Rapport DS ROS

Evaluation pratique

1. Préparation de l'environnement

Réponse aux questions

1.1 :

ROS (Robot Operating System), est un outils open source rassemblant de nombreuses bibliothèques et modules permettant de controler ou de s'interfacer avec des robots. Il s'agit d'une couche haut niveau permettant aux développeurs d'utiliser des fonctions simples permettant d'activer certains fonctions d'un robot. C'est surtout utilisé dans le cas de robots mobiles, mais peut être utilisé pour d'autres types de robots.

1.2 :

ROS1 fonctionnait avec un noeud **maitre** (roscore) avec lequel devaient interagir tous les autres noeuds de notre application. Sur ROS2, ce n'est plus le cas, tous les noeuds, qu'ils soient développés par le programmeur ou non, communiquent tous entre eux en même temps. Faisons un bref tour de quelques avantages et inconvénients de chaque version :

ROS1	ROS2
Linux uniquement	Linux, Windows, Mac
Deux langages de programmation disponibles (C++ et Python)	Plusieurs langages de programmation disponibles (C++, Python, Javascript, Go, Rust, C#)

1.3 :

Il faut lancer la commande `source devel/setup.bash` pour que le package soit ajouté au path de ROS. Cela permet à ROS de trouver le package et de pouvoir l'utiliser. Dans le cas contraire, ROS ne pourra pas utiliser le package, puisque ce dernier ne se trouvera pas dans ses dossiers de recherche.

1.4 :

Un Topic est un élément auquel un noeud peut s'abonner ou sur lequel il peut publier des informations. On peut aussi interagir avec un Topic par le biais du terminal avec la commande `rostopic pub <topic> <type> <message>`. Ils permettent de faire communiquer des informations entre les noeuds. Par exemple, un topic `/takeoff` permet de faire décoller un drone (bebop par exemple) et un topic `/land` permet de le faire atterir. On utilisera respectivement les commandes suivantes pour le faire décoller et atterrir : `rostopic pub /takeoff std_msgs/Empty "{}"` et `rostopic pub /land std_msgs/Empty "{}"`.

Les services ROS permettent de faire communiquer des noeuds entre eux. Ils permettent de faire des requêtes et d'y répondre. Par exemple, on peut demander à un noeud de faire décoller un drone, et ce

dernier nous répondra si la requête a été acceptée ou non. On peut utiliser la commande `rosservice call <service> <message>` pour faire une requête à un service. Par exemple, pour faire décoller un drone, on utilisera la commande `rosservice call /bebop/takeoff "{}"`.

1.5 :

Commandes entrées pour créer un workspace :

```
$ mkdir -p ds_ros_batiste.laloi_ws_2022_2023/src
$ cd ds_ros_batiste.laloi_ws_2022_2023/
$ source /opt/ros/melodic/setup.bash
$ catkin_make
```

Commande entrée pour démarrer le noeud `turtlesim` :

```
# Dans un premier terminal :
$ roscore

# Dans un second terminal :
$ rosrun turtlesim turtlesim_node
```

1.6 :

Afficher la liste des topics disponibles :

```
# Dans un terminal autre que celui du roscore et du turtlesim_node :
$ rostopic list
```

Résultats :

```
$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

1.7 :

Connaitre le type de message du topic `/turtle1/pose` :

```
$ rostopic info /turtle1/pose
```

Résultats :

```
$ rostopic info /turtle1/pose
Type: turtlesim/Pose

Publishers:
 * /turtlesim (http://tpres15.cpe.lan:38385/)

Subscribers: None
```

Ce topic utilise donc des message de type `turtlesim/Pose`, une sorte de JSON avec des arguments de positions, d'angle et de vitesse linéaire/angulaire, voici un exemple de message de ce type :

```
$ rostopic echo /turtle1/pose
x: 5.544444561
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
```

1.8 :

Determiner les bornes de l'environnement de la tortue :

```
# Dans un terminal autre que celui du roscore et du turtlesim_node :
$ rosrun turtlesim turtle_teleop_key

# Dans un second terminal :
$ rostopic echo /turtle1/pose
```

Le premier nous permet de controler la tortue avec les touches directionnelles du clavier, et le second nous permet de voir en direct l'évolution de la position de la tortue

Voici par exemple le résultat lorsque la tortue est dans le coin supérieur gauche de la fenetre :

```
x: 0.0
y: 11.088889122
```

En se déplaçant dans les quatres coins on peut établir les coordonnées maximales que peut avoir la tortue :

Point	valeur
min_x	0

Point	valeur
max_x	11.088889122
min_y	0
max_y	11.088889122

2. Distance aux bords

Création d'un package `distance_mng`

```
$ catkin_create_pkg distance_mng std_msgs turtlesim rospy roscpp
```

2.1 : Subscribe to Pose

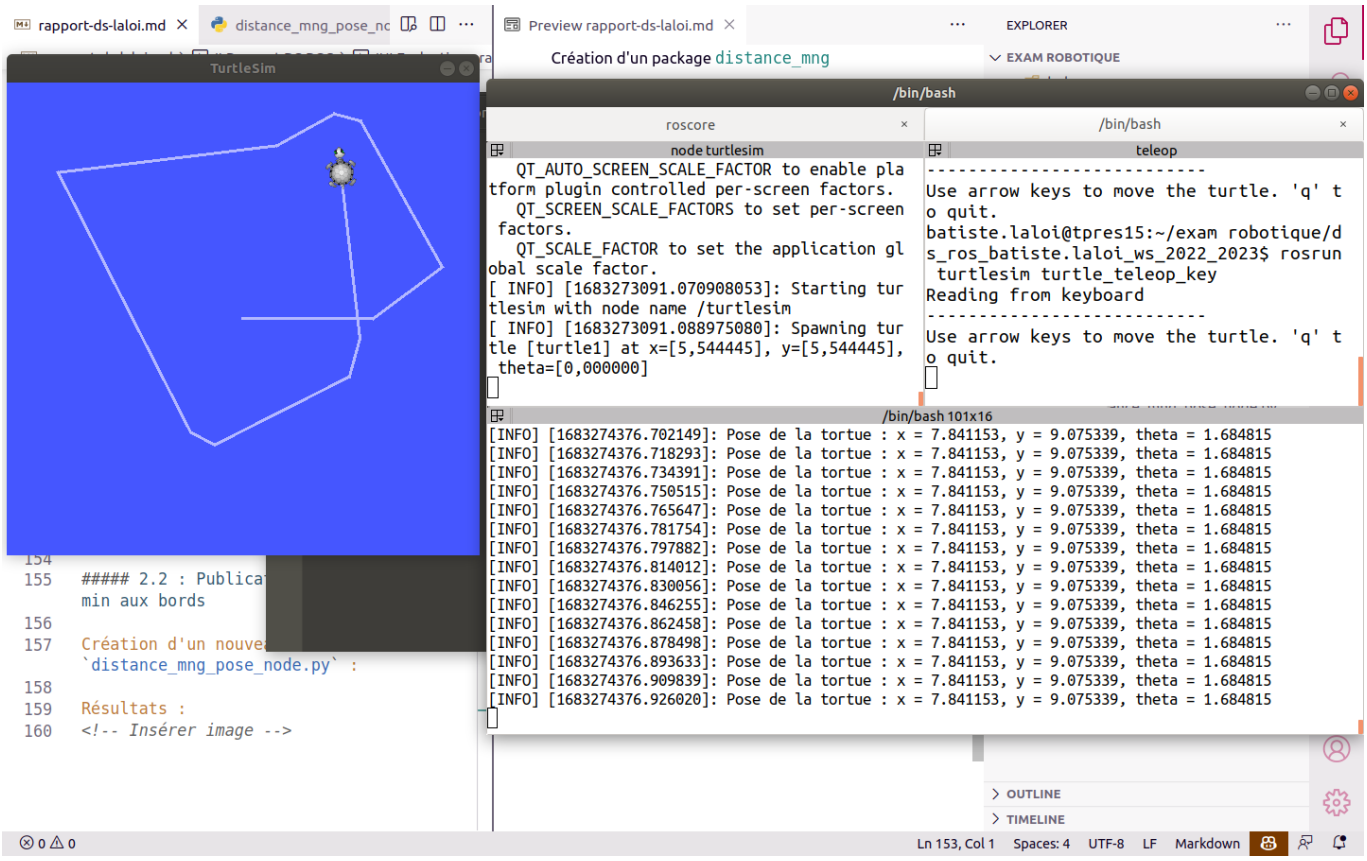
Nom du package : `distance_mng`

Nom du noeud : `distance_mng_pose_node_v0.py`

Fonctions réalisées par le noeud :

- Le noeud permet d'afficher en continue la position de la tortue dans la fenêtre, grâce à un Subscribe sur le topic `/turtle1/pose`

Capture montrant le comportement :



rqt_graph montrant le comportement du noeud :

2.2 : Publication de la distance min aux bords

Nom du package : `distance_mng`

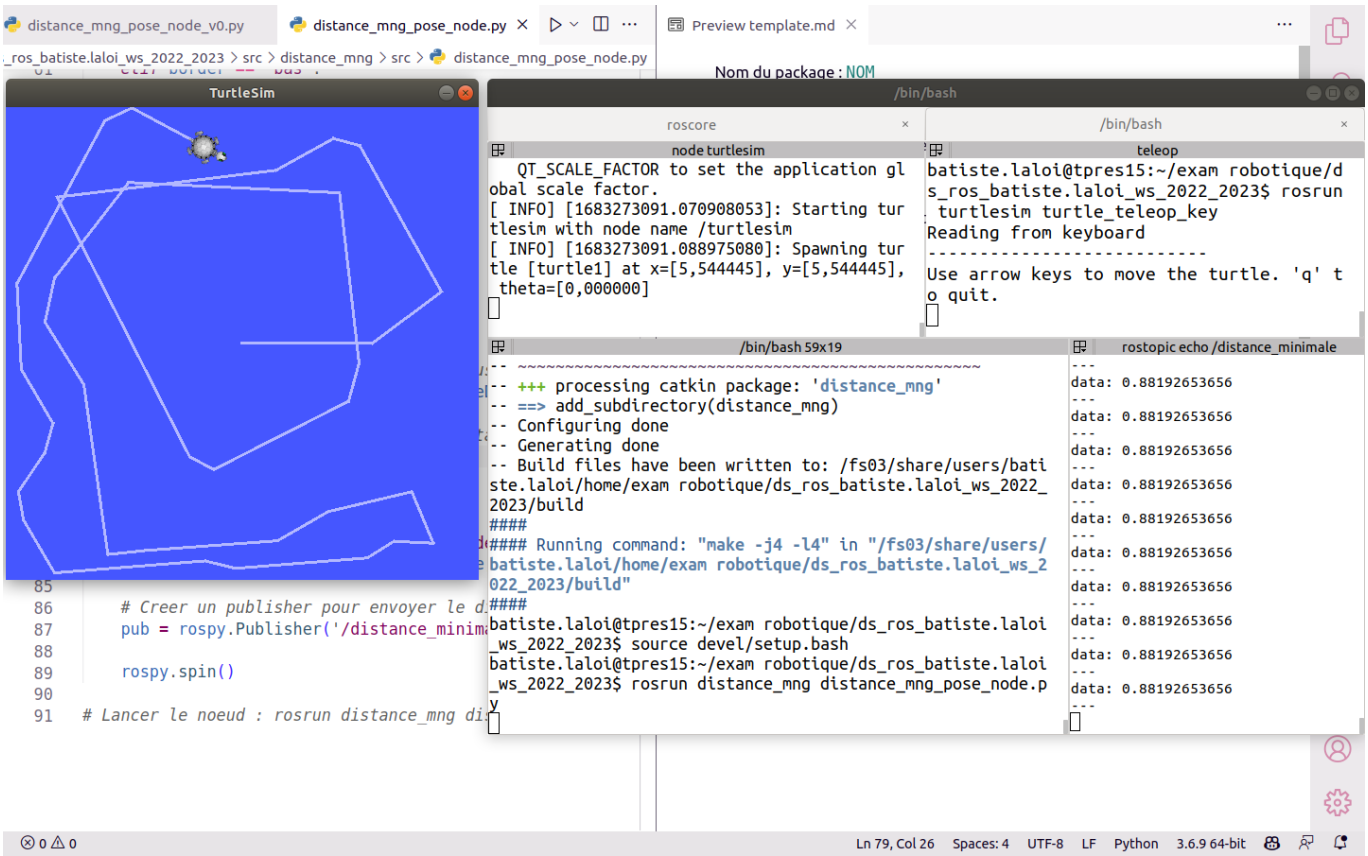
Nom du noeud : `distance_mng_pose_node.py`

Fonctions réalisées par le noeud :

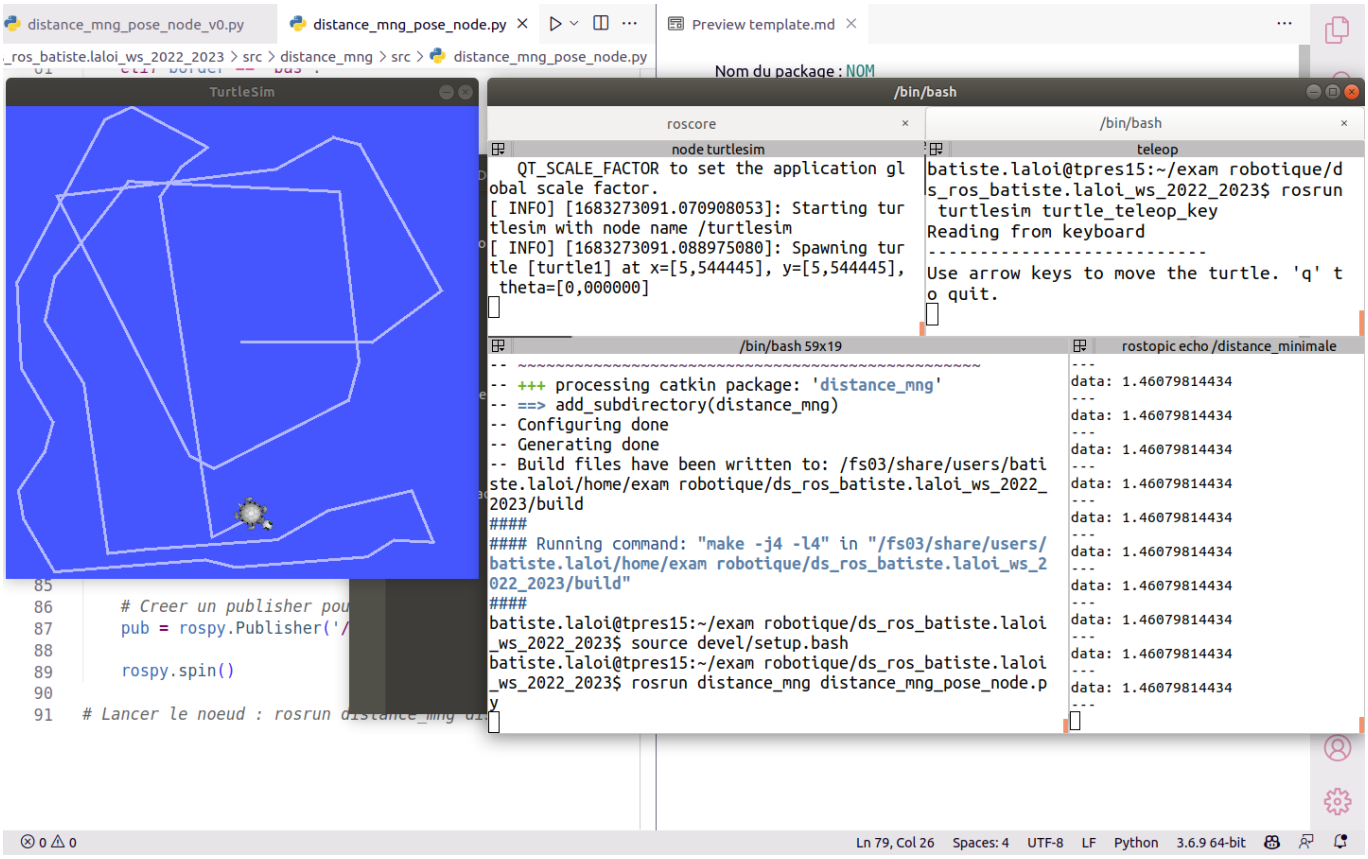
- Calcule le coin de la fenêtre dans lequel se trouve la tortue
- Calcule le bord le plus proche de la tortue
- Publie la distance entre la tortue et le bord le plus proche dans le topic `/distance_minimale`

Capture montrant le comportement, en bas à gauche des terminaux tourne le noeud `distance_mng_pose_node.py` et en bas à droite des terminaux se trouve un shell qui écoute sur le topic `/distance_minimale` grâce à la commande `rostopic echo /distance_minimale` :

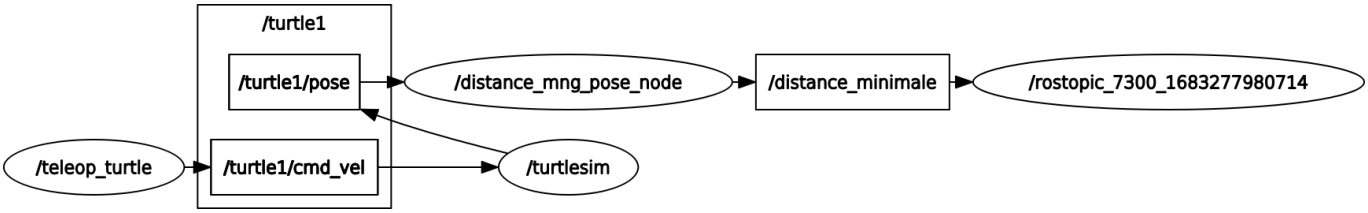
Tortue en haut :



Tortue en bas :



rqt_graph montrant le comportement du noeud :



3 : Détection de collision

3.1 : Création de noeud

Nom du package : `distance_mng`

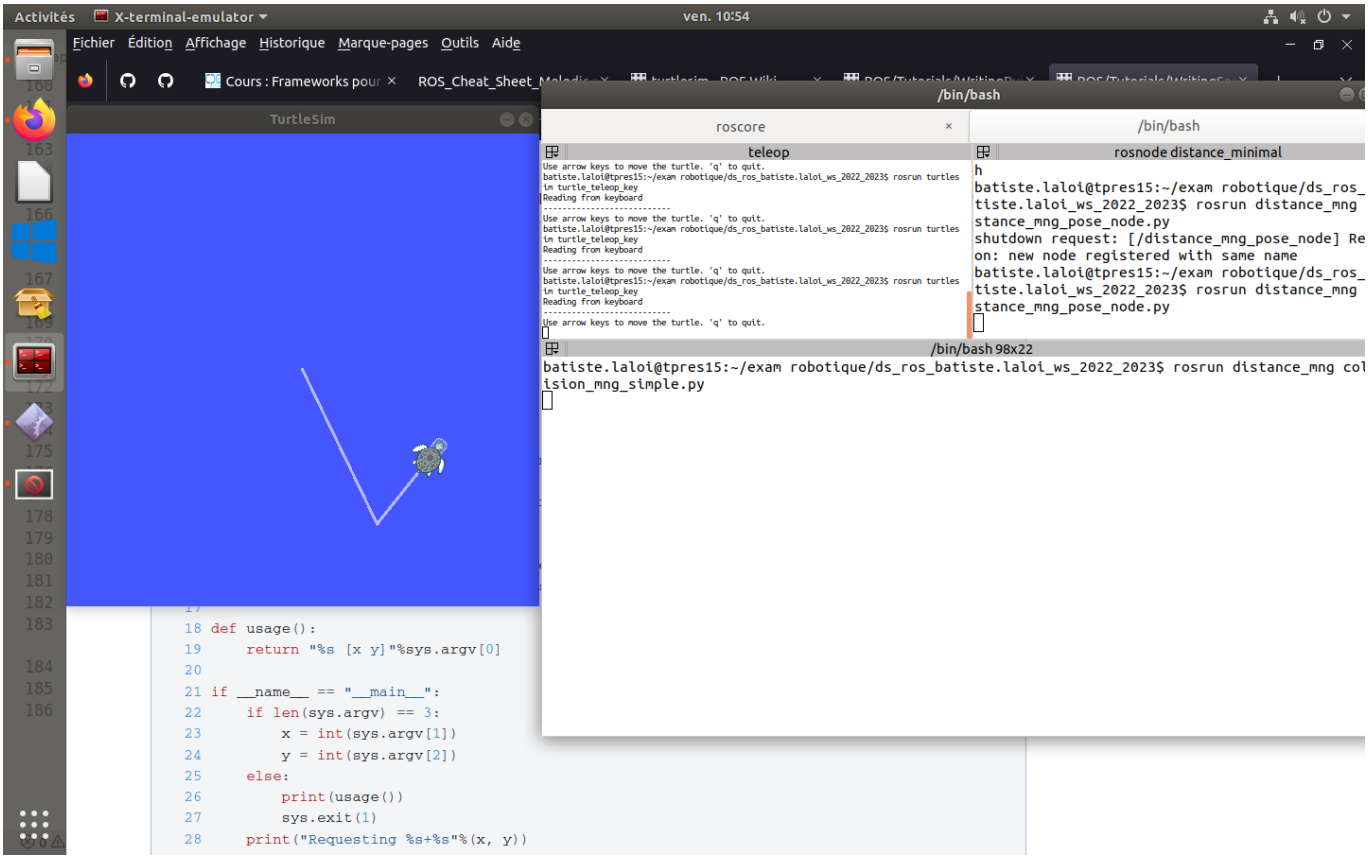
Nom du noeud : `collision_mng_simple.py`

Fonctions réalisées par le noeud :

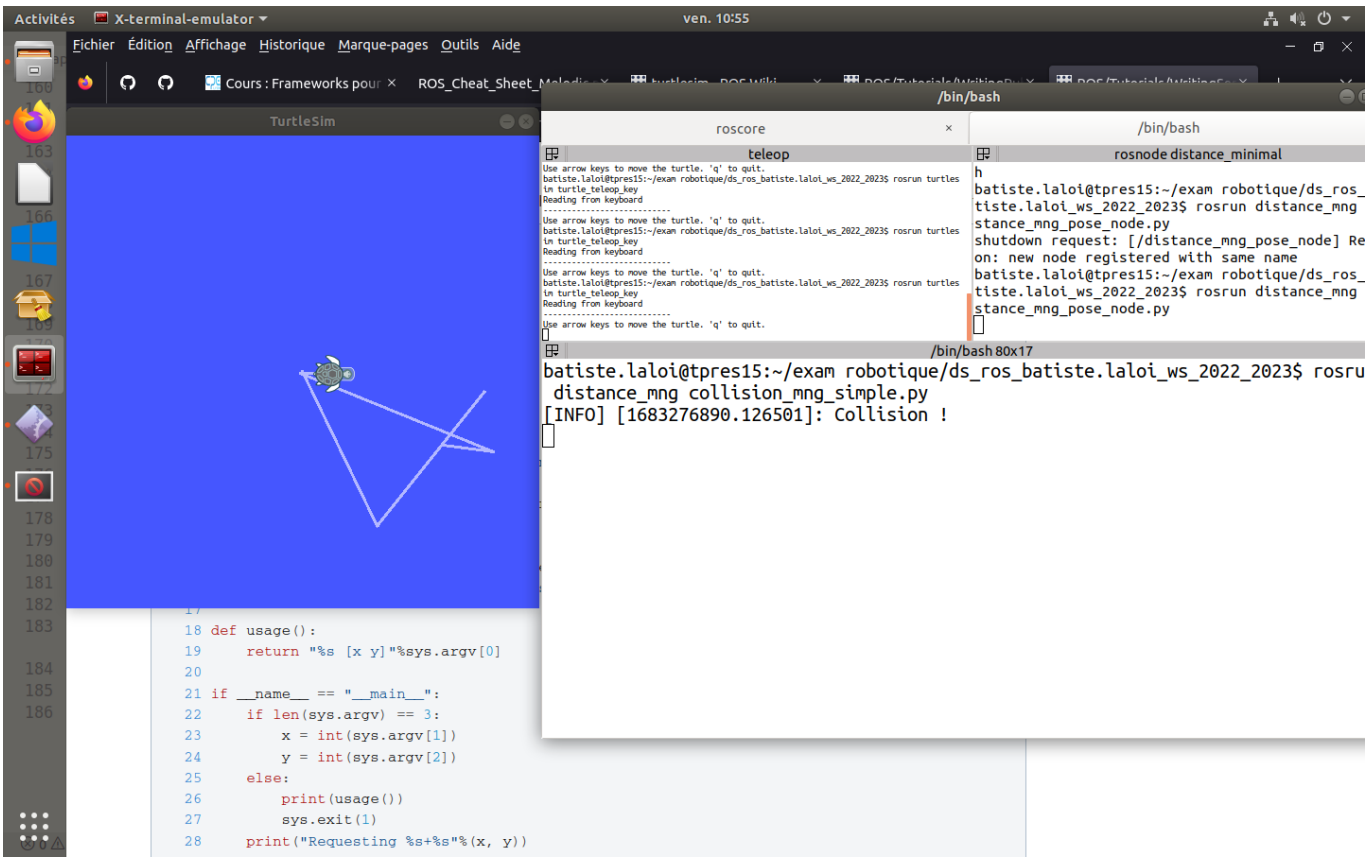
- Le noeuf utilise le service `turtle1/teleport_absolute` pour téléporter la tortue au centre de la fenêtre lorsqu'elle se rapproche trop d'un bord

Capture montrant le comportement :

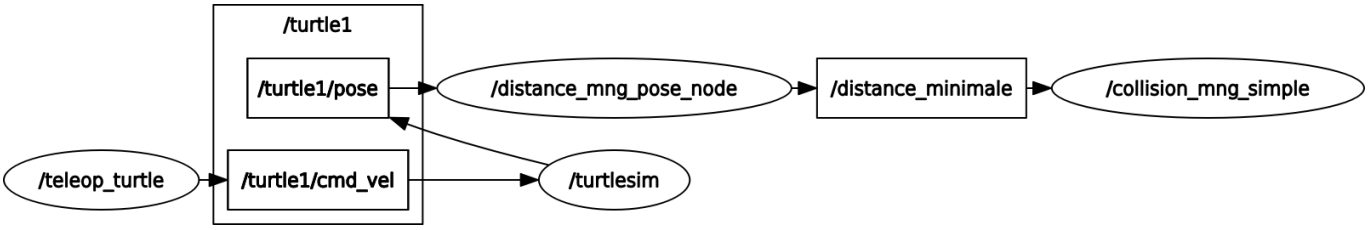
Avant la téléportation :



Après la téléportation :



rqt_graph montrant le comportement du noeud :



3.2 : Confuguration du noeud :

Nom du package : `distance_mng`

Nom du noeud : `collision_mng_service.py`

Fonctions réalisées par le noeud :

- Le noeud doit permettre de configurer les coordonnées de respawn, et le seuil de proximité pour déclencher la téléportation

Voici le code que j'ai fait, mais je n'arrive pas à passer les informations en arguments lors de l'appel à mon service :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import rospy
from std_msgs.msg import Float32
from turtlesim.srv import TeleportAbsolute
from std_srvs.srv import SetBool, SetBoolResponse

HEIGHT = 11.0
WIDTH = 11.0
X_RESPAWN, Y_RESPAWN = WIDTH/2, HEIGHT/2
SEUIL = 0.1

# Ce noeud represente un service "config_respawn_pose" qui permet de
modifier la position de respawn de la tortue et de modifier le seuil de
collision

def teleport(x, y):
    teleportAbsolute = rospy.ServiceProxy('/turtle1/teleport_absolute',
TeleportAbsolute)
    teleportAbsolute(x, y, 0.0)

def distanceCallback(data):
    if data.data <= SEUIL:
        rospy.loginfo("Collision !")
        teleport(X_RESPAWN, Y_RESPAWN)

def handleConfig(req):
    # Res = "x,y,seuil"

    global X_RESPAWN, Y_RESPAWN, SEUIL
```

```
    data = req.data.split(",")
    X_RESPAWN, Y_RESPAWN, SEUIL = float(data[0]), float(data[1]),
float(data[2])

def confServiceServer():
    rospy.init_node('collision_mng_service')
    rospy.Subscriber('/distance_minimale', Float32, distanceCallback)
    rospy.Service('config_respawn_pose', #String ?#, handleConfig)
    rospy.loginfo("Service config_respawn_pose lance")

if __name__ == '__main__':
    confServiceServer()
    rospy.spin()
```

3. Lancement de plusieurs noeuds

Pas fait

4. Safe Controller

Le code du noeuf `safe_controller.py` ne semble pas avoir d'effet sur la vitesse transmise par `teleop`, je ne sais pas vraiment pourquoi

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Float32

# Ce node permet de diminuer la vitesse transmise par turtle_teleop_key
lorsque que la valeur se trouvant sur le topic /distance_minimale est
inferieure a 1.0

def distanceCallback(data):
    if data.data <= 1.0:
        # rospy.loginfo("Collision !")
        rospy.set_param('/turtle_teleop_key/turtle_laser_linear', 0.1)
    else:
        rospy.set_param('/turtle_teleop_key/turtle_laser_linear', 1.0)

if __name__ == '__main__':
    rospy.init_node('collision_mng')
    rospy.Subscriber('/distance_minimale', Float32, distanceCallback)
    rospy.spin()
```