

Travaux sur la partie GPS/ IMU :

Lors de ce projet, je me suis principalement concentré sur la partie GPS / IMU via ROS2 foxy.

Dans une première partie, j'ai essayé de faire fonctionner le GPS RTK de centipède sur linux, puis dans une seconde partie, je me suis intéressé sur la partie GPS, j'ai créé un launch file afin de lancer plus facilement le nœud ros « ublox_gps », et enfin j'ai créé un package ROS, pour l'IMU BNO055.

I) Le GPS RTK :

Ce type de GPS présente un sérieux avantage : il est bien plus précis qu'un GPS normal : celui-ci ne permettant que d'avoir une précision de l'ordre du mètre, le système RTK permet alors d'avoir une précision de l'ordre du cm(la précision nominale typique pour les GPS RTK est de 1 cm horizontalement et 2 cm verticalement).

1) Explication du fonctionnement d'un GPS RTK :

Il utilise d'un récepteur fixe (station de base dont la position est connue) et un certain nombre de récepteur mobile... La station envoie la position au système. L'erreur de position entre la position calculée et la celle mesurée permet, par la suite, au système de proposer une correction aux récepteurs. Deux modes de communication sont possible:

_ la radio, mais cela n'est utilisable que pour les courte distance (moins de 10 km). Cependant il y a tout de même un avantage: nous n'aurons pas de surcoût de communication

On pourrait également utiliser le signal GSM (signaux mobiles), nous n'aurons alors plus de limite en termes de distance, mais les coûts de communication sont élevés... Bien évidemment la réception d'information dépendra de la qualité du réseau.

Pour accéder aux corrections, le GPS de précision doit disposer d'un accès internet mobile. Après s'être identifié sur le serveur et avoir donné sa position initiale, le GPS va recevoir un flux de données chaque seconde contenant les paramètres de correction à intégrer à son calcul de positionnement. Il nous faudra donc une connexion internet. Les coordonnées des points seront données en coordonnées MGRS WGS84 (normes dans les coordonnées GPS).

2) Utilisation du GPS RTK :

Nous avons réussi à faire fonctionner le GPS RTK sur Windows mais pas sur Linux : pour la configuration, il faut tout d'abord installer u-center (<https://www.u-blox.com/en/product/u-center> , prendre « For F9 / M9 products and below »), il faut alors configurer le baudrate à 9600 et suivre les instructions de ce document (https://www.ensta-bretagne.fr/lebars/Share/uCenter_RTK_guide.pdf).

Une fois le GPS RTK configuré, nous pouvons alors utiliser le gps RTK sur windows ...

II) Mise en place du nœud ROS pour le GPS :

1) Explication de l'utilisation du nœud et du launch file

Dans la suite de ce paragraphe, nous travaillerons avec le nœud ROS2 « ublox_gps », qui est disponible sur github ([https://github.com/KumarRobotics/ublox](https://github.com/KumarRobotics/ublox_gps)). Tout d'abord, il faut vérifier que la VM Linux est à jour : `sudo apt update && sudo apt upgrade` . Ensuite, il faut créer, si cela n'est pas déjà fait, un workspace, on utilisera les commandes suivantes :

```
mkdir -p ~/dev_ws/src
```

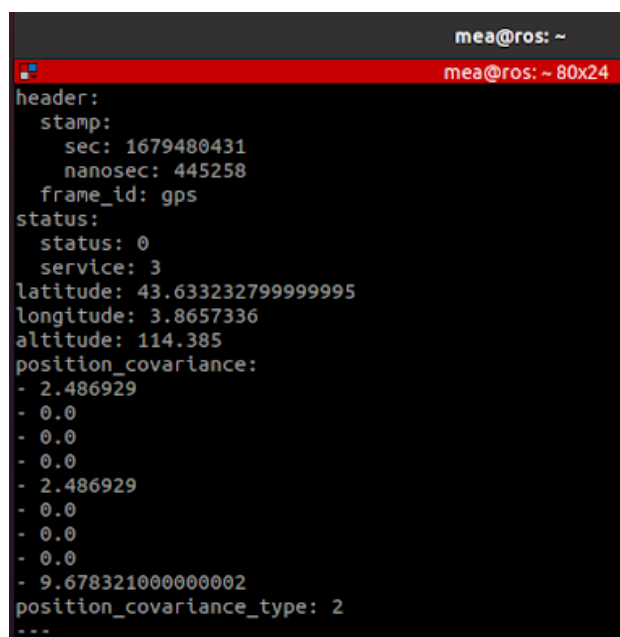
```
cd ~/dev_ws
```

```
colcon build
```

Une fois le workspace créé, on se place dans le dossier source du workspace `cd src`, et on va importer le nœud ROS dans ce dossier : `git clone git@github.com:KumarRobotics/ublox_gps.git`, Normalement github va demander de s'identifier : Attention, on ne peut plus se connecter via les mot de passe sur un terminal, il faut donc utiliser les tokens de github pour se connecter. On va donc sur la page web de github(<https://github.com/login>) , puis cliquer sur settings / developer settings / Personal access tokens. Une fois le token généré, on le copie et on le colle à l'emplacement du mot de passe.

Une fois le nœud ROS installé, il faut sortir du dossier source (`cd ..`), puis compiler (soit on compile tout le workspace `colcon build`, soit on ne compile que le nœud `colcon build --packages-select ublox_gps`) et enfin on source le nœud (`source install/setup.bash`). Maintenant le nœud ROS est prêt à être utilisé, afin de lancer celui-ci, on utilise la commande suivante : `ros2 run ublox_gps ublox_gps_node` (attention d'être bien passer en mode ROS avant, grâce à la commande `foxy`). Afin de pouvoir écouter le nœud, on pourra ouvrir un second terminal, en parallèle, et utiliser la commande `ros2 topic echo /fix` .

Voici ce que l'on obtient après avoir exécuter le nœud :



```
mea@ros: ~  
mea@ros: ~ 80x24  
header:  
  stamp:  
    sec: 1679480431  
    nanosec: 445258  
  frame_id: gps  
status:  
  status: 0  
  service: 3  
latitude: 43.633232799999995  
longitude: 3.8657336  
altitude: 114.385  
position_covariance:  
- 2.486929  
- 0.0  
- 0.0  
- 0.0  
- 2.486929  
- 0.0  
- 0.0  
- 0.0  
- 9.678321000000002  
position_covariance_type: 2  
---
```

Figure 1 : Ecoute du topic « /fix » provenant du nœud ublox_GPS

Comme nous le remarquons sur la figure ci-dessus, le nœud nous donne beaucoup d'informations, et nous n'aurons besoin que de la latitude et de la longitude pour notre projet, c'est cela qui nous a motivé à créer un launch file, afin d'épurer la sortie en console.

La suite du paragraphe porte donc sur le launch file que j'ai réalisé afin de simplifier l'utilisation de ce nœud, et ne sortir que la latitude et la longitude. Pour ce qui est de l'utilisation du launch file, il s'exécute grâce à la commande `ros2 launch ublox_gps ublox_gps_node-launch.py` (il faudra faire attention à être bien placé dans le dossier où est le launch file).

2) Quelques explications à propos du launch file :

Tout d'abord, il faut que l'on importe les librairies rclpy et NavSatFix, qui vont nous permettre de traiter les messages.

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import NavSatFix
```

Ensuite On développe une classe que l'on a nommé GPSNode, où on va définir la fonction `_init_` qui va initialiser le système et la fonction `gps_callback`. Dans la fonction d'initialisation, on définit quelle va être le type de message que l'on veut recevoir, ici ce sont des NavSatFix : ce qui correspond à des données non corrigées, d'où le nom « Fix », on définit la taille du buffer à 10 messages, si le buffer est complet, on supprimera le plus ancien. On a également ouvert un fichier texte grâce à la fonction

```
self.file = open('gps_data.txt', 'w')
```

Maintenant, intéressons nous à la fonction `gps_callback`, c'est cette fonction qui va nous servir à n'afficher que la latitude et la longitude : `msg` étant une structure, on accède à ces données en écrivant `msg.longitude` ou `msg.latitude`, et on utilisera la fonction `self.get_logger().info(...)`, afin d'afficher la latitude et la longitude en console.

III) Réalisation d'un nœud ROS pour la lecture de données de l'IMU.

Ce nœud sera non seulement utilisé pour la fusion de données, mais aussi pour les drones qui ont besoin de données d'accélération et de vitesse, tel que le bunker.

Au début de la création de ce nœud, nous nous étions orienté vers une communication UART, qui est possible sur le capteur BNO055, cependant la communication I2C s'est avérée plus simple à utiliser pour la création du nœud. Nous utiliserons donc une Raspberry pi, afin de le concevoir.

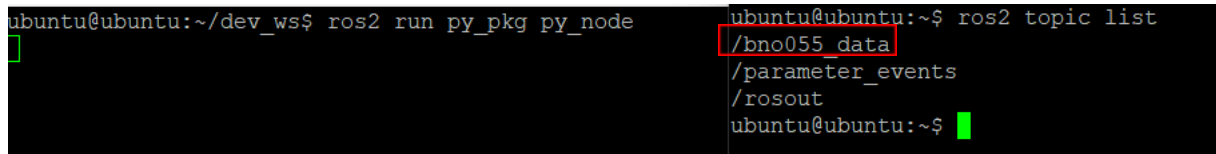
1) Comment utiliser le nœud :

Il faut tout d'abord se connecter au réseau de la Raspberry, et ouvrir un terminal. On se place dans le workspace (`cd dev_ws`) et on lance le nœud en utilisant la commande suivante (en faisant attention d'être en mode ROS foxy (`foxy`), `ros2 run py_pkg py_node`).

Voici ce que l'on observe en console :

```
---
header:
  stamp:
    sec: 1683711902
    nanosec: 413184771
  frame_id: imubno055
orientation:
  x: 0.0
  y: 1.9375
  z: 1.5
  w: 1.0
orientation_covariance:
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
angular_velocity:
  x: 0.003272492347489368
  y: 0.00545415391248228
  z: -0.002181661564992912
angular_velocity_covariance:
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
linear_acceleration:
  x: 0.33
  y: -0.25
  z: 9.42
linear_acceleration_covariance:
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
```

Figure 2 : donnée sortant du nœud ROS2



```
ubuntu@ubuntu:~/dev_ws$ ros2 run py_pkg py_node
ubuntu@ubuntu:~$ ros2 topic list
/bno055_data
/parameter_events
/rosout
ubuntu@ubuntu:~$
```

Figure 3 : sorti de la commande `ros2 topic echo`

Lorsque l'on a lancé le nœud ROS, on ouvre un nouveau terminal, on peut exécuter la commande suivante `ros2 topic list`, (figure 3) afin de faire apparaître la liste des topics que l'on peut écouter. Afin d'écouter le nœud, on utilisera la commande `ros2 topic echo /bno055_data`.

2) Explication du contenu du nœud ROS :

Le nœud n'est constitué que d'un fichier python, c'est celui-ci qui assure la communication entre la Raspberry et le capteur BNO055 ainsi que le traitement des données. Pour ce faire, nous avons besoin de créer une classe avec les définition des fonctions d'initialisation, de de publish.

Dans la fonction d'initialisation, nous allons donc configurer la communication I2C : via cette fonction :

`adafruit_bno055.BNO055_I2C(busio.I2C(board.SCL, board.SDA))`, où on définit les pins SCL et SDA.

On a également besoin de créer un Publisher: nous allons donc publier les données dans le topic 'bno055_data'. `self.publisher_ = self.create_publisher(Imu, 'bno055_data', 10)`, dans ce topic, on retrouvera donc des messages de type `sensors_msg.IMU`, le buffer a été défini sur 10, autrement dit, on ne peut attendre que 10 messages, si on en reçoit un nouveau, c'est alors le plus ancien qui est supprimé.

Ensuite, on a défini un timer : toute les 0.1 seconde, on publiera des données, si cela est trop rapide, il faudra augmenter le 0.1.

Nous avons aussi la fonction `publish_data`, c'est elle qui définit ce que l'on va écrire dans le topic. Le nom du message publié est « `imu_msg` », on a défini un header pour celui-ci : il est défini par son nom « `imubno055` » et la date de publication. Pour accéder aux données de l'IMU, nous allons avoir besoin de la variable `self.sensor` c'est dans cette variable, que nous allons retrouver les orientations, les vitesses linéaires et angulaires.

```
orientation = self.sensor.euler
imu_msg.orientation.x = orientation[0]
imu_msg.orientation.y = orientation[1]
imu_msg.orientation.z = orientation[2]
angular_velocity = self.sensor.gyro
imu_msg.angular_velocity.x = angular_velocity[0]
imu_msg.angular_velocity.y = angular_velocity[1]
imu_msg.angular_velocity.z = angular_velocity[2]
linear_acceleration = self.sensor.acceleration
imu_msg.linear_acceleration.x = linear_acceleration[0]
imu_msg.linear_acceleration.y = linear_acceleration[1]
imu_msg.linear_acceleration.z = linear_acceleration[2]
self.publisher_.publish(imu_msg)
```

Figure 4 : code qui permet de choisir les données publiés dans le topic

Bien sûr, toutes ces données ne nous seront pas forcément utiles pour faire une fusion de données, mais peuvent l'être pour le bunker.

Ce sera ensuite à la fonction `self.publisher_.publish(imu_msg)` de publier dans le topic les différentes informations.

Ensuite, comme dans tout nœud ROS, nous retrouvons la fonction main qui initialise le framework de ROS2, crée une instance de la classe BNO055Node, démarre la boucle des tâches que nous avons coder auparavant et lorsque l'exécution du nœud est terminé (lors du keyboard interrupt control + 'c'), libère les ressources utilisées et allouées par le nœud.