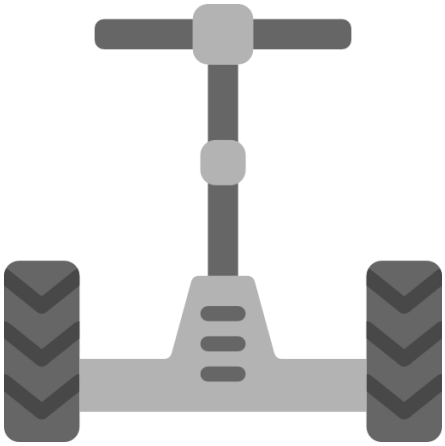
Département de Génie Electrique & Informatique

RAPPORT DE PROJET DE RECHERCHE TUTORE

4ème année Ingénierie des Systèmes

Contrôle d'un gyropode en temps réel



Léo LAUDOUARD **Margaux SEBAL**

Emeric MOLERE

Juin 2018

Tuteurs : **Claude BARON**

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université de Toulouse

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université de Toulouse

Table des matières

INTRODUCTION

Dans le cadre de notre 4ème année à l'INSA, nous avons un projet tutoré à réaliser. Il est intitulé "Commande d'un gyropode en temps réel" et notre tutrice est Mme Baron. L'objectif de notre projet est d'améliorer la maquette du gyropode conçue par des étudiants et professeurs des départements Génie Mécanique et Génie Electrique et Informatique.

Durant le premier semestre, nous avons effectué un état de l'art des gyropodes et des diverses améliorations et fonctionnalités à apporter à notre système. Notre tutrice nous a dirigés vers une fonctionnalité précise à ajouter : la commande à distance du gyropode. Ceci permettra de faire une démonstration lors des journées portes ouvertes de l'INSA de Toulouse.

Nous nous sommes donc efforcés durant le second semestre à mettre en place un système permettant d'assurer cette commande à distance.

I - MISE EN PLACE DU PROJET

I.1 - Réflexion d'amélioration

Dans notre état de l'art, nous avions décrit plusieurs idées de fonctionnalités à améliorer ou à ajouter à la maquette du gyropode. Nous allons vous parler de quelques-unes d'entre elles.

I.1.1 - Changement de la détection de présence

Tout d'abord, nous avons pensé à remplacer la détection de présence qui consiste à appuyer sur un des deux boutons de sécurité aux extrémités des poignées par une balance. Celle-ci limiterait l'utilisation du gyropode à des personnes de masse supérieure à une valeur choisie et aurait bloqué l'utilisation du gyropode en cas d'absence d'utilisateur. Ceci aurait été particulièrement facile à réaliser car nous avons trouvé un tutoriel permettant de le faire.

Seulement, ce n'est pas l'ajout de fonctionnalité primordial. Comme le gyropode n'est pas encore très sûr, il aurait fallu mettre cette condition de présence en parallèle de celle existante. Ce sera donc une idée à développer lorsque le contrôle en temps réel du gyropode sera efficace.

I.1.2 - Améliorer la maniabilité du gyropode

Le maquette du gyropode est dirigée à l'aide du guidon. Lorsque l'on penche le guidon vers la gauche ou la droite, une jauge de contraintes placée à l'extrémité de celui-ci au niveau du socle fléchit. La carte STM32 récupère alors une valeur de déformation et la transforme en consignes de couple indépendantes pour chaque moteur. Nous nous sommes rendu compte que le gyropode tournait difficilement, surtout d'un côté. En effet, cette jauge n'est pas très précise et le jeu laissé pour pivoter le guidon n'est pas égal des deux côtés.

Ce sont des problèmes à prendre en compte mais comme le gyropode n'est pas encore sûr pour recevoir un utilisateur, ce n'est pas primordial.

I.1.3 - Interface utilisateur WEB

Nous avons également pensé à créer un site web connecté aux données du gyropodes, permettant à un ou plusieurs utilisateurs d'avoir accès aux données du gyropode à distance via un téléphone ou un ordinateur.

Cette application informerait l'utilisateur des valeurs de nombreux paramètres tels que le niveau de batterie, la vitesse du gyropode et sa localisation par exemple. Elle lui permettrait de configurer certains paramètres du gyropode tels que la vitesse maximale, la sensibilité de la prise en main, ou l'activation d'un mode "économie d'énergie", "veille" ou "sport". Il serait également possible d'inclure un contrôle parental.

I.1.4 - Système de géolocalisation

Nous avons également pensé à implémenter un module GPS au gyropode pour rendre possible l'ajout de nouvelles fonctionnalités telles que :

- la géolocalisation du gyropode ;
- le retour vers l'utilisateur ;
- le guidage de l'utilisateur d'un point A à un point B.

I.1.5 - Système antivol

Afin d'éviter tout vol du Segway lorsqu'il est laissé en lieu public, nous avons pensé à implémenter un système antivol. Le principe serait d'intégrer un code PIN ou autre méthode d'identification au démarrage du Segway via l'interface graphique.

I.2 - Cahier des charges du contrôle à distance

Compte tenu du temps imparti pour finir ce projet, il nous était totalement impossible d'implémenter toutes les solutions décrites précédemment. Mme Baron, notre tutrice, nous a donc orientés vers le contrôle du gyropode à distance.

Le code implémenté sur la maquette du gyropode jusqu'à maintenant permet un contrôle de la vitesse à partir de l'inclinaison vers l'avant ou l'arrière de l'utilisateur (mesurée par un gyroscope et un accéléromètre). Pour diriger le gyropode, l'utilisateur fait varier l'inclinaison du guidon vers la gauche ou la droite, dont l'angle est mesuré grâce à une jauge de contrainte. Ce fonctionnement sera à présent dénommé mode *nominal* en opposition au mode *contrôle à distance* que nous désirons implémenter.

Nous devons alors changer le code des cartes Raspberry Pi 3 et STM32 pour réussir à piloter le gyropode sans utilisateur, et donc sans les consignes calculées à partir des valeurs d'angle et de

vitesse angulaire mesurées par les capteurs. Le système à étudier est l'application* permettant de contrôler le gyropode à distance.

Les exigences que nous avons fixées sont les suivantes :

- Exigences opérationnelles :

  - L'utilisateur doit pouvoir contrôler* le gyropode sans aucun contact physique direct.
  - Le gyropode doit pouvoir être commandé avec un temps de réponse inférieur à 0,1s.
  - L'utilisateur doit pouvoir intervertir entre le mode *contrôle à distance* et le mode *nominal* en moins de 1s.
  - L'utilisateur doit pouvoir commander le gyropode sans interruption tant que les batteries le permettent.
  - L'utilisateur doit avoir accès à une interface graphique simple afin de contrôler le gyropode.
  - L'utilisateur doit pouvoir se déplacer tout en contrôlant le gyropode.

- Exigences dues à la sécurité :

  - La vitesse maximale du gyropode ne doit pas dépasser $2 m.s^{-1}$.
  - Le gyropode doit être contrôlable à distance par au maximum un utilisateur simultanément.
  - En cas de perte de connexion, le gyropode doit s'arrêter.
  - Les consignes envoyées ne doivent pas mettre en péril le hardware du gyropode (protection contre les surtensions ou sur-courants).
  - L'utilisateur doit pouvoir utiliser le gyropode en mode *nominal* sans que ce dernier soit perturbé par l'implémentation du mode *contrôle en distance*.

- Exigences fonctionnelles :

  - L'application doit se connecter au gyropode.
  - L'application doit permettre de contrôler* le gyropode.
  - L'application doit envoyer les valeurs de consigne en moins de 0,07s.
  - L'application doit envoyer les commandes de changement de mode en moins de 1s.
  - L'application ne doit pas s'arrêter tant que l'utilisateur ne l'a pas demandé.
  - L'application doit contenir une interface graphique avec un joystick.

- L'application doit pouvoir être lancée à partir d'un appareil mobile de type téléphone ou tablette.
- L'application doit envoyer aux moteurs une consigne de tension inférieure à 12V.
- L'application doit empêcher la commande simultanée du gyropode par deux utilisateurs différents.
- L'application doit envoyer une consigne de couple nulle en cas de perte de connexion avec la carte Raspberry.
- L'application doit autoriser le fonctionnement mode *nominal* lorsque le mode *contrôle à distance* est désactivé.

Lexique :

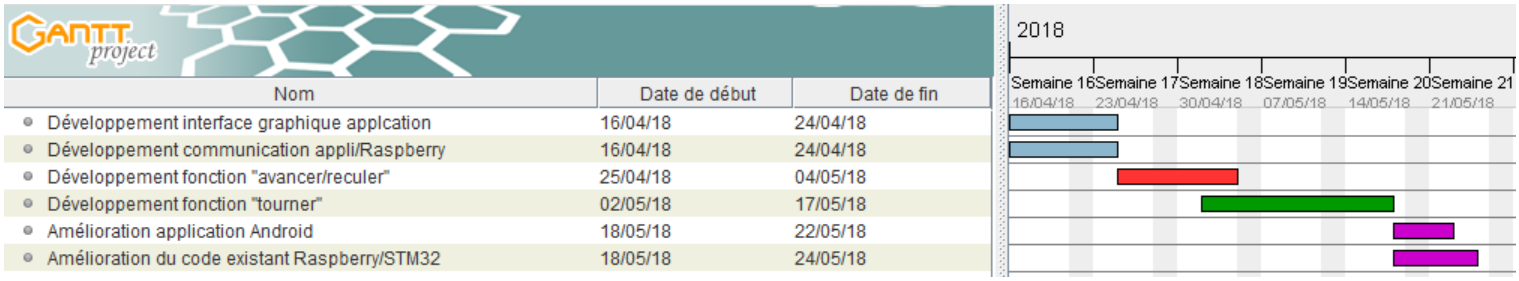
- *Contrôler :* Faire avancer, reculer ou tourner le gyropode à la vitesse souhaitée par l'utilisateur et être en mesure de l'arrêter.
- *Application :* Comprend l'application distante et le code implémenté sur les cartes Raspberry Pi et STM32 permettant de commander le gyropode à distance.

Pour rendre notre gyropode connecté et le contrôler à distance, nous avons choisi de passer par une application mobile Android au lieu d'un site web comme nous l'avions pensé lors de notre recherche bibliographique. Cette application Android et le code implémenté sur les cartes Raspberry Pi 3 et STM32 permettront de répondre à notre cahier des charges.

I.3 - Organisation du projet et problèmes rencontrés

I.3.1 - Organisation du projet

Compte tenu de notre cahier des charges, nous avons défini des tâches à effectuer. Pour organiser le projet dans le temps et se répartir les tâches, nous avons réalisé un diagramme de GANTT ci-dessous.



Nom	Date de début	Date de fin
Développement interface graphique applcation	16/04/18	24/04/18
Développement communication appli/Raspberry	16/04/18	24/04/18
Développement fonction "avancer/reculer"	25/04/18	04/05/18
Développement fonction "tourner"	02/05/18	17/05/18
Amélioration application Android	18/05/18	22/05/18
Amélioration du code existant Raspberry/STM32	18/05/18	24/05/18

Figure 1 : Diagramme de GANTT de notre projet

Nous avons parallélisé certaines tâches car elles peuvent être effectuées en même temps. Par exemple, nous avons prévu de faire d'un seul coup le développement complet de l'application comprenant l'interface graphique et la communication avec la carte Raspberry Pi 3 de la maquette du gyropode. De ce fait, nous avons gagné du temps. Ensuite, nous avons décidé de ne pas trop parallélisé les deux fonctions à implémenter au niveau du code de la carte Raspberry et de la carte STM32. En effet, nous voulions bien avancer la première fonction avant de se lancer dans la deuxième car elles sont dépendantes l'une de l'autre. De même, les deux dernières tâches en parallèle concernent l'amélioration, donc nous avons décidé de ne pas les paralléliser avec la partie développement.

Nous n'avons pas eu le temps de finir toutes les tâches que nous avons prévues, notamment à cause des problèmes détaillés ci-dessous. En effet, nous n'avons pas effectué les deux tâches d'amélioration. Nous avons peut-être été ambitieux parce que nous n'avions pas pensé rencontrer autant de problèmes.

I.3.2 - Problèmes liés au temps accordé au projet

Dans un premier temps, jusqu'à début avril, nous avons aidé à la réalisation d'un sujet de Travaux Dirigés et de Travaux Pratiques concernant l'UF Programmation orientée objet et Temps réel, et plus particulièrement le contrôle du gyropode en temps réel. Les Travaux Pratiques consistaient à développer les tâches du gyropode sur un simulateur comprenant une carte Raspberry Pi 3 et une carte STM32 reliées entre elles à l'aide d'une liaison série UART. Nous avons donc commencé réellement notre travail sur la maquette du gyropode mi-avril. Nous avons réussi notre mission, mais nous aurions pu implémenter beaucoup plus de fonctionnalités si nous avions eu plus de temps.

De plus, nous avons eu besoin de la maquette du gyropode pour travailler. En effet, afin de trouver les erreurs dans notre code implémenté, il fallait lancer le programme sur la maquette. Nous avons donc travaillé en grande partie à l'INSA. Pour cela, nous avions dans l'emploi du temps des créneaux réservés au projet tutoré. Malheureusement, ces créneaux étaient placés tous les mardis après-midi alors que deux d'entre nous avaient cours de langue de 15h30 à 18h15. Le créneau se limitait donc à 1h30 en début d'après-midi, ce qui n'était pas suffisant pour avancer correctement. C'est pourquoi nous revenions le jeudi après-midi pour compenser.

I.3.3 - Problèmes liés à la répartition des tâches

La répartition des tâches au sein de notre groupe a été compliquée. En effet, nous étions quatre au début du projet, puis l'un d'entre nous n'est plus venu en cours à partir de mars et nous ne l'avons pas revu depuis.

De plus, la maquette du gyropode étant à l'INSA, nous n'avons pas pu tester notre code chez nous. Cependant, grâce à la plate-forme GitHub, nous avons pu centraliser le code et le changer à partir de différents ordinateurs. Il suffit de récupérer le code sur la plate-forme, de le modifier et de le déposer à nouveau sur la plate-forme.

Nous avons donc pu travailler en parallèle sur le même code. Comme l'ensemble de notre travail était basé sur l'amélioration du code, il était primordial de le développer à l'aide d'une plate-forme de centralisation de ce code.

II – REALISATION DE L'APPLICATION DE CONTROLE A DISTANCE

II.1 - L'application mobile

Afin de commander le gyropode à distance, nous avons choisi de développer une application mobile Android en JAVA que nous avons développé sous Android Studio. Nous avons choisi cet IDE car il est facile et agréable à utiliser et il y a de nombreuses ressources à son sujet facilement accessible sur internet. Le but de cette application était donc dans un premier temps, de récupérer la consigne de l'utilisateur (avancer, reculer, tourner) et dans un second temps, de la transmettre au gyropode.

II.1.1 - Réalisation d'un joystick virtuel

Afin de commander le gyropode, il fallait récupérer une valeur de puissance à transmettre allant de 0 à 100%, un angle allant de 0 à 180°, ainsi qu'un sens de déplacement. Nous avons donc pensé que dessiner un joystick sur l'interface de l'application serait une bonne option, car celui-ci nous permettrait de récupérer ces trois informations. Afin de le réaliser, nous avons suivi des tutoriels sur internet de manière à comprendre le fonctionnement d'Android Studio et de Java.

La classe "JoystickView", est une "SurfaceView" qui est une classe définie dans les librairies d'Android Studio, permettant de dessiner sur une surface de l'écran. Pour des soucis de lisibilité, la définition de cette classe est donnée en Annexe 1.

Dans le "main" de notre application, le programme affiche la vue définie dans "activity_main.xml" que vous trouverez en Annexe 2. Ce fichier comporte un "Layout" contenant deux "Button" : un pour se connecter et un pour se déconnecter, un "EditText" permettant de rentrer l'adresse IP sur laquelle on veut se connecter ainsi que l'objet défini par notre classe JoystickView. Le placement de ces objets par rapport à la taille de l'écran utilisé est réalisé dans le fichier "MainActivity.java". La partie de code plaçant ces objets est disponible en Annexe 3. Cette partie de code nous permet d'avoir une interface principale, étant l'activité principale, affichée ci-dessous.

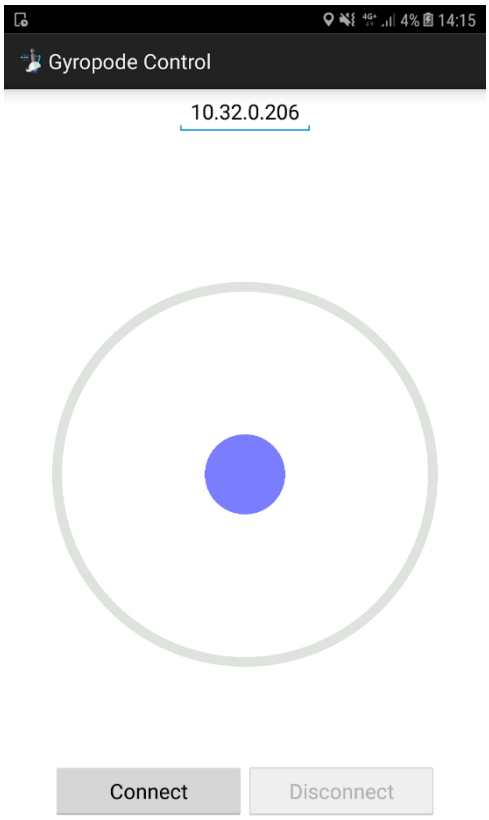


Figure 2 : Interface graphique principale de l'application distante

II.1.2 - Envoi de donnée

Une fois le joystick réalisé et opérationnel sur l'application, nous avons dû envoyer les valeurs de consigne au gyropode. Nous avons choisi de créer un socket pour établir une connexion WI-Fi. Nous avons mis le client côté application Android.

Nous avons ensuite ajouté à l'interface contenant uniquement le Joystick un espace texte permettant d'entrer l'adresse IP du serveur auquel on veut se connecter. Nous avons également ajouté un bouton "Connect" permettant de créer le socket et de se connecter au serveur. Si l'application n'arrive pas à se connecter au serveur au bout de 5 secondes, elle en informe l'utilisateur de l'application et n'essaie plus de s'y connecter. L'application bénéficie aussi d'un bouton "Disconnect" permettant de fermer le socket. Les fonctions gérant les évènements d'appui de bouton sont disponibles en Annexe 4.

Afin d'ouvrir un socket et de s'y connecter, nous avons créé une Thread que vous trouverez en Annexe 5, nommée "clientThread", qui crée le socket et tente de s'y connecter avec un timeout de 5 secondes.

Afin d'envoyer les messages de consigne une fois le socket ouvert et la communication établie, nous avons créé une Thread nommée MessageThread, que vous trouverez en Annexe 6, qui prend en paramètre le socket sur lequel elle doit envoyer les informations ainsi que le message à envoyer, qui est un tableau de caractère. Nous avons défini une méthode permettant de modifier ce tableau de caractère. Lorsque l'on lance cette Thread, elle envoie les messages de consigne, d'angle et de sens à une fréquence de 50Hz.

Ce diagramme explique le fonctionnement du programme Android :

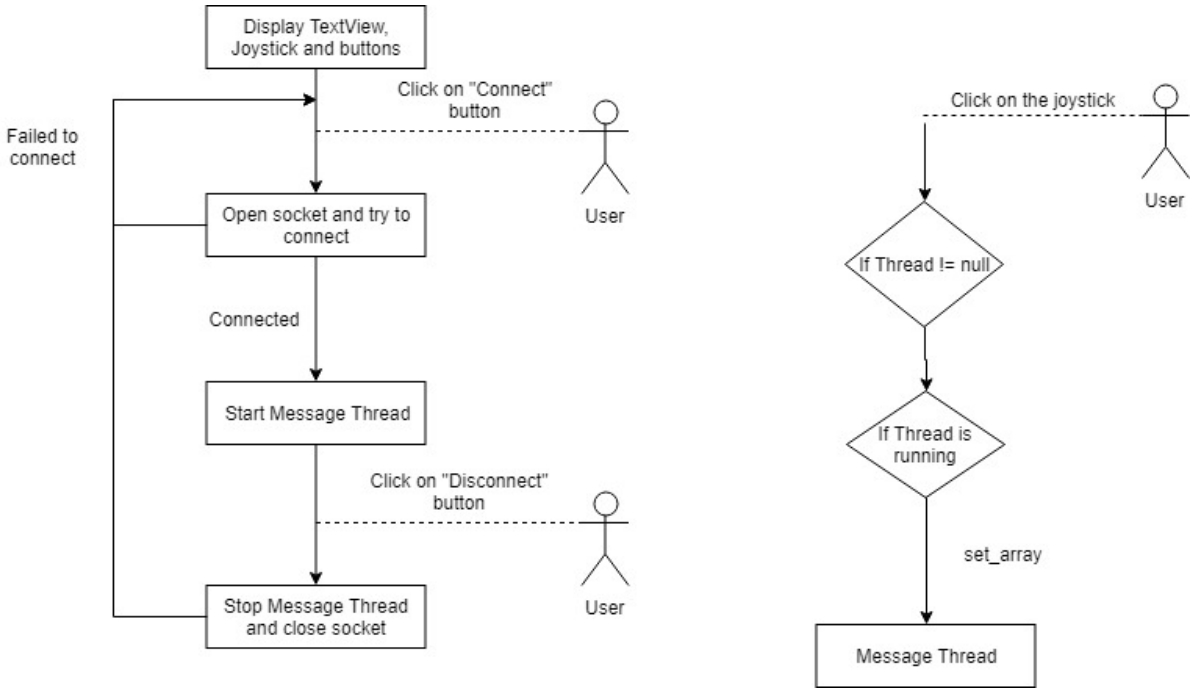


Figure 3 : Diagramme application Android

II.2 - Le code de la Raspberry Pi 3

Nos prédécesseurs avaient organisé la commande du gyropode de la manière suivante : une carte Raspberry gère toutes les consignes de sécurité, (présence utilisateur, arrêt d'urgence, niveau de batterie) ainsi que la consigne de couple pour avancer ou reculer et l'interface graphique et parallèlement, la carte STM32 récupère les données des capteurs physiques (accéléromètre, gyroscope, jauge de contrainte pour les virages) et commande en courant les moteurs. La commande en courant est effectuée grâce à la consigne de couple envoyée par la carte Raspberry. La carte STM32 gère aussi le calcul de la consigne pour tourner.

Nous avons dû dans un premier temps modifier le code déjà existant, hébergé sur la carte Raspberry Pi 3 afin de prendre en compte la consigne utilisateur dans l'asservissement. Nous avons modifié les conditions de fonctionnement comme par exemple, ne pas prendre en compte la non

présence de l'utilisateur lorsque l'on commande le gyropode à distance. Sinon, le gyropode ne pourrait pas se déplacer sans que l'un des boutons de sécurité soient enfoncés.

Afin de pouvoir mettre notre code sur la carte, le compiler et le tester, nous avons principalement utilisé les commandes de GIT ainsi qu'une liaison ssh, grâce à la commande scp nous pouvions directement copier les fichiers de notre machine vers la carte. Nous avons codé sous l'IDE NetBeans.

Nous avons ajouté au programme de la carte une nouvelle tâche, nommée Communication Android. Cette tâche se lance à l'exécution du programme, crée un socket, rentre dans sa boucle, fixe la variable globale "android" à 0, puis attend une connexion. Lorsqu'un client se connecte, la variable globale "android" est fixée à 1, puis la tâche lit les messages qu'elle reçoit de son client, en déduit les valeurs de consigne de couple et d'angle et les mets dans la queue de message du STM32. Lorsque le client se déconnecte, la tâche repart en début de boucle, fixe la variable android à 0 et attend une connexion. Vous trouverez le détail de cette tâche en Annexe 7.

Nous avons conditionné les tâches d'Asservissement et d'Arrêt d'Urgence de manière que celles-ci ne mettent leurs valeurs de consigne ou d'arrêt d'urgence dans la queue de message du STM32 uniquement si la variable android est à 0 et donc qu'aucun utilisateur n'est connecté au gyropode via l'application Android.

II.3 - Code du STM32

Nous avons dû ensuite modifier le code du STM32 afin de pouvoir contrôler à distance les virages du gyropode. Nos prédécesseurs l'avaient codé de manière à ce que celui-ci reçoive la consigne de couple de la part de la carte Raspberry, puis qu'il adapte lui-même la consigne entre le moteur 1 et le moteur 2 en fonction de la flexion de la jauge de contrainte située sur le bas du guidon.

Par manque de temps, nous n'avons pas modifié la manière dont l'asservissement du gyropode est fait et nous nous sommes uniquement intéressés à la commande via notre application.

Grâce aux modifications apportées sur la carte Raspberry, maintenant, lorsqu'un utilisateur est connecté à la carte Raspberry via l'application Android, les valeurs de consigne et d'angle sont transmises depuis l'application vers la carte Raspberry via un socket, puis de la carte Raspberry à la carte STM32 via la liaison UART.

Afin de pouvoir coder sur la carte STM32, il a fallu installer l'IDE µVision et le configurer de manière à pouvoir compiler et loader le code sur la carte.

Nous avons donc rajouté une variable globale "android" dans le code du STM32. Celle-ci passe à 1 lorsque le STM32 reçoit une valeur d'angle et elle repasse à 0 lorsque le STM32 reçoit une valeur d'arrêt d'urgence (signifiant que personne n'est connecté à la carte Raspberry). Une partie du code de la réception de message du STM32 via la liaison série est disponible ci-dessous.

```
//cas de réception de trame de consigne
        case 'c' : Consigne_OK=1;
                   Consigne_new=value/0.80435f;
                  // détection si le consigne ancien est le même
                  // précaution de erreur de communication
                  if(old_cons == Consigne_new){
                            cmpt_egal++;
                  } else {
                            cmpt_egal = 0;
                            STM_EVAL_LEDOff(LED5);
                  }
                  old_cons = Consigne_new;
                  STM_EVAL_LEDOn(LED3);
                  cmpt_arr++;
                  break;
//cas de réception de trame de arret urgence
        case 'a' : arret=1;
                  cmpt_arr = 0;
                  android = 0;
                  break;
//cas de réception d'angle de consigne
        case 'd' : android=1;
                  angleAndroid = value;
                  break;

// des autres cas pas défini
        default : Consigne_OK= 0;
                  cmpt_arr++;
```

Dans l'asservissement des moteurs, nous avons donc rajouté une condition : si android est à 1, et donc qu'un utilisateur est connecté, nous adaptons la consigne des moteurs en fonction de la valeur d'angle reçue. La partie du code concernant l'asservissement en courant des moteurs est disponible ci-dessous.

```
if (Consigne_OK==1){                              // Si c'est bien une nouvelle trame reçu
        if (android){                             // Si la communication android est établie
                  if (angleAndroid < 90){   //On a donc une valeur d'angle
                            Consigne2 = Consigne_new;
                            Consigne1 = (angleAndroid/90)*Consigne_new;
                  } else if (angleAndroid >= 90){
                            Consigne2 = ((180.0-angleAndroid)/90)*Consigne_new*0.5;
                            Consigne1 = Consigne_new;
                  }
                  Consigne_OK = 0;
                  } else {
                            Consigne1=Consigne_new;
                            Consigne2=Consigne_new;
                            Consigne_OK = 0;
                  }
        }
```

II.4 - Difficultés techniques rencontrés lors de la réalisation du projet

Nous pensions que le contrôle de position du gyropode nous prendrait que quelques semaines, sachant que, comme expliqué précédemment, il suffisait de créer une application Android relativement basique et d'ajouter quelques lignes de codes sur la Raspberry pi puis sur le STM32. Cependant les difficultés suivantes que nous n'avions pas prévues nous ont énormément ralenti dans la progression de notre projet.

II.4.1 Difficultés liées à l'environnement de travail

Tout d'abord, comme nous sommes repartis d'un travail préexistant, il a été difficile de s'approprier le code déjà en place, comprendre les choix d'architectures qui avaient été fait et naviguer au milieu des diverses contraintes que ceux-ci imposent. Nous avions un peu de documentation afin de nous aiguiller sur le fonctionnement du code et de l'architecture choisie, mais il nous manquait de nombreuses informations.

De plus l'architecture réseau de l'INSA ne nous a pas aidé. Au début nous voulions utiliser l'IDE NetBeans afin de pouvoir compiler et exécuter le code sur la carte Raspberry à partir d'un ordinateur. Cependant les ordinateurs INSA sont sur le réseau IOT alors que la carte Raspberry ne l'était pas et les sécurités du groupe INSA ne permettent pas d'utiliser des commandes tel que ssh -x depuis le réseau IOT sur un autre réseau. Nous avons donc dû chercher un moyen pour éditer le code de la carte Raspberry, le compiler et le transférer sur la carte afin de l'exécuter. La carte étant connecté au réseau InviteINSA, nous avons opté pour l'utilisation de GitHub.

II.4.2 Mise en place et débug de l'application Android

Le développement de l'application ainsi que la mise en place de la communication avec la carte Raspberry et l'échange de données ont rapidement été mise en place sans soucis majeur.

Nous avons donc tenté de modifier la consigne d'asservissement sur la carte Raspberry afin de contrôler la marche avant et arrière. La transmission de message et le changement de la consigne fonctionnaient modulo des singularités détaillées ci-dessous.

Tout d'abord, nous avons remarqué qu'après un certain temps d'utilisation pouvant varier de quelques secondes à plusieurs dizaines de secondes, le terminal sur lequel s'exécutait l'application s'interrompait brusquement sans aucun message d'erreur.

Après de nombreux tests et modifications afin de débuguer ce problème, tel que le changement de l'organisation des files de messages entre les cartes Raspberry et STM32, nous sommes arrivés à la conclusion que le problème venait certainement d'une mauvaise gestion des messages lors de leur émission par l'application Android.

Nous avons donc modifié le code Android afin de mieux définir la taille des messages envoyés, et le bug a été résolu.

Nous avons aussi dû modifier la gestion de la consigne 0 couples qui n'était pas correctement prise en compte lors du traitement du message. En effet, nos prédécesseurs avaient codé la carte STM32 tel que s'il recevait 10 fois la même consigne, il prenait la main sur la carte Raspberry Pi, ceci ne pose pas de problème en général puisque n'ayant pas une précision humaine élevée, les consignes que nous envoyons n'étaient jamais exactement égale. Sauf quand il s'agit de la consigne 0,0. Nous avons donc ajouté quelques lignes de code pour faire varier d'une fois à l'autre la consigne de plus ou moins 0,001.

A la suite de ces modification, l'application Android était tout à fait opérationnelle.

II.4.3 Débug du code de la carte Raspberry Pi

Nous avons ensuite dû résoudre un problème que nous avons mis du temps à analyser. Après un temps d'utilisation variable ou une série répétée de connexion et déconnexion de l'application Android, le gyropode prenait le contrôle et avançait tout seul.

Nous avons mis de nombreux "flag" afin de s'assurer de l'envoi correct des consignes, jusqu'à comprendre que le problème venait de la gestion des mutex et des priorités.

La tâche « Communication Android » lisait et écrivait sur des variables globales partagées, et de temps en temps, deux Thread voulant modifier la même variable cette dernière ne s'actualisait pas comme nous le pensions.

Il arrivait donc un moment où lorsque la connexion entre l'application Android et la carte Raspberry s'arrêtait (volontairement ou non) la variable "etat_Android" passait à 0, sans mettre la variable "arrêt" à 1 et le gyropode passait en mode "nominal" sans prendre en compte la non-présence de l'utilisateur.

Nous avons donc tenté de jouer sur les priorités afin de résoudre ce problème sans succès. En mettant une plus haute priorité à la Thread "Communication Android" qu'à la Thread "Asservissement", celle-ci n'avait plus accès assez fréquemment à la consigne et avait un temps de réponse trop lent et dans le cas inverse, de temps en temps "Asservissement" ne permettait pas à "Communication Android" d'actualiser correctement sa variable "etat_android" ce qui était à l'origine du comportement observé (le gyropode avançait sans consigne de notre part).

Nous avons donc tenté d'envoyer directement la consigne calculée au STM32 sans la stocker dans une variable globale mais cela n'a pas fonctionné. Nous avons donc pris le parti de ne pas protéger la variable "etat android" par un mutex afin que même une Thread de faible priorité puisse la modifier momentanément malgré la "présence" d'une tâche de plus forte priorité.

Après avoir totalement débugué notre code, le gyropode avançait et reculait sans soucis en respectant toutes les exigences que nous nous étions fixés en début de projet.

II.4.3 Modification du code du STM32

Il ne restait plus qu'à commander les virages en modifiant légèrement le code du STM32. Ce fut rapide et après quelques optimisations de temps de réponse et de comportement "transitoire", le gyropode marche parfaitement en mode "contrôle à distance".

La plupart des difficultés rencontrées viennent du fait que nous débutons en programmation de temps réel, en gestion d'une application Android et d'une communication entre deux applications comme celles-là. Nous avons donc passé du temps à comprendre certains comportements non-désirés du gyropode et à résoudre les problèmes rencontrés.

Nous avons également dû faire face à d'autres contretemps tel que la subite augmentation du temps de réponse du système qui n'était en fait dû qu'à un niveau de batterie trop bas et fût réglé dès lors que nous avons recharger les batteries.

Finalement, bien que nous n'ayons pas correctement évalué le temps que cela demandait, nous sommes parvenus au résultat, c'est-à-dire contrôler le gyropode à distance.

CONCLUSION

Ce projet tutoré nous a permis de travailler sur un projet ambitieux et intéressant. Bien qu'encadrer par notre tutrice Mme Baron, le travail en autonomie nous a demandé de développer une organisation de notre travail personnel et une gestion du temps alloué à ce projet. Tout cela en prenant en compte les emplois du temps et disponibilités de chacun.

Ce n'a pas été évident d'estimer le temps dont nous avions besoin pour développer le mode "contrôle à distance" puisque nous n'avions jamais fait de projet tel que celui-là et que nous n'avions pas l'expérience nécessaire pour anticiper les problèmes que nous avons rencontré.

D'un point de vue personnel, ce projet nous a demandé une capacité d'adaptation et d'apprentissage par nous-même importante. En effet nous avons été formés au codage en C et C++ mais nous ne connaissions pas Xenomai et les singularités du code en temps réel avant le début de cette année. Et bien que les cours de ce semestre nous aient permis de comprendre les bases de ce type de programmation, nous avons dû nous auto-former pour mener notre projet à bien.

Nous n'avions jamais codé en JAVA non plus et nous n'avions qu'une petite expérience de la création de socket et de l'envoie de données via WIFI entre deux applications. Ce sont en partie les raisons pour lesquelles nous avons eu beaucoup de bugs au début, mais nous avons pu apprendre de nos erreurs et les corriger au fur et à mesure que notre compréhension du système évoluait.

Afin de résoudre les problèmes de notre programme nous avons dû mettre en place des phases de test importantes afin de vérifier le bon fonctionnement de nos fonctions, de chaque étape du programme et pour détecter les instructions qui posaient problèmes ou les singularités que nous n'avions pas pensé traiter en premier lieu.

Ce projet a donc été très formateur. Nous aurions aimé avoir plus de temps à consacrer au développement de nouvelles fonctions car nous commençons à bien nous approprier le système et nous sommes convaincus que nous aurions pu développer une interface utilisateur bien plus performante avec quelque semaines supplémentaires.

REMERCIEMENTS

Nous tenons à remercier tout particulièrement Mme Baron qui nous a accompagnés tout au long du projet. De plus, nous souhaitons aussi remercier M.Senaneuch, M. Hladik, M.Di Mercurio, M.Martin et M.Lombard de nous avoir aidés à la réalisation de notre projet.

BIBLIOGRAPHIE

Notre travail étant principalement pratique, nous n'avons pas utilisé de nombreuses sources externes excepté les documentations d'Android Studio, de Xenomai et de Keil µVision.

ANNEXES

ANNEXE 1 : CODE DE LA CLASSE JOYSTICKVIEW

```
public class JoystickView extends SurfaceView implements SurfaceHolder.Callback,
View.OnTouchListener {
    private float centerX, centerY, baseRadius, hatRadius;
    private JoystickListener joystickCallback;

//Fonction permettant de définir les tailles et positions des deux cercles définissant
notre joystick
    private void setupDimensions() {
        centerX = getWidth() / 2;      //Centre selon X
        centerY = getHeight() /2;      //Centre selon Y
//Rayon du grand cercle délimitant le joystick
        baseRadius = Math.min(getWidth(), getHeight())/2.5f;
//Rayon du petit cercle
        hatRadius = Math.min(getWidth(), getHeight())/12;

    }

//Constructeur
    public JoystickView(Context context) {
        super(context);
        getHolder().addCallback(this);
        setOnTouchListener(this);
        if (context instanceof JoystickListener)
            joystickCallback = (JoystickListener) context;
    }

//Constructeur
    public JoystickView(Context context, AttributeSet attributes) {
        super(context, attributes);
        getHolder().addCallback(this);
        setOnTouchListener(this);
        if (context instanceof JoystickListener)
            joystickCallback = (JoystickListener) context;
    }

//Constructeur
    public JoystickView(Context context, AttributeSet attributes, int style) {
        super(context, attributes, style);
        getHolder().addCallback(this);
        setOnTouchListener(this);
        if (context instanceof JoystickListener)
            joystickCallback = (JoystickListener) context;
    }

//Fonction permettant de dessiner notre joystick
    private void drawJoystick(float newX, float newY) {
        if (getHolder().getSurface().isValid()) {
            Canvas myCanvas = this.getHolder().lockCanvas(); //Canvas à dessiner
            Paint colors = new Paint();      //Objet de couleurs
            myCanvas.drawColor(Color.WHITE); //Efface l'arrière-plan
            colors.setARGB(25,50,50,50);//Définit la couleur du grand cercle
            myCanvas.drawCircle(centerX, centerY, baseRadius, colors);
//Dessine le grand cercle
            colors.setColor(Color.WHITE);
```

```
            myCanvas.drawCircle(centerX,              centerY,              (baseRadius-
baseRadius*((float)0.05)), colors);
//Dessine un deuxième grand cercle blanc de manière à avoir un fin trait
            colors.setARGB(125,0,0,255); //Définit la couleur de la base
            myCanvas.drawCircle(newX, newY, hatRadius, colors); //Dessine la base
            getHolder().unlockCanvasAndPost(myCanvas);
//Écrit le nouveau dessin dans la SurfaceView
        }
    }

//Fonction appelée à la création de la surface
//Dessine le joystick complet
    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        setupDimensions();
        drawJoystick(centerX, centerY);
    }

//Fonction appelée lorsque la surface change
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height)
{
    }

//Fonction appelée lorsque la surface est détruite
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
    }


//Fonction appellee dès que l'utilisateur touche la SurfaceView
    public boolean onTouch(View v, MotionEvent e) {
        if (v.equals(this)) {
//Si l'événement n'est pas un levé de doigt
            if (e.getAction() != e.ACTION_UP) {
//Distance entre le centre et la position de l'évènement
                float displacement = (float) Math.sqrt((Math.pow(e.getX() - centerX, 2))
+ Math.pow(e.getY() - centerY, 2));
                float ratio = baseRadius / displacement;
      //La consigne à envoyer est donc le rapport entre le déplacement et le rayon
                float consignePercent = displacement / baseRadius;
//Au cas où l'utilisateur soit en dehors du joystick
                if (consignePercent > 1)
                    consignePercent = 1;
      //Calcul de l'angle
                double angled = Math.acos((e.getX() - centerX) / displacement);
      //Conversion en degrés
                angled = angled * 180 / 3.14;
      //Conversion en float
                float angle = (float) angled;
                int sens = 0;
                if ((e.getY() - centerY) >= 0)
                    sens = 0; //Mode marche arrière
                else
                    sens = 1; //Mode marche avant
//Si on est à l'intérieur du joystick
                if (displacement < baseRadius) {
                    drawJoystick(e.getX(), e.getY());//Dessine le joystick
```

```
                    joystickCallback.onJoystickMoved(consignePercent,    angle,    sens,
getId());//Appelle la fonction permettant de mettre à jour les variables à envoyer
                } else { //On est en dehors du joystick
       //On dessine le joystick sur les bords du grand cercle
                    float constrainedX = centerX + (e.getX() - centerX) * ratio;
                    float constrainedY = centerY + (e.getY() - centerY) * ratio;
                    drawJoystick(constrainedX, constrainedY);
                    joystickCallback.onJoystickMoved(consignePercent,    angle,    sens,
getId());//Appelle la fonction permettant de mettre à jour les variables à envoyer
                }
            } else if (e.getAction() == e.ACTION_UP) {//Si l'action est le levé de doigt
       //On trace le joystick au centre
                drawJoystick(centerX, centerY);
                joystickCallback.onJoystickMoved((float)   0.10,(float)   90.0,   (int)1,
getId());//Appelle la fonction permettant de mettre à jour les variables à envoyer
            }
        }
        return true;
    }

//Définit la fonction appelée lorsqu'on touche l'écran
    public interface JoystickListener {
        void onJoystickMoved(float Puissance, float Angle, int sens, int id);
    }
}
```

ANNEXE 2 : FICHIER XML PRINCIPAL

```
//Joystick
<com.androidsrc.client.JoystickView
       android:layout_centerInParent="true"
        android:id="@+id/testJoystick"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

//LinearLayout contenant les deux boutons
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:id="@+id/button_layout"
    android:layout_width="match_parent"
    android:gravity="center_horizontal"
    android:layout_below="@id/testJoystick"
    android:layout_height="match_parent" >

    //Button connect
    <Button
        android:id="@+id/button_connect"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_connect_text" />

    //Button disconnect
    <Button
        android:id="@+id/button_disconnect"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_disconnect_text" />
</LinearLayout>

//Edit Text pour l'adresse IP
<EditText
    android:id="@+id/ip_adress"
    android:layout_centerHorizontal="true"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

ANNEXE 3 : CODE GERANT L'AFFICHAGE

```
//Fonction appelé lors de la création de l'activité « MainActivity »
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  //Affiche la vue définie dans le fichier xml « activity_main »
  setContentView(R.layout.activity_main);

  //Récupère la longueur et largeur de l'écran de l'utilisateur
  screen = new DisplayMetrics();
  getWindowManager().getDefaultDisplay().getMetrics(screen);
  screenWidth = screen.widthPixels;
  screenHeight = screen.heightPixels;

//Récupère l'objet Joystick et fixe sa hauteur et largeur à la largeur de          l'écran
  myJoystick = (JoystickView) findViewById(R.id.testJoystick);
  myJoystick.getLayoutParams().height = (int) (screenWidth);
  myJoystick.getLayoutParams().width = (int) (screenWidth);

  //Fixe une marge en dessous du joystick
  RelativeLayout.LayoutParams      marginParams      =      (RelativeLayout.LayoutParams)
myJoystick.getLayoutParams();
  marginParams.setMargins(0, 0, 0, (int) (0.1f * screenWidth));
  myJoystick.setLayoutParams(marginParams);

//Récupère les boutons, leur affecte une largeur de 40% de l'écran et y affecte un «
OnClickListener » pour pouvoir récupérer des évènements sur ces boutons
  connect = (Button) findViewById(R.id.button_connect);
  connect.getLayoutParams().width = (int) (0.4f * screenWidth);
  connect.setOnClickListener(connectionManager);
  disconnect = (Button) findViewById(R.id.button_disconnect);
  disconnect.getLayoutParams().width = (int) (0.4f * screenWidth);
  disconnect.setOnClickListener(connectionManager);
  disconnect.setEnabled(false);

//Récupère l'objet EditText et y écris l'adresse IP prédéfinie
  ip_adress = (EditText) findViewById(R.id.ip_adress);
  ip_adress.setText(SERVER_IP);
}
```

ANNEXE 4 : CODE DE GESTION DES EVENEMENTS BOUTONS

```
View.OnClickListener connectionManager = new View.OnClickListener() {
    //Fonction appelé lorsque l'utilisateur appuie sur l'écran
    @Override
    public void onClick(View v) {
         //Si on appuie sur le bouton connect
        if (v.getId() == connect.getId()) {

            //Création du socket
            socket = new Socket();

            //Création de la Thread de connection
            Thread clientThread = new Thread(new ClientThread());

            //Rend le bouton connect indisponible
            connect.setEnabled(false);
            Toast.makeText(MainActivity.this, "Trying to connect to" + SERVER_IP + "...",
Toast.LENGTH_SHORT).show();
            //Démarre la Thread et on attend qu'elle soit terminée
            clientThread.start();
            Thread.State state = clientThread.getState();
            while (state != Thread.State.TERMINATED) {
                state = clientThread.getState();
            }
            //Si la connection a réussi
            if (connected) {
                //Rend le bouton disconnect disponible
                disconnect.setEnabled(true);
                Toast.makeText(MainActivity.this,                          "Connected",
Toast.LENGTH_SHORT).show();

//Défini le message à envoyer grâce a la fonction setArray définie en dessous
                array = setArray(array, "0.10", "90.0", "1");
                //Démarre la Thread d'envoi de message
                myThread = new MessageThread(socket, array);
                myThread.start();
            //Si la connection a échoué
            } else {
                //Rend le bouton de connection disponible
                connect.setEnabled(true);
                Toast.makeText(MainActivity.this, "Failed to connect to " + SERVER_IP,
Toast.LENGTH_SHORT).show();
            }
        }
        //Si on appuie sur le bouton disconnect
        if (v.getId() == disconnect.getId()) {
            try {
                //Fermeture du socket
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            connected = false;
            //Rend le bouton connect disponible
            connect.setEnabled(true);
      //Rend le bouton disconnect indisponible
```

```
            disconnect.setEnabled(false);
            //Interrompt la Thread d'envoie de message
            myThread.interrupt();
        }
    }
};


public char[] setArray(char[] array, String puissance, String angle, String sens) {
    while (puissance.length()<4) {
        puissance = puissance + '0';
    }

    while (angle.length()<4) {
        angle = angle+ '0';
    }
    array[0] = '<';
    array[1] = 'p';
    array[2] = puissance.charAt(0);
    array[3] = puissance.charAt(1);
    array[4] = puissance.charAt(2);
    array[5] = puissance.charAt(3);
    array[6] = '<';
    array[7] = 'a';
    array[8] = angle.charAt(0);
    array[9] = angle.charAt(1);
    array[10] = angle.charAt(2);
    array[11] = angle.charAt(3);
    array[12] = '<';
    array[13] = 's';
    array[14] = sens.charAt(0);
    array[15] = '\0';
    return array;
}
```

ANNEXE 5 : CLIENT THREAD

```
class ClientThread implements Runnable {

    @Override
    public void run() {

        try {
            SERVER_IP = String.valueOf(ip_adress.getText());
            InetAddress serverAddr = InetAddress.getByName(SERVER_IP);

            int timeout = 5000;   // 5000 millis = 5 seconds
            // Créé un socket non connecté
            SocketAddress sockaddr = new InetSocketAddress(serverAddr, SERVERPORT);
            // Connecte ce socket au serveur avec un timeout
            // Si le timeout s'écoule, SocketTimeoutException est renvoyé
            socket.connect(sockaddr, timeout);

            if (socket.isConnected()) {
                connected = true;
            }

        } catch (UnknownHostException e1) {
            e1.printStackTrace();
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

ANNEXE 6 : MESSAGE THREAD

```
public class MessageThread extends Thread {

    private Socket _socket;
    private char[] _array;

    //Constructeur
    public MessageThread(Socket socket, char[] array) {
        _socket = socket;
        _array = array;
    }


    @Override
    public void run() {
        try {
             //PrintWriter permettant l'envoi de message
            PrintWriter out = new PrintWriter(new BufferedWriter(
                    new OutputStreamWriter(_socket.getOutputStream())),
                    true);
            while (true) {
                try
                {
                    //Attend une période de 20ms pour une fréquence de 50Hz
                    Thread.sleep(20);
                    //Envoie le message
                    out.println(_array);
                }
                catch (InterruptedException e)
                {
                    // Interromps la Thread si il y a un problème
                    Thread.currentThread().interrupt();
                }
            }
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    // Méthode permettant de changer le message qui s'envoie
    public void set_array(char[] array) {
        this._array = array;
    }


}
```

ANNEXE 7 : COMMUNICATION ANDROID

```
void Communication_Android (void *arg){

    rt_printf("Thread ANDROID : Debut de l'execution\n");
    log_task_entered();
    float puissance = 0.0;
    float angle = 0.0;
    int socket_desc , client_sock , c , read_size, sens;
    struct sockaddr_in server , client;

    //Create socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1){
        rt_printf("Could not create socket\n");
    } else {
        rt_printf("Socket created\n");
    }

    //Prepare the sockaddr_in structure
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 8888 );

    //Bind
    if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0){
        rt_printf("bind failed. Error\n");
    } else{
        rt_printf("bind done\n");
    }

    while (1){
         //Début de boucle : pas de connexion établie donc android = 0
        android = 0;

        //Listen
        listen(socket_desc , 1);

        //Accept and incoming connection
        rt_printf("Waiting for incoming connections...\n");
        c = sizeof(struct sockaddr_in);

        //accept connection from an incoming client
        client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c);
        if (client_sock < 0)
        {
            rt_printf("accept failed\n");
        } else {
            rt_printf("Connection accepted\n");
             android = 1;
        }



        int MAX_SIZE = 100;
```

```
int i, len;
int noerror = 0;
char tab[MAX_SIZE];
char subtab[4];
char senstab[2];
char string[MAX_SIZE];
/* lecture d'un message de taille max MAX_SIZE, information dans string */
    while( (len = recv(client_sock , string , MAX_SIZE , 0)) > 0 ){
            read_size=strlen((char*)string);
            if(read_size > 0){
     //Si le message n'est pas vide, on copie ces informations dans tab
                memcpy(tab,string,read_size);
                for(i=0 ; i<(read_size-1) ; i++){
                    if(tab[i] == '<'){
                        switch (tab[i+1]){
                            //puissance
                            case 'p':
                                    subtab[0] =  tab[i+2];
                                    subtab[1] =  tab[i+3];
                                    subtab[2] =  tab[i+4];
                                    subtab[3] =  tab[i+5];
                                    puissance = atof(subtab);
                                    break;
                            //angle
                            case 'a':
                                    subtab[0] =  tab[i+2];
                                    subtab[1] =  tab[i+3];
                                    subtab[2] =  tab[i+4];
                                    subtab[3] =  tab[i+5];
                                    angle = atof(subtab);
                                    break;
                            //sens
                            case 's':
                                    senstab[0] = tab[i+2];
                                    sens = atoi(senstab);
                                    break;
                        }//case
                    } //<\n
                } //for
                    if (!sens){
                        puissance = -puissance;
                    }
                //En réalité on peut aller jusqu'à 10A, on se limite ici à 6A
                float c = puissance*6;
                int err=0;

                message_stm m;
                m.label = 'c';

                //Alterne les consignes car il y a une sécurité au niveau du STM32 :
                si  il  reçoit  10  fois  la  même  consigne,  il  prend  le  relais  de
                l'asservissement
                if (noerror){
                    noerror = 0;
                    m.fval = c+0.1;
                } else {
                    noerror = 1;
                    m.fval = c;
                }
```

```
                    //Met les messages dans le queue de message du STM32
                     err=rt_queue_write(&queue_Msg2STM,&m,sizeof(message_stm),Q_NORMAL);

                     message_stm d;
                     d.label = 'd';
                     d.fval = angle;
                     err=rt_queue_write(&queue_Msg2STM,&d,sizeof(message_stm),Q_NORMAL);
                } //if taille*/
            } //while
            if(len <= 0)
            {
                android = 0;
                rt_printf("Client disconnected\n");
            }
            else if(len == -1)
            {
                android = 0;
                rt_printf("recv failed\n");
            }

        }
}
```

L

Département de Génie Electrique & Informatique

[CONTROLE D'UN GYROPODE EN TEMPS REEL]

RESUME :

Ce rapport présente le travail effectué par des étudiants de 4ème année en ingénierie des systèmes sur le contrôle d'un gyropode en temps réel. Il explique comment une application Android peut contrôler à distance un gyropode muni d'une carte Raspberry Pi 3 permettant de récupérer les données et d'une carte STM32 permettant de contrôler les moteurs du gyropode.

MOTS-CLES : *TEMPS REEL, XENOMAI, ANDROID, STM32, RASPBERY PI 3*

ABSTRACT :

This report presents the study of the real time control of a gyropode, realized by a group of fourth year student in Systems Engineering. It explains how an Android application can remotely control a gyropode using a Raspberry Pi 3 to get all the datas and a STM32 to control the motors of the gyropode.

KEYWORDS : *Real time, Xenomai, Android, STM32, Raspberry Pi3*