RAPPORT C10-4

LE ROUZIC Ameline, HOYER Hadrien

13/03/16

Cartographie d'obstacles avec un GPS, une IMU et un télémètre laser

Présentation du problème

Nous souhaitions nous servir de ce projet C10-4 pour avancer dans notre projet autonome. Notre projet autonome se nomme MANTA (Module Autonome de Navigation des Techniques Avancées), il s'agit d'un drone de surface qui a pour but de cartographier les fonds marins de faible profondeur (<100m). Le capteur utilisé pour voir la profondeur est un échosondeur, et les capteurs pour se positionner sont un GPS et une IMU.

Pour traiter les données, nous disposons d'un Raspberry Pi 2 model v1.1, et nous commandons les moteurs à travers un Arduino Uno.

Pour développer l'algorithme de cartographie, que dans un premier temps nous séparons complètement de l'algorithme de navigation, nous allons adapter et restreindre notre projet autonome pour constituer un sous-projet que nous réaliserons pour ce cours de C10-4 :

(1) la composante de déplacement autonome sera omise, vu que la coque et les moteurs dont nous disposons ne fonctionnent que sur l'eau, et que nous avons déjà toute une électronique fonctionnelle que nous souhaiterions utiliser plutôt qu'un Turtlebot. Nous déplacerons la boîte qui contient l'électronique à la main. Le focus sera mis sur la localisation et le dessin de la carte.

(2) l'échosondeur n'est pas encore entre nos mains à cause d'errements administratifs qui nous ont fait perdre beaucoup de temps. Nous allons donc remplacer ce capteur par un télémètre laser, prêté temporairement par M Filliat. Nous n'allons donc pas cartographier en profondeur, mais faire une carte 2D des alentours du robot.

Il s'agit donc d"utiliser le GPS et l'IMU, de fusionner leurs données pour localiser le robot dans un repère absolu, d'exprimer les obstacles repérés pas le télémètre laser dans ce repère absolu, et enfin de dessiner une carte pour visualiser l'environnement du robot.

La localisation sera réalisée en traitement direct et le dessin de la carte en post-traitement. En effet, nous n'avons qu'un dongle radio pour communiquer avec le drone, et celui-ci est utilisé pour reprendre la main et pouvoir télécommander le bateau. Notre moyen de communication avec le robot étant déjà utilisé, nous décidons de visualiser la carte une fois que le robot a fini sa mission. Le drone final (car pour l'instant nous travaillons sur un prototype) est censé intégrer la visualisation de la carte en direct.

I. Nodes et topics

Pour la localisation du robot, nous utilisons le package *robot_localization* [1]. Il permet de fusionner les données d'odométrie, d'un GPS et de plusieurs capteurs (dont des IMUs) pour localiser le robot. Notre boîte étant portée à la main, nous n'avons pas utilisé l'odométrie, mais seulement un GPS et une IMU.

Nous avions tout d'abord téléchargé la librairie *minimu9-ahrs* [2] qui permet de lire les données de l'IMU et de les afficher selon 3 modes : matrix, euler et quaternion. Cette librairie ne fonctionne que sur processeur ARM, donc sur la Pi et pas sur le Notebook. *minimu9-ahrs* permet de calibrer l'IMU et de faire un peu de fusion (entre les données de l'accéléromètre et du gyroscope). Malheureusement, cette librairie en C++ ne s'interface pas aisément avec un publisher python (qui devrait transmettre les données au package *robot_localization*).

Pour cela, nous avons téléchargé une autre librairie Python, *RTIMULib* [2], qui fait également la calibration de l'IMU, et qui renvoie les données de manière plus simple à interfacer avec un publisher. Toutefois, *robot_localization* prenant en entrée uniquement des quaternions, nous avons du compléter notre publisher IMU avec des fonctions issues de *minimu9-ahrs*, recodées en Python.

Le format des messages associés à des données de l'IMU est le suivant :

```
Header header

geometry_msgs/Quaternion orientation
float64[9] orientation_covariance

geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance

geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance
```

Le publisher de l'IMU publie un quaternion représentant l'orientation de l'IMU, ainsi que deux vecteurs de 3 composantes représentant respectivement la vitesse angulaire et l'accélération linéaire pour chaque axe. On transmet aussi les matrices de covariances associées à ces valeurs. Les codes pour le publisher et le subscriber sont donnés dans le dossier du rapport.

Le GPS est lu à l'aide du package *gpsmon*, inclus dans *gpsd* [3]. On code de même un publisher qui envoie les données GPS à *robot_localization*. Le format des données transmises est :

```
Header header

float64 latitude
float64 longitude
float64 altitude
```

On n'utilise pas l'altitude par la suite car le drone se déplace sur l'eau, qu'on suppose plane et à altitude constante. L'altitude ne sert pas dans la fusion mais il est prévu de l'utiliser dans l'initialisation pour connaître l'altitude de référence du plan d'eau.

Le télémètre est lu par la librairie *urg_node* [4], qui publie les distances sur un topic "scan" (codé directement dans la librairie). Le format de transmission des données est lui aussi donné par la librairie, sous sensor_msgs/LaserScan.msg :

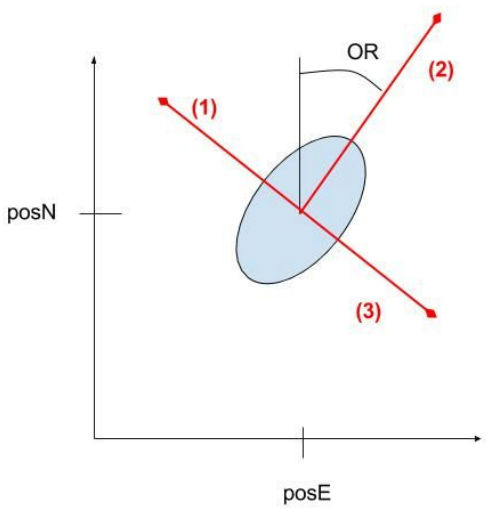
```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

II. Architecture

Voir schéma en annexe (5).

III. Tracé de la carte en post-traitement

Les données sont stockées dans un fichier sous cette forme :

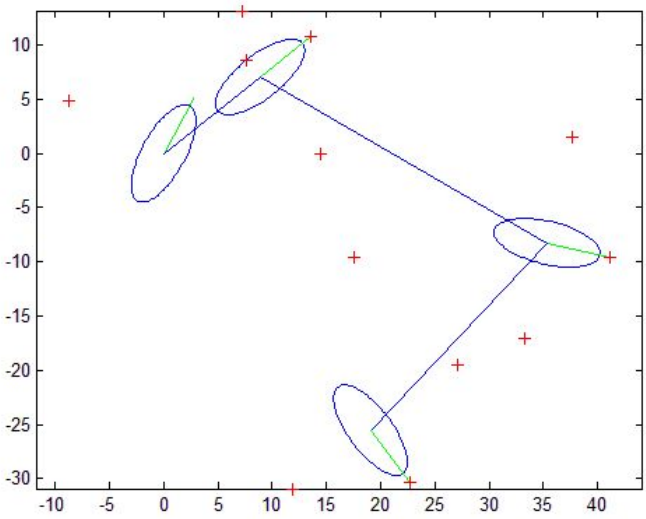


T 1314826776 GPS 2219.278 48711.0442 OR 0.5 DI 10 15 20

- T : temps au format timestamp Linux
- GPS : coordonnées GPS filtrées par l'IMU selon l'est, puis le nord, multipliées par 1000
- OR : orientation par rapport au Nord dans le sens horaire en radians, obtenue grâce au magnétomètre
- DI : distances vues par (1) le faisceau de gauche, (2) faisceau de devant, (3) faisceau de droite, en mètres

Décision du tracé du repère : nous avions commencé par un repère en coordonnées GPS décimales, mais il aurait alors fallu traduire le vecteur robot - obstacle (que l'on a grâce aux distances mesurées par le télémètre laser) en coordonnées GPS des obstacles. Nous avons décidé qu'il serait plus aisé de tracer le robot dans un repère R dont l'origine serait la

position initiale du robot, l'axe des ordonnées orienté vers le nord et l'axe des abscisses vers l'est. On stocke les positions du drone dans le repère R en calculant la distance en mètres de la position actuelle par rapport à l'origine (= la position de départ), grâce à une formule de conversion des coordonnées GPS en mètres et l'orientation par rapport au nord.



Exemple test de tracé MATLAB de tracé de carte : les axes sont gradués en mètres, et la position initiale est en (0,0). Le nord est en ordonnée et l'est en abscisse. Le robot est représenté par l'ellipse bleue ; son orientation est matérialisée par le trait vert ; les obstacles qu'il voit sont les croix rouges ; et les positions successives sont reliées par un trait bleu. On affiche une nouvelle position en appuyant sur n'importe quelle touche.

IV. Difficultés rencontrées

Le principal souci que nous avons eu avec ROS, c'est le temps. Même en y passant des journées complètes, on ne progresse pas vite, car le temps passé à télécharger des packages est bien plus important que le temps à vraiment coder.

Nous avons passé un temps considérable à convertir les données de l'IMU en quaternions, pour des raisons aussi peu facilement résolubles que "les définitions de Vector3 et Quaternion étaient concurrentes entre les messages ROS et les librairies python" ou "la librairie pyrr appelle la librairie numpy, qui elle-même renvoyait une erreur, il fallait en fait installer la dernière version de numpy donc réinstaller tout python et aller reconfigurer les path".

Le package *robot_localization* met 5 heures à s'installer avec toutes ses dépendances. Nous avons essayé de l'installer deux fois car une fois toutes les données d'entrée du GPS et de l'IMU mises en forme, il était impossible de rosmake ou cmake le package *ekf_localisation*. Après une demi-journée à débugger sans succès, nous n'avions plus que l'alternative de la réinstallation. Celle-ci n'a pas fonctionné et nous avons donc décidé de nous passer du package, de récupérer les fichiers source et de coder notre noeud ROS nous-mêmes en modifiant les sources.

urg_node met également 3 heures à s'installer car comme *robot_localization*, il nécessite la librairie poco qui prend 2h30 à compiler sur notre Pi en Ethernet.

Notre installation de ROS nous a également posé problème. Il a fallu modifier les variables d'environnement $ROS_PACKAGE_PATH et $PYTHONPATH pour permettre à ROS de fonctionner correctement et de reconnaître nos packages. Trouver ces erreurs et comment les résoudre nous a pris une bonne journée, étant donné leur multiplicité et la faible documentation disponible sur internet.

V. Expériences

Étant donné que le GPS ne fonctionne qu'en extérieur, il est hors de question de conduire des expériences à l'intérieur de l'école, nous avons décidé de travailler en extérieur, derrière l'école (au niveau des arbres face aux fenêtre de l'U2IS, du côté de la zone de livraison). L'objectif est de démontrer la précision de la localisation et de la construction de la carte en testant le système dans un environnement connu et mesurable. Par exemple, deux troncs d'arbres apparaissant sur la carte générée auront une distance mesurée que l'on peut vérifier à l'aide d'un mètre dans l'environnement réel.

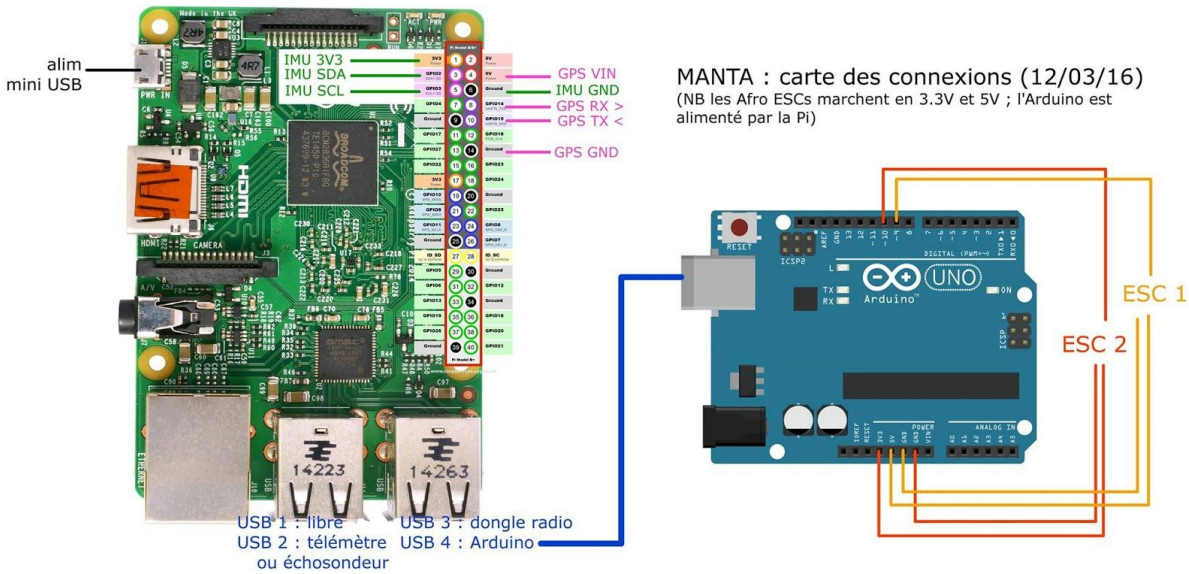
Nous présenterons les résultats des expériences lors de la soutenance.

Annexes

(1) Sources ROS : https://github.com/Epsichaos/mantOS/tree/master/src/manta/scripts

(2) Code MATLAB : joint au dossier

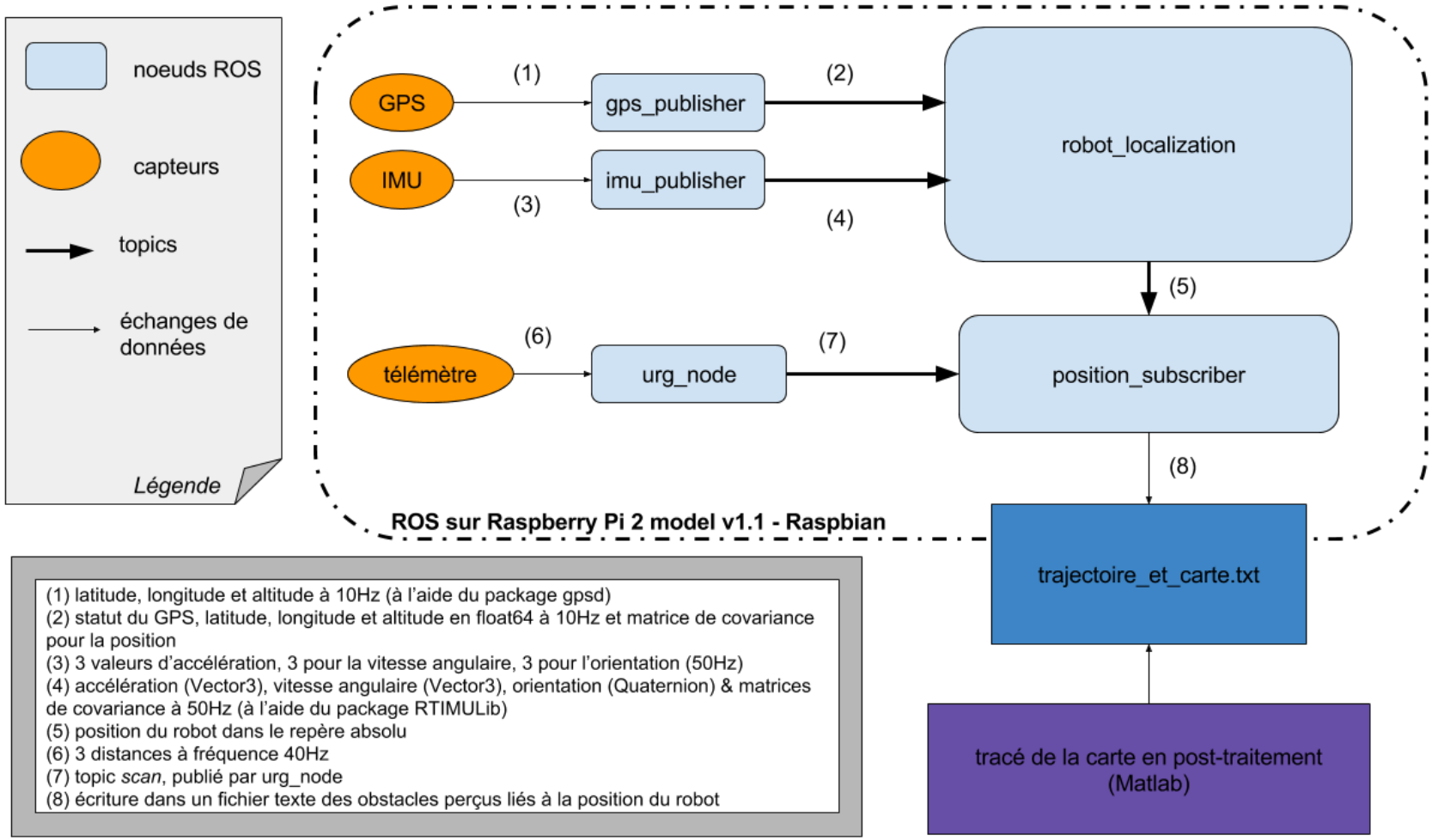
(3) Carte des connexions :



(4) Liens :

[1] Package robot_localization http://wiki.ros.org/robot_localization
[2] minimu-ahrs https://github.com/DavidEGrayson/minimu9-ahrs
RTIMULib https://github.com/richards-tech/RTIMULib
[3] GPSD http://www.catb.org/gpsd/
[4] Node pour la collecte des données télémétriques http://wiki.ros.org/urg_node Architecture ROS

(5) Architecture ROS : voir page suivante


noeuds ROS
capteurs
topics
échanges de données
Légende
GPS
IMU
télémètre
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
gps_publisher
imu_publisher
urg_node
robot_localization
position_subscriber
ROS sur Raspberry Pi 2 model v1.1 - Raspbian
trajectoire_et_carte.txt
tracé de la carte en post-traitement (Matlab)
(1) latitude, longitude et altitude à 10Hz (à l'aide du package gpsd)
(2) statut du GPS, latitude, longitude et altitude en float64 à 10Hz et matrice de covariance pour la position
(3) 3 valeurs d'accélération, 3 pour la vitesse angulaire, 3 pour l'orientation (50Hz)
(4) accélération (Vector3), vitesse angulaire (Vector3), orientation (Quaternion) & matrices de covariance à 50Hz (à l'aide du package RTIMULib)
(5) position du robot dans le repère absolu
(6) 3 distances à fréquence 40Hz
(7) topic *scan*, publié par urg_node
(8) écriture dans un fichier texte des obstacles perçus liés à la position du robot