_mm_s/1000.0)
    except :
        if a == "I" or a == "i" :
            direction_prop = -direction_prop
            print("nouvelle direction : " + str(direction_prop))
        elif a == "R" :
            recule()
            print("recule")
        elif a == "p" :
            delta_pwm_max_prop -=0.1
            print("nouveau delta_pwm_max_prop : " + str(delta_pwm_max_prop))
            pwm_prop.change_duty_cycle(pwm_stop_prop+direction_prop*(point_mort_prop+\
            delta_pwm_max_prop))
        elif a == "P" :
            delta_pwm_max_prop +=0.1
            print("nouveau delta_pwm_max_prop : " + str(delta_pwm_max_prop))
            pwm_prop.change_duty_cycle(pwm_stop_prop + direction_prop*(point_mort_prop+\
            delta_pwm_max_prop))
        elif a == "z" :
            pwm_stop_prop -=0.01
            print("nouveau pwm_stop_prop : " + str(pwm_stop_prop))
            pwm_prop.change_duty_cycle(pwm_stop_prop)
        elif a == "Z" :
            pwm_stop_prop +=0.01
            print("nouveau pwm_stop_prop : " + str(pwm_stop_prop))
            pwm_prop.change_duty_cycle(pwm_stop_prop)
        elif a == "m" :
            point_mort_prop -=0.01
            print("nouveau point_mort_prop : " + str(point_mort_prop))
            pwm_prop.change_duty_cycle(pwm_stop_prop + direction_prop*(point_mort_prop))
        elif a == "M" :
            point_mort_prop +=0.01
            print("nouveau point_mort_prop : " + str(point_mort_prop))
            pwm_prop.change_duty_cycle(pwm_stop_prop + direction_prop*(point_mort_prop))
        else :
            break

pwm_prop.change_duty_cycle(pwm_stop_prop)
print("nouvelles valeurs")
print("direction : "         +      str(direction_prop))
print("delta_pwm_max_prop : "    + str(delta_pwm_max_prop))
print("point zero 1,5 ms : "     + str(pwm_stop_prop))
print("point mort : "            + str(point_mort_prop))
```

4 - Codage d’un suivi de ligne simple

4.1 - Code de base

Comme dans le simulateur, les données du lidar sont acquises dans un tableau **tableau_lidar_mm[360]** et la commande de la direction et de la propulsion se fait via les fonctions **set_direction_degre()** et **set_vitesse_m_s()**.

On peut importer la zone de code correspondant à l’algorithme de conduite du simulateur.

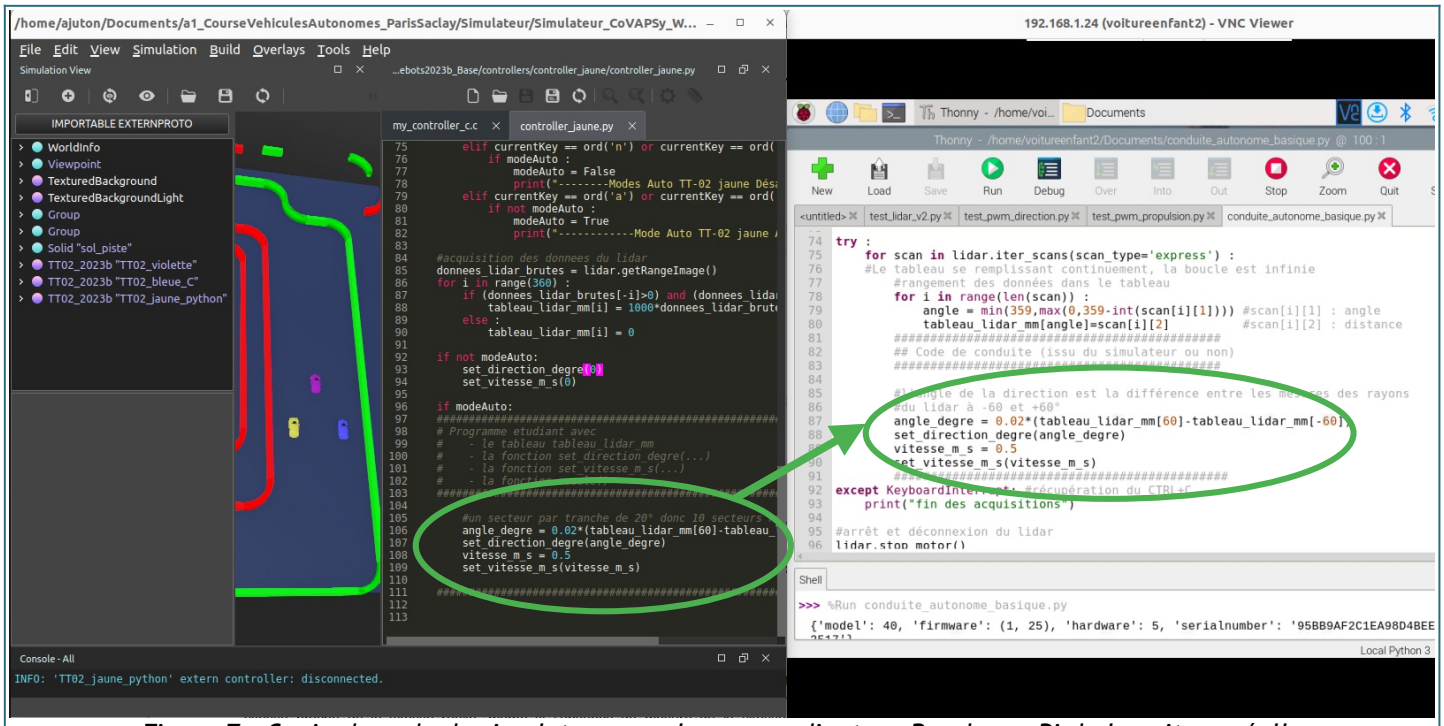


Figure 7 : Copie du code du simulateur vers le nano-ordinateur Raspberry Pi de la voiture réelle

Le code de base fourni (conduite_autonome_basique.py) est le suivant :

```
from rplidar import RPLidar
import time
from rpi_hardware_pwm import HardwarePWM

#paramètres de la fonction vitesse_m_s, issus des essais précédents
direction_prop = -1 # -1 pour les variateurs inversés
pwm_stop_prop = 8.17
point_mort_prop = 0.13
delta_pwm_max_prop = 1.5 #pwm à laquelle on atteint la vitesse maximale

vitesse_max_m_s_hard = 8 #vitesse que peut atteindre la voiture
vitesse_max_m_s_soft = 2 #vitesse maximale que l'on souhaite atteindre


#paramètres de la fonction set_direction_degre, issus des essais précédents
direction = 1 #1 pour angle_pwm_min a gauche, -1 pour angle_pwm_min à droite
angle_pwm_min = 6   #min
angle_pwm_max = 9   #max
angle_pwm_centre= 7.5

angle_degre_max = +18 #vers la gauche
angle_degre=0

pwm_prop = HardwarePWM(pwm_channel=0, hz=50)
pwm_prop.start(pwm_stop_prop)

def set_vitesse_m_s(vitesse_m_s):
    # cf définition de la fonction précédemment


def recule():
    # cf définition de la fonction précédemment

pwm_dir = HardwarePWM(pwm_channel=1,hz=50)
pwm_dir.start(angle_pwm_centre)

def set_direction_degre(angle_degre) :
    # cf définition de la fonction précédemment

#connexion et démarrage du lidar
lidar = RPLidar("/dev/ttyUSB0",baudrate=115200)
lidar.connect()
print (lidar.get_info())
lidar.start_motor()
time.sleep(1)

tableau_lidar_mm = [0]*360 #création d'un tableau de 360 zéros
```

```
try :
    for scan in lidar.iter_scans(scan_type='express') :
    #Le tableau se remplissant continuement, la boucle est infinie
        #rangement des données dans le tableau
        for i in range(len(scan)) :
            angle = min(359,max(0,359-int(scan[i][1]))) #scan[i][1] : angle
            tableau_lidar_mm[angle]=scan[i][2]  #scan[i][2] : distance
        ################################################
        ## Code de conduite (issu du simulateur ou non)
        ################################################

        #l'angle de la direction est la différence entre les mesures
        #des rayons du lidar à -60 et +60°
        angle_degre = 0.02*(tableau_lidar_mm[60]-tableau_lidar_mm[-60])
        set_direction_degre(angle_degre)
        vitesse_m_s = 0.5
        set_vitesse_m_s(vitesse_m_s)
        ################################################
except KeyboardInterrupt: #récupération du CTRL+C
    print("fin des acquisitions")

#arrêt et déconnexion du lidar et des moteurs
lidar.stop_motor()
lidar.stop()
time.sleep(1)
lidar.disconnect()
pwm_dir.stop()
pwm_prop.start(pwm_stop_prop)
```

4.2 - Améliorations possibles

On donne ici les mêmes pistes que pour l'amélioration de l'algorithme du simulateur :

- Il est possible de regrouper les rayons en secteur de 10° pour chercher le secteur dont le rayon le plus court est le plus long parmi les plus courts des autres secteurs.
- Il est possible d'adapter sa vitesse à la distance de l'obstacle devant.
- Il est intéressant de détecter un obstacle pour réussir à l'éviter. On peut pour cela ajouter des morceaux de bordure de piste au milieu de la piste.
- A la différence de celui du simulateur, le lidar réel fait moins de 250 mesures par tour, donc il ne met pas à jour chaque valeur du tableau. Des valeurs moins « fraîches » restent dans le tableau. Il pourrait être intéressant de vider le tableau avant sa mise à jour et de compléter les valeurs manquantes par une estimation.
- Il est possible de reculer quand on est dans un mur, en surveillant une valeur minimale des rayons du lidar à l'avant et sur les côtés. Pour déterminer les situations de quasi-collision, on peut se baser sur 3 valeurs du Lidar : la mesure à 0°, celle à -30° et celle à 30°. Si les distances captées par le Lidar sur ces angles spécifiques sont inférieures à un certain seuil, on considère que la voiture s'est crashée. On a alors 3 possibilités, qui demande un peu de travail car il n'est pas possible d'utiliser time.sleep, cette fonction bloquante mettant aussi en pause le moteur physique (l'utilisation de time.time() peut alors être intéressante) :
    - Seuil franchi pour l'angle 0° (mur devant) : On replace les roues pour aller droit puis on recule jusqu'à ce que la valeur de lidar franchisse un seuil ou pendant 0,5s.
    - Seuil franchi pour l'angle 30° (mur à gauche) : On tourne complètement à gauche puis on recule jusqu'à ce que la valeur de lidar franchisse un seuil ou pendant 0,5s.
    - Seuil franchi pour l'angle -30° (mur à droite) : On tourne complètement à droite puis on recule jusqu'à ce que la valeur de lidar franchisse un seuil ou pendant 0,5s.
- Des méthodes avancées sont bien évidemment possible, en sortant du cadre du lycée, avec des trajectoires en forme de tentacules (https://doi.org/10.1002/rob.20256), ou avec de l'apprentissage par renforcement (voir « Apprentissage par renforcement et transfert simulation vers réalité pour la conduite de voitures autonomes » [3]).

L'utilisation de la fonction de recul ou d'une temporisation un peu longue n'est pas possible dans la boucle de scan, en effet, pendant la temporisation, le buffer du lidar se remplit. Il faut donc, comme sur le simulateur utiliser une machine à état avec la lecture du temps (time.time()) non bloquante.

Une autre solution consiste à utiliser les threads de python, comme dans le code suivant, avec un thread pour l'acquisition lidar et un thread pour la conduite autonome.

```
from rplidar import RPLidar
import numpy as np
import time
from rpi_hardware_pwm import HardwarePWM
import threading

#paramètres fonction vitesse_m_s, étalonnés avec test_pwm_propulsion.py
stop_prop = 7.5
point_mort_prop = 0.5
pwm_max = 9 #pwm à laquelle on atteint la vitesse maximale

vitesse_max_m_s_hard = 8 #vitesse que peut atteindre la voiture
vitesse_max_m_s_soft = 2 #vitesse maximale que l'on souhaite atteindre

#paramètres fonction set_direction_m_s, étalonnés via test_pwm_direction.py
direction = 1 #1 pour angle_pwm_min a gauche, -1 sinon
angle_pwm_min = 6.2   #min
angle_pwm_max = 8.7   #max
angle_pwm_centre= 7.4

angle_degre_max = +18 #vers la gauche
angle_degre=0

pwm_prop = HardwarePWM(pwm_channel=0, hz=50)
pwm_prop.start(stop_prop)

def set_vitesse_m_s(vitesse_m_s):
    # cf définition de la fonction précédemment

def recule():
    # cf définition de la fonction précédemment

pwm_dir = HardwarePWM(pwm_channel=1,hz=50)
pwm_dir.start(angle_pwm_centre)

def set_direction_degre(angle_degre) :
    # cf définition de la fonction précédemment

acqui_tableau_lidar_mm = [0]*360 #création d'un tableau de 360 zéros
tableau_lidar_mm = [0]*360
drapeau_nouveau_scan = False
Run_Lidar = False

def lidar_scan() :
    global drapeau_nouveau_scan
    global acqui_tableau_lidar_mm
    global Run_Lidar
    global lidar
    print ("tâche lidar_scan démarrée")
    while Run_Lidar == True :
        try :
            for scan in lidar.iter_scans(scan_type='express') :
            #Le tableau se remplissant continuement, la boucle est infinie
                #rangement des données dans le tableau
                for i in range(len(scan)) :
                    angle = min(359,max(0,359-int(scan[i][1]))) #scan[i][1]:angle
                    acqui_tableau_lidar_mm[angle]=scan[i][2]    #scan[i][2]:distance
                drapeau_nouveau_scan = True
                time.sleep(0.01)
                if(Run_Lidar == False) :
                    break
        except :
            print("souci acquisition lidar")

def conduite_autonome():
    global drapeau_nouveau_scan
    global acqui_tableau_lidar_mm
    global tableau_lidar_mm
    global Run_Lidar
    print ("tâche conduite autonome démarrée")
```

```
    while Run_Lidar == True :
        if(drapeau_nouveau_scan == False) :
            time.sleep(0.01)
        else :
            #récupération du tableau_lidar acquis par l'autre thread
            for i in range(-100,101) :
                tableau_lidar_mm[i] = acqui_tableau_lidar_mm[i]
            drapeau_nouveau_scan = False

            ######################################################
            # programme de conduite avec détection des murs et marche arrière
            ######################################################

            if tableau_lidar_mm[0]>0 and tableau_lidar_mm[0]<150:
                print("mur devant")
                set_direction_degre(0)
                recule()
                time.sleep(0.5)


            elif tableau_lidar_mm[-30]>0 and tableau_lidar_mm[-30]<150 :
                print("mur à droite")
                set_direction_degre(-18)
                recule()
                time.sleep(0.5)

            elif tableau_lidar_mm[30]>0 and tableau_lidar_mm[30]<150 :
                print("mur à gauche")
                set_direction_degre(+18)
                recule()
                time.sleep(0.5)

            else :
                #l'angle de la direction est la différence entre les mesures des rayons
                #du lidar à -60 et +60°
                angle_degre = 0.02*(tableau_lidar_mm[60]-tableau_lidar_mm[-60])
                set_direction_degre(angle_degre)
                vitesse_m_s = 0.5
                set_vitesse_m_s(vitesse_m_s)

            ######################################################


#connexion et démarrage du lidar
lidar = RPLidar("/dev/ttyUSB0",baudrate=115200)
lidar.connect()
print (lidar.get_info())
lidar.start_motor()
time.sleep(2)

#démarrage des 2 thread
Run_Lidar = True
thread_scan_lidar = threading.Thread(target= lidar_scan)
thread_scan_lidar.start()
time.sleep(1)
thread_conduite_autonome = threading.Thread(target = conduite_autonome)
thread_conduite_autonome.start()

while True :
    try :
        pass
    except KeyboardInterrupt: #récupération du CTRL+C
        print("arrêt du programme")
        Run_Lidar = False
        break

#attente de l'arrêt des tâches
thread_conduite_autonome.join()
thread_scan_lidar.join()

#arrêt et déconnexion du lidar
lidar.stop_motor()
lidar.stop()
time.sleep(1)
lidar.disconnect()
pwm_prop.stop()
pwm_dir.stop()
```

5 - Ouvertures

La programmation de la voiture 1/10ème avec seulement un nano-ordinateur Raspberry Pi et un lidar permet de manipuler un capteur avancé et la raspberry pi en programmation python, avec des tableaux et des fonctions. En programmation, avec du multi-tâche, du travail simulation / réalité, de l'apprentissage… le champ des possibles est immense.

Côté systèmes embarqués, profitant de la connectivité i2c de la raspberry pi, Il est possible d'ajouter des capteurs : ultrason (SRF10) à l'arrière, une centrale intertielle (BNO055) pour mesurer l'orientation ou les chocs, un afficheur OLED (TF051). Il est possible d'utiliser le bluetooth pour ajouter une manette de playstation, pour faire une course entre voiture autonome et voiture commandée, etc.

Enfin, côté asservissement, il est possible de mettre en place un correcteur PID ou des correcteurs plus avancés pour le contrôle de la voiture. En monovariable, la direction est calculée à partir de la différence entre les distances mesurées par le lidar à +60 et -60°.

Des exemples sont fournis sur le dépôt gît [4] de la course et les nouvelles contributions sont les bienvenues.

Références :

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

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

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

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

Ressource publiée sur Culture Sciences de l'Ingénieur : https://eduscol.education.fr/sti/si-ens-paris-saclay