Recommandations

I. Introduction

Ce document contient les informations que je veux transmettre qui sont soit essentielles, soit qui n'ont pas leur place dans la documentation

II. Généralités

A. Synergie avec l'élec

Comme dans n'importe quel projet, il est important d'être au courant de ce qu'il se passe dans les autres pôles et de les tenir au courant de ce que vous faîtes. Il est encore plus important de comprendre comment fonctionne l'électronique car cela est indispensable pour expliquer au pôle élec ce dont vous avez besoin et pour résoudre les bugs qui peuvent provenir à la fois du hardware et du software. De plus, quand vous pensez à quelque chose qu'il faudrait modifier ou ajouter, dites-le-leur immédiatement pour qu'ils puissent y réfléchir et s'adapter en conséquence.

B. Débogage

C'est en déboguant qu'on apprend à déboguer, donc je vais juste mettre les 3 point les plus importants :

- Procéder par élimination, on valide les morceaux de code un par un jusqu'à tomber sur l'erreur.
- Lire la doc, surtout si on croit la connaître par cœur et se renseigner sur les forums, en essayant tous les mots clés possibles dans google.
- Ne pas s'acharner, et aller se coucher. La moitié du temps je trouve la solution quand je ne suis pas en train de déboguer.

C. Quelques astuces

- Quand vous travaillez sur MPLAB, clique droit → « set as main project » pour qu'il se passe moins de trucs bizarres.
- N'oublie pas d'ajouter 'config_bits.c' à ton projet, sinon tu vas te faire très très mal.
- Pas d'Arduino ou de Python dans la fusée.
- Si c'est pas la faute à l'élec, c'est celle à la méca.

III. Lisibilité du code

A. Nommage

1. Ne pas hésiter à renommer les variables et les fonctions

Je suis très mauvais pour donner des noms aux fonctions et aux variables, donc changez les si vous les trouvez trop nuls. Utilisez la fonctionnalité 'refactor→rename' de MPLAB (ctrl+R) pour le faire proprement. Attention cependant, les modifications ne seront appliquées que dans votre 'main project'. Par exemple si votre 'main project' est 'Trajecto_Sequencer' et que vous renommez une fonction de can.c, les changements ne seront pas appliqués dans 'Ejection_Sequencer'.

2. Être consistant

Choisir une convention pour le nommage et s'y tenir. Les deux principales conventions sont :

- nom_de_la_variable_ou_fonction
- NomDeLaVariableOuFonction

Ecrivez les macros constantes tout en majuscules et les macros fonctions en minuscule avec un underscore au début :

```
#define MACRO_CONSTANTE
#define _macro_fonction()
```

3. Utiliser des notations et macro intuitives

Les noms doivent être explicites. Si une macro ou une variable à une logique 'inversée' (par exemple il faut mettre *LED1=0* pour allumer la *LED*), mettez un petit « n » devant (remplacer *LED1* par *nLED1*) ou utilisez une macro (*#define _led1_on() LED1=0 ;*).

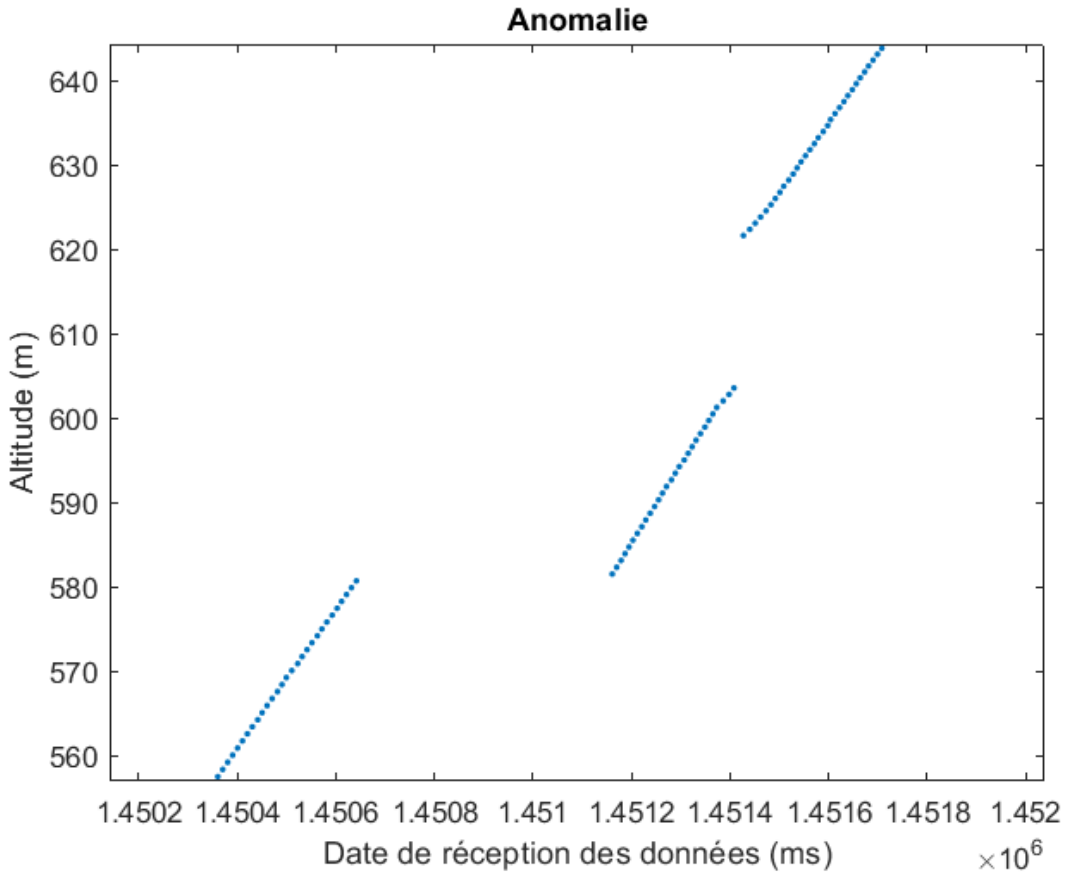
IV. Bus CAN et Gestion des Données

A. Analyse coupures ou latences CAN

En analysant les données du vol, on voit que le CAN a des coupures, et le comportement est assez étrange :

- Le CAN se coupe entre T_1 et T_2 (T_2-T_1 de l'ordre de 0.5s à 1.5s).
- Entre T_2 et T_2+T_{wtf} la Raspberry reçoit les données envoyées entre T_1 et T_1+T_{wtf} (T_{wtf} de l'ordre de 0.25 à 0.5s).
- Les données envoyées entre T_1+T_{wtf} et T_2+T_{wtf} sont perdues.
- A partir de T2+T_{wtf} on reçoit de nouveau les données normalement.

Voici ce que cela donne sur un graphe :



Il faudrait quantifier la fréquence et l'importance de ces coupures, et chercher une explication.

B. Datation des mesures

Pour les mesures de pression j'ai considéré que la date de réception des données était la date de la mesure. Cela permet de ne pas avoir à transmettre une date par le CAN en plus de la mesure, mais les observation des latences du CAN remettent en cause ce choix. On pourrait donc choisir d'envoyer la date de la mesure avec la mesure mais cela pose un problème : Quelle référence de temps choisir pour la date ?

- La date d'allumage du microcontrôleur est différente pour chaque microcontrôleur
- La date du décollage est *à priori* la même pour tous les microcontrôleurs (et encore, il faudrait le vérifier), mais cela pose le problème de dater les mesures au sol.

La solution que je propose est d'utiliser l'allumage de chaque microcontrôleur comme référence, d'enregistrer la date du décollage détecté par chacun, et de l'utiliser pour recaler les données lors du post-traitement.

C. Données à enregistrer

1. Séquenceur Trajectographie

- Mesures brutes Xsens
- Données trajectographie
- Log : Mode, mise à feu autorisée
- Date de début de l'intégration

2. Séquenceur Allumage

- Log : Mode, Ordre de mise à feu, Coiffe détectée, Séparation Détectée

3. Séquenceur Ejection

- Pression
- Dérivée pression
- Log : Mode, Baromètre opérationnel

D. Mise en forme messages

Un paquet CAN ne peut contenir que 8 bytes. Il faut donc découper les gros messages en paquets de 8 bytes et les recoller dans la Raspberry. Cela pose plusieurs difficultés :

- Les paquets ne sont pas toujours transmis les uns à la suite des autres, d'autre paquets d'origine différentes peuvent s'intercaler.
- Certains paquets peuvent être perdus : il faut un moyen de détecter les erreurs
- Il faut pouvoir identifier le début et la fin d'un message.

Dans l'état actuel du code, j'envoie un petit message (un *header*) avant chaque gros message pour indiquer le début, le type et la taille du message. Pour vérifier l'intégrité du message, je me contente de compter les bytes reçus entre deux *headers*.

Le protocole CAN est énormément utilisé dans l'industrie automobile, il doit donc être possible de trouver les procédures déjà faites pour transmettre de gros messages.

V. Séquenceur Trajectographie

A. Organisation du code

Actuellement la lecture et le traitement des données de la Xsens est un gros bordel (c'est le 1er truc que j'ai fait au SCube, un vrai travail de bizuth) :

- La Xsens est configurée en utilisant les fonctions du fichier 'xsens_config' qui utilise le fichier 'xbus' pour communiquer.
- Pour recevoir les mesures de la Xsens, on utilise directement la fonction *receive_xbus_buf()* du fichier 'xbus' qui stocke le message dans la variable globale *xbus_receive_buf* définie et déclarée dans 'xbus.h'.
- Pour décoder le message on appelle la fonction read_xbus_data() du fichier 'data_processing' (nom inadapté) qui lit directement *xbus_receive_buf* et stocke les données décodées dans la variable globale *acquisition_data* définie et déclarée dans 'trajecto.h'
- Pour traiter les données décodées on utilise les fonctions du fichier 'trajecto.c' qui utilisent directement les variables globales *acquisition_data* et *trajecto_data*.

Ce code abuse des variables globales, ce qui cause plusieurs problèmes : on est obligé d'inclure les headers définissant les structures et déclarant ces variables partout, et le code n'est pas utilisable ailleurs car il est directement lié aux variables globales. Je propose donc la nouvelle organisation suivante :

- Créer un fichier 'trajecto_types.h' définissant les structures *Acquisition_Data* et *Trajecto_Data* mais qui ne déclare pas de variables globales. Ce fichier pourra ensuite être inclus dans 'trajecto.h' et 'xsens.h' (voir la suite)
- Fusionner 'xsens_config' et 'data_processing' en un fichier 'xsens' permettant de configurer la Xsens et the recevoir les données avec une fonction du genre :
  *bool get_measure(Acquisition_Data *measure)*.
- Modifier les fonctions de 'trajecto.c' pour qu'au lieu d'utiliser une variable globale, elles prennent en paramètre un pointeur sur *Acquisition_Data*.

Si quelqu'un est intéressé par la trajectographie, ce projet pourrait être un bon moyen de lui faire prendre en main le code avant d'améliorer les calculs (cf. partie suivante).

B. Calibration

Les accéléromètres de la Xsens n'avaient pas été calibré car le calcul de la position n'était pas une priorité et que le processus demandait une précision que l'on n'avait pas le temps de fournir. Le principe de la calibration en lui-même est assez simple, on oriente la Xsens de façon a ce qu'un seul des axes ne mesure l'accélération de la gravité. On note ensuite l'accélération résiduelle sur les 2 axes horizontaux et l'erreur de l'axe vertical par rapport à la valeur théorique. On répète l'opération sur chacun des 3 axes et dans les 2 sens pour pouvoir faire une moyenne et vérifier que la non-linéarité est négligeable. Enfin on intègre le biais mesuré au code pour corriger les mesures brutes.

C. Algorithme de trajectographie

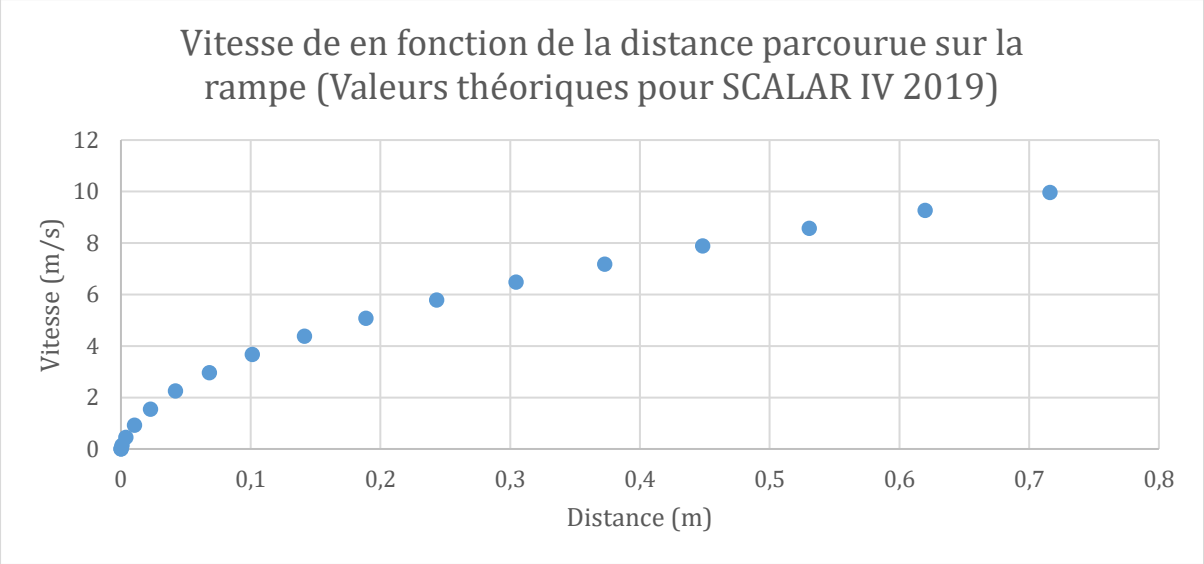
Selon moi, la trajectographie se divise chronologiquement en 3 étapes : l'attente au sol, le décollage et le vol.

1. Attente au sol

Durant l'attente au sol on profite de la relative immobilité de la fusée pour calibrer les gyroscopes calculant la moyenne glissante de la vitesse angulaire résiduelle qu'ils mesurent. On détermine aussi l'orientation initiale de la fusée grâce à la mesure de la gravité.

2. Décollage

Le décollage est la partie la plus critique selon moi. En effet l'ombilical ne s'arrache pas immédiatement après l'allumage du moteur, ce qui fait que la fusée a déjà une vitesse non négligeable à ce moment-là.

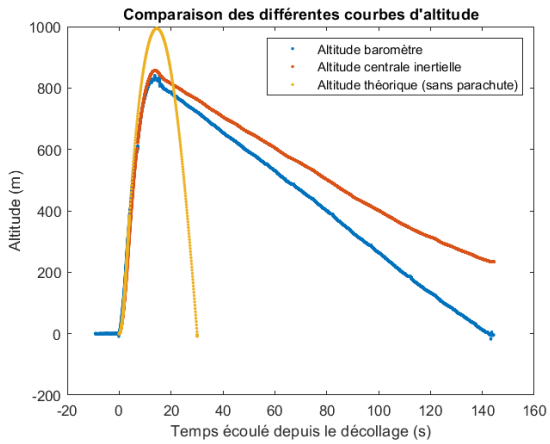


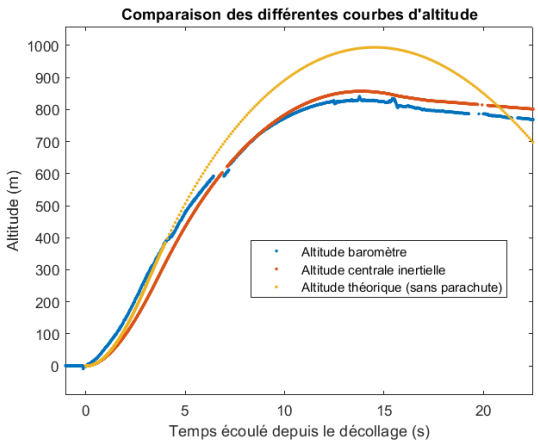
La distance d'arrachage de l'ombilical dépend de la façon dont il est attaché sur la rampe, elle était de 30-50cm cette année. Une erreur de 8m/s sur la vitesse correspond à une dérive de 8m par seconde de la trajectoire. Il faut donc commencer à intégrer avant l'arrachage de l'ombilical.

La méthode que j'ai implémentée consiste à commencer à intégrer dès lors que l'accélération dépasse un certain seuil, et réinitialiser les valeurs si on repasse sous ce seuil sans que l'ombilical ne se soit arraché. Cela semble fonctionner d'après les enregistrements du vol, cependant il faut choisir une bonne valeur seuil d'accélération.

3. Vol

Durant le vol on intègre pas à pas la vitesse angulaire et l'accélération de la fusée pour déterminer l'orientation et la position. Le calcule de l'attitude lors du vol semble valide, même s'il n'est pas évident de le vérifier à partir du vidéo embarqué. Le calcul de la position quant à lui souffre d'une grosse dérive vers la fin du vol, mais pendant la montée l'altitude calculée suit à peu près celle obtenue à partir des baromètres (même si on a parfois jusqu'à 50m d'écart entre le baromètre et la trajectographie pendant la montée) :





On remarque que l'accélération oscille énormément, ce qui peut nuire à l'intégration (Je vous invite à regarder vous-même les données du vol).

Il y a différentes façons d'améliorer l'algorithme, par exemple filtrer l'accélération ou intégrer les données d'autres capteurs. L'idéal serai d'implémenter un filtre Kalman, mais c'est une tâche très longue et complexe. Selon moi, complexifier l'algorithme d'intégration n'est pas une nécessité, il vaut mieux investir du temps à calibrer la Xsens et s'assurer que le décollage est bien intégré.

D. Fusée à eau

Le meilleur moyen de simuler la poussée que va devoir mesurer la Xsens est de la tester dans une fusée à eau. Comme l'on avait jamais fait cela avant, il faudra que vous conceviez la carte de test et le code de zéro.

VI. Télémétrie

A. Intégration

Durant toute l'année dernière, j'ai testé la télémétrie en reliant directement le module rfm95w à la Raspberry avec des fils et tout fonctionnait parfaitement : La télémétrie est prête et fonctionnelle. Cependant lorsque le module rfm95w a été intégré à la fusée il a cessé de fonctionner, peut-être à cause d'un problème au niveau de l'alimentation. La recherche du problème relève principalement des compétences du pôle électronique, mais il faudra les aider pour tester leur prototypes (cf. Synergie avec l'élec).

B. Test de portée

Comme la télémétrie ne fonctionnait dans la fusée, nous n'avons pas pu faire de test de portée, il faudra donc les faire cette année. Il faudra impérativement faire le test de portée quand le module sera intégré dans la coiffe, mais en attendant, un test de portée avec seulement les Raspberry, les modules et les antennes ne peut pas faire de mal.

Il sera également possible de mesurer le diagramme d'émission du module intégré à la coiffe dans la chambre anéchoïque du DEOS.

C. Choix des données à envoyer

La télémétrie a un débit limité, on ne peut donc pas tout envoyer. L'an dernier j'avais choisi d'envoyer tous les logs, et d'utiliser le débit restant pour envoyer le plus de quaternions et de positions possible. Pour connaître le débit disponible, il faut avoir fait le test de portée.

D. Station sol

J'ai déjà écrit et testé un programme pour la Raspberry servant de station sol. Cependant ce programme se contente de lire les données reçues par télémétrie et de les 'printer' dans une console. Il faudrait donc créer en C++ une interface stylée pour afficher les données reçues. Pour cela deux solutions :

- Utiliser une librairie graphique.
- Utiliser ncurses pour créer une console dynamique ultra stylée.

VII. Séquenceurs Récupération et Charge Utile

Le code des séquenceurs Récupération et Charge Utile est testé et validé. Cependant il faut tout de même que quelqu'un le prenne en main car la récupération est la partie la plus critique.

A. Détection de l'apogée

La détection de l'apogée pour l'éjection du parachute ou du Cansat se fait grâce à la dérivée de la pression mesurée par le baromètre. Cette dérivée est obtenue en appliquant un filtre passe-bas de 1er ordre aux mesures de pression, en dérivant la pression obtenue, puis en appliquant une moyenne glissante à cette dérivée. On éjecte ensuite le parachute ou le Cansat lorsque cette dérivée passe au-dessus d'un certain seuil. Cependant, ce seuil n'est pas égal à 0 pour deux raisons :

- On ne veut pas éjecter à l'apogée, mais un peu avant. Le parachute est éjecté 1s avant l'apogée pour qu'il ait plus de temps pour s'ouvrir. Le Cansat est éjecté quelques secondes avant le parachute pour qu'il ne se prenne pas dedans.
- Le processus permettant d'obtenir la dérivée induit un retard dans la valeur de la dérivée.

Nous nous sommes aussi rendu compte que l'éjection du parachute et du Cansat provoquait des pics de pression :

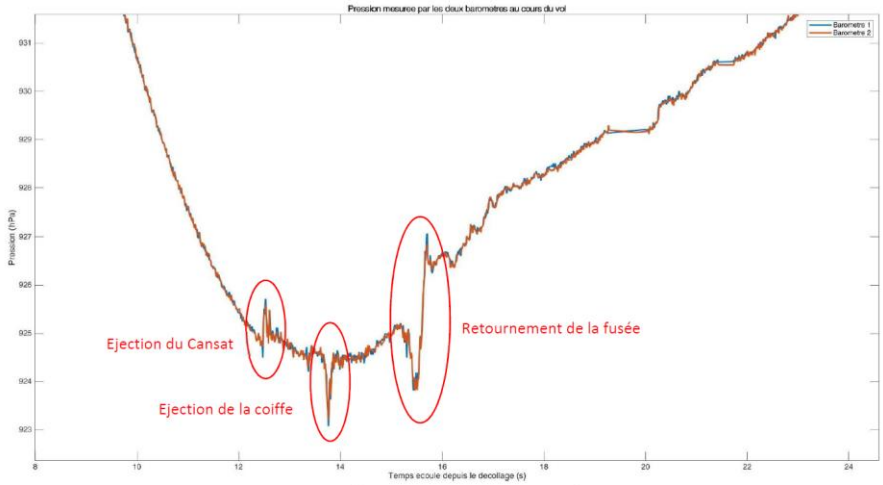


***Fig. 19** Détail sur les pics de pressions mesurés.*

Il faut s'assurer que le pic de pression dû à l'éjection du Cansat ne déclenche pas l'éjection du parachute. Le plus simple est de faire en sorte que les fenêtres d'éjection du Cansat et du parachute ne se chevauchent pas.

B. Adaptation pour le Biétage

Pour un vol biétage actif, il faudrait prévoir 2 fenêtres d'éjection du parachute haut. Une fenêtre nominale à l'apogée du 2ème étage si tout se passe bien, et une autre fenêtre plus tôt, s'il n'y a pas de séparation ou si le 2ème moteur ne s'allume pas. En conséquence, il faudra trouver un moyen de détecter l'allumage du moteur (accélération, capteur de luminosité, fil qui brûle, …).

C. Driver Herkulex

Les Herkulex sont le modèle de servomoteur utilisé dans la fusée. Leur driver à été écrit par mon prédécesseur et m'a été transmit sans documentation. J'ai essayé de le documenter et de rajouter des fonctionnalités, mais leur datasheet est très mal faite et difficile à comprendre. Finalement, la transmission d'ordres vers les Herkulex fonctionne et j'arrive à savoir si elles ont terminé leur rotation. Cependant je n'ai pas réussi à écrire une fonction permettant de lire leur position actuelle. Dans sont état actuel le driver est suffisant pour ce que l'on veut en faire, mais il serai intéressant d'être capable de lire leurs informations (pour les calibrer, vérifier qu'elles ne sont pas coincée, …).

VIII. Séquenceur Séparation

Le code du séquenceur Séparation est fonctionnel et a été testé au sol avant que la séparation soit abandonnée. Il faudra qu'il soit modifié pour le nouveau système de séparation.

IX. Suggestion de tâches à réaliser :

A. Important

- Rendre la trajectographie plus fiable :
    - Réorganisation du code
    - Calibration de la Xsens
    - Ajustement de la gestion du décollage
    - **Tester avec des fusées à eau**
- Chaine d'enregistrement des données :
    - Quelles données enregistrer ?
    - Comment transmettre de gros messages par le CAN ?
    - Quelles données transmettre par télémétrie et sous quel format ?
- Télémétrie :
    - Faire fonctionner dans la fusée
    - Tester la portée
    - Station sol basique
- Prise en main et adaptation des séquenceurs aux nouveaux systèmes de la fusée.
    - Dépend de ce que vous comptez faire cette année

B. Pas essentiel

- Interface station sol stylée
- Filtre Kalman (compliqué)
- Lire les information des Herkulex

C. Réflexions futures

Ceci est une liste des choses auxquelles réfléchir quand tout le reste fonctionnera parfaitement.

- Prendre une meilleure centrale inertielle : Les performances de la Xsens mti-1 sont limitées, prendre un modèle plus performant (filtre Kalman intégré, GPS, …) peut être une bonne idée.
- Changer de microcontrôleur : Passer à un microcontrôleur 32 bits programmable en C++ peut changer la vie. Cependant cela implique de réécrire tout le code
- Changer de bus de communication : Le CAN n'est pas très adapté à notre application, principalement à cause de la limite de taille des packets.