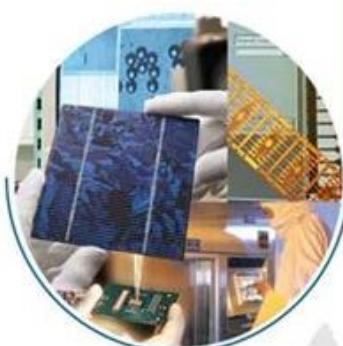
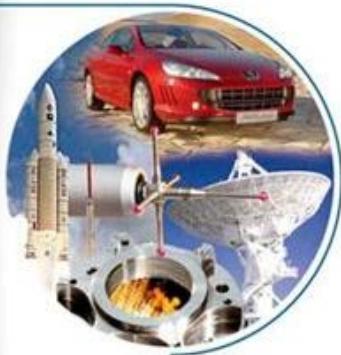


SLAM par TurtleBot

Compte-rendu Projet

CHAUVREAU Nicolas
DUPONT Cédric
KAMBOU TCHAKAM Danielle Lyne
REMUSATI Héloïse
THOMAS Loïc

SMR
2020-2021



SOMMAIRE

I)	TurtleBot	5
1)	Objectifs	5
2)	Présentation de l'hardware du robot	5
3)	Travail réalisé	6
A)	Installation de ROS	6
(i)	Qu'est-ce que ROS ?	6
(a)	Présentation générale	6
(b)	Architecture	7
(c)	ROS2 Graph Structure	9
(ii)	Fonctionnement du robot	10
(iii)	Mise en place de l'installation	11
B)	Utilisation basique de ROS	12
C)	Première automatisation	12
D)	Traitement de la carte et algorithme A*	15
E)	Automatisation complète de la cartographie	16
(i)	Principe de l'algorithme	16
(ii)	Détails du script	18
(iii)	Limites	23
II)	Second robot	24
1)	Hardware	24
2)	Asservissement des moteurs	25
A)	Logique de programmation	26
B)	Expérience et résultats	27
3)	Suivi de trajectoire	28
A)	Définition du problème	28
B)	Logique de programmation	28
C)	Résultats	30
4)	Interprétation des ordres de la base de données	31
5)	Boucle finale	32
III)	Communication	33
1)	Objectifs	33
2)	Démarches	33
A)	Recherche préliminaire	33
B)	La base de données	34
C)	Communication SFTP	34
3)	Difficultés	34
IV)	Interface utilisateur	35
1)	Objectifs	35
2)	Démarches et résultats	35
A)	Commande	35
B)	Carte	36
C)	Retour et notification	37

3) Difficultés	37
 Annexes	38
Annexe 1: Cahier des charges	38
Annexe 2: savemap_on_turtlebot.py	39
Annexe 3: dll_ssh.py	40
Annexe 4: Algorithme A*	40
Annexe 5: Calcul des points proches des zones à explorer	43
Annexe 6: Envoi du robot dans les zones à explorer	44
Annexe 7: Exemple de dimensions de la plateforme	44
Annexe 8: Code du codeur fournissant la vitesse	45
Annexe 9: Code du gyroscope	46
Annexe 10: Boucle infinie	47
Annexe 11: Forme de la base de données	48
Annexe 12: Exemple de connexion SSH Python et Java	49
Annexe 13: Interface utilisateur complète	50
Annexe 14: Description de la partie commande	50
Annexe 15: Format « pgm »	51
Annexe 16: Exemple de connexion SQL sous Java	52
Annexe 17: Programme pour afficher un format pgm dans un label	52

INTRODUCTION

Le projet « SLAM par TurtleBot » se compose de deux phases. D'une part, nous devons cartographier une pièce avec un robot TurtleBot et la mettre à disposition de l'utilisateur. D'autre part, nous désirons ordonner un déplacement d'un autre robot sur celle-ci. Pour cela, nous aurons aussi besoin d'une interface utilisateur simplifiant les commandes.

L'équipe de projet est constituée de :

- CHAUVREAU Nicolas, chef de projet et aidant sur le second robot et la communication entre les dispositifs ;
- Cédric DUPONT et Héloïse REMUSATI, travaillant sur le TurtleBot et l'algorithme de planification de chemin ;
- Danielle Lyne KAMBOU TCHAKAM, travaillant sur le second robot ;
- Loïc THOMAS, réalisant l'interface utilisateur et la communication entre les dispositifs.

Pour répondre au cahier des charges (cf. annexe 1), le développement est divisé en plusieurs étapes :

- Pour le TurtleBot, réalisation d'une carte immobile, puis en étant contrôlé par un utilisateur et enfin sans aide extérieure ;
- Pour le second robot, choix des pièces, conception de la plateforme, assemblage, asservissement des moteurs, planification de trajectoire et suivi de trajectoire ;
- Pour la communication, mise en place d'un protocole FTP, d'une base de données et des méthodes de récupérations de données ;
- Pour l'interface utilisateur, conception d'une première interface et modification de celle-ci tout au long du projet pour répondre aux attentes.

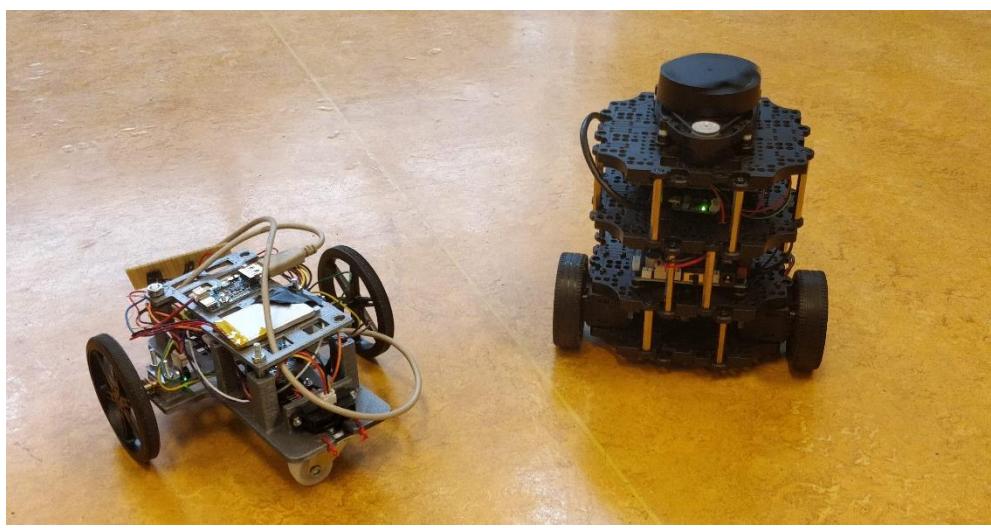


Figure 1 : Photo des deux robots

I - TurtleBot

Le TurtleBot est le robot principal du projet dont le but est de cartographier une pièce de manière autonome.

1. Objectifs

Comme énoncé précédemment, le but principal de cette partie du projet est d'automatiser complètement le processus de cartographie d'une pièce pour ensuite mettre à disposition de l'utilisateur la carte établie.

Dans la mesure où nous utilisons un robot vendu en kit par Robotis (celui-ci sera présenté dans une partie ultérieure), qui a déjà été utilisé dans un autre projet les années précédentes, nous n'avons donc pas eu à nous soucier de la partie hardware du robot. Ainsi, la majeure partie du travail concerne la programmation du robot.

Cependant, cette mission est assez dense et surtout nécessite plusieurs étapes pour être accomplie. Ainsi, pour ce faire, nous avons partitionné ce travail en plusieurs jalons :

- La première étape est de réaliser une carte manuellement, soit d'exploiter directement le robot cartographieur TurtleBot3 Burger.
- Le deuxième objectif est de réaliser une cartographie interactive, c'est-à-dire de commencer à automatiser la cartographie du robot en rendant autonome le lancement du robot et en facilitant la cartographie manuelle du robot ainsi que la récupération de la carte réalisée.
- La troisième étape consiste à traiter la carte réalisée par notre robot afin de la rendre exploitable pour les seconds robots.
- Le quatrième objectif est de mettre en place un algorithme de pathfinding pour permettre aux robots d'optimiser leur chemin entre deux points sur la carte.
- Enfin, l'objectif optimal est d'automatiser complètement la cartographie en réalisant un script Python capable d'assister au maximum l'utilisateur.

2. Présentation de l'hardware du robot

Afin de pouvoir répondre à nos objectifs, nous avions besoin d'un robot mobile avec un LIDAR pour pouvoir faire de la cartographie. Mr KANTY RABENOROSOA nous a donc conseillé de prendre le robot d'un ancien projet. Nous avons donc récupéré le Turtlebot3 Burger déjà monté. Ainsi, nous n'avions pas à nous occuper de la partie Hardware (dans un premier temps).

Rapidement, le robot est composé de :

- Un RPI avec les packages ROS2 installé dessus.
- Un microcontrôleur OpenCR

- Un LIDAR (télédétection par laser) qui permet d'avoir des mesures précises de distance.

TurtleBot3 Burger

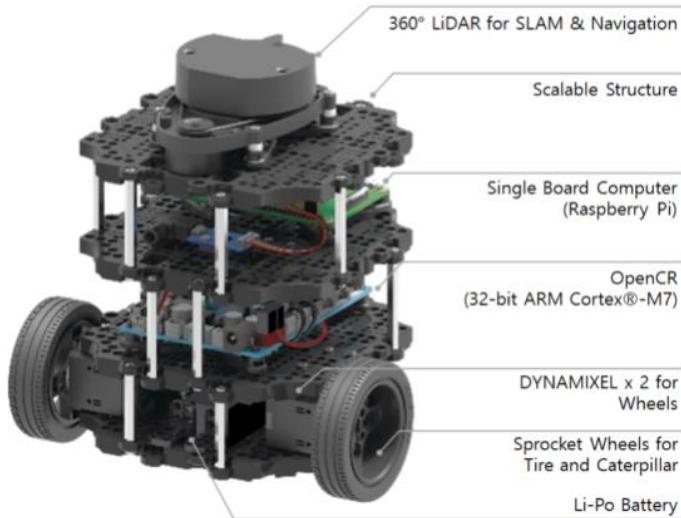


Figure 2 : Composants du TurtleBot3 Burger

3. Travail réalisé

Maintenant que les objectifs ont été fixés et que nous avons présenté le robot sur lequel nous avons travaillé, nous allons vous présenter en détail la démarche que nous avons menée pour mener à bien cette partie du projet.

A. Installation de ROS

Le robot que nous avons récupéré est initialement en kit. Toutefois, il a déjà été monté par un groupe de projet des années précédentes. Ce qui nous restait à faire et qui a donc été la première étape de notre travail était donc d'installer tous les softwares nécessaires au fonctionnement du robot et l'environnement sur lequel le robot fonctionne : ROS2.

Nous allons ainsi commencer par vous présenter ROS.

Nous souhaitons préciser que nous ne prétendons pas donner une présentation exhaustive de ROS2, mais nous souhaitons juste donner une idée du fonctionnement de cet environnement.

i. Qu'est-ce que ROS ?

a. Présentation générale

Tout d'abord, ROS signifie Robot Operating System. C'est un framework flexible destiné à la programmation de robot. En fait, c'est un ensemble de bibliothèques logicielles et d'outils

open source qui permettent de faciliter l'écriture d'applications robotiques, notamment destinées pour la navigation et la cartographie.

Bien plus qu'un système d'exploitation, l'univers ROS comprend :

- o Un système de communication ;
- o Un cadre et des outils ;
- o Un écosystème entier.

En fait, ROS est un environnement très complet.

Actuellement, il existe deux versions de cet environnement : ROS1 et ROS2. La première version est plutôt utilisée dans le cadre de projets académiques alors que la deuxième version, plus récente, vise, quant à elle, les projets commerciaux. Leur fonctionnement est, par ailleurs, différent. Pour des raisons d'installation de systèmes d'exploitation (Ubuntu 16.04 ne s'installant pas) et donc de compatibilité, nous avons choisi de travailler sur la deuxième version. Je m'attarderai donc uniquement sur l'architecture de ROS2.

b. Architecture

Tout d'abord, il faut avoir à l'esprit que le but de ROS2 est de simplifier le contrôle du robot, que nous avons imaginé dans la figure 3.

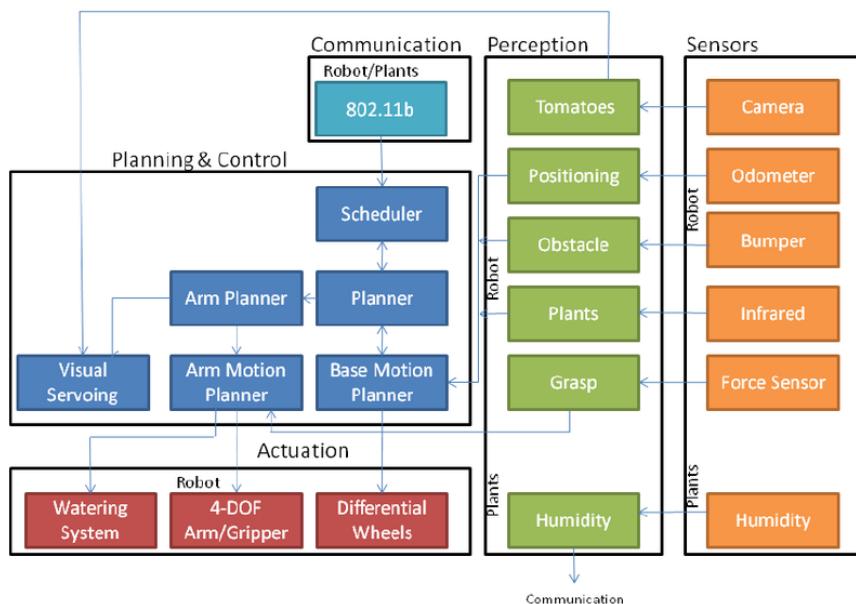


Figure 3 : Structure du logiciel de contrôle du robot (Source : <https://medium.com/software-architecture-foundations/robot-operating-system-2-ros-2-architecture-731ef1867776>)

Mais comment ROS2 fonctionne-t-il concrètement ?

La figure 4 permet de visualiser le rôle de ROS2 dans la programmation du robot.

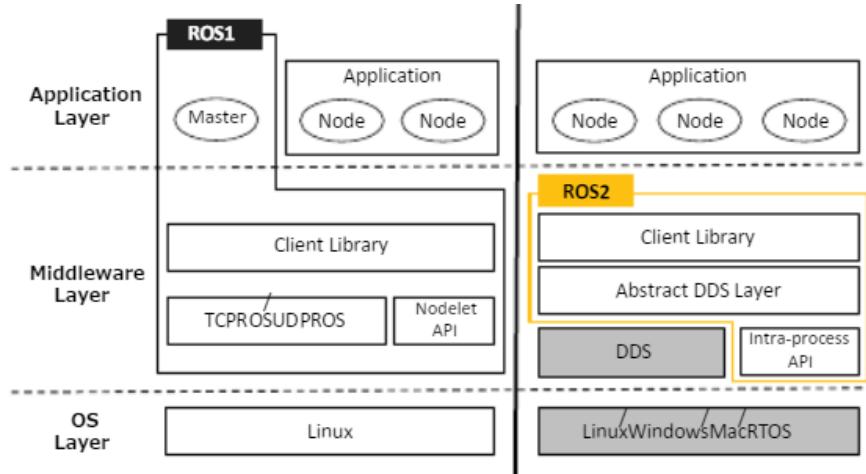


Figure 4 : Structure ROS (Source : https://www.researchgate.net/figure/ROS1-ROS2-architecture-for-DDS-approach-to-ROS-We-clarify-the-performance-of-the-data_fig1_309128426)

La particularité de ROS2 est qu'il possède une architecture actualisée en temps réel. En effet, les capteurs présents sur le robot, les contrôleurs, etc. sont les composants (appelés nœuds) de cette architecture distribuée et peuvent communiquer entre eux grâce à l'intergiciel DDS.

DDS, ou Data Distribution Service, est une norme, proposée par l'Object Management Group, dont le rôle est de proposer une technologie évoluée d'échanges de données via un réseau.

Réalisons un petit zoom sur le fonctionnement de la zone encadrée en orange dans la figure 5.

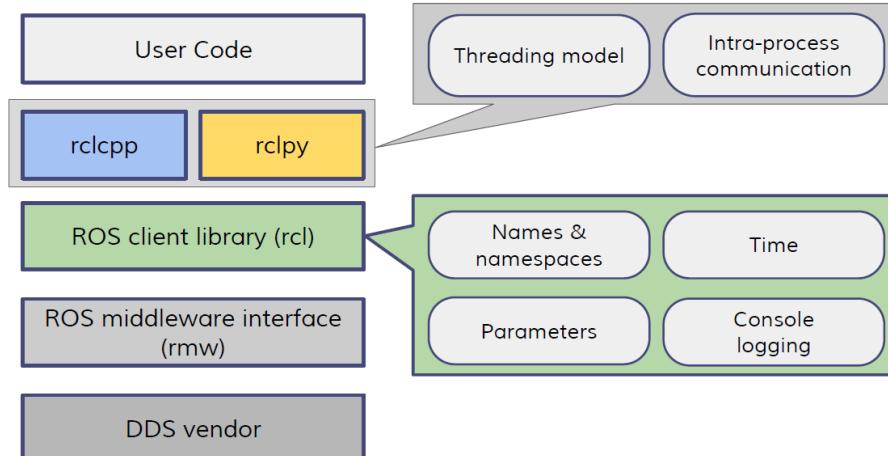


Figure 5 : Zoom sur le fonctionnement de ROS2 (Source : <https://medium.com/software-architecture-foundations/robot-operating-system-2-ros-2-architecture-731ef1867776>)

User Code

Ce sont simplement les codes que va rédiger l'utilisateur, et qui vont, par exemple, lancer des nodes et des fonctionnalités ROS2.

ROS Client Library

Les applications ROS2 accèdent aux fonctionnalités ROS2 par le biais du ROS Client Library, qui est écrite en C.

RCLCPP et RCLPY

Au-dessus de la RCL, il y a les librairies client RCLCPP et RCLPY. Ce sont respectivement des librairies client écrites en C++ et en Python.

Toute cette bibliothèque client est initialement fournie avec une interface standard qui assure l'échange des données entre les Topics et les Services (nous détaillerons ce que c'est plus tard).

ROS Middleware Interface

ROS2 a proposé sa propre couche d'abstraction matérielle (rmw) au-dessus du DDS au lieu d'utiliser directement l'intericiel DDS. Une couche d'abstraction matérielle est simplement un logiciel intermédiaire entre le système d'exploitation et le matériel informatique.

DDS Vendor

L'utilisateur a le choix entre les différentes librairies qui permettent d'implémenter le protocole RTPS, ou encore le Real Time Publish Subscribe, qui permet d'assurer les communications « écoute-publication » entre les différents composants.

Pour conclure, le système d'exploitation Linux héberge ROS2 ainsi que ses packages. Les nœuds, lancés par l'utilisateur par le biais de code, sont traités par ROS2 qui assure leur réalisation grâce à sa rmw. Enfin, les nœuds communiquent entre eux grâce au DDS.

c. ROS2 Graph Structure

L'architecture distribuée de ROS2, appelée graph, repose sur différents concepts.

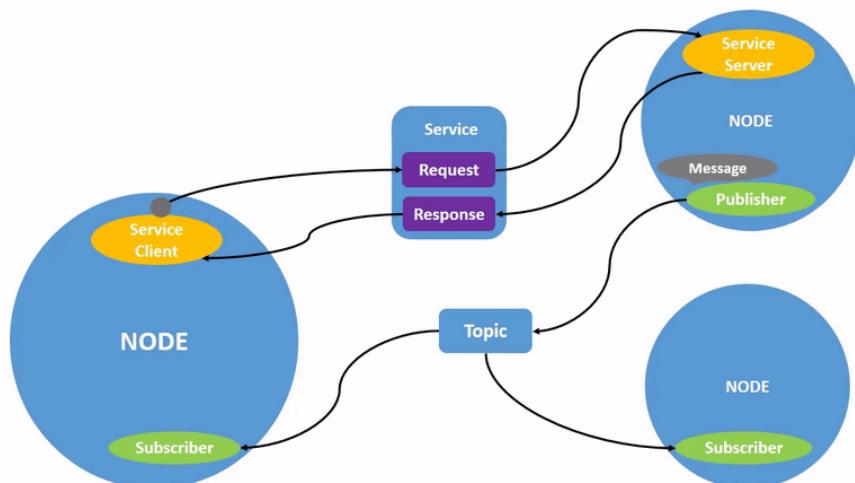


Figure 6 : Principe de communication entre nœuds ROS2

◊ Node ou nœud

Les nœuds sont simplement les applications utilisées lors du contrôle du robot. Les capteurs, les algorithmes de planification de routes peuvent être des nœuds.

Par exemple, on compte parmi les nœuds l'initialisation du robot, mais aussi l'interface de visualisation en temps réel de la carte du robot (RViz). Il y a donc des nœuds utiles pour le robot, et d'autres utiles pour le Remote PC.

Ces nœuds sont répertoriés dans des packages dont la structure est scrupuleusement établie, comme on peut l'observer en figure 7.

```
workspace_folder/
  src/
    package_1/
      CMakeLists.txt
      package.xml

    package_2/
      setup.py
      package.xml
      resource/package_2
    ...
    package_n/
      CMakeLists.txt
      package.xml
```

Figure 7 : Structure environnement de travail/packages (Source : ROS Index)

Les packages peuvent être écrits avec CMake ou avec Python. On a toujours un espace de travail dans lequel se trouve le dossier src qui contient tous les packages. Chaque package contient au minimum un nœud.

◊ Outils de communication

Les nœuds peuvent communiquer entre eux par le biais des Topics, des appels de Service et des Actions.

Un topic est un bus de communication asynchrone où des nodes publient des données tandis que d'autres s'abonnent à la lecture de ces données. Par exemple, un des topics basiques de ROS2 est le topic « /turtle1/cmd_vel geometry_msgs/msg/Twist ». C'est le topic qui exprime la vitesse du robot, avec les composantes linéaires et angulaires. Écouter ce topic permet donc de connaître la vitesse du robot et publier dans ce topic permet de commander le robot en vitesse.

Enfin, un service est simplement un mécanisme de communication synchrone entre deux nodes.

ii. Fonctionnement du robot

Ainsi, le TurtleBot3 utilise l'environnement ROS2 que je viens de décrire.

En fait, on a, d'une part, un Remote PC, qui est l'ordinateur qui permet à l'utilisateur de contrôler le robot. Linux ainsi que ROS2 sont donc installés sur cet ordinateur. L'intérêt

d'avoir un Remote PC est d'avoir une interface graphique pour contrôler le robot et aussi de contrôler le robot à distance par le biais d'une connexion SSH. Le robot peut donc se déplacer librement, nous n'avons pas besoin d'un clavier ou d'un écran pour donner des commandes au Turtlebot.

D'autre part, pour que l'on puisse contrôler le robot, il faut que celui-ci ait aussi ROS2 d'installé sur sa carte Raspberry Pi. On a donc également Linux et ROS2 installés sur la RPi.

La différence entre les deux dispositifs est que nous n'avons pas besoin des mêmes packages installés sur les deux. Par exemple, le logiciel de visualisation de carte RViz n'est pas nécessaire sur la RPi.

iii. Mise en place de l'installation

L'installation s'est découpée en plusieurs parties, celles-ci correspondant aux différents composants.

REMOTE PC

Tout d'abord, il faut savoir que ROS2 fonctionne de manière optimale sous Linux.

La première étape de l'installation a donc été d'installer Linux, et nous avons choisi la version Ubuntu 18.04, compatible avec la version de ROS2 que nous avons choisie : « Dashing ». Cédric a donc réalisé un Dual Boot sur son ordinateur, et c'est cet appareil que nous avons utilisé durant tout le projet pour programmer le robot.

Ensuite, nous avons installé ROS2 ainsi que les packages associés au TurtleBot3 par le biais de lignes de commande.

SBC

En premier lieu, il a fallu flasher le système d'exploitation sur la micro-SD de la Raspberry.

De plus, la manière la plus pratique de contrôler le robot est de se connecter en SSH afin d'écrire directement les commandes sur la Raspberry. Conséquemment, il a fallu configurer la connexion réseau de la carte pour lui indiquer de se connecter au réseau wifi que nous allons utiliser durant l'intégralité du projet : le partage de connexion sans fil du téléphone de Cédric. Ce n'est pas le choix le plus judicieux dans la mesure où ce n'est pas la manière optimale pour communiquer avec le robot mais vu que nous étions parfois contraints de ne pas travailler à l'école, ce choix nous a permis de travailler où nous voulions sans avoir à changer la configuration réseau.

Maintenant que notre Raspberry est connectée à Internet, on a pu installer ROS2 sur la Raspberry du TurtleBot3.

Enfin, nous avons installé le logiciel OpenCR sur la carte correspondante. Cette carte est un microcontrôleur qui se charge de la commande des moteurs.

B. Utilisation basique de ROS

Une fois l'installation terminée, nous pouvons maintenant utiliser les modules de ROS2 afin de nous familiariser avec ce Framework et aussi apprendre à utiliser les modules de cartographie.

On s'est très rapidement rendu compte que l'utilisation de ROS2 n'est pas du tout conviviale. En effet, pour pouvoir utiliser les modules de ROS2, il faut déjà initialiser les capteurs en lançant une commande sur la Raspberry du TurtleBot. Pour cela, il faut créer une connexion SSH entre l'ordinateur et la RPI (la RPI n'a pas de GUI, il s'agit juste d'un noyau Debian). Dans un premier temps sur l'ordinateur maître, il faut que le terminal courant prenne en compte les paramètres, pour cela on doit utiliser la commande source « /opt/ros/dashing/setup.bash » afin d'exécuter le script bash pour pouvoir utiliser les modules ROS2. Ensuite, on doit spécifier le type de TurtleBot, dans notre cas il suffit d'exécuter la commande : export TURTLEBOT3_MODEL=burger (toujours dans le même script). C'est uniquement à ce moment-là que nous pouvons commencer à utiliser les modules ROS2.

Une fois l'initialisation de l'ordinateur et du TurtleBot faite, nous pouvons passer à l'utilisation des modules de cartographie SLAM de ROS2. Dans notre cas, nous avons surtout utilisé 2 modules qui sont :

- Teleop : Ce module permet de contrôler le robot grâce au clavier ;
- Cartographer : Permet de récupérer les informations reçues du lidar, et des algorithmes SLAM afin de cartographier une pièce et de permettre au robot de se localiser dans l'environnement.

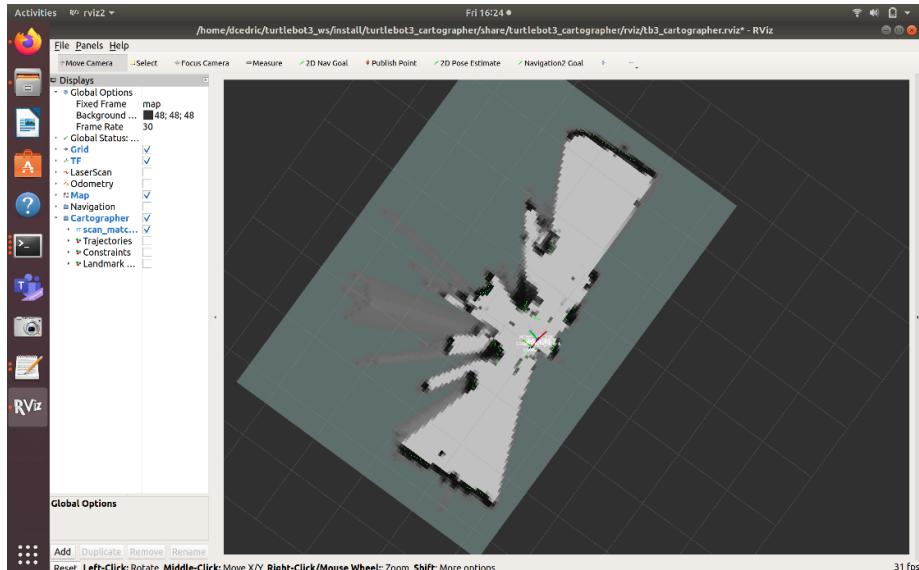


Figure 8 : Interface de Cartographer

C. Première automatisation

L'objectif de cette partie est de simplifier au plus possible l'utilisation de ROS2 en mettant en place des scripts. Nous avons donc écrit des programmes python qui permettent d'interagir avec l'environnement linux. De ce fait, l'idée est de pouvoir lancer l'environnement

ROS sur l'ordinateur et sur le TurtleBot juste en exécutant un seul script. On a donc créé un script connexion_ssh_test.py qui permet cela.

Dans un premier temps il faut exécuter le bring-up sur le TurtleBot ce qui implique une connexion ssh entre les deux appareils et ensuite faire le bring-up :

```
HOST = '172.20.10.4'
USER = 'ubuntu'
PASSWORD = 'turtlebot'

client = pm.SSHClient()
client.load_system_host_keys()
client.load_host_keys(os.path.expanduser('~/ssh/known_hosts'))
client.set_missing_host_key_policy(AllowAllKeys())
client.connect(HOST, username=USER, password=PASSWORD)

channel = client.invoke_shell()
stdin = channel.makefile('wb')
stdout = channel.makefile('rb')

stdin.write('')
ls
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_bringup robot.launch.py
'''
```

Figure 9 : Initialisation des capteurs sur le TurtleBot

Ensuite, il faut lancer des processus en parallèle sur l'ordinateur. Nous avons besoin de Cartographer qui est l'interface permettant de voir la carte en temps réel et de la sauvegarder. Pour cela, nous avons développé un code qui ouvre un nouveau terminal et qui exécute la commande pour lancer Cartographer, mais aussi une fonction qui permet de fermer ce terminal une fois le script principal interrompu.

```
def close_cartographer():
    cmd = """ps -ef | awk '$NF=="run_cartographer"'"""\n    #Avoir PID de la fenetre run cartographer
    os.environ['PYTHONUNBUFFERED'] = "1"
    proc = subprocess.Popen(cmd,
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,
                           universal_newlines=True,
                           shell=True
                           )
    stdout = proc.stdout
    stderr = proc.stderr
    mix = []
    out=stdout.readlines()
    out=out[8]
    out=out[9:15]
    cmdkill='kill -s 9 '+str(out)
    os.system(cmdkill)

    process = subprocess.Popen(
        "gnome-terminal -x python3 run_cartographer.py",
        stdout=subprocess.PIPE,
        stderr=None,
        shell=True
    )
```

Figure 10 : Code pour exécuter Cartographer en parallèle du processus principal

Maintenant, tous les processus primordiaux sont lancés, il reste maintenant à faire intervenir l'utilisateur pour savoir ce qu'il veut faire avec le TurleBot. L'utilisateur a trois choix :

- Sauvegarder la map
- Contrôler le robot
- Quitter le processus

```

a=0
b=0

msg="""
SOFT ROS2 PAR Cédric et Héloïse
-----"""

Paramètre :
    alt : Contrôler le robot
    |           Moving around:
    |           |
    |           w
    |           a   s   d
    |           x
    |
    |           space key, s : force stop
tab : save map
esc : quitte le programme
"""

print(msg)
status=0
with keyboard.Events() as events:
    for event in events:
        if event.key == keyboard.Key.esc:
            print('break')
            break
        elif (event.key== keyboard.Key.tab and a==0):
            print("space save map")
            os.system("python3 savemap_on_turtlebot.py")
            os.system("python3 dll_ssh.py 1")
            a=a+1
        elif (event.key== keyboard.Key.alt and b==0):
            print("contrôleur")
            os.system("python3 teleop.py")
            b=b+1
        else:
            print('Received event {}'.format(event))
            status=status+1
            if (status==10):
                print(msg)
                status=0
a=0
b=0

```

Figure 11 : L'utilisateur à le choix du script qu'il veut exécuter

La variable « status » permet d'afficher le message « msg » toutes les 10 lignes sur le terminal pour ne pas perdre le visuel sur celui-ci.

Finalement, lorsque l'utilisateur décide d'interrompre le script, nous forçons la sauvegarde de la « map courante » (pour éviter les mauvaises manipulations) et fermons tous les processus en cours (Cartographer et connexion SSH).

Comme vous pouvez le voir, il y a trois fichier python qui apparaissent dans la figure 10 (savemap_on_turtlebot.py, teleop.py et dll_ssh.py).

Savemap_on_turtlebot.py (cf. annexe 2) et dll_ssh.py (cf. annexe 3) sont de la même forme que le code de la figure 9. Il s'agit simplement d'exécuter des lignes de commande en ssh (pour savemap_on_turtlebot.py) et de créer une connexion sftp pour récupérer (ou envoyer) des fichiers par la connexion ssh.

Visuellement, le résultat sur un terminal donne la figure 12 :

```

cedric@cedric-MacBookPro:~/Documents/projet/scripts$ python3 connexion_ssh test.py
Please wait for the initialization of the TurtleBot | 35/35
SOFT ROS2 PAR Cédric et Héloïse
-----"

Paramètre :
    alt : Contrôler le robot
    |           Moving around:
    |           |
    |           w
    |           a   s   d
    |           x
    |
    |           space key, s : force stop
tab : save map
esc : quitte le programme

space save map
Please wait while the map is saved | 30/30
Received event Release(key=Key.tab)

Control Your TurtleBot3!
-----"
Moving around:
    w
    a   s   d
    x
w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)
space key, s : force stop
CTRL-C to quit

```

Figure 12 : Interface homme-machine

D. Traitement de la carte et algorithme A*

Nous allons maintenant voir comment nous avons mis en place un algorithme pour choisir le chemin le plus optimal entre un point de départ et d'arrivée. Pour cela, nous avons utilisé Python et en partie le module OpenCV.

L'image sauvegardée est sous le format « pgm » et chaque pixel de l'image correspond à la valeur RGB de celui-ci (cf. Annexe 13). Comme on peut le voir, il y a 3 couleurs différentes qui correspondent à :

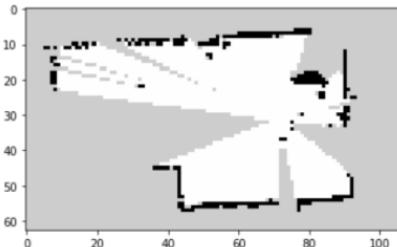
- gris : Incertain
- noir : obstacle
- blanc : accessible

Dans un premier temps, nous allons traiter l'image afin de transformer tous les gris en noir. En effet, étant donné que nous n'avons pas d'information précise sur les pixels gris nous partirons du principe que ceux-ci ne sont pas accessibles. Pour cela, nous appliquons un seuillage sur l'image :

```
In [2]: import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
import tifffile as tiff

In [3]: im = cv.imread('maptest.pgm')
plt.imshow(im)

Out[3]: <matplotlib.image.AxesImage at 0x7fc36100d2e8>

  
In [4]: ret,imbin = cv.threshold(im,210,255,cv.THRESH_BINARY)
plt.imshow(imbin,'gray');plt.show()


```

Figure 13 : Seuillage de l'image afin de n'avoir que des pixels gris et noirs

Ensuite, l'idée est de binariser l'image afin de la transformer en une matrice 2x2 sur laquelle on peut appliquer notre futur algorithme de « pathfinding ». Cette transformation facilitera la mise en place de l'algorithme. Pour cela, nous avons créé une fonction « imRGBbin_to_matrix » dans python :

```
In [7]: def imRGBbin_to_1dmatrix(im,valueamin,valueamax):
    ##Obstacle sont les 1
    ##libre = 0

    colonne=len(im[:,1])
    ligne=len(im[1,:])
    mat=np.zeros((ligne,colonne))
    mat=mat+1
    blanc=[valueamax,valueamax,valueamax]
    noir=[valueamin,valueamin,valueamin]

    for i in range(ligne):
        for j in range(colonne):
            if all(im[i,j]==blanc):
                mat[i,j]=0

    return mat
```

Figure 14 : Binarisation de l'image

Une fois le traitement de l'image terminé nous pouvons désormais passer à la mise en place d'un algorithme de « pathfinding ». Nous avons choisi un algorithme appelé A* (voir annexe 4) qui est très efficace et très utilisé, donc facile à récupérer et modeler suivant nos envies. L'idée est que notre algorithme renvoie une liste avec tous les pixels parcourus par l'algorithme pour aller du point de départ au point d'arrivée.

Nous avons donc modelé un code A* afin qu'il nous retourne la liste de tous les pixels parcourus (cf. annexe 4). Nous avons également ajouté un visuel du chemin :

```
[(50, 60), (49, 60), (48, 60), (47, 60), (46, 60), (45, 60), (44, 60), (43, 60), (42, 60), (41, 60), (40, 60), (39, 60), (39, 59), (38, 59), (37, 59), (36, 59), (36, 60), (35, 61), (35, 62), (35, 63), (35, 64), (34, 64), (34, 65), (34, 66), (33, 66), (33, 67), (33, 68), (33, 69), (32, 69), (31, 69), (31, 68), (31, 67), (31, 66), (31, 65), (30, 65), (30, 64), (30, 63), (30, 62), (30, 61), (30, 60), (30, 59), (30, 58), (29, 58), (29, 57), (29, 56), (29, 55), (29, 54), (29, 53), (29, 52), (29, 51), (28, 51), (28, 50), (28, 49), (28, 48), (27, 48), (27, 47), (26, 47), (26, 46), (25, 46), (25, 45), (24, 45), (24, 44), (23, 44), (23, 43), (22, 43), (22, 42), (21, 42), (21, 41), (20, 41), (20, 40)]
False
```

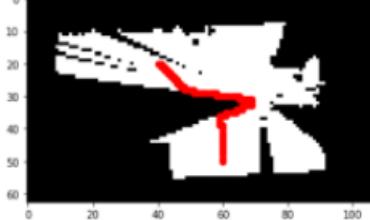


Figure 15 : Résultat de l'algorithme A*

E. Automatisation complète de la cartographie

i. Principe de l'algorithme

Maintenant que la cartographie pseudo-automatisée fonctionne, nous nous sommes penchés sur l'autonomisation complète du processus.

Tout d'abord, il fallait réfléchir à la démarche, à la manière dont nous voulions réaliser ce processus. Ainsi, voici les étapes pour lesquelles nous avons optées :

- Premièrement, l'utilisateur place le robot dans la pièce (à l'emplacement de son choix)
- Ensuite, on télécharge la première map que le robot a réalisée. En fait, le robot cartographie en permanence, ici, on cherche juste à récupérer la carte

qu'il a créée. Pour l'instant, nous avons une cartographie uniquement partielle de la pièce.

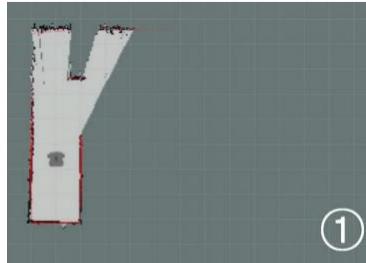


Figure 16 : Cartographie incomplète de la pièce

- Maintenant que nous avons une première carte, on va l'utiliser afin de déterminer les zones inconnues que le robot doit explorer.

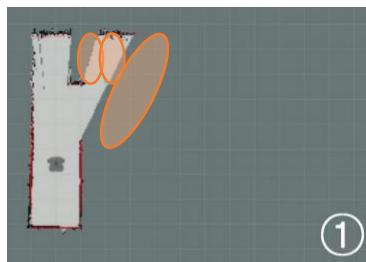


Figure 17 : Détermination des zones à explorer

- Après avoir déterminé tous les points où le robot doit aller, on lui ordonne d'y aller un par un. La carte se complète au fur et à mesure sur le robot.

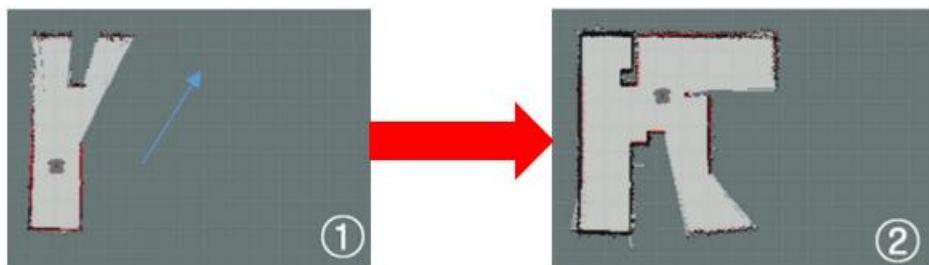


Figure 18 : Complétion de la carte

- Quand le robot a fini d'aller à tous les points, on télécharge la map.
- Ici, on peut être face à deux cas de figure : soit la cartographie est terminée, la pièce a été complètement cartographiée (cas de pièces simples), soit la cartographie n'est pas encore terminée et on a encore des zones à explorer (cas de pièces plus complexes). Pour finir la cartographie, on va donc appliquer le même principe qu'auparavant :
 - À partir de la dernière carte téléchargée (la plus complète, celle à jour), on va recalculer les points à explorer.
 - On envoie le robot vers toutes les zones à explorer.
 - On met à jour la carte que peut voir l'utilisateur en la téléchargeant lorsque le robot est allé à tous les points. On la récupère sur le Remote PC.

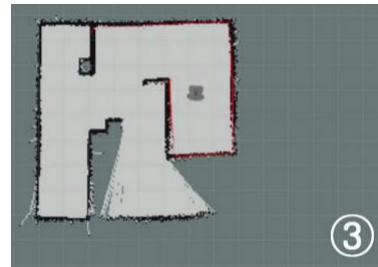


Figure 19 : Dernière carte téléchargée depuis le robot

- On répète ces étapes jusqu'à ce que la cartographie soit complète, soit jusqu'à ce que l'on n'ait plus de zones à explorer.

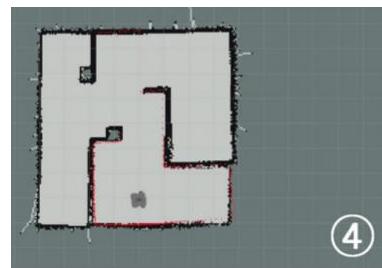


Figure 20 : Carte complète

Maintenant que nous vous avons présenté le principe, je vais rentrer un peu plus dans le détail de la programmation.

Tout d'abord, comme le but est d'automatiser complètement le processus de cartographie, nous sommes partis sur le même principe que dans l'automatisation partielle : rédiger un script Python qui réalise toutes les commandes nécessaires. Ainsi, l'utilisateur n'aura besoin que de lancer le script, et le reste s'effectuera sans son intervention.

Rentrons donc plus dans le détail du script.

ii. Détails du script

```

1 ## Imports utiles
2
3 import sys
4 import paramiko as pm
5 import os
6 from pynput import keyboard
7 import subprocess
8 import time
9 from progress.bar import IncrementalBar
10
11 sys.stderr = open('/dev/null')      # Silence silly warnings from paramiko
12 sys.stderr = sys.__stderr__
13
14 class AllowAllKeys(pm.MissingHostKeyPolicy):
15     def missing_host_key(self, client, hostname, key):
16         return
17
18 ## Définition de la fonction permettant de fermer les processus liés aux nodes slam et nav2
19
20 def close_slam():
21     cmd = """ps -ef | awk '$NF~"online_async_launch.py"'"""
22     os.environ['PYTHONUNBUFFERED'] = "1"
23     proc = subprocess.Popen(cmd,
24                            stdout=subprocess.PIPE,
25                            stderr=subprocess.PIPE,
26                            universal_newlines=True,
27                            shell=True
28                            )
29     stdout = proc.stdout
30     stderr = proc.stderr
31     mix = []
32
33     out=stdout.readlines()
34     out=out[0]
35     out=out[9:15]
36
37     cmdkill='kill -s 9 '+str(out)
38     os.system(cmdkill)
39
40 def close_nav():
41     cmd = """ps -ef | awk '$NF~"nav2_navigation_launch.py"'"""
42     os.environ['PYTHONUNBUFFERED'] = "1"
43     proc = subprocess.Popen(cmd,
44                            stdout=subprocess.PIPE,
45                            stderr=subprocess.PIPE,
46                            universal_newlines=True,
47                            shell=True
48                            )
49     stdout = proc.stdout
50     stderr = proc.stderr
51     mix = []
52
53     out=stdout.readlines()
54     out=out[0]
55     out=out[9:15]
56
57     cmdkill='kill -s 9 '+str(out)
58     os.system(cmdkill)

```

Figure 21 : Programme : Imports utiles et fermeture propre des processus

Tout d'abord, on importe ce dont on va avoir besoin. Ensuite, comme expliqué précédemment, on a besoin de fermer proprement les processus à la fin de la cartographie. On a donc créé deux méthodes dont le but est de fermer les processus qui vont nous servir : SLAM et Nav2.

```

60 ## Automatisation de la cartographie
61
62 msg="""#
63 Bonjour et bienvenue dans le module de cartographie autonome !
64 """
65 print(msg)
66

```

Figure 22 : Programme : cartographie autonome

On rentre ensuite dans le cœur de la programmation de la cartographie autonome.

```
67 # Lancement du node SLAM et Nav2 sur le remote PC
68
69 process = subprocess.Popen(
70     "gnome-terminal -x python3 nav2_bringup nav2_navigation_launch.py",
71     stdout=subprocess.PIPE,
72     stderr=None,
73     shell=True
74 )
75
76 process2 = subprocess.Popen(
77     "gnome-terminal -x python3 slam_toolbox online_async_launch.py",
78     stdout=subprocess.PIPE,
79     stderr=None,
80     shell=True
81 )
82
```

Figure 23 : Programme : lancement des nodes SLAM et Nav2

Pour se repérer dans la carte, le robot utilise l'algorithme SLAM. On lance donc le node correspondant au SLAM et aussi le node Navigation2 qui vont permettre tous les deux d'assurer la navigation du robot.

Comme ce sont des processus qui vont fonctionner tout le long de la cartographie, on les ouvre dans des terminaux différents afin qu'ils tournent en parallèle, jusqu'à ce qu'on les ferme à la fin de la cartographie.

```
83 # Connexion SSH avec le Turtlebot3
84
85 HOST = '172.20.10.4'
86 USER = 'ubuntu'
87 PASSWORD = 'turtlebot'
88
89 client = paramiko.SSHClient()
90 client.load_system_host_keys()
91 client.load_host_keys(os.path.expanduser('~/.ssh/known_hosts'))
92 client.set_missing_host_key_policy(AllowAllKeys())
93 client.connect(HOST, username=USER, password=PASSWORD)
94
95 channel = client.invoke_shell()
96 stdin = channel.makefile('wb')
97 stdout = channel.makefile('rb')
98
99 # Source et bring up du robot
100
101 stdin.write('''
102 export TURTLEBOT3_MODEL=burger
103 ros2 launch turtlebot3_bringup robot.launch.py
104 ''')
```

Figure 24 : Programme : initialisation du robot

Ensuite, on doit initialiser le robot. On se connecte donc en SSH avec le robot sur le Remote PC et on lance ce qu'on appelle le « bring up ».

```
106 # Attente de la fin de l'initialisation du robot
107
108 bar = IncrementalBar('Please wait for the initialization of the TurtleBot', max=35)
109
110 for i in range(35):
111     bar.next()
112     time.sleep(1)
113
114 bar.finish()
--
```

Figure 25 : Programme : Attente de la fin du bring up

Le « bring up » est un processus relativement long dans la mesure où c'est l'initialisation du robot. On force donc un temps d'attente pour être sûrs que le robot soit opérationnel avant de passer à la suite de la cartographie.

```

116 # Attente de la première carte
117
118 bar = IncrementalBar('Please wait for the realization of the first map', max=20)
119
120 for i in range(20):
121     bar.next()
122     time.sleep(1)
123
124 bar.finish()
125
126 # Enregistrement de la première carte
127
128 os.system("python3 savemap_on_turtlebot.py")
129 os.system("python3 dll_ssh.py 1")
130
131 msg=""""
132 La première carte a été enregistrée.
133 """
134 print(msg)
135

```

Figure 26 : Programme : Réalisation et téléchargement de la première carte

De même, on laisse le temps au robot de réaliser une première carte propre. Ensuite, on l'enregistre sur le robot en faisant appel au fichier Python savemap_on_turtlebot.py expliqué dans la partie précédente. Enfin, on récupère la carte sur le Remote PC à l'aide du fichier dll_ssh.py, également évoqué dans la partie précédente.

```

136 # Calcul des coordonnées à explorer
137
138 cmd = """python3 pts_a_explorer.py"""
139 pts_a_explorer=os.popen(cmd).readlines()
140

```

Figure 27 : Programme : Calcul des premiers points à calculer

Ici, on fait appel au fichier pts_a_explorer.py qui va permettre, à partir de la carte qu'on vient d'établir, de calculer les points où le robot doit se diriger afin d'explorer les zones inconnues.

Afin de ne pas nous perdre dans les explications, nous allons simplement rapidement expliquer le principe de détermination des points et mettre le fichier commenté en annexe [5].

Le principe est le suivant. Nous avons une carte composée de trois types de zones : les zones inconnues, les zones découvertes et les murs. On va simplement stocker ces informations dans une matrice appelée « map_grise » de même taille que l'image. Cette matrice contient :

- Des 0 : représentant les murs ;
- Des 255 : représentant les zones découvertes ;
- Des 150 : représentant les zones inconnues.

Dans la carte utilisée actuellement, il y a des pixels blancs, considérés comme accessibles, qui sont, par exemple, entourés par des murs (tel un encadrement de porte) et qui ne sont, donc, dans les faits pas accessibles par le robot.

On vient donc modifier les valeurs des pixels accessibles par le robot dans la matrice map_grise : on met pour chaque pixel accessible la valeur 10 plutôt que 255. Ce processus s'effectue par « diffusion ». En effet, on part d'un point central de la carte par exemple, et on regarde à partir de ce pixel quelles zones sont accessibles par le robot. Ceci évite donc de considérer les zones derrière des murs ou même des zones inconnues comme des zones accessibles par le robot.



Figure 28 : Programme : Représentation des zones accessibles par le robot

Enfin, on vient calculer les points où le robot doit aller pour explorer la pièce. Pour ce faire, on stocke dans la liste « pts_a_explorer » tous les pixels qui, sont, à la fois, accessibles par le robot, et, à la fois, à côté d'une zone inconnue.

```

140
141 # Aller à chaque point et mettre à jour la carte
142
143 for line in pts_a_explorer:
144     os.system("envoi_goal.py")
145

```

Figure 29 : Programme : Envoyer le robot à côté de chaque zone à explorer

Maintenant que nous avons calculé tous les points où le robot doit aller, nous allons envoyer le robot à chacun de ces points. Pour cela, nous faisons appel au fichier Python envoi_goal.py. Ce fichier, mis en annexe [6], permet de publier dans le topic « /goal_pose geometry_msgs/PoseStamped ». En faisant cela, on ordonne au robot de se déplacer jusqu'au point que l'on désire. Ici, on lui donne les coordonnées des points à explorer.

```

146 # Sauver la map pour la mettre à jour
147
148 os.system("python3 savemap_on_turtlebot.py")
149 os.system("python3 dll_ssh.py 1")
150
151 msg=""""
152 La carte a été mise à jour.
153 """
154 print(msg)
155

```

Figure 30 : Programme : Mettre à jour la map de l'utilisateur

Une fois tous les points explorés, on sauvegarde la carte et on l'enregistre sur le Remote PC.

```

156 # Fin de la cartographie
157
158 cmd = """python3 pts_a_explorer.py"""
159 pts_a_explorer=os.popen(cmd).readlines()
160
161 while pts_a_explorer != []:
162     for line in pts_a_explorer:
163         os.system("envoi_goal.py")
164         os.system("python3 savemap_on_turtlebot.py")
165         os.system("python3 dll_ssh.py 1")
166         cmd = """python3 pts_a_explorer.py"""
167         pts_a_explorer=os.popen(cmd).readlines()
168
169 msg=""""
170 La cartographie est terminée !
171 """
172 print(msg)
173
174
175 close_slam()
176 close_nav()
177 stdout.close()
178 stdin.close()
179 client.close()

```

Figure 31 : Programme : Fin de la cartographie

Enfin, si la cartographie n'est pas terminée, on continue à déterminer les points à explorer et on envoie le robot en chacun de ces points, jusqu'à ce qu'on n'ait plus de zones à explorer.

On a enfin la carte complète !

Pour finir, on ferme bien toutes les connexions et processus.

iii. Limites

Bien que nous ayons réfléchi à ce script, nous n'avons pas encore eu le temps de le tester.

En effet, découvrir et comprendre ROS nous a pris beaucoup de temps et nous sommes parfois restés bloqués sur du code sans pouvoir avancer. De fait, comprendre comment fonctionne la création d'un package est assez fastidieux.

D'autre part, nous avons mis du temps pour mettre en place la première automatisation de la cartographie. En effet, pour que la gestion des processus soit correctement réalisée, nous avons dû passer quelques heures sur le sujet.

Nous avons, de plus, rencontré des problèmes de connexion avec le Turtlebot3 une semaine avant la soutenance et nous avons donc perdu du temps à réinstaller Ubuntu sur la RPi.

Par ailleurs, nous avons également eu du mal au début, ne serait-ce que pour la cartographie manuelle, dans la mesure où le robot n'arrivait pas à cartographier correctement. Nous nous sommes rendu compte que le Lidar était en fait monté à l'envers. Après correction de ce montage, le robot était fonctionnel.

Ainsi, nous avons commencé la partie cartographie autonome bien plus tard que ce que nous pensions.

Cette partie a d'abord été abordée en réalisant un tutoriel dont le but était simplement de cartographier en même temps que l'on déplaçait le robot. C'est, lors de cette séance, que nous nous sommes rendu compte que la publication dans le topic « /goal_pose geometry_msgs/PoseStamped » ne fonctionne pas. En effet, lorsque l'on publie le message, le robot n'avance pas. Nous n'avons pas encore eu le temps de régler ce problème (qui est bloquant pour notre script Python) mais nous pensons que c'est lié au driver du Lidar qui n'envoie pas le bon nombre de lectures. Nous pensons donc qu'il faut mettre à jour ce driver.

D'autre part, comme nous n'avons pas pu tester le code, nous ne savons pas encore s'il est fonctionnel. Nous avons travaillé sur le principe, il nous reste encore à le tester et à le déboguer si nécessaire.

II -Second robot

Le but de ce robot est de pouvoir communiquer avec l'utilisateur et de se servir d'une carte pour suivre un parcours prédéfini.

1. Hardware

Etant donné que ce robot doit se déplacer, on doit d'abord se décider sur le type de robot mobile à faire. Il existe quatre grands types de robot mobile : les robots type unicycle, type tricycle, type voiture et omnidirectionnel. Comme on désirait ne pas avoir à programmer un modèle trop compliqué, on s'est décidé de faire un modèle unicyle avec une roue folle pour la stabilité.

Maintenant, que l'on connaît le modèle, on doit savoir de quel type de capteur dont on a besoin. Pour ce projet, on désire avoir :

- Les vitesses des roues afin de calculer le déplacement de la plateforme ;
- L'orientation du robot dans l'espace pour avoir sa direction de déplacement avec une bonne précision ;
- La distance à un obstacle afin d'avertir l'utilisateur et de ne pas avancer sans fin dans l'obstacle.

Ainsi, on a décidé d'utiliser des roues codeuses sur les roues, un gyroscope et un capteur de distance infrarouge. Cependant, le choix de capteur proprioceptif pour calculer la position du robot dans la pièce amène une incertitude au cours du temps. Il faudra donc permettre à l'utilisateur de redéfinir la position du robot à des endroits stratégiques.

Par conséquent, après ces décisions, on a fait la demande d'achat du matériel suivant :

- Carte Raspberry PI Zero WH ;
- Accumulateur LiPo ;
- Carte Lipo rider plus, permettant le rechargement de la batterie et la protection de la carte si un problème arrive avec la batterie ;
- Deux moto-réducteurs de rapport 50:1 avec roue encodeuse. Les moteurs pris fonctionnent normalement sous une tension de 6Vcc mais les alimentent avec du 5Vcc,

leurs caractéristiques (vitesse, couple) s'en trouve réduit. Ainsi, le choix du rapport 1/50, permet d'avoir une vitesse acceptable du point de vue du cahier des charges malgré la sous-tension ;

- Un motordriver L298N ;
- Un accéléromètre et gyroscope MPU6050 ;
- Une paire de roue.

Le capteur de distance et la roue folle étant disponibles dans la salle de projet, on n'a pas eu à en faire l'acquisition.

A partir de là, on devait faire le choix de fabrication de la plateforme contenant tout cela. Puisque le robot se déplace en intérieur, la plateforme n'a pas besoin d'avoir une grande résistante aux événements extérieurs. On a donc choisi de l'imprimer en 3D et donc d'en faire un modèle sous un logiciel. Ce modèle a été fait par Nicolas sous CREO. On a d'abord fait un premier modèle avec les mesures des composants décrites par les fournisseurs. A la réception des composants, on les a mesurés afin de s'assurer des valeurs mises dans le modèle.

Afin de gérer l'encombrement des composants et des fils, il a été choisi de faire une plateforme sur deux étages. Le premier contient les moteurs et leur driver, la carte RPi, l'accéléromètre et le capteur de distance tandis que le second, la batterie et la carte LiPo.

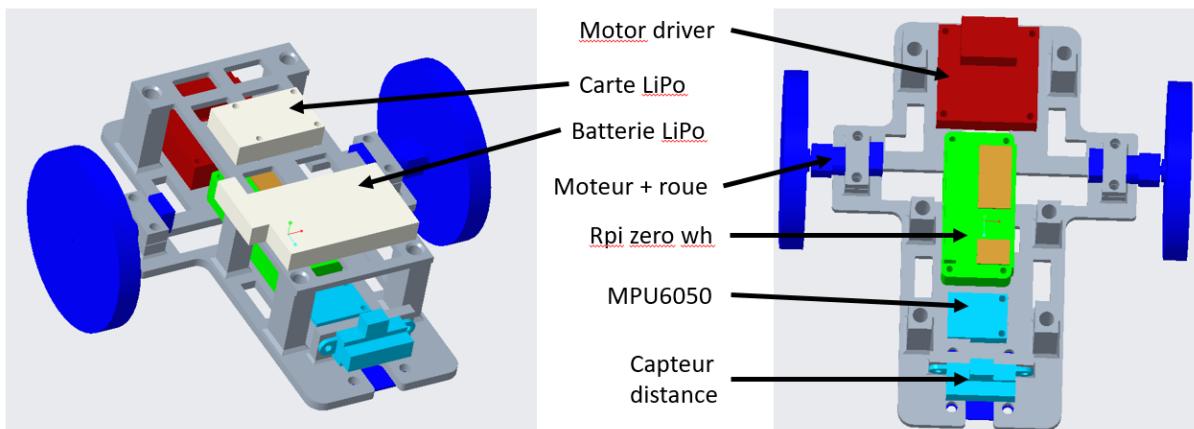


Figure 32 : Modèle Créo du robot

Suite à l'impression, on a remarqué quelques erreurs sur le modèle qui ont été révisées :

- Les dimensions et le positionnement des trous de fixation de la roue folle sont erronés, des trous supplémentaires seront faits pour corriger cela ;
- Les trous n'étaient assez gros pour faire passer les vis, ils ont été agrandis grâce aux matériels de la salle de projet ;
- Les enlèvements de matière soutenant les moteurs n'étaient pas assez large, ce qui a été corrigé en retirant de la matière avec une lime.

2. Asservissement des moteurs

Comme dit précédemment, on désire déterminer la position de ce robot avec une certaine précision. Pour ce faire, on doit asservir les moteurs pour ne pas avoir d'erreur avec la consigne ni de différence de vitesse entre les deux moteurs. En effet, une différence de vitesse

entre les moteurs amène une rotation du robot ce qui peut nuire à la détermination de la position.

A. Logique de programmation

Pour l'asservissement des moteurs, on a choisi de faire un correcteur PI. L'action proportionnelle du correcteur permet de réduire le temps de réponse et l'action intégrale l'erreur avec la consigne. Le correcteur est implémenté selon le schéma suivant et le code est fourni en annexe 8:

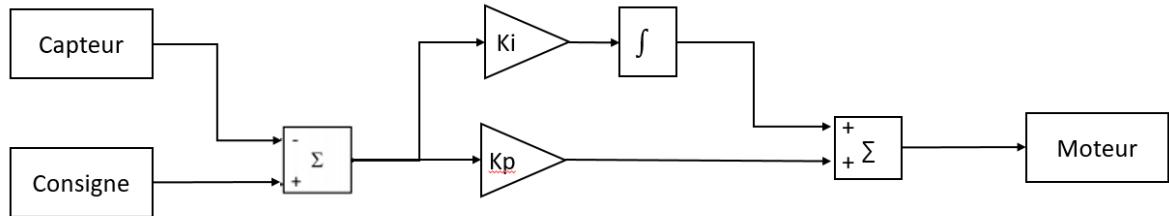


Figure 33 : Schéma du correcteur de vitesse

Pour faire l'asservissement, on doit d'abord obtenir la vitesse grâce au codeur incrémental du moteur. Ce dernier est composé de deux voies et de sept zones détectables. En comptant les fronts montant et descendant des capteurs, on obtiendra ainsi une vitesse de rotation. Afin de réaliser cela, on décide d'utiliser une logique de détection d'événement sous RPi. Lorsque la carte détectera un front (montant ou descendant) sur la première voie du capteur, on lance un processus parallèle. Ce dernier ajoute 1 à un compteur global si la valeur lue sur la première voie est égale à celle de la deuxième. Il retire 1 si les valeurs sont différentes. Puis, toutes les n ms, dans le processus principal, on exécutera une fonction « correcteur ». Cette fonction permet de calculer la vitesse et la nouvelle commande envoyée au moteur. Le calcul de la vitesse est fait à partir de la dernière valeur utilisée du compteur, du dernier temps où la fonction s'est lancée, de la valeur du compteur actuel et du temps actuel. Elle continue par déterminer l'erreur entre la consigne et la vitesse pour appliquer le correcteur et envoyer la nouvelle commande au moteur.

Cependant, la vitesse d'un moteur en courant continu se modifie grâce à la PWM. Dans le cas d'un programme sous Python, la valeur de PWM est comprise entre 0 et 100. On doit donc commencer par déterminer une loi pour convertir la valeur de commande trouvée précédemment en une valeur pour le PWM. Après des recherches, on a pu avoir que cette loi est linéaire. Ainsi, on mesure la vitesse pour une PWM et on applique un produit en croix.

Maintenant que la logique d'exécution est programmée, nous pouvons trouver le couple (K_p ; K_i) pour chaque moteur. Le but est d'obtenir un moteur assez rapide avec un dépassement faible, peu d'oscillation et une erreur statique nulle. On commence donc par un premier moteur en fixant un gain K_p expérimentalement (avec K_i nul). On détermine ensuite le gain intégrateur (quitte à modifier le gain proportionnel) garantissant la dynamique désirée. On termine par trouver le couple de gain pour le second moteur de manière à avoir une dynamique identique au premier moteur.

B. Expériences et résultats

Lors des premiers essais expérimentaux, on a pu constater que le driver ne fonctionnait pas. En effet, quelles soit la consigne envoyée au moteur, il ne tournait pas. On a d'abord pensé à un problème lié à la sous-tension. On a essayé de faire un amplificateur opérationnel (avec le matériel de la salle) fournissant une tension 6V. Mais, cela n'a pas réglé le problème. On a ensuite testé un autre driver (L293D possédé par Nicolas) et le moteur fonctionnait. Ainsi, pour la fin du projet, on utilise ce driver.

On réalise maintenant les différentes mesures nécessaires au détermination des gains du correcteur. Ces mesures sont faites à l'aide d'une carte Arduino pour traiter les résultats rapidement sur ordinateur.

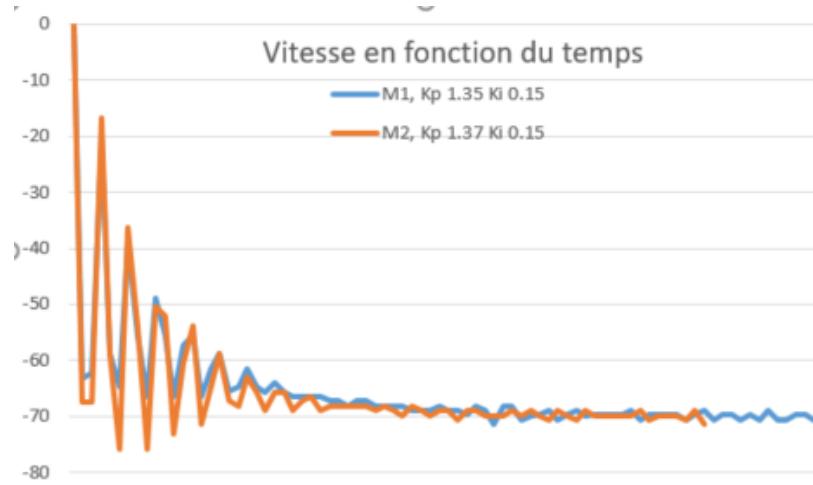


Figure 34 : Premier test de couple de valeurs

Lors d'une première détermination, nous avons obtenu les résultats précédents. On peut voir que les moteurs sans correcteur n'ont pas la même dynamique et que le correcteur permet de rectifier cela. Cependant, les oscillations au démarrage sont trop importantes. Il faut donc déterminer un nouveau couple de valeur ou un programme pouvant les filtrer.

Lors de l'implémentation du code dans la Raspberry, on a pu constater un problème avec le compteur de vitesse. En effet, si le moteur tournait trop rapidement, il ne réussissait pas à suivre et pouvait afficher une valeur d'environ 1 RPM. Afin de corriger cela, on ne se sert que de l'une des deux voies du codeur. Dans ce cas, la valeur affichée est plus proche de la réalité mais on ne peut plus obtenir le sens de rotation du moteur. Cependant, la vitesse lire par la RPi varie beaucoup. En effet, on peut passer d'une valeur lire de 45 RPM à 52 RPM à 48 RPM pour une consigne de 50 RPM. Afin de stabiliser cela, on décide de mettre en place une moyenne glissante. Ainsi, lorsque l'on calcule l'erreur entre la consigne et la vitesse, on utilise une moyenne des quatre dernières vitesses lues. Par ailleurs, cela permet de diminuer les oscillations au démarrage.

Par conséquent, avec les expériences précédentes, nous avons pu déterminé les couples de gain de chaque moteur, amélioré le programme pour réduire les variations de vitesses utilisées et résolu certains problèmes de fonctionnement.

3. Suivi de trajectoire

Après le Hardware et l'asservissement des moteurs, la prochaine étape a été la mise en œuvre du suivi de trajectoire sur le robot. Globalement, lorsque nous demandons au robot d'aller dans une certaine configuration, ayant déjà connaissance de la trajectoire calculée au préalable avec l'algorithme A*, il effectue un certain nombre d'opérations entre sa configuration courante et la configuration demandée. Il impose ensuite les vitesses à ses roues tout en effectuant la régulation nécessaire.

A. Définition du problème

Comme il a été précisé plus haut, nous avons considéré un robot de type unicycle. Afin d'établir les lois de contrôle, une trajectoire a été calculée au préalable et les coordonnées en pixels des points à atteindre durant le déplacement ont été sauvegardées. Le problème principal est de faire partir le robot d'une configuration donnée A à une configuration désirée B en passant par les points identifiés lors du calcul de la trajectoire. Par configuration, on entend la valeur de l'ensemble des paramètres du robot : la vitesse résiduelle V , son orientation θ , sa position en x et y . Pour répondre à ce problème, nous avons décidé de faire parcourir au robot des mini-trajets entre les points de la trajectoire afin de ne faire que des mouvements linéaires. Nous avons ensuite défini certaines contraintes :

- Le robot doit rester dans un certain périmètre.
- Le dérapage des roues doit être évité (sans glissement).
- Les déplacements doivent pouvoir s'enchaîner.

B. Logique de programmation

Le modèle cinématique considéré pour la programmation du suivi de trajectoire est représenté sur la figure suivante.

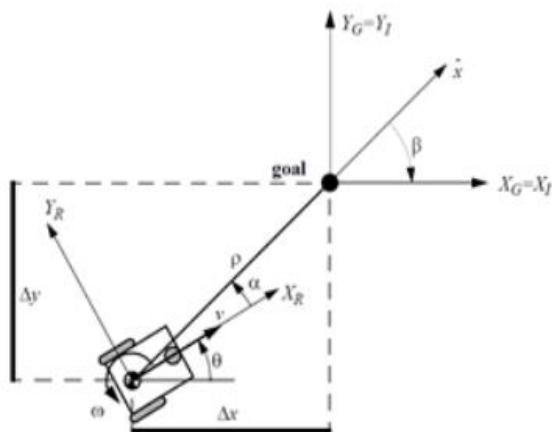


Figure 35 : Modèle cinématique

Les paramètres utiles sont la posture (x, y, θ) du robot. Après avoir converti les données de la trajectoire en mètres, nous avons programmé une boucle afin que le robot se déplace suivant les points de la trajectoire. L'idée globale est de calculer la rotation à effectuer à partir de la direction du robot pour savoir si le robot doit tourner à droite ou à gauche. Pour cela, le

robot calcule à chaque itération l'angle entre son point de départ et le point successif donné par la formule : $\beta = \arctan\left(\frac{y_B - y_A}{x_B - x_A}\right)$. Ensuite, il compare cet angle avec son orientation θ donnée par le gyroscope « MPU6050 ». Après plusieurs essais, nous avons considéré une marge de $\pm 1^\circ$ en ce qui concerne les rotations à effectuer par le robot pour rejoindre une configuration. Le robot devra donc se déplacer en tenant compte de l'angle obtenu : $\alpha = \beta - \theta$. Si $\alpha < -1$, alors le robot doit tourner à gauche d'un angle β étant donné que les angles sont absous, puis avancer vers le point suivant. Si $\alpha = 1$, le robot n'effectue aucune rotation et doit juste avancer. Si $\alpha > 1$, il devra tourner à droite d'un angle β , puis avancer vers la configuration suivante. Cette logique est valable pour des angles β compris entre -180° et 180° . Les rotations sur place du robot vers la gauche ou vers la droite sont effectuées en régulant à chaque fois les vitesses des moteurs gauche « vg » et du moteur droit « vd ».

```

def calculTempsTeta(path, speedMS):
    global Xactuel, Yactuel, currentAngle
    x=path[0][0]-Xactuel
    y=path[0][1]-Yactuel
    dis=dist(x,y)
    t=dis/speedMS
    if x!=0:
        teta=math.degrees(math.atan2(y,x))
    else :
        if y>=0:
            teta=90
        else:
            teta=-90
    print('teta :'+str(teta))
    return t,teta

def rotation(teta_a_atteindre, speedRPM):
    global currentAngle
    if teta_a_atteindre <=180:
        print('C1')
        if(teta_a_atteindre-currentAngle>1):
            turn_right(speedRPM+20,teta_a_atteindre)
            stop()
            #print('C1-1')
        elif(teta_a_atteindre-currentAngle<-1):
            turn_left(speedRPM+20,teta_a_atteindre)
            stop()
            #print('C1-2')
    elif teta_a_atteindre>180:
        print('C2')
        if (currentAngle-teta_a_atteindre>1):
            turn_left(speedRPM+20,teta_a_atteindre)
            stop()
            #print('C2-1')
        elif(currentAngle-teta_a_atteindre<-1):
            turn_right(speedRPM+20,teta_a_atteindre)
            stop()
            #print('C2-2')
    return True

```

Figure 36 : Fonctions pour le calcul de l'angle à atteindre

Afin de vérifier, l'angle de rotation du robot, on se sert de l'interface I2C de la RPi communiquant avec le gyroscope. Comme on ne s'intéresse qu'à la rotation selon l'axe z, on ne lit que les données gyroscopiques provenant de cette rotation. Ainsi, au lancement du robot, on commence par calibrer le gyroscope. Lors d'une rotation, on vient lire les données pour obtenir une valeur en deg/s à laquelle on retire la valeur de calibrage. Puis, on intègre cette valeur par rapport au temps pour avec l'angle du robot (cf. annexe 9).

A chaque fois que le robot se déplace vers une configuration de la trajectoire, sa position est recalculée et son positionnement est vérifié pour éviter des imprécisions. Pour cela, une marge d'erreur de $\pm 0,05$ m est tolérée. Ensuite, les points où le robot est déjà passé sont enlevés et ne seront plus pris en compte.

```

def convertPath(chemin):
    for k in range(len(chemin)):
        chemin[k][0]=0.05*chemin[k][0]
        chemin[k][1]=0.05*chemin[k][1]
    return chemin

def enlèvePointAtteint(path):
    if(len(path)>1):
        newPath=np.zeros((len(path)-1,2))
        for i in range(len(newPath)):
            newPath[i][0]=path[i+1][0]
            newPath[i][1]=path[i+1][1]
    else:
        newPath=0
    return newPath

def isPositionOK(path):
    global Xactuel, Yactuel
    Xok=False
    Yok=False
    positionOK=False
    if path==0 :
        positionOK=True
    else :
        if(Xactuel<=path[0][0]+0.05 and Xactuel>=path[0][0]-0.05):
            Xok=True
        if(Yactuel<=path[0][1]+0.05 and Yactuel>=path[0][1]-0.05):
            Yok=True
        if(Xok and Yok):
            positionOK=True
    return positionOK

```

Figure 37 : Fonctions utiles respectivement pour la conversion des coordonnées en mètres, l'enlèvement des points déjà atteints et la vérification du positionnement du robot quand il arrive à une configuration

Pour finir, nous avons ajouté au programme une fonctionnalité permettant au robot de signaler la présence d'obstacles éventuels sur la trajectoire. Cette fonctionnalité devient utile si des obstacles ont été rajoutés après la définition de la carte. C'est à l'aide d'un capteur infrarouge « Sharp GP2Y0A41SK0F » que la détection d'obstacles a été ajoutée. C'est un capteur permettant la mesure de distance absolue ou relative avec plage de détection allant de 40 à 300 mm. La fonction « `getDistance()` » est implémentée afin de permettre au robot de détecter les éventuels obstacles à une distance inférieure à 20 cm, ensuite il les signalera à l'utilisateur à travers le server grâce à la fonction « `signalementObstacleBDD(Xactuel,Yactuel)` ». La Figure 38 présente la ditea-fonctions.

```

def signalementObstacleBDD(Xactuel, Yactuel):
    BDD=connect.connect(host="192.168.4.1",user="root",password="rpiSMR2",database="projet")
    curseur=BDD.cursor()
    query="INSERT INTO ordre (heure,destinataire,etat,typeOrdre,syntaxeOrdre) VALUES (NOW(),%s,%s,%s,%s)"
    syntaxe=str(Xactuel)+";"+str(Yactuel)
    data=(0,0,5,syntaxe)
    curseur.execute(query,data)
    BDD.commit()
    BDD.close()

```

Figure 38 : Fonction de signalement d'obstacle

Etant donné que la trajectoire a été calculée à travers un algorithme de path-finding qui permet de calculer un certain nombre de points de passage pour atteindre notre objectif final tout en évitant les obstacles, on a voulu optimiser le programme en prenant en compte le cas où un obstacle est placé sur la trajectoire déjà calculée. C'est à l'aide d'un capteur infrarouge que la détection d'obstacles a été implémentée.

C. Résultats

Lors des tests, nous avons eu des erreurs avec le gyroscope. En effet, au début, la valeur rentrée par le gyroscope variait grandement même sans mouvement. On a donc augmenté la résolution du calibrage (en considérant plus de valeur). Ensuite, l'intégration donnait un

angle de temps à autre incohérent (on passait de 33° à 35° à 27° en gardant la même rotation). Pour pallier à ce problème, on a décidé de considérer une moyenne de vitesse de rotation au moment de l'intégration. Enfin, le capteur (gyroscope-accéléromètre) prenait les angles de manière absolue jusqu'à l'infini. On a donc dû modifier nos programmes afin de garder des angles compris entre -180 et 180° et de retirer la relativité de calcul de rotation.

Par ailleurs, après plusieurs déplacements, on a remarqué que le robot à une précision acceptable. On a donc réduit la tolérance de position à $\pm 5\text{mm}$. Cependant, à certain moment, une roue tourne moins rapidement que l'autre pour une même consigne. Cela provoque un changement de trajectoire non contrôlé et une dérive sur la position par la suite. Puisqu'on pense que ce problème provient des roues ou des moteurs mêmes, on n'a pas pu corriger ce dysfonctionnement.

4. Interprétation des ordres de la base de données

Le second robot suit le diagramme de la figure 39 afin de comprendre les informations de la base de données. Par des soucis d'exécution et de temps, on a par la suite donné tout le chemin à suivre (déterminé par l'algorithme A*) sous la forme suivante : «X1;Y1!X2;Y2!X3!Xn;Yn». Cette écriture est donc décomposée est inscrite dans le tableau du parcours à l'aide de l'algorithme de la figure 40.

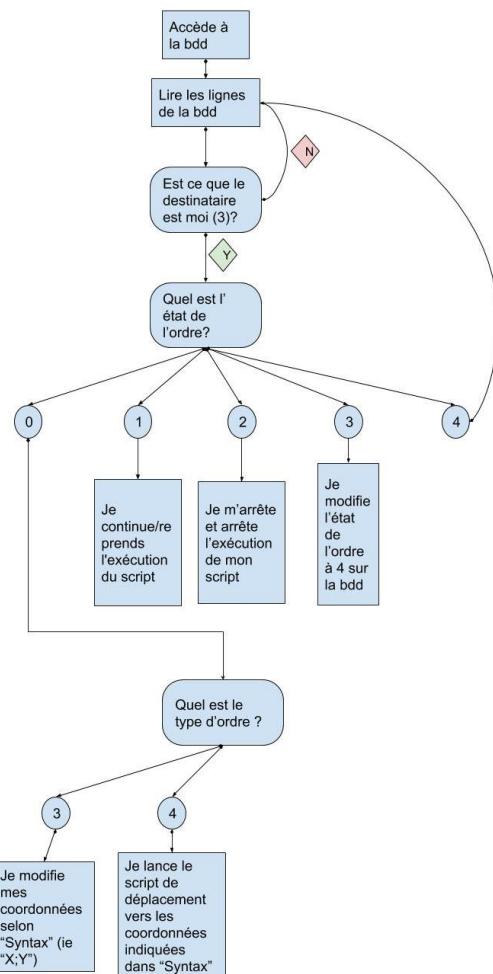


Figure 39 : Schéma de lecture des ordres de la base de données pour le second robot

```

def separePoints(listeTot):
    listeSplt=listeTot.split("!")
    listeDest=np.zeros((len(listeSplt),2))
    for i in range(len(listeSplt)):
        listeCoord=convertListeStr(listeSplt[i])
        listeDest[i][0]=listeCoord[0]
        listeDest[i][1]=listeCoord[1]

def convertListeStr(listeString):
    liste=listeString.split(" ; ")
    list2=np.zeros(len(liste))
    for i in range(len(liste)):
        list2[i]=float(liste[i])
    return list2

```

Figure 40 : Algorithme décomposant le texte de la base de données en chemin

5. Boucle finale

Dans l'exécution totale du code, on décide de lancer les appels à la base de données dans un deuxième processus. Pour cela, on utilise le module « multiprocessing ». Lorsque l'on utilise ce dernier, les variables ne sont pas communes entre les processus. Ainsi, on doit employer une structure de « Queue » ou de « Pipe » pour communiquer les résultats. Pipe ne peut contenir qu'un résultat. Queue est une structure de liste « FIFO », on peut donc mettre un grand nombre de résultat et on récupère le premier entré en le retirant dans la liste. On décide donc d'utiliser la structure de « Queue » qui permet plus de liberté.

Lors du lancement du robot, il faut commencer par initialiser tous les pins, les variables et calibrer le gyroscope. Ensuite, on regarde si un premier ordre a été déposé et on lance la boucle d'exécution. Dans celle-ci, on regarde si :

- La queue est vide. Si elle ne l'est pas c'est qu'un résultat de l'appel à la base de données y est stocké et on doit l'interpréter. Il est important de lancer ce script en début dans le cas de changement d'ordre ;
- Le robot est autorisé à se déplacer. S'il ne l'est pas, on calcule les vitesses des moteurs, sa position et arrête les moteurs. On réinitialise aussi l'autorisation pour le lancement d'un début de trajet ;
- Le chemin n'est pas vide et qu'on peut lancer un début de trajet. Alors, on calcule le temps de trajet pour atteindre la position et l'angle de rotation du robot. Le robot effectue ensuite la rotation. Quand la rotation est finie, l'autorisation de lancement de trajet est mise sur « Faux ». Le robot réinitialise ensuite les variables permettant le calcul de vitesse. Cette opération permet de ne pas prendre en compte l'avancement des compteurs dû à la rotation et de ne pas dépasser la longueur de stockage de la variable. On lance ensuite le déplacement et on sauvegarde l'heure ;

- On vérifie la distance à un obstacle. Si l'obstacle est situé à 7cm du robot, on envoie un message à l'utilisateur grâce à la base de données (l'envoi d'information a été testé et est fonctionnel tandis que la détection n'est pas encore implémentée) ;
- Le chemin n'est pas vide et que le temps de déplacement n'est pas dépassé (. On réalise la correction des moteurs afin d'atteindre la vitesse voulue et on calcule la position ;
- Le temps de déplacement est dépassé. On calcule les vitesses des moteurs, on les stoppe et on calcule la position. On autorise le lancement de trajet. On vérifie si la position est atteinte avec la précision posée par le cahier des charges ;
 - Si la position est atteinte. On enlève le point atteint du chemin à parcourir, on vérifie si tout le trajet est effectué. Si c'est le cas, on interdit le déplacement ;
 - Si la position n'est pas atteinte, on ne fait rien. Au retour du début de boucle, on repassera par le lancement de déplacement qui recalculera l'angle et le temps de trajet pour atteindre le point de manière plus précise.
- On doit appeler de nouveau la base de données. On a décidé de l'appeler toutes les 5 secondes afin de ne pas surcharger le système et de pouvoir effectuer la boucle assez rapidement.

Le script décrit par cette partie est fourni en annexe 10.

III - Communication

1. Objectifs

Le but de la communication est de trouver un moyen pour que chacune de nos parties (TurtleBot, second robot et interface utilisateur) puissent interagir entre elles. La communication doit pouvoir gérer plusieurs types de programmation car l'interface utilisateur est en Java alors que nos Raspberry Pi utilisent le langage Python.

2. Démarche

A. Recherche préliminaire

Dans un premier temps, il y eu un brainstorming afin de déterminer comment nous allions envoyer nos ordres. Nous sommes partis en premier lieu pour utiliser gRPC, un Framework open-source permettant de gérer plusieurs client-serveur sous plusieurs langages de programmation différents. Cependant, nous nous sommes très vite aperçus que l'apprentissage de ce Framework allait nous prendre un temps précieux. Nous avons donc finalement opté pour repartir sur une forme de communication que nous connaissons mieux : une base de données en langage SQL.

B. La base de données

Notre base de données (appelée plus communément BDD) est un tableau où chaque ligne est un ordre et chaque colonne un paramètre de cet ordre (cf. Annexe 11). Cette BDD est créée et stockée sur une « Raspberry PI Zero wh » grâce au système de gestion « MariaDB ». Nous pouvons y accéder car cette même Raspberry a été configurée en mode point d'accès, c'est-à-dire qu'elle devient son propre réseau auquel nous pouvons nous connecter.



Figure 41 : Carte Raspberry Pi Zero wh

Une fois nos machines connectées à ce réseau, nous pouvons effectuer deux actions : l'insertion et la lecture.

- L'insertion d'ordre consiste simplement à envoyer dans la BDD un ordre (rappel : une ligne dans un tableau). Principalement utilisée par l'interface utilisateur car des paramètres peuvent être demandés à ce dernier.
- La lecture de la BDD consiste à récupérer les ordres de cette dernière. Il convient de créer un programme capable de repérer si l'ordre est bien prévu pour cette machine avant de commencer à l'effectuer. La lecture s'effectue de manière régulière afin de pouvoir gérer les mises à jour d'ordres.

C. Communication SFTP

Nous pouvons donc interagir en envoyant des ordres dans notre base de données qui peuvent être lus. Mais il faut aussi que nous puissions envoyer et récupérer des fichiers comme par exemple la carte ou des fichiers texte sur le TurtleBot.

Pour cela nous allons utiliser le SFTP. C'est un protocole de transfert SSH permettant l'échange de fichier entre deux machines sur un même réseau. Cela ne demande pas une programmation compliquée, seulement une adaptation entre le langage python et Java (cf. Annexe 12).

3. Difficultés

Les deux principales difficultés rencontrées ont été la recherche de solution ainsi que la gestion du projet en ces temps troublés. Premièrement, trouver notre fonctionnement pour communiquer impliquait d'avoir une vision globale sur tout ce qu'on devrait pouvoir faire avec notre système, et anticiper quel genre de problème nous allions rencontrés. Deuxièmement,

en cette période de crise sanitaire, notre travail était plus souvent en télétravail de notre côté ce qui rendait le travail dessus plus compliqué à tester afin de tout faire fonctionner.

IV - Interface utilisateur

1. Objectifs

L'interface graphique (voir Annexe 13) se doit d'être un environnement ergonomique afin que l'utilisateur puisse de façon aisée pouvoir avoir accès aux outils mis en place pour interagir avec les robots. Dans le cadre de notre projet, il nous fallait une interface pour pouvoir envoyer les ordres dans la base de données facilement, une zone prévue pour afficher la carte créée par le TurtleBot ainsi qu'une zone de texte pour recevoir des informations complémentaires. De plus, cette interface devait être aisément installable sur un autre ordinateur pour une utilisation plus ouverte.

2. Démarche et résultats

Afin d'avoir une interface facilement exécutable sur plusieurs machines, il a été choisi de la faire sous Java avec la bibliographie Swing, en plus de l'avoir déjà pratiquée plusieurs fois au cours de ces dernières années d'études. Swing permet une implémentation rapide et facile d'actionneurs tels que des boutons ou des panels pour des affichages d'image. Comme introduit précédemment, notre interface graphique se décompose en 3 parties :

A. Commande

Cette zone est entièrement composée d'actionneurs de type bouton (cf. figure 42). Chacun permet d'envoyer un ordre spécifique à la base de données. Certains envoient un ordre direct (ex : « Afficher carte » affiche directement la carte après appui), d'autres demandent d'entrer des données supplémentaires à l'utilisateur (ex : « Pause » demande sur quels robots demander de mettre pause cf. figure 43).



Figure 42 : Partie commande de l'interface utilisateur



Figure 43 : Demande utilisateur pour le destinataire de l'ordre Pause

La description de chaque bouton se fait dans l'Annexe 14.

B. Carte

Étant donné que l'un des points clés de notre projet est la cartographie d'une pièce, il est évident que l'utilisateur doit pouvoir visualiser cette carte sur son interface. Il existe donc deux boutons dans la partie commande pour cette fonctionnalité. Le bouton « Sauvegarder Carte » récupère directement la carte au format « pgm » sur le TurtleBot via un transfert de fichier sftp. Le bouton « Afficher Carte », lui, permet comme son nom l'indique d'afficher la carte dans un Panel de l'interface utilisateur (cf. figure 44).

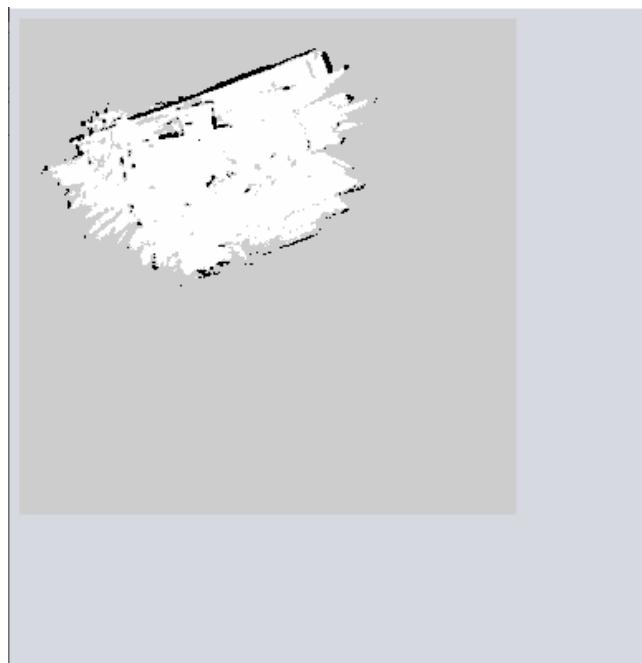


Figure 44 : Carte affichée sur l'interface utilisateur

Certaines commandes demandent d'interagir avec la carte pour envoyer un ordre (cf. figure 45).

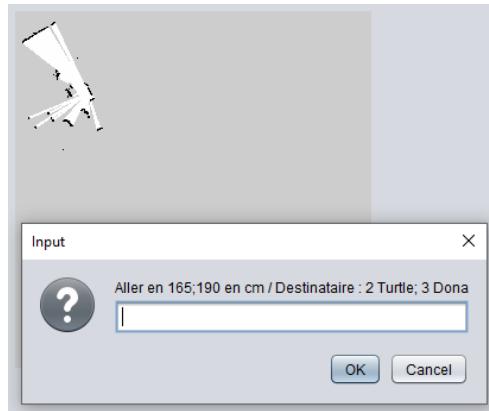


Figure 45 : « Enregistrer Destination » demande de cliquer sur la carte pour indiquer un point, avant d'enchaîner sur la suite à savoir le destinataire

C. Retour et notifications

En dessous des deux zones précédentes se trouve une zone de texte (cf. figure 46) qui permet à l'utilisateur de recevoir des notifications de ce qu'il se passe, que ce soit une confirmation de ce qui vient d'être fait, un retour d'un robot pour une erreur, ou une instruction supplémentaire particulière (cf. figure 47).



Figure 46 : Zone de texte pour les notifications



Figure 47 : On notifie l'utilisateur qu'il doit cliquer sur la carte pour continuer. On lui confirme ensuite l'ordre qu'il vient d'envoyer

3. Difficultés

La principale difficulté pour réaliser cette interface a été d'afficher la carte. Comme dit précédemment, c'est un format « pgm », un format peu usuel (du moins dans un cadre grand public). Il a donc fallu faire des recherches et comprendre comment ce format était codé et ainsi trouver comment l'adapter à un affichage en Java (voir Annexe 15 & Annexe 17). Pour ce qui est des ordres envoyés par la partie commande, il s'agit de commandes pour accéder à notre base de données SQL assez simple (voir Annexe 16).

Annexes

Annexe 1: Cahier des charges

FP1	Ordonner au robot ouvrier (Donatello) d'aller à une position donnée	FC1	Respecter l'environnement
FP2	Permettre l'interaction entre l'utilisateur et le robot par un serveur	FC2	Séduire l'acheteur
FP3	Communiquer à Donatello la carte et la destination par le biais du serveur	FC3	Se déplacer dans l'environnement d'utilisation
FP4	Contourner les objets non indiqués sur la carte	FC4	Alimenter le système avec une source d'énergie locale
		FC5	Communiquer les informations et la position du TurtleBot (Léonardo) au serveur

Fonction	Milieu mis en jeu	Verbe	Critère	Performance	Flexibilité
FP1	Utilisateur		Âge	À partir de 10 ans	0
			Précision de la position	<=10cm	1
			Vitesse en trajectoire rectiligne	Comprise entre 5km/h et 10km/h	0
	Donatello	Ordonner	Vitesse pour autres trajectoires	Inférieure à 5km/h	0
			Portée de la commande	Possible à 25m	0
			Position	Point sur carte	1
FP2	Serveur		Localisation	Sur un ordinateur	2
			Temps de communication	30s	1
			Interface	Web ou mobile	0
	Permettre		Coordonnées des robots	Coordonnées cartésiennes	0
			Affichage de la position du robot	Croix	1
			Lancer la cartographie	/	/
			Modifier la carte	Ajouter une interdiction	2
FP3	Milieu		Temps de communication	1 min	1
			Coordonées	Cartésiennes (dans ref carte)	0
	Contourner		Intérieur	Plat	0
			Superficie	?	?
FP4	Milieu		Attente de changement	30s	0
			Distance de signalement	20 cm	1
	Contourner		Se déplacer sous un meuble	Taille du robot inférieure à 30 cm	0
			Reconnaitre petits obstacles	Taille minimale reconnaissable : 5 cm	1
FP5	Source d'énergie		Portable	Batterie rechargeable	0
			Autonomie	30 min minimum	0
	Alimenter		Distance de communication	Possible à 25m	0
			Coordonées	Cartésiennes (dans ref carte)	0

Valeurs numériques potentiellement à rediscuter

Figure 48 : Cahier des charges

Annexe 2: savemap_on_turtlebot.py

```
import sys
import paramiko as pm
import os
from pynput import keyboard
import time
from progress.bar import IncrementalBar
sys.stderr = open('/dev/null') # Silence silly warnings from paramiko
sys.stderr = sys.__stderr__
class AllowAllKeys(pm.MissingHostKeyPolicy):
    def missing_host_key(self, client, hostname, key):
        return

HOST = '172.20.10.4'
USER = 'ubuntu'
PASSWORD = 'turtlebot'

client = pm.SSHClient()
client.load_system_host_keys()
client.load_host_keys(os.path.expanduser('~/ssh/known_hosts'))
client.set_missing_host_key_policy(AllowAllKeys())
client.connect(HOST, username=USER, password=PASSWORD)

channel = client.invoke_shell()
stdin = channel.makefile('wb')
stdout = channel.makefile('rb')

stdin.write('''
ls
export TURTLEBOT3_MODEL=burger
ros2 run nav2_map_server map_saver -f ~/maptest
''')

bar = IncrementalBar('Please wait while the map is saved', max=30)
for i in range(30):
    bar.next()
    time.sleep(1)
bar.finish()

stdout.close()
stdin.close()
client.close()
```

Annexe 3: dll_ssh.py

```
import sys
sys.stderr = open('/dev/null')           # Silence silly warnings from paramiko
import paramiko as pm
sys.stderr = sys.__stderr__
import os
from pynput import keyboard

class AllowAllKeys(pm.MissingHostKeyPolicy):
    def missing_host_key(self, client, hostname, key):
        return

HOST = '172.20.10.4'
USER = 'ubuntu'
PASSWORD = 'turtlebot'

client = pm.SSHClient()
client.load_system_host_keys()
client.load_host_keys(os.path.expanduser('~/ssh/known_hosts'))
client.set_missing_host_key_policy(AllowAllKeys())
client.connect(HOST, username=USER, password=PASSWORD)

sftp=client.open_sftp()
localpath='maptest.pgm'
remotepath='maptest.pgm'
sftp.get(localpath, remotepath)
sftp.close()
client.close()
```

Annexe 4: Algorithme A*

L'idée de l'algorithme A* est très simplement sur [un article du site Medium par Nicolas Swift](#) (<https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2>):

- On note G la distance entre le nœud courant et le point de départ.
 - On note H la distance heuristique, tel que $H=\text{distance}_x^2+\text{distance}_y^2$
 - On note F, tel que $F=H+G$.
1. Ajouter le nœud de départ à la liste ouvertes
 2. Répétez l'algorithme suivant :
 - a) Recherchez le nœud de coût F le plus bas dans la liste ouverte. Il s'agit du nœud courant.
 - b) Basculez vers la liste fermée.
 - c) Pour chacune des 8 cases adjacentes au nœud courant...
 - Si elle n'est pas accessible à pied ou si elle figure sur la liste fermée, ignorez-la. Sinon, procédez comme suit.
 - Si elle ne figure pas dans la liste ouverte, ajoutez-la à la liste ouverte. Faites du nœud courant le parent de cette case. Enregistrez les coûts F, G et H du carré.

- Si elle est déjà sur la liste ouverte, vérifiez si le chemin vers cette case est meilleur, en utilisant le coût G comme mesure. Un coût G inférieur signifie que c'est une meilleure voie. Si tel est le cas, remplacez le parent du carré par le carré actuel et recalculez les scores G et F du carré.

d) Arrêtez-vous :

- Lorsque vous ajoutez le carré cible à la liste fermée, auquel cas le chemin a été trouvé
- S'il est impossible de trouver le carré cible et la liste ouverte est vide. Dans ce cas, il n'y a pas de chemin.

3. Enregistrez le chemin. En reculant à partir de la case cible, passez de chaque case à sa case parent jusqu'à ce que vous atteigniez la case de départ. C'est votre chemin.

Algorithme :

```
In [9]: import numpy as np

class Node:
    """
        A node class for A* Pathfinding
        parent is parent of the current Node
        position is current position of the Node in the maze
        g is cost from start to current Node
        h is heuristic based estimated cost for current Node to end Node
        f is total cost of present node i.e. : f = g + h
    """

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position

#This function return the path of the search
def return_path(current_node):
    path = []
    current = current_node
    while current is not None:
        path.append(current.position)
        current = current.parent
    return path[::-1] # Return reversed path

def search(maze, cost, start, end):

    if(maze[start[0],start[1]]==1 or maze[end[0],end[1]]==1):
        print("Impossible c'est un mur")
        return None

    # Create start and end node with initialized values for g, h and f
    start_node = Node(None, tuple(start))
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, tuple(end))
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both yet_to_visit and visited list
    # in this list we will put all node that are yet_to_visit for exploration.
    # From here we will find the lowest cost node to expand next
    yet_to_visit_list = []
    # in this list we will put all node those already explored so that we don't explore it again
    visited_list = []

    # Add the start node
    yet_to_visit_list.append(start_node)

    # Adding a stop condition. This is to avoid any infinite loop and stop
    # execution after some reasonable number of steps
    outer_iterations = 0
    max_iterations = (len(maze) // 2) ** 10
    #max_iterations=20000

    # what squares do we search . serach movement is left-right-top-bottom
    #(4 movements) from every positon
    move = [[-1, 0], # go up
            [0, -1], # go left
            [1, 0], # go down
            [0, 1]] # go right

    #find maze has got how many rows and columns
    no_rows, no_columns = np.shape(maze)
```

```

#find maze has got how many rows and columns
no_rows, no_columns = np.shape(maze)

# Loop until you find the end
while len(yet_to_visit_list) > 0:

    # Every time any node is referred from yet_to_visit list, counter of limit operation incremented
    outer_iterations += 1

    # Get the current node
    current_node = yet_to_visit_list[0]
    current_index = 0
    for index, item in enumerate(yet_to_visit_list):
        if item.f < current_node.f:
            current_node = item
            current_index = index

    # if we hit this point return the path such as it may be no solution or
    # computation cost is too high
    if outer_iterations > max_iterations:
        print ("giving up on pathfinding too many iterations")
        return return_path(current_node)

    # Pop current node out off yet_to_visit list, add to visited list
    yet_to_visit_list.pop(current_index)
    visited_list.append(current_node)

    # test if goal is reached or not, if yes then return the path
    if current_node == end_node:
        return return_path(current_node)

    # Generate children from all adjacent squares
    children = []

    for new_position in move:

        # Get node position
        node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

        # Make sure within range (check if within maze boundary)
        if (node_position[0] > (no_rows - 1) or
            node_position[0] < 0 or
            node_position[1] > (no_columns -1) or
            node_position[1] < 0):
            continue

        # Make sure walkable terrain
        if maze[node_position[0]][node_position[1]] != 0:
            continue

        # Create new node
        new_node = Node(current_node, node_position)

        # Append
        children.append(new_node)

    # Loop through children
    for child in children:

        # Child is on the visited list (search entire visited list)
        if len([visited_child for visited_child in visited_list if visited_child == child]) > 0:
            continue

        # Make sure this child is not in the yet_to_visit list
        if child not in yet_to_visit_list:
            yet_to_visit_list.append(child)

```

Annexe 5: Calcul des points proches des zones à explorer

```

pts_a_explorer.py
 1 ## Imports utiles
 2
 3 import cv2 as cv
 4 import numpy as np
 5 import matplotlib.pyplot as plt
 6 import tifffile as tiff
 7 from PIL import Image
 8
 9 ## Définition des fonctions
10
11 # Fonction qui calcule l'accessibilité d'un point pour certaines coordonnées : on lui donne des coordonnées et si ce point possède
12 # des murs autour de lui pour un certain rayon, on renvoie False. Ceci évite d'ordonner au robot de passer par une porte par
13 # exemple
14 rayon = 4;
15 def carte_accessibilite(x, y, rayon):
16     if(map_grise[x][y] == 255): # Si le point est une zone accessible, on regarde si autour de lui il y a des murs (pour un
17         # certain rayon choisi arbitrairement)
18         for m in range(-rayon,rayon+1):
19             for n in range(-rayon,rayon+1):
20                 if(map_grise[x+m][y+n] == 0):
21                     return False # S'il y a un mur, on renvoie False
22             return True # Sinon on renvoie True
23     elif(map_grise[x][y] == 150): # Renvoie False si le point est un point inconnu, car le but est de regarder les points
24         decouverts qui ne sont pas accessibles réellement par le robot
25     return False
26 else :
27     return False # Si le point est un mur, on renvoie False, puisque l'on ne peut pas aller dans un mur !
28
29 # Fonction qui permet de "diffuser" l'accessibilité de points. On part d'un point dans la pièce, et on regarde si tout ce qui est
30 # autour est accessible. Ceci permet de calculer l'accessibilité des points et d'éliminer les points qui se trouvent derrière les
31 # murs de la pièce !
32
33 def accessibilite_diff(): # Diffuse la zone d'accessibilité en prenant en compte les points considérés comme non accessibles
34     pile=[]
35     milieu_l = int(l/2);
36     milieu_h = int(h/2);
37     for i in range(-3,4):
38         for j in range(-3,4):
39             pile.append([milieu_l+i,milieu_h+j]) # On part d'un point dans la carte, que l'on sait accessible (par exemple le centre
40             # de la pièce où on a mis le robot au début
41     while pile != []:
42
43
44 [x,y]=pile.pop()
45 map_grise[x,y] = 10 # La valeur arbitraire "10" correspond à une zone accessible pour le robot
46 for k in range(-1,2):
47     for j in range(-1,2): # On regarde si chaque pixel autour du pixel considéré est accessible
48         if(k != 0 or j != 0):
49             if carte_accessibilite(x+k,y+j,rayon) == True :
50                 pile.append([x+k,y+j])
51
52 return
53
54 def a_explorier(x,y): # On détermine si le point considéré est à explorer
55     if(map_grise[x,y] == 10): # On regarde si le point est accessible par le robot et on renvoie False si ce n'est pas le cas
56         for k in range(-1,2):
57             for j in range(-1,2): # On regarde les pixels adjacents
58                 if(k != 0 or j != 0):
59                     if(map_grise[(x+k),(y+j)] == 150): # Si un des pixels adjacents est inconnu, on retourne True
60                         print("Bordure trouvée")
61                         return True
62             return False
63 else:
64     return False
65
66 ## Calcul des points à explorer
67
68 img = cv.imread('maptest.pgm') # On lit la carte téléchargée
69 global l
70 global h
71 (l, h) = img.shape[0], img.shape[1]
72
73 map_grise = np.zeros((l, h));
74
75 for x in range(l):
76     for y in range(h):
77         valeur = img[x,y][0]
78         if (valeur >= 250 and valeur <= 255):
79             map_grise[x,y] = 255 # Zone découverte
80         elif (valeur >= 0 and valeur <= 5):
81             map_grise[x,y] = 0 # Mur
82         elif (valeur > 5 and valeur < 250):
83             map_grise[x,y] = 150 # Pixel inconnu
84
85 remplissage_diff() # On cherche les zones accessibles pour le robot
86
87 global points_a_explorier = [] # On calcule les points à aller explorer
88 for i in range(l):
89     for j in range(h):
90         if a_explorier(i,j):
91             points_a_explorier.append([i,j])

```

Annexe 6: Envoi du robot dans les zones à explorer

```
envoi_goal.py
```

```
envoi_goal.py No Selection
1 import sys
2 import paramiko as pm
3 import os
4 from pynput import keyboard
5 import time
6 from progress.bar import IncrementalBar
7
8 sys.stderr = open('/dev/null') # Silence silly warnings from paramiko
9 sys.stderr = sys._stderr_
10
11 class AllowAllKeys(pm.MissingHostKeyPolicy):
12     def missing_host_key(self, client, hostname, key):
13         return
14
15 ## Connexion SSH
16 HOST = '172.20.10.4'
17 USER = 'ubuntu'
18 PASSWORD = 'turtlebot'
19
20 client = pm.SSHClient()
21 client.load_system_host_keys()
22 client.load_host_keys(os.path.expanduser('~/ssh/known_hosts'))
23 client.set_missing_host_key_policy(AllowAllKeys())
24 client.connect(HOST, username=USER, password=PASSWORD)
25
26 channel = client.invoke_shell()
27 stdin = channel.makefile('wb')
28 stdout = channel.makefile('rb')
29
30 ## Envoi du robot au point désiré
31
32 stdin.write('''
33 export TURTLEBOT3_MODEL=burger
34 ros2 topic pub /goal_pose geometry_msgs/PoseStamped "{header: {stamp: {sec: 0}, frame_id: 'map'}, pose: {position: {x: '''+pts_a_explorer[line][0]+''', y: '''+pts_a_explorer[line][1]+''', z: 0.0}, orientation: {w: 1.0}}}"
35 ''')
36
37
38
```

Annexe 7: Exemple de dimensions de la plateforme

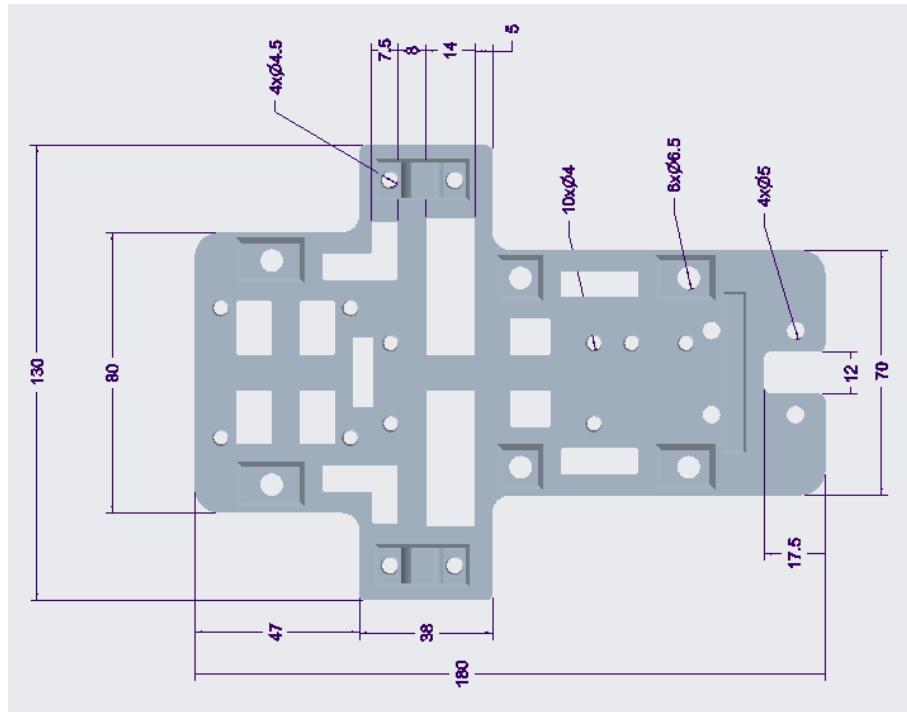


Figure 49 : Dimensions du premier étage

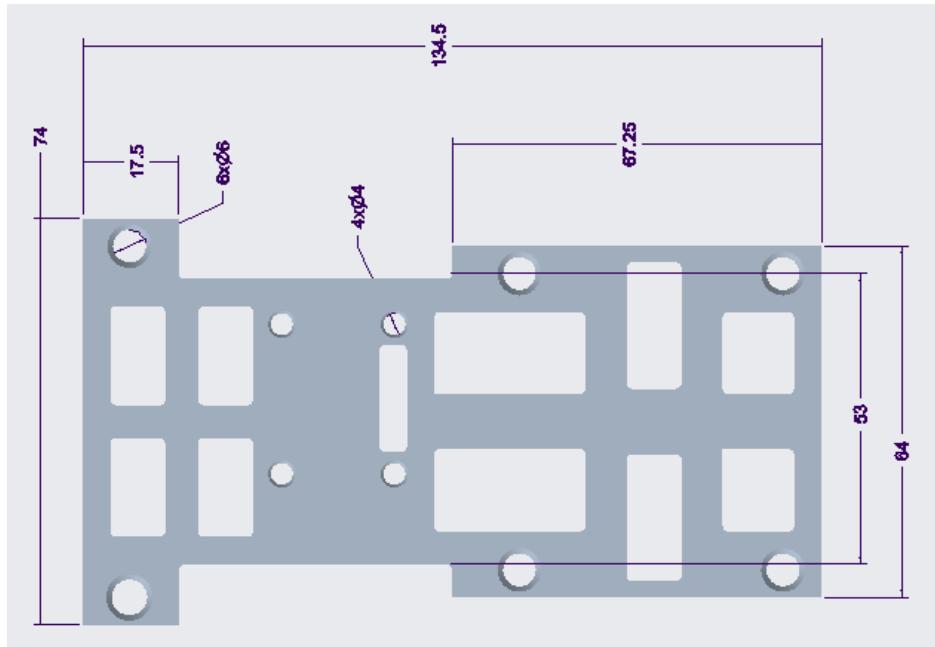


Figure 50 : Dimensions du second étage

Annexe 8: Code du codeur fournissant la vitesse

```

def correcteurG():
    global vitesseG, moyenneG, tG, previousG, posG, previousPosG, erreurG, sommeErreurG, KpG, KiG, consigneG, pwmG, moteurGPositif
    tG=time.time_ns()
    #print(posG)
    vitesseG=1e9*(posG-previousPosG)/(tG-previousG)
    vitesseG=vitesseG*60/700
    previousG=tG
    previousPosG=posG
    if(moteurGPositif==False):
        vitesseG=-vitesseG
    moyenneG,vitesseG=moyenneGlissante(moyenneG,vitesseG)
    erreurG=consigneG-vitesseG
    sommeErreurG=sommeErreurG+erreurG
    #print('Vitesse G: '+str(vitesseG))
    #print('Erreur :'+str(erreurG))
    #print('SommeErreur :'+str(sommeErreurG))
    commandSpeedG=erreurG*KpG+sommeErreurG*KiG
    if(commandSpeedG>102):
        commandSpeedG=102
    if(commandSpeedG<-102):
        commandSpeedG=-102
    #print(math.floor(commandSpeedG*100/141))
    pwmG.ChangeDutyCycle(math.floor(commandSpeedG*42/102))

def frontG(self):
    #print("front")
    global moteurGEncoder1, moteurGEncoder2, posG
    posG=posG+1
    ##Le compteur ne fonctionne pas bien avec les 2 voies
    ##On ne compte donc qu'avec une
    ##Le sens de rotation est connu de base donc ça n'est pas un problème
    ##Ce code se lançant à chaque front, on compte chaque front
    #sa=GPIO.input(moteurGEncoder1)
    #sb=GPIO.input(moteurGEncoder2)
    #if(sa==sb):
    #    posG=posG+1
    #else :
    #    posG=posG-1
    #print(posG)

    GPIO.add_event_detect(moteurGEncoder1,GPIO.BOTH,callback=frontG)

```

Figure 51 : Code de vitesse moteur

L'événement à détecter est spécifié à l'initialisation des pins de la Raspberry.

Annexe 9: Code du gyroscope

```
def getYaw():
    global GzCal
    Gz=0
    #GzCal=-1.9
    for i in range(300):
        Gz=Gz+read_word_2c(0x47)/131.0-GzCal
    Gz=Gz/300
    print("Gz="+str(Gz))
    return Gz

def calibrateMPU():
    global GxCal, GyCal, GzCal
    z=0
    for i in range(10000):
        #x=x+read_word_2c(0x43)
        #y=y+read_word_2c(0x45)
        z=z+read_word_2c(0x47)
    #x=x/100
    #y=y/100
    z=z/10000
    #GxCal=x/131.0
    #GyCal=y/131.0
    GzCal=z/131.0
    print(GzCal)

def turn_right(speed,alpha):
    global currentAngle
    start=time.time()
    arret=0
    forward_mg(speed)
    reverse_md(speed)
    #print(alpha)
    while(arret==0):
        currentYaw=-getYaw()
        elapse=time.time()-start
        currentAngle=currentAngle+(elapse*currentYaw)
        start=time.time()
        print('Angle :'+str(currentAngle))
        if currentAngle<-180:
            currentAngle=currentAngle+360
        elif currentAngle>180:
            currentAngle=currentAngle-360
        if(currentAngle>=alpha-1):
            stop()
            arret=1
```

Annexe 10: Boucle infinie

```

##boucle infinie
init()
pwmG=GPIO.PWM(moteurGpwm,100)
pwmD=GPIO.PWM(moteurDpwm,100)
pwmG.start(0)
pwmD.start(0)

#while executeDep=True autorise le déplacement du robot
if __name__=='__main__':
    queueBDD=multiprocessing.Queue()
    process=multiprocessing.Process(target=recupererBDD,args=(queueBDD,))
    while(process.is_alive()):
        time.sleep(0.1)
        dernierTpBDD=time.time()
        print(Xactuel)
        print(Yactuel)
        print(currentAngle)
        if(queueBDD.empty()==False):
            interpreterBDD(queueBDD.get())
    while True:
        #time.sleep(1)
        #print(' new boucle')
        #print(process.is_alive())
        #print(Xactuel)
        #print(Yactuel)
        #print(currentAngle)
        if(queueBDD.empty()==False):
            #si je n'ai pas de message de la bdd, je ne fais rien
            interpreterBDD(queueBDD.get())
            print('interpreter')
        if(executionDep==False):
            #si mon déplacement n'est pas autorisé je me stop
            correcteurG()
            correcteurD()
            calculPosition(vitesseG,vitesseD)
            stop()
            rotationFini=False #pour la reprise après une pause de bdd
        if(path.any() and executionDep and (rotationFini==False)):
            # si j'ai un chemin à effectué et que mon déplacement est autorisé
            # et que je ne suis pas déjà tourner
            # je calcule l'angle à atteindre ainsi que le temps de déplacement
            tpsDep,teta=calculTempsTeta(path,speedMoyMS)
            #je fais ma rotation
            print('rotate : '+str(teta))
            rotationFini=rotation(teta,speedMoyRPM)
            #je réinitialise mes compteur pour le correcteur
            time.sleep(0.1)
            posG=0

            posD=0
            previousPosG=0
            previousPosD=0
            previoustG=time.time_ns()
            previoustD=time.time_ns()
            moyenneG=np.zeros(3)
            moyenneD=np.zeros(3)
            #correcteurG()
            #correcteurD()
            #je lance mon déplacement
            heureDep=time.time()
            forward(speedMoyRPM)
        if(path.any() and executionDep and (time.time()-heureDep<tpsDep+0.1)):
            # si je n'ai pas fait le temps de déplacement nécessaire,
            # je corrige mes vitesses et calcule ma position
            print(' dep en cours')
            correcteurG()
            correcteurD()
            calculPosition(vitesseG,vitesseD)
            print(Xactuel)
            print(Yactuel)
            print(currentAngle)
            #time.sleep(0.05)
        if(path.any() and executionDep and (time.time()-heureDep>tpsDep+0.1)):
            # si j'ai dépassé mon temps de déplacement, je calcule mes vitesses et ma position
            correcteurG()
            correcteurD()
            stop()
            calculPosition(vitesseG,vitesseD)
            print("pause")
            # je réinitialise mon flag de rotation
            rotationFini=False
            if(isPositionOK(path)):
                print('pos ok')
                # si j'ai atteint ma position avec la précision nécessaire
                # je définit mon nouveau chemin total
                path=enlevePointAtteint(path)
                # je regarde sur le trajet est fini
                test=trajetFini(path) # a réaliser l'appel à la bdd si nécessaire
                if(test==True):
                    # si tous les déplacements sont faits, j'annule mon autorisation de déplacement
                    # il est autorisé à l'appel de la bdd si je dois de nouveau me déplacer
                    executionDep=False
            if(time.time()>dernierTpBDD+5):
                #appel bdd toutes les 5 sec
                print('Lecture bdd')
                recupererBDD(queueBDD)

```

```

#process=multiprocessing.Process(target=recupererBDD,args=(queueBDD,))
#Lors de premier test, le résultat de la bdd ne rentrait pas dans le queue
#Tant que cela n'a pas été corrigé, on lance la lecture de bdd dans le process principal
dernierTpsBDD=time.time()
time.sleep(0.05)

```

Annexe 11: Forme de la base de données

Paramètre	Heure	Destinataire	Etat	Type d'ordre	syntaxeOrdre
Type	temps	int	int	int	String
Exemples	15:05:41	3	0	4	50.5;24
	18:15:22	0	1	2	

Heure : indique l'heure (horloge Raspberry PI) à laquelle l'ordre est arrivé.

Destinataire : entier donnant à qui est destiné l'ordre.

Destinataire
0 utilisateur
1 serveur
2 TurtleBot
3 Donatello

Etat : entier donnant à quel stade d'avancement est l'ordre.

Etat
0 pas fait
1 en cours
2 en pause
3 annulé
4 fait

Type d'ordre : entier déterminant le type d'ordre à effectuer. Cela sous-entend que c'est au receveur d'ordre d'interpréter ce chiffre pour lancer les bons processus.

SyntaxeOrdre : inclut les paramètres nécessaires pour l'ordre, peut être vide si l'ordre n'en requiert pas.

Annexe 12: Exemple de connexion SSH Python et Java

PYTHON :

```
import paramiko
ssh_client=paramiko.SSHClient()
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh_client.connect(hostname='172.20.10.4',username='ubuntu',password='turtlebot')

ftp_client=ssh_client.open_sftp()
localpath='maptest.pgm'
remotepath='maptest.pgm'
ftp_client.get(remotepath,localpath)
ftp_client.close()
```

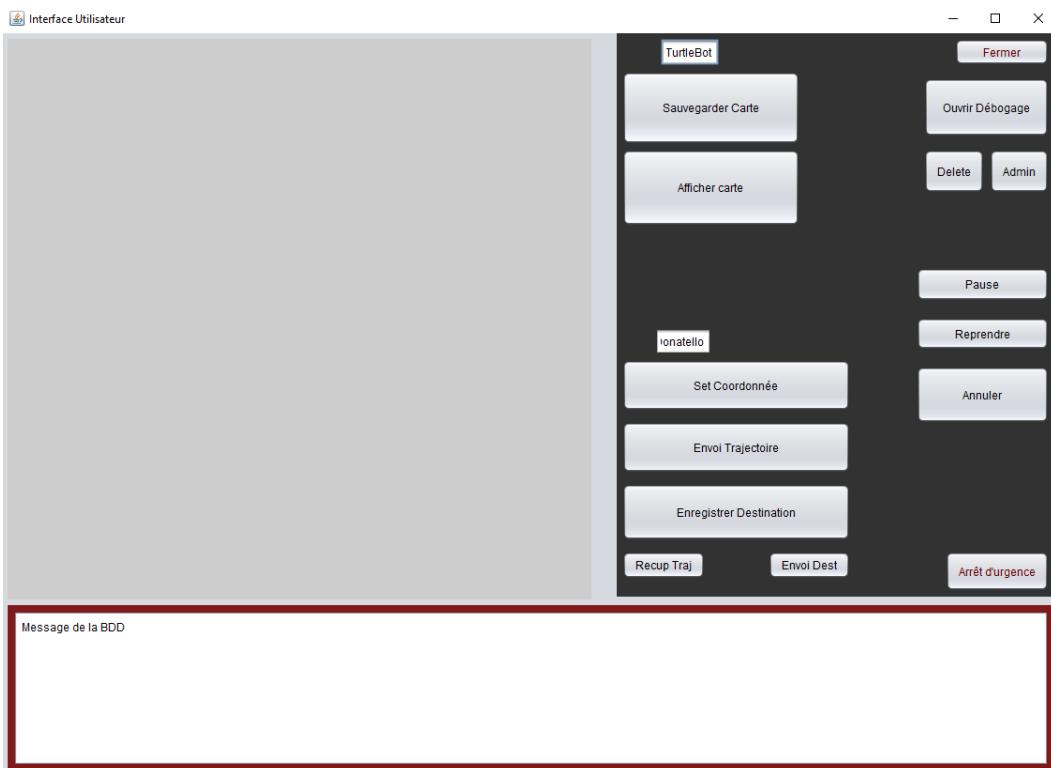
JAVA :

```
public static void main(String[] args) {
    // TODO code application logic here

    String user = "ubuntu";
    String password = "turtlebot";
    String host = "172.20.10.4";
    int port = 22;
    String remoteFile = "path.txt";
    String localFile = "path.txt";
    try {
        JSch jsch = new JSch();
        Session session = jsch.getSession(user, host, port);
        session.setPassword(password);
        session.setConfig("StrictHostKeyChecking", "no");
        System.out.println("Establishing Connection...");
        session.connect();
        System.out.println("Connection established.");
        System.out.println("Crating SFTP Channel.");
        ChannelSftp sftpChannel = (ChannelSftp) session.openChannel("sftp");
        sftpChannel.connect();
        System.out.println("SFTP Channel created.");

        sftpChannel.get(localFile,remoteFile);
        sftpChannel.disconnect();
        session.disconnect();
    } catch (JSchException | SftpException e) {
        e.printStackTrace();
    }
}
```

Annexe 13: Interface utilisateur complète



Annexe 14: Description de la partie commande

Sauvegarder Carte : Permet de récupérer le fichier au format pgm de la carte du TurtleBot.

Afficher Carte : Affiche la carte sur la zone grise de gauche.

Set Coordonnée : Change les coordonnées du second robot.

Enregistrer Destination : Enregistre un point de destination en cliquant sur la carte dans un fichier texte.

Envoi Dest : Envoie le fichier texte de 'Enregistrer Destination' au TurtleBot.

Recup Traj : Récupère le fichier texte de la trajectoire jusqu'au point de destination depuis le TurtleBot.

Envoi Trajectoire : Demande au second robot d'aller à la destination voulue via la trajectoire obtenue du TurtleBot.

Pause : Met un ordre en pause.

Annuler : Annule un ordre

Arrêt d'urgence : Arrête tous les ordres de la BDD.

Fermer : Ferme l'interface et les connexions ouvertes.

Ouvrir Débogage : Ouvre une nouvelle fenêtre qui affiche le contenu de la BDD en temps réel. Un second appui permet de la fermer.

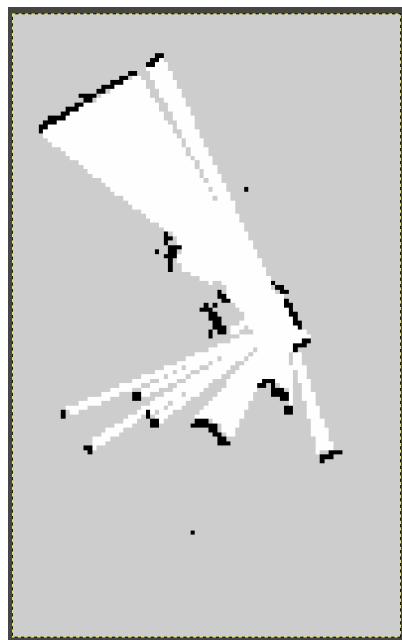
Delete : Supprime toutes les données de la BDD.

Admin : Permet d'envoyer un ordre en choisissant tous ces paramètres.

Annexe 15: Format « pgm »

Le format « pgm » est un format d'image pouvant être interpréter de plusieurs façons. Voici comme celui que nous fournit le TurtleBot fonctionne.

Voici une image type pgm :



C'est une image en niveau de gris. Son code interne est composé de deux parties, un Header et le descriptif de l'image.

Le Header de cette image est :

P5
87 140
255

P5 indique la manière dont est interprété la suite ; 87 et 140 indique la largeur et hauteur de l'image en pixel, 255 indique sur combien de niveau est codé le gris.

Le descriptif de l'image est codé en langage ASCII sur une seule ligne, décrivant chaque pixel en partant du haut-gauche, ligne par ligne. Chaque pixel est associé à une valeur de niveau de gris (0 à 254) qui a un équivalent en caractère ASCII. Par exemple, un pixel blanc (niveau de gris 254) sera codé par le caractère ASCII 254 soit **b**.

Annexe 16: Exemple de connexion SQL sous Java

```
try {
    PreparedStatement requete = connexion.prepareStatement ("INSERT INTO ordre VALUES (NOW(),?, ?, ?, ?)");
    requete.setInt(1,2); /*destinataire*/
    requete.setInt(2,0); /*etat*/
    requete.setInt(3,2); /*type d'ordre*/
    requete.setString(4,Syntx); /*syntaxe de l'ordre*/
    requete.executeUpdate();
} catch (SQLException ex) {
    Logger.getLogger(InterfaceUser.class.getName()).log(Level.SEVERE, null, ex);
    textAreaLogBdd.append(ex + System.lineSeparator());
}
```

Annexe 17: Programme pour afficher un format pgm dans un label

```
import java.awt.Frame;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.MemoryImageSource;
import java.io.IOException;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

/**
 *
 * @author loict
 */
public class AfficherMap /*extends Frame*/{

    private JLabel jlab;

    public AfficherMap( String name,BytePixmap p){
        // fabrication des pixels gris au format usuel AWT : ColorModel.RGBdefault
        int[] pixels = new int[p.size];
        for (int i = 0; i < pixels.length; i++)
            pixels[i] = 0xFF000000 + Pixmap.intValue(p.data[i]) * 0x010101; // réplique l'octet 3 fois
        // construit une image avec ces pixels
        MemoryImageSource source = new MemoryImageSource(p.width, p.height, pixels, 0, p.width);
        Image img = Toolkit.getDefaultToolkit().createImage(source);
        ImageIcon ico = new ImageIcon(img);
        jlab = new JLabel(ico);
        jlab.setBounds(0,0, p.width,p.height);
        this.jlab = jlab;
    }

    public AfficherMap( String filename) throws IOException {
        this( filename, new BytePixmap(filename));
    }

    public JLabel label(){
        return(this.jlab);
    }
}
```

Ce programme fait appel à d'autres classes trouvées sur ce site :

https://www.enseignement.polytechnique.fr/informatique/profs/Philippe.Chassagnet/PGM/pgm_java.html