TOMMY ANDREWS

JÉRÉMY CHARLAND

Projet d'intégration en sciences informatiques et mathématiques

420-204-RE

Gr. 101

Rapport final :
Drone contrôlé par un gant

Travail présenté à

M Robert TURENNE

Département d'informatique

Cégep de Saint-Jérôme

Rapport remis le 18 mai 2018

Table des matières

Description du projet

L'objectif initial était de construire un drone contrôlé par un gant. Cependant, un tel projet nécessite trop de dépenses. En effet, la base du drone à elle seule, excluant les pièces servant au contrôle du drone en soi, coûterait au-dessus de 200 $ (moteurs, corps du drone, hélices, piles, contrôleurs de vitesse, etc.). Pour débuter, il serait donc préférable de tenter de *hack* les commandes d'un drone existant à faible coût à partir de la manette. Ainsi, cela consisterait en la version 1.0 du projet : réussir à faire passer les commandes provenant d'un gant équipé d'un gyroscope au travers du transmetteur radio de la manette, de façon à remplacer les commandes de base (analogues ou digitales). Si les concepts nécessaires au fonctionnement du contrôleur gyroscopique sont assimilés assez rapidement et que la version 1.0 vient à terme assez tôt dans la période de réalisation, il serait alors envisageable de passer au développement de la version 2.0, c'est-à-dire la construction d'un drone avec l'intégration du concept du gant.

Version 1.0

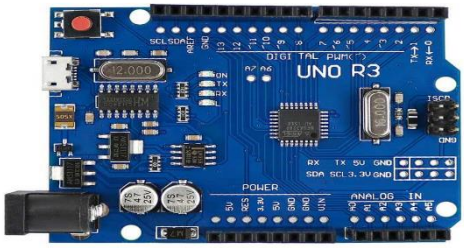
Nous avons sélectionné ce mini drone FuriBee H815, car il semble stable et bien répondre aux commandes de la manette. C'est un modèle miniature, ce qui semble être la meilleure option quant à l'expérimentation. La première étape consiste à ouvrir la manette et à comprendre le fonctionnement des commandes, c'est-à-dire identifier le type de commande (analogue ou digitale) des *joysticks*. Il est presque assuré que les commandes soient analogues, alors la prochaine étape serait fort probablement de remplacer les potentiomètres/interrupteurs par un potentiomètre digital lié au Arduino du gant. La direction du drone (joystick de droite) serait contrôlée par les données du gyromètre du gyroscope, tandis que la hauteur du drone (joystick de gauche) serait contrôlée par l'accéléromètre du gyroscope.

Version 2.0

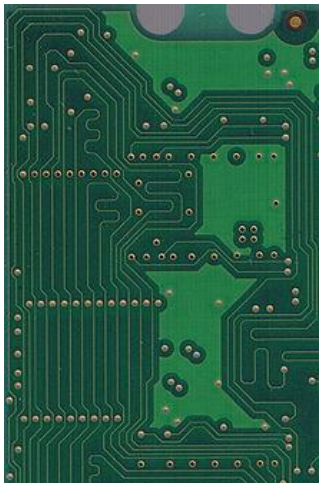
Puisque le drone de la version 1.0 est trop instable et que la construction d'un drone semble un objectif irréalisable pour le temps que nous avons, nous optons pour le contrôle par Wifi d'un plus gros drone. Le Holy Stone HS200 comprend un protocole d'envoi avec une application mobile et quelqu'un a réussi à le comprendre et à contrôler le drone avec son ordinateur. Nous prendrons son projet sur GitHub (détails plus loin) et l'adapterons de façon à pouvoir communiquer les données recueillies par le MPU 6050 au drone pour pouvoir le diriger. Nous allons également afficher les données utiles au contrôle du drone à l'écran. Nous accomplirons cela dans un sketch Processing, qui agira également comme relai entre Arduino et le drone. Éventuellement, il serait possible de délaisser l'ordinateur et de communiquer directement avec le drone en utilisant un Arduino Wifi (ceci consisterait en la version 3.0).

Glossaire

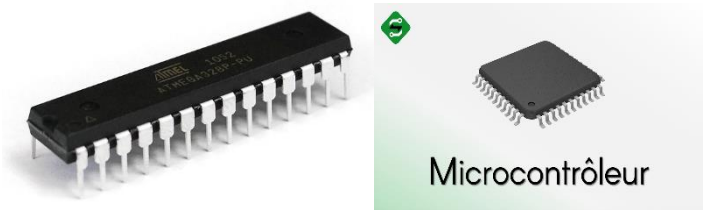
Arduino : Un Arduino comprend un circuit imprimé sur lequel se trouve un microcontrôleur. Il peut être programmé à l'aide du programme Arduino, installé préalablement sur un ordinateur, qui est gratuit et qui utilise un langage Java. Il suffit de connecter l'Arduino avec un fil USB avec l'ordinateur pour établir un lien.



Circuit imprimé (carte électronique) : Un circuit imprimé est une planche constituée de plusieurs fils de cuivre. Il permet de connecter des composants électroniques ensemble en créant un circuit électrique complexe.



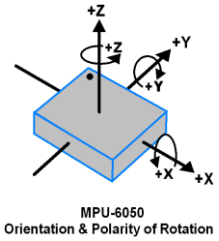
Microcontrôleur : Un microcontrôleur est un circuit intégré (puce électronique) qui agit comme un ordinateur miniature et qui comprend les composantes de bases de celui-ci.



Processing : Processing est un logiciel qui est principalement utilisé pour créer des environnements graphiques interactifs 2D ou 3D sur l'ordinateur. Ce logiciel est entièrement gratuit et utilise Java.

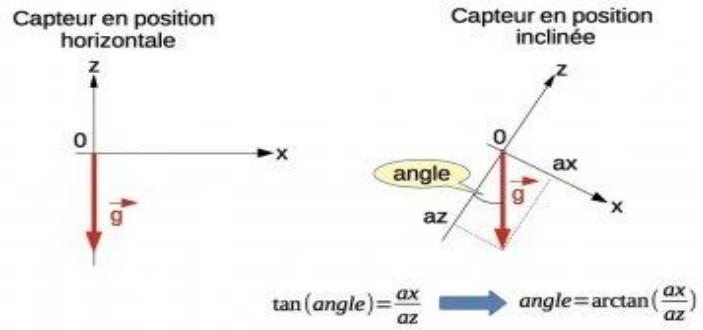
MPU 6050 : Le MPU 6050 est un dispositif contenant un gyromètre, un accéléromètre, un DMP et un FIFO. Les données de celui-ci seront recueillies à l'aide d'un Arduino et de la librairie I2Cdev.

Gyromètre : Quand les gyroscopes tournent autour de l'un des axes de détection, une vibration est provoquée et détectée par un microsystème électromagnétique à l'intérieur du MPU-6050. Le signal résultant est filtré pour produire une tension (V) proportionnelle à la vitesse angulaire qui est digitalisée par la suite. Cette vitesse angulaire peut être utilisée pour trouver l'angle d'inclinaison du MPU-6050.



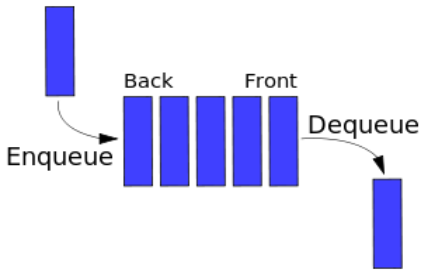
MPU-6050
Orientation & Polarity of Rotation

Accéléromètre : Le MPU-6050 utilise une masse de référence (*proof mass*) pour chacun des 3 axes. Une accélération au long d'un des axes résulte en un déplacement de la masse de cet axe qui est détecté par les capteurs de façon différentielle. Il est possible de trouver l'angle d'inclinaison par rapport au vecteur de poids.



DMP (Digital Motion Processor) : Le DMP se trouve dans le MPU 6050. Il combine les données de l'accéléromètre et du gyromètre afin de minimiser l'erreur et de trouver une donnée plus réaliste concernant la vitesse, l'accélération et les angles du MPU tel un filtre complémentaire. Donc, il traite les données et les envoie ensuite au stockage FIFO.

FIFO buffer (tampon en file d'attente) : File d'attente / queue / First in, first out. Le FIFO buffer du MPU 6050 stocke les données reçues et traitées par le DMP dans un buffer (tampon) pour que celles-ci puissent ressortir en ordre lors de l'extraction des données par l'Arduino.

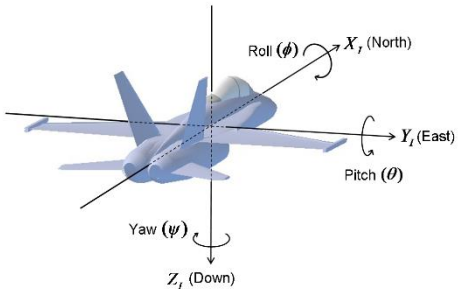


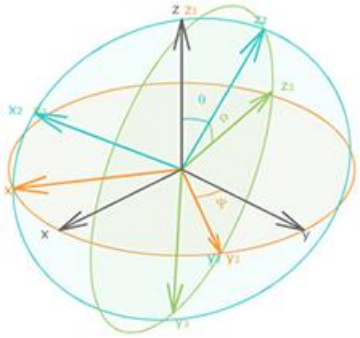
HC-SR04 : Le HC-SR04 est un capteur et émetteur d'ultrason conçu pour envoyer des ultrasons et le recevoir par la suite. Il calcule le temps qu'a pris l'ultrason pour revenir, s'il revient. Ceci est utile pour savoir la distance qui sépare le senseur et un objet plat comme un mur ou le sol. L'Arduino peut être programmé pour calculer la distance que l'ultrason a traversé, puisqu'on sait qu'il a une vitesse de 330m/s et qu'on a le délai.

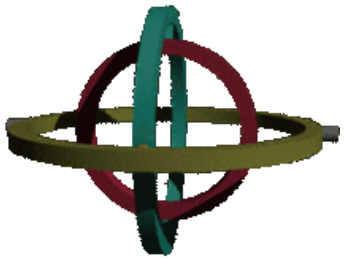
Ultrason : Un ultrason est un son qui a une fréquence trop élevée pour être pour être entendue par l'oreille humaine. En effet, les ultrasons se situent généralement entre 16'000 et 10'000'000 de Hertz tandis que les fréquences audibles par l'homme se situent dans les 20 à 20'000 Hertz.

Port série : Maintenant majoritairement remplacé par l'USB, le port série, aussi appelé RS-232, est un système utilisé pour le transfert de données entre certains appareils électriques. Ces ports séries sont représenté sur Windows par les noms "COM" suivis par un chiffre. Cette forme de liaison sera utilisée pour communiquer de l'Arduino vers l'ordinateur pour le programme de représentation graphique du MPU sur Processing. Bien que cette liaison puisse se faire via des connecteurs DE-9, DB-25 ou RJ-25, elle se fait également avec des connecteurs plus communs de nos jours, tels que par USB et par Wifi.

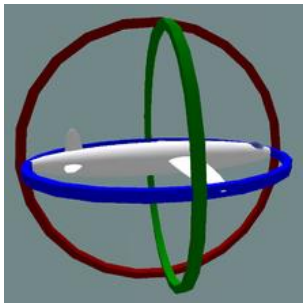
Angles d'Euler : Les angles d'Euler sont utilisés comme cadrans de positions. Ils nous indiquent la profondeur (pitch), le roulement (roll) et la rotation (yaw) du MPU.
https://www.youtube.com/watch?v=zc8b2Jo7mno#action=share



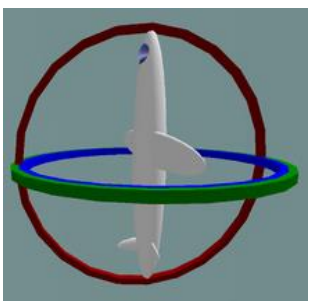




Blocage de cadran (gimbal lock) : Problème lorsque deux des trois cadrans de positions (x,y,z) se retrouvent à la même position (parallèles). Ceci résulte en un bug qui fait qu'un des cadrans ne réagira plus selon son axe de départ (x, y ou z) et réagira comme l'autre cadran, perdant ainsi un cadran de position.



→



Quaternion : Les Quaternions sont une façon de contrer l'effet de blocage de cadran et résultent de mathématiques avancées. Ils représentent une rotation en trois dimensions avec quatre valeurs ; 3 valeurs représentent un axe et la dernière est la rotation alentour de cet axe.

LIBRAIRIES :

I2Cdev : Le senseur MPU-6050 exploite le bus I2C (*Inter-Integrated Circuit*). Ce bus permet la liaison entre un microprocesseur (le Arduino dans ce cas-ci) et différents circuits (le MPU-6050 dans le cas présent). Ainsi, cette librairie permet le transfert de données à partir du senseur vers l'Arduino.

Wire : Pour utiliser le bus I2C mentionné ci-haut, il faut d'abord inclure la librairie Wire.h de Arduino. La fonction Wire.begin(), utilisée en début de programme, initialise la librairie et joint le bus I2C comme "maître" ou "esclave". Dans le cas présent, on désigne le Arduino comme maître.

MPU6050 : Pour utiliser le senseur MPU-6050, il faut définir une multitude d'adresses, dont les adresses utilisées par le bus I2C pour la transmission des données brutes. En ce sens, la librairie MPU6050.h nous évite cette tâche. La librairie permet aussi de définir notre senseur comme un objet, qui peut être initialisé avec la fonction initialize(). Les fonctions provenant de cette librairie qui ont également été utilisées sont getAcceleration() et getRotation(), qui prennent en paramètre l'adresse de 3 entiers à 16 bits pour leur assigner les valeurs respectivement des accélérations en x, y et z et des vitesses angulaires en x, y et z.

MPU6050_6Axis_MotionApps20 : Librairie incluant la librairie MPU6050.h, mais en y ajoutant des fonctions intéressantes comme la prise en charge des Quaternions, qui nous sera très utile.

NewPing : Pour utiliser le senseur à ultrasons HC-SR04, il était nécessaire d'inclure cette librairie. Nous avons essayé d'implémenter l'envoi et le retour de l'onde pour en calculer la durée et ainsi trouver la distance entre le gant et le sol, mais cela ralentissait l'exécution du code à un tel point que le sketch Processing ne fonctionnait plus. Précisément, ce que nous utilisons de cette bibliothèque est la méthode ping_cm() de notre objet NewPing. Cette méthode demande au senseur d'envoyer une onde, puis de capter l'écho pour mesurer la durée qu'a pris l'onde pour faire l'aller-retour. Avec la durée, la distance d'un aller est calculée puis assignée à la variable désirée.

Toxiclibs : Toxiclibs est une librairie faite en Java pour des tâches de conception de calculs dans Processing.

toxi.geom : C'est une sous-catégorie de la librairie Toxiclibs. Nous allons utiliser cette librairie principalement pour les calculs de quaternions dans Processing.

oscp5 : Cette librairie est une implémentation OSC, soit d'Open Sound Control (protocole de communication entre systèmes informatiques pour la technologie de réseautique moderne).

netP5 : C'est une sous-catégorie d'oscp5. Nous allons l'utiliser pour l'ajout de client UDP.

Prise de données

Prise de données MPU-6050

Les accélérations de rotation et les vitesses de rotation sont facilement accessibles avec les méthodes de la librairie MPU6050.h contenue dans MPU6050_6Axis_MotionApps20.h . En effet, il suffisait de déclarer un objet MPU6050, puis de l'initialiser dans le setup():

```
while (!Serial) {}  //Si le port série n'est pas connecté/initialisé.
senseurMPU.initialize();  // initialisation du MPU-6050.
```

Ensuite, dans la boucle principale, il suffisait d'appeler les fonctions suivantes:

```
senseurMPU.getAcceleration(&axT, &ayT, &azT);  //Prend les accélérations de rotation du MPU6050
senseurMPU.getRotation(&vxT, &vyT, &vzT);  //Prend les vitesses de rotation du MPU6050
```

Les variables passées en paramètres se voient ainsi assigner les multiples mesures par lecture (plus tard, il faudra appliquer une division pour obtenir la moyenne des mesures et avoir la mesure unique et valide qu'on pourra afficher):

```
const float sensibiliteVitesse = 131.0; //Selon la documentation du senseur, pour une valeur maximum de 250 degrés/seconde
const float erreurAccelXY = 0.1f, erreurAccelXYmax = 0.1f;  //Erreur d'arrondissement pour les accélérations en x et y
```

Pour obtenir des valeurs stables plus tard lors du transfert vers le sketch Processing, nous avons écrit une fonction qui instaure une marge d'erreur pour toutes les données lues à chaque lecture.
Si la valeur temporaire est plus grande que l'erreur, elle est assignée à la "vraie" variable et le compteur approprié est remis à zéro:

```
if(abs(mesureT/sensibilite-mesure) > erreur)
{
  mesure = mesureT/sensibilite;
  tabCompteurs[indexTab] = 0;
}
```

Le compteur de cette mesure augmente si la valeur est plus petite que l'erreur:

```
else if(tabCompteurs[indexTab] < nbLecturesCompteurs)
    tabCompteurs[indexTab]++;
}
```

Si la valeur est suffisamment petite pendant un certain nombre de lectures, on lui accorde la valeur de référence, soit la valeur lorsque le MPU ne bouge pas et le compteur approprié est remis à zéro:

```
else if(abs(valReference-mesureT/sensibilite) < erreur)
{
  mesure = valReference;
  tabCompteurs[indexTab] = 0;
}
```

Prise de données HC-SR04

Premièrement, dans le setup(), il a fallu ajouter les informations sur les pins du Arduino que l'on utilise avec le HC-SR04. Il a fallu dire que la pin 8 émet l'information et que la pin 7 prend l'information. De plus, il faut dire que la pin d'envoi émette, au repos, toujours un voltage bas pour lui dire de ne pas prendre de données au repos, car, quand le voltage est haut, c'est pour indiquer d'envoyer un ultrason afin de prendre une donnée et de l'envoyer.

```
//Variables pour la lecture du senseur ultrasonique
#define ENVOI_PIN 8 //Pin utilisée pour l'envoi
#define RETOUR_PIN 7
#define MAX_DISTANCE 200

//Initialisation des pins du senseur ultrasonique
pinMode(ENVOI_PIN, OUTPUT);
digitalWrite(RETOUR_PIN, LOW); // La pin ENVOI doit être à LOW au repos
pinMode(RETOUR_PIN, INPUT);
```

C'est la fonction priseDonneeHauteur(), que l'on appelle dans la loop(), qui s'occupe de la prise de données du HC-SR04 et de la position par rapport au sol. Elle est aidée par la librairie "NewPing" qui s'occupe de la prise d'une donnée du HC-SR04 et qui la transforme en distance en centimètre à l'aide de la fonction ping_cm().

```
float distanceSol = 0.0f; //Distance du gant par rapport au sol
```

Au départ, nous ne faisions qu'envoyer un voltage haut en un très court laps de temps pour prendre la donnée qu'elle émettait et la traduire en longueur. Cependant, nous avons remarqué que cette façon n'était pas bien optimisé et qu'il était plus facile de le faire de la façon expliquée ci-haut.
Aussi, lors des tests avec Processing, nous avons remarqué que le sonar n'était pas aussi rapide que le MPU pour la prise de données et l'envoi de celles-ci. Alors, c'est pourquoi nous avons fait une fonction pour la prise de données de l'hauteur au lieu de simplement appeler faire la ligne ci-haut dans la loop().
priseDonneeHauteur() regarde si l'intervalle de la dernière fois que le sonar a envoyé des données est plus grande ou égale à 50 millisecondes. On initialise l'intervalle à -50 pour que le programme puisse entrer prendre une donnée au départ. Comme cela, il ne prendra l'information que vingt fois par secondes, ce qui est vraiment moindre que ce qu'il faisait précédemment. Cela ralentit la prise de donnée de la distance quelque peu, mais elle reste assez rapide pour prendre de bonnes données en temps réel. En plus, le programme ne ralentit pas en faisant cela.

```
float fIntervalleSonar = -50; //Intervalle de temps depuis la dernière prise de donnée du sonar

//Prise de donnée du senseur ultrasonique: assignation de la hauteur du gant relative au sol
void priseDonneeHauteur()
{
  if(millis() - fIntervalleSonar >= 50)
  {
```

Ensuite, on prend la donnée de distance du sol dans une variable test. Pourquoi? Parce que la valeur retournée n'est pas nécessairement bonne. En effet, si le sonar ne reçoit pas l'ultrason qui est supposé revenir pour une raison quelconque ou que celle-ci prend trop de temps à

revenir à cause que la distance est trop loin, la donnée que l'on reçoit devient 0. Ceci peut causer un problème, surtout si l'on prend cette donnée pour monter ou descendre le drone : celui-ci pourrait descendre même si l'utilisateur ne veut pas. C'est pourquoi on prend la valeur dans une variable test : Si elle égale 0, on ne la considère pas et on garde l'ancienne valeur.

```
float testDistanceSol = sonar.ping_cm();  //On prend la mesure de la distance
if(testDistanceSol != 0)  //Si elle n'égale pas 0, on la prend en considération
{ //On la stocke si la valeur est bonne (si c'est 0, c'est, la plupart du temps,
  // à cause que l'onde n'est pas revenu
  distanceSolT = testDistanceSol;
  erreurMesure(distanceSolT, distanceSol, erreurHauteur, 1.0, 6, 0.0);
}
```

Pour finir la fonction, on remet l'intervalle au temps actuel.

```
fIntervalleSonar = millis();
```

Interaction Arduino-Processing

https://github.com/jrowberg/i2cdevlib/issues/252

Après avoir complété la prise de données du MPU-6050 avec un Arduino. Nous voulions voir les mouvements que celui-ci effectuait à l'aide d'un environnement graphique comme Processing.

En premier lieu, il fallait trouver comment transférer nos données du Arduino vers Processing. Le MPU Teapot, un exemple d'interaction entre Arduino et Processing pour un espace graphique représentant les mouvements angulaires du MPU, était exactement l'exemple parfait pour comprendre cette pratique. En effet, le programme aurait pu pratiquement être copié, mais il est était beaucoup mieux de refaire le programme comme on le voulait pour pouvoir mieux avancer ensuite en ajoutant plus facilement des fonctions dont on aurait besoin.

Pour le code Arduino, on a commencé par les **librairies**. Il fallait télécharger des librairies additionnelles. Les librairies "Wire" et "I2Cdev" étaient déjà installées dû à l'étape de récupération de données (voir documentation: Librairies MPU6050). Cependant, la librairie "MPU6050" était utilisé, mais celle-ci n'incluait pas l'utilisation des quaternions, alors il a fallu télécharger la librairie "MPU6050_6Axis_MotionApps20", qui était plus complète et comprenait la librairie "MPU6050".

```
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h" //Librairie plus complète que MPU6050.h pour quaternion
```

Ensuite, on crée une variable quaternion pour plus tard, ainsi qu'une pour le senseur MPU que nous allons appeler quand nous voudrons prendre des données de celui-ci.

```
Quaternion quat1; // [w, x, y, z]         conteneur de quaternion
MPU6050 senseurMPU; //0x68 : high / Fait appel au MPU6050, ses fonctions nous donnent des données
```

On commence ensuite le **setup**(). Il faut commencer en initialisant la librairie Wire pour pouvoir utiliser les librairies Wire et I2C. Il faut aussi commencer une liaison avec le port série. On met 115'200 baud et il faudra que le lecteur du port série lise en 115'200 baud aussi. Cette étape était déjà faite dû à la prise de données qui en dépendait.

```
//Initialisation du MPU, des librairies et des offsets sur le MPU.
void setup()
{
  Wire.begin(); //Initialise la librairie Wire et joint I2C avec elle.
  Serial.begin(115200); // liaison série avec une fréquence de 155'200 Baud.
```

Ensuite, on met une ligne d'attente que le port série soit prêt, au cas où il prendrait un certain temps à établir le lien. On peut ensuite initialiser le senseur MPU qui est connecté au Arduino. Après, on regarde si le DMP a bien été initialisé et on stocke son statut dans une variable qui sera utilisée dans la fonction qu'on appelle ensuite, soit statutDMP().

```
uint8_t iStatutDMP;       // Contient le statut du DMP après chaque opératon (0 = success, !0 = error)
```

```
while (!Serial) {}  //Si le port série n'est pas connecté/initialisé.
senseurMPU.initialize();  // initialisation du MPU-6050.
iStatutDMP = senseurMPU.dmpInitialize(); // Va chercher le statut du DMP (0 = success, !0 = error)
statutDMP();
```

On met ensuite les offsets, qui ont été calculés dans un autre sketch (https://codebender.cc/sketch:97892#MPU6050%20calibration.ino), pour l'accélération et la vitesse après ces étapes :

```
// Calibration du senseur MPU en éliminant les décalages/offsets
senseurMPU.setXAccelOffset(309);
senseurMPU.setYAccelOffset(2773);
senseurMPU.setZAccelOffset(1421);
senseurMPU.setXGyroOffset(-218);
senseurMPU.setYGyroOffset(-32);
senseurMPU.setZGyroOffset(15);
```

Pour la fonction **statutDMP**(), on regarde quel est le statut du DMP que l'on a pris précédemment : on regarde s'il a été initialisé ou s'il y a eu une erreur.

```
void statutDMP()
{  //Met le DMP en marche et regarde s'il est prêt à prendre et convertir les données
```

S'il n'y a pas eu d'erreur, le statut sera à 0. Dans ce cas, On met le DMP en marche.

```
if (iStatutDMP == 0) // Si le statut du DMP n'est pas une erreur.
{
     // Met le DMP en marche
     Serial.println(F("Mise en marche du DMP..."));
     senseurMPU.setDMPEnabled(true);
```

On met en marche la détection d'interruption d'Arduino pour qu'Arduino sache quand le MPU s'est interrompu et n'envoie plus de données. dmpDataReady est une fonction qui est expliqué plus bas. Cette fonction laisse Arduino interagir n'importe quand, au moment où il s'en aperçoit.

```
//Mise en marche la détection d'interruption d'Arduino
attachInterrupt(0, dmpDataReady, RISING);
```

On crée une variable nous disant si le DMP est prêt à l'utilisation (hors de la fonction, en haut avec les autres variables) et on lui accorde vrai dans notre fonction.

```
bool dmpReady = false;  // True si l'initialisation du DMP se fait correctement, pour ne pas avoir de bugs
```

```
// Le DMP est maintenant prêt
Serial.println(F("DMP prêt!"));
dmpReady = true;
```

On peut créer une autre variable à mettre avec les autres qui nous indiquera le statut du MPU. On lui donne son statut en int dans la fonction.

```
uint8_t iStatutMPU;    // Contient le statut d'interruption du MPU en Byte
```

```
iStatutMPU = senseurMPU.getIntStatus();
```

Il nous reste seulement à regarder la taille du FIFO Packet. Nous allons faire une comparaison plus loin et on prend cette valeur ici puisque c'est une fonction qui est appelé seulement dans le

---

La fonction **dmpDataReady**() ne fait qu'allouer la valeur *true* à la variable boolean de l'interruption du MPU lorsqu'elle est appelée. Elle est appelée selon la détection d'interruption d'Arduino.

```
volatile bool mpuInterrupt = false;  // Fonction : Si le DMP est prêt, retourne vrai. Sinon, faux par défaut
```

```
void dmpDataReady()
{
    mpuInterrupt = true;
}
```

On peut maintenant passer à la **loop**(). On regarde, en premier lieu, si le DMP est prêt. On a qu'à regarder dans la variable où on a stocké cette information plus tôt dans le setup(). S'il n'est pas prêt, rien ne se fait dans la boucle.

```
//Appelle les fonctions principales à chaque boucles
void loop()
{
  if (!dmpReady) return;  // Si la configuration n'a pas fonctionné, la boucle ne sera pas utilisé
  {
```

On a déjà vérifié le statut du DMP dans le setup(), mais il faut regarder le statut du MPU à chaque boucle pour savoir s'il n'y a pas trop de données dans le buffer par exemple (cette fonction est expliquée plus bas). On appelle cette fonction au début de la boucle, juste après la condition.

```
statutMPU();
```

Le reste de la fonction demeure comme elle était auparavant, excepté pour les angles. En effet, à cause de l'effet du blocage de cadran et du fait que la rotation dans le programme Processing ne peut se faire bien avec des angles de bases x,y et z, on a eu recours aux quaternions pour ensuite avoir des angles qui sont beaucoup plus adaptés à notre programme : soit les angles d'Euler. En effet, ces angles représentent les effets de "pitch", "yaw" et "roll" en radians. Pour avoir ces angles, on doit d'abord prendre un quaternion du MPU et le stocker dans notre variable. On peut ensuite prendre les angles Euler du MPU à partir du quaternion.

```
senseurMPU.dmpGetQuaternion(&quat1, fifoBuffer); //Crée un quaternion des données du fifoBuffer
senseurMPU.dmpGetEuler(euler, &quat1); //Crée les angles Euler à partir du quaternion

senseurMPU.getAcceleration(&axT, &ayT, &azT);  //Prend l'accélération du MPU6050
senseurMPU.getRotation(&vxT, &vyT, &vzT);  //Prend la rotation du MPU6050

restrictionMargesErreur();
ecritureDonnee();
```

Premièrement, la fonction **statutMPU**() regarde si le MPU a été interrompu avec la variable volatile qui peut être changée à tout moment. Elle regarde aussi si le nombre de byte FIFO utilisé est plus grand que la taille du packet FIFO. Si le MPU n'est pas interrompu et qu'il manque de place dans le packet FIFO, on rentre dans la boucle qui ne fait qu'attendre que de la place se libère ou que le MPU s'interrompt. On réinitialise la variable à faux ensuite, puisque le MPU n'est plus ou pas interrompu.

```
void statutMPU()
{ //Prends des info du MPU pour savoir s'il est prêt à donner des données
  //Si le MPU n'est pas interrompu et qu'il manque de place
  //dans le packet pour l'information lue
  while (!mpuInterrupt && nbByteFIFO < taillePacket)
    {
      //  Attendre que le MPU ne soit plus interrompu ou qu'il y ait de la place disponible dans le packet
    }

    mpuInterrupt = false;   // Réinitialise le drapeau d'interruption
```

Après on va chercher le statut en Int (en byte) du MPU et on va reprendre le nombre de byte dans le FIFO.

```
iStatutMPU = senseurMPU.getIntStatus();  // Va chercher le statut du MPU en Byte
nbByteFIFO = senseurMPU.getFIFOCount(); // Va chercher le compte FIFO
```

On va vérifier aussi si le statut de MPU réfère à une trop grande quantité de donnée et que le nombre de byte indique le maximum. Si c'est le cas, on indique dans le port série qu'il y a eu un débordement de données (overflow). On réinitialise aussi les données du FIFO en les supprimant pour pouvoir repartir à zéro.

```
if ((iStatutMPU & 0x10) || nbByteFIFO == 1024)
{ // Vérifie s'il y a trop de données en liste (ne devrait pas arriver)
    Serial.println(F("Débordement du FIFO (Overflow)!"));
    senseurMPU.resetFIFO();  // Réinitialise les données pour recommencer
}
```

Pour la fonction **ecritureDonnee**(), il faut écrire les données dans le port série d'une façon méthodique et constante pour que le code processing (le récepteur) puisse interpréter l'information de la bonne manière. Au début, on écrivait l'indice de la donnée au complet suivi de deux points et de la donnée, ce qui ressemblait à ceci : "Accélération angulaire en x : 1.00" et on changeait de ligne ensuite. À lire l'information caractère par caractère comme on le fait, ça ne donne pas une rapidité de lecture optimale. Alors on a changé pour une écriture beaucoup plus simple, soit : "ax1.00". En lisant ceci, on sait que l'on considère l'accélération en x à une valeur de 1.00 avec beaucoup moins de caractères. Aussi on pensait devoir faire l'écriture en byte avec Serial.write(), ce qui aurait compliqué la tâche drastiquement. Cependant, avec Serial.print(), l'envoi est aussi rapide et peut se faire par l'envoi de String, ce qui ne nous oblige pas à transférer en bytes explicitement.

```
void ecritureDonnee() //Écrit dans le port série les informations recueillis
{
    //Accélérations Angulaires
    Serial.print("ax");
    Serial.print(ax);
    Serial.println();

    Serial.print("ay");
    Serial.print(ay);
    Serial.println();

    Serial.print("az");
    Serial.print(az);
    Serial.println();


    //Vitesses angulaires
    Serial.print("vx");
    Serial.print(vx);
    Serial.println();

    Serial.print("vy");
    Serial.print(vy);
    Serial.println();

    Serial.print("vz");
    Serial.print(vz);
    Serial.println();
```

```
    //Angles
    Serial.print("tx");
    Serial.print(euler[1]);
    Serial.println();

    Serial.print("ty");
    Serial.print(euler[0]);
    Serial.println();

    Serial.print("tz");
    Serial.print(euler[2]);
    Serial.println();

    //Hauteur par rapport au sol
    Serial.print("dz");
    Serial.print(distanceSol);
    Serial.println();
}
```

Pour le code Processing, la **librairie** “processing.serial” est très importante pour établir un lien avec le port série, il faut donc l’avoir absolument. De plus, pour les quaternions, il a fallu télécharger la librairie ToxicLibs et importer dans le code “toxi.geom” pour l’aspect géométrie. La librairie “toxi.processing” est nécessaire pour dessiner, mais “toxi.opengl” n’est pas nécessaire pour l’aspect graphique si on le remplace par P3D dans la fonction size() dans le setup().

```
import processing.serial.*;
//import processing.opengl.*;
import toxi.geom.*;
import toxi.processing.*;


void setup() {
    // 300px square viewport using OpenGL rendering
    size(300, 300, OPENGL);
```

->

```
//size(800, 600, OPENGL);
  //size(Largeur de l'écran, Hauteur de l'écran, rendu de l'interface (3D dans notre cas))
  size(800, 600, P3D);
```

Il faut aussi nommer notre port série et appeler notre support de Toxiclibs pour les dessins. De plus, la librairie de “toxi.geom” nous donne accès à l'appellation d’un quaternion qu’on utilisera plus tard sûrement pour enregistrer nos données.

```
ToxiclibsSupport gfx;  //Pour dessiner des objets

Serial portSerie;  // Crée un objet de la classe Serial
Quaternion quat1 = new Quaternion(1, 0, 0, 0);
```

Ensuite, on peut continuer vers le **setup**().

Nous avons déjà fait le size(), qui est pour définir l'écran. Il faut maintenant initialiser le support Toxiclibs.

```
gfx = new ToxiclibsSupport(this);
```

Il faut aussi initialiser le port série pour pouvoir savoir le port qu'il faut prendre et mettre le bon nombre de baud pour comprendre ce qui est écrit. Sinon, le moniteur série ne comprend pas bien ce qui est lue. Pour notre cas, ce sera toujours le premier port série disponible que l'on prendra et on a ajuster le code en conséquence. En effet, Serial.list() donne la liste des ports disponibles et on prend le premier de cette liste.

```
portSerie = new Serial(this, Serial.list()[0], 115200); // mettre le bon nombre de baud comme 3e donnée
```

Pour la fonction **Draw**(), on mettra tout l'aspect graphique.

```
void draw()
{
```

Pour l'instant, on va copier un peu plus le code pour pouvoir voir si ça marche et changer le dessin pour quelque chose de mieux ensuite.

Donc, pour commencer, on met un fond d'écran. 0 est noir et ça varie jusqu'au blanc à 255.

```
 //Couleur du fond d'écran
 background(0);  //Noir dans notre cas
```

Il faut ensuite utiliser pushMatrix(), qui va nous aider à dessiner des objets plus facilement. On écrira PopMatrix(); après le dessin terminé.

```
//PushMatrix() doit être utilisé avec PopMatrix()
//Écrire un code entre ces deux fonctions bouge les images sans bouger le reste
pushMatrix();  //Enregistre le point d'origine du plan
```

Ensuite, il a fallu décider de l'endroit où le dessin sera placé à l'écran. Il sera au milieu pour l'instant.

```
translate(width/2, height/2 + 50, 0);  //Mettre le dessin à l'endroit souhaité au départ
```

L'espace graphique est initialisé pour pouvoir y placer le dessin. Il faut maintenant savoir l'orientation du dessin avant de le dessiner. Pour ce faire, il nous faut les angles d'Euler, que l'on prendra du port Série dans une fonction serialEvent plus tard. Une fois les angles stockés, il faut pouvoir le faire tourner et c'est ce qu'on va faire. Il faut commencer en initialisant le quaternion avec les angles d'Euler (première ligne ci-bas). Ensuite, il faut prendre les angles d'axe qui sont calculé grâce au quaternion dans un tableau de float nommé axis. On fait alors la rotation du plan complet selon la rotation des angles d'axe que l'on a avec la fonction rotate(). La raison de ne pas utilisé rotateX(), rotateY(), et rotateZ() avec les angles d'Euler à la place, sans le passer dans un quaternion, est que le blocage de cadran peut se produire avec cette façon, ce qui n'est pas le cas en le passant par un quaternion pour avoir les angles d'axe.

```
quat1 = Quaternion.createFromEuler(-roulementXT, -rotationYT, profondeurZT);

float[] axis = quat1.toAxisAngle();  //Stocke les angles d'axe
rotate(axis[0], axis[1], axis[3], axis[2]);  //Rotation du cube avec les angles d'axe
```

Ensuite, pour le moment, nous allons seulement copié le code du MPU Teapot pour le dessin d'une sorte d'avion coloré.

```
// draw main body in red
fill(255, 0, 0, 200);
box(10, 10, 200);

// draw front-facing tip in blue
fill(0, 0, 255, 200);
pushMatrix();
translate(0, 0, -120);
rotateX(PI/2);
drawCylinder(0, 20, 20, 8);
popMatrix();

// draw wings and tail fin in green
fill(0, 255, 0, 200);
beginShape(TRIANGLES);
vertex(-100,  2, 30); vertex(0,  2, -80); vertex(100,  2, 30);  // wing top layer
vertex(-100, -2, 30); vertex(0, -2, -80); vertex(100, -2, 30);  // wing bottom layer
vertex(-2, 0, 98); vertex(-2, -30, 98); vertex(-2, 0, 70);  // tail left layer
vertex( 2, 0, 98); vertex( 2, -30, 98); vertex( 2, 0, 70);  // tail right layer
endShape();
beginShape(QUADS);
vertex(-100, 2, 30); vertex(-100, -2, 30); vertex(  0, -2, -80); vertex(  0, 2, -80);
vertex( 100, 2, 30); vertex( 100, -2, 30); vertex(  0, -2, -80); vertex(  0, 2, -80);
vertex(-100, 2, 30); vertex(-100, -2, 30); vertex(100, -2,  30); vertex(100, 2,  30);
vertex(-2,   0, 98); vertex(2,   0, 98); vertex(2, -30, 98); vertex(-2, -30, 98);
vertex(-2,   0, 98); vertex(2,   0, 98); vertex(2,   0, 70); vertex(-2,   0, 70);
vertex(-2, -30, 98); vertex(2, -30, 98); vertex(2,   0, 70); vertex(-2,   0, 70);
endShape();
```

De même que la fonction **drawCylinder**() :

```
void drawCylinder(float topRadius, float bottomRadius, float tall, int sides) {
    float angle = 0;
    float angleIncrement = TWO_PI / sides;
    beginShape(QUAD_STRIP);
    for (int i = 0; i < sides + 1; ++i) {
        vertex(topRadius*cos(angle), 0, topRadius*sin(angle));
        vertex(bottomRadius*cos(angle), tall, bottomRadius*sin(angle));
        angle += angleIncrement;
    }
    endShape();

    // If it is not a cone, draw the circular top cap
    if (topRadius != 0) {
        angle = 0;
        beginShape(TRIANGLE_FAN);

        // Center point
        vertex(0, 0, 0);
        for (int i = 0; i < sides + 1; i++) {
            vertex(topRadius * cos(angle), 0, topRadius * sin(angle));
            angle += angleIncrement;
        }
        endShape();
    }
```

```
    // If it is not a cone, draw the circular bottom cap
    if (bottomRadius != 0) {
        angle = 0;
        beginShape(TRIANGLE_FAN);

        // Center point
        vertex(0, tall, 0);
        for (int i = 0; i < sides + 1; i++) {
            vertex(bottomRadius * cos(angle), tall, bottomRadius * sin(angle));
            angle += angleIncrement;
        }
        endShape();
    }
}
```

On arrive ensuite à **serialEvent**() : la fonction pour lire le contenu du port série. On va lire une valeur (char) à la fois et la stocker dans la bonne variable. Cette partie a été beaucoup plus complexe à cause de la transmission de donnée qui est vraiment différente avec celui du MPU Teapot. Il a fallu créer des fonctions pour une lecture de ce que l'on envoyait. Il faut toujours envoyer la même base pour que le programme comprenne, comme s'il déchiffrait un code et qu'il avait besoin de la clé de décodage. Après quelques essais, il était clair que la lecture des données était plus facile à faire en lisant un caractère à la fois qu'une ligne à la fois. En effet, le fait que l'ordinateur soit plus rapide que l'Arduino fait en sorte que les lignes n'étaient pas toujours écrites au complet lors de la lecture. Ceci compliquait la tâche et rendait difficile la lecture lorsqu'on l'imprimait sur la console. On s'est donc concentré dans la lecture d'un char à la fois.

```
void serialEvent(Serial port)
{
```

Il a aussi fallu ajouter des variables pour les fonctions ci-dessous :

```
float fAx, fAy, fAz;   //Accélérations mesurées en x, y et z
float fVx, fVy, fVz;   //Vitesses angulaires mesurées en x, y et z
float roulementX = 0, rotationY = 0, profondeurZ = 0;   //Angles

String sIndiceValeur;  // Données reçues du port série
String sAx, sAy, sAz;   //Accélérations mesurées en x, y et z
String sVx, sVy, sVz;   //Vitesses angulaires mesurées en x, y et z
String sAngleX, sAngleY, sAngleZ;   //Angles
String sDistanceSol; //Distance entre le gant et le sol
```

Pour commencer, on crée une variable int qui stockera la valeur lue. Puisque c'est un int, le chiffre ou la lettre sera accessible sous la forme d'un chiffre dans la table ASCII. On initialise à 0 pour l'instant.

```
int iValeurLue = 0; //Valeur lue dans le port série
```

Il faudra ensuite regarder s'il y a une valeur de prête à être lue dans le port série avant de continuer. L'ordinateur est supposé être plus rapide que l'Arduino. Ça ne devrait donc pas déranger le système si c'est une boucle ou un if, car il devrait être souvent en attente de réception de données de toute façon.

```
//if (port.available() > 0)
while (port.available() > 0) // tant que les données sont disponibles dans le port
{
```

En entrant dans cette boucle, on peut maintenant lire une valeur du port série.

```
iValeurLue = port.read();  //Lire un int dans le port série
```

**Update : Avec la nouvelle donnée de distance par rapport au sol, on regarde aussi si c'est un d. Le texte, les fonctions et les images qui suivent ont été remplacées pour correspondre avec le programme.

On regarde ensuite si la valeur est "a", "v", "t" ou "d" pour voir si c'est la première lettre d'un indice de ce que va être la donnée qui suivra. Si c'est le cas, on appelle la fonction stockageIndiceAVTD().

```
if (estAVTD(iValeurLue)) //Si la valeur = a, v, t ou d
{
  stockageIndiceAVTD(iValeurLue);
}
```

La fonction **estAVTD**() remplace le fait d'écrire les nombre ASCII dans le if :

```
boolean estAVTD(int iValeur)
{ // 97 = a, 118 = v, 116 = t, 100 = d dans la table ASCII
  if (iValeur == 97 || iValeur == 118 || iValeur == 116 || iValeur == 100)
  {
    return true;
  } else
  {
    return false;
  }
}
```

La fonction **stockageIndiceAVTD**() (accélération, vitesse, angle (thêta) et distance) regarde, au début, si l'indice de la valeur précédente est null. si ce n'est pas le cas, ça veut dire que le stockage dans un String de la valeur précédente est terminée. Il faut changer l'ancienne valeur dans le stockage float pour la nouvelle que l'on a obtenu avec le stockage String. Ensuite, on supprime la valeur dans le stockage String pour y laisser place à l'écriture de la nouvelle. Cependant, avant de faire ces étapes, il faut regarder quel est l'indice précédent complet que l'on a noté pour pouvoir associer nos valeurs aux bonnes variables. Il faut aussi s'assurer, dans la même condition, que la valeur en String que l'on a n'est pas vide et qu'elle concorde avec l'indice. À la fin, il faut réinitialiser la valeur indice en String avec la valeur que l'on a présentement, car "a", "v", "t" ou "d" sont nécessairement les premières lettres d'une nouvelle donnée. Avec ces précautions et le fait que le programme s'exécute plus rapidement que l'Arduino, il ne devrait pas y avoir de problèmes dans l'écriture des bonnes données en float.

```
void stockageIndiceAVTD(int iValeur)
{
  if (sIndiceValeur != null)
  {
    if (sIndiceValeur.equals("ax") && sAx != null)
    {
      fAx = float(sAx);
      sAx = null;
    } else if (sIndiceValeur.equals("ay") && sAy != null)
    {
      fAy = float(sAy);
      sAy = null;
    } else if (sIndiceValeur.equals("az") && sAz != null)
    {
      fAz = float(sAz);
      sAz = null;
    } else if (sIndiceValeur.equals("vx") && sVx != null)
    {
      fVx = float(sVx);
      sVx = null;
    } else if (sIndiceValeur.equals("vy") && sVy != null)
    {
      fVy = float(sVy);
      sVy = null;
```

```
    } else if (sIndiceValeur.equals("vz") && sVz != null)
    {
      fVz = float(sVz);
      sVz = null;
    } else if (sIndiceValeur.equals("tx") && sAngleX != null)
    {
      roulementXT = float(sAngleX);
      sAngleX = null;
    } else if (sIndiceValeur.equals("ty") && sAngleY != null)
    {
      rotationYT = float(sAngleY);
      sAngleY = null;
    } else if (sIndiceValeur.equals("tz") && sAngleZ != null)
    {
      profondeurZT = float(sAngleZ);
      sAngleZ = null;
    }
    else if (sIndiceValeur.equals("dz") && sDistanceSol != null)
    {
      fDistanceSol = float(sDistanceSol);
      sDistanceSol = null;
    }
  }
  sIndiceValeur = Character.toString(char(iValeur));
}
```

Pour continuer dans le serialEvent(), il faut maintenant vérifier si la valeur est “x”, “y”, ou “z”. Ces valeurs sont les deuxièmes qui devraient être obtenu de l’indice d’une donnée et la donnée devrait suivre ensuite. C’est pour ça qu’il faut regarder si l’indice de la valeur est null. Si oui, on ne sait pas si la donnée est une vitesse, une accélération, un angle ou une distance, ce qui fait de cette valeur une donnée invalide. Sinon, on entre dans la fonction stockageIndiceXYZ().

```
if (estXYZ(iValeurLue) && sIndiceValeur != null)  //Si la valeur = x, y ou z
{
  stockageIndiceXYZ(iValeurLue);
}
```

La fonction **estXYZ**() est une boolean et retourne vrai ou faux (dans la table ASCII, x = 120, y = 121, z = 122).

```
boolean estXYZ(int iValeur)
{
  if (iValeur >= 120 && iValeur <= 122) //Si la valeur dans la table ASCII correspond à x,y ou z
  {
    return true;
  } else
  {
    return false;
  }
}
```

Pour la fonction **stockageIndiceValeurXYZ**(), si l’indice de la valeur a déjà une valeur d’indice qui est soit “a”, “v”, “t” ou “d”, on stocke la valeur “x”, “y”, ou “z” après dans le String.

```
void stockageIndiceXYZ(int iValeur)
{
  if (sIndiceValeur.equals("a") || sIndiceValeur.equals("v") || sIndiceValeur.equals("t") || sIndiceValeur.equals("d"))
  {
    sIndiceValeur += Character.toString(char(iValeur));
  }
}
```

Finalement, on regarde avec la table ASCII, dans le SerialEvent(), si la valeur lue est un chiffre, un point, ou un signe négatif. Il faut aussi que l'indice de la valeur ne soit pas nulle, car on ne saurait pas où stocker la valeur. Si les conditions sont bonnes, on appelle la fonction stockageValeur(). La fonction se termine ensuite.

```
    if (estChiffreOuPoint(iValeurLue) && sIndiceValeur != null) //Si la valeur = un chiffre ou un point
    {
      stockageValeur(iValeurLue);
    }
  }
}
```

La fonction **estChiffreouPoint**(), qui est boolean,considère aussi le signe négatif et retourne un vrai ou un faux dépendamment de la véracité des conditions.

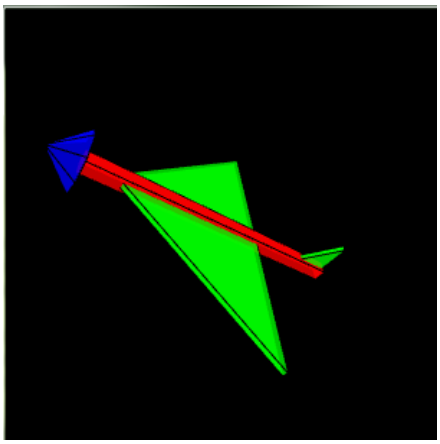
```
// La valeur intéressée est en int et on regarde dans la table ASCII pour voir si c'est les données qu'on recherche.
// L'indice de la valeur nous indique si cette valeur est pour une vitesse, accélération ou angle. Sinon, on ne la considère pas.
boolean estChiffreOuPoint(int iValeur)
{ //Si la valeur dans la table ASCII correspond à un nombre, un point ou un signe négatif
  if ((iValeur >= 48 && iValeur <= 57) || iValeur == 46 || iValeur == 45)
  {
    return true;
  } else
  {
    return false;
  }
}
```

La fonction **stockageValeur**() regarde, en premier lieu, quel est l'indice de la valeur. S'il n'est pas trouvé, on ne la considère pas. Une fois l'indice trouvée, il regarde si la valeur stockée de cet indice est nulle. Si c'est le cas, c'est que c'est la première valeur du chiffre complet, alors on égalise le String avec la valeur. Sinon, ça veut dire que ce n'est pas la première valeur lue du String, alors on l'ajoute à la fin de celui-ci.

```
void stockageValeur(int iValeur)
{
  if (sIndiceValeur.equals("ax"))
  {
    if (sAx == null)
    {
      sAx = Character.toString(char(iValeur));
    } else
    {
      sAx += Character.toString(char(iValeur));
    }
  } else if (sIndiceValeur.equals("ay"))
  {
    if (sAy == null)
    {
      sAy = Character.toString(char(iValeur));
    } else
    {
      sAy += Character.toString(char(iValeur));
    }


 } else if (sIndiceValeur.equals("az"))
 {
   if (sAz == null)
   {
     sAz = Character.toString(char(iValeur));
   } else
   {
     sAz += Character.toString(char(iValeur));
   }
 } else if (sIndiceValeur.equals("vx"))
 {
   if (sVx == null)
   {
     sVx = Character.toString(char(iValeur));
   } else
   {
     sVx += Character.toString(char(iValeur));
   }


} else if (sIndiceValeur.equals("vy"))
{
  if (sVy == null)
  {
    sVy = Character.toString(char(iValeur));
  } else
  {
    sVy += Character.toString(char(iValeur));
  }
} else if (sIndiceValeur.equals("vz"))
{
  if (sVz == null)
  {
    sVz = Character.toString(char(iValeur));
  } else
  {
    sVz += Character.toString(char(iValeur));
  }
```

```
} else if (sIndiceValeur.equals("tx"))
{
  if (sAngleX == null)
  {
    sAngleX = Character.toString(char(iValeur));
  } else
  {
    sAngleX += Character.toString(char(iValeur));
  }
} else if (sIndiceValeur.equals("ty"))
{
  if (sAngleY == null)
  {
    sAngleY = Character.toString(char(iValeur));
  } else
  {
    sAngleY += Character.toString(char(iValeur));
  }

} else if (sIndiceValeur.equals("tz"))
{
  if (sAngleZ == null)
  {
    sAngleZ = Character.toString(char(iValeur));
  } else
  {
    sAngleZ += Character.toString(char(iValeur));
  }
}
else if (sIndiceValeur.equals("dz"))
{
  if (sDistanceSol == null)
  {
    sDistanceSol = Character.toString(char(iValeur));
  } else
  {
    sDistanceSol += Character.toString(char(iValeur));
  }
}
```

Après ces étapes, notre programme fonctionnait bien et on pouvait obtenir les erreurs du MPU qui se manifestaient, mais l'aspect graphique était clairement à améliorer :



Représentation graphique du MPU 6050

Maintenant que le MPU Teapot est finalisé, on peut se concentrer sur l'aspect graphique de notre représentation des mouvements du MPU. En regardant des exemples sur internet, nous avons pu constater un programme qui agissait sur le même but et avec relativement les mêmes outils que celui du MPU Teapot. De plus, ce programme représentait un MPU-6050, le même modèle qui est utilisé pour notre projet. Bien sûr, mettre un gant ou une main comme objet à la place aurait pu être bien, mais ceci aurait été très long à faire, compte tenu de la basse information qu'il y a sur internet à ce sujet. Cela aurait prit du temps que nous n'avions pas et qui n'allait pas en parallèle avec le but principal du projet qui est de contrôler un drone avec un gant. Finalement,

le code que l'on s'est basée a été trouvé sur : https://www.youtube.com/watch?v=L_soFcZ5pWo et est beaucoup mieux que celui du MPU Teapot. Les images données avec le programme sont vraiment bonne et ne seront pas changées. On les a placé dans un dossier "data" dans le même dossier que le programme Processing.

Pour commencer, il faut des variables d'image pour venir stocker nos images dans notre dossier.

```
PImage imageDessusMPU, imageDessousMPU, imageDevantMPU, imageCoteMPU;  //Images du MPU
```

On va aussi, en même temps, ajouter les infos en texte à l'écran, il nous faut donc une police de caractère. Pour cela, il faut appuyer (en haut du programme Arduino) sur Outils->Générer la police… Ensuite on choisit la police et on appuie sur ok. Nous allons maintenant ajouter une police de caractère.

```
PFont police;  //Police de caractère du texte de l'interface (Pour les infos)
```

Dans le **setup**(), on doit rajouter une ligne indiquant le mode de la texture vers NORMAL pour les images png puissent fonctionner avec l'interface. On va aussi ajouter une ligne pour spécifier qu'on ne veut pas de contour sur nos images pour plus de réalisme. On va aussi mettre fill(255); pour mettre la couleur des caractères le plus blanc possible.

```
textureMode(NORMAL);  // Ne pas changer NORMAL pour IMAGE(défaut),
                      // les images png utilisées ne fonctionnent pas sinon.
fill(255);  //Clareté du blanc dans la police (0 à 255) (255 = max)
noStroke();  //Ne pas avoir de contour sur le MPU de l'interface
```

Aussi, il faut initialiser les lumières et l'effet d'anticrénelage afin d'avoir des polices plus lisses.

```
// initialise les lumières et l'effet d'anticrénelage (lissage de police)
lights();
smooth();
```

Ensuite, il faut charger la police que l'on a choisit et les images, qui devraient être dans le dossier Data.

```
// Mettre la police de caractère désirée (Doit être dans le fichier Data)
// une police peut être acquise en haut du programme dans Outils -> Générer la police...
police = loadFont("AngsanaNew-Bold-48.vlw");

// Chargement des textures du MPU (prisme)
// Les fichiers PNG nous donne la possibilité d'avoir des trous dans l'image, ce qui augmente le réalisme.
imageDessusMPU = loadImage("MPU6050 A.png");   //Dessus
imageDessousMPU = loadImage("MPU6050 B.png");  //Dessous
imageDevantMPU = loadImage("MPU6050 E.png");   //Devant
imageCoteMPU = loadImage("MPU6050 F.png");     //Côté
```

La fonction **draw**() change complètement. En fait, on garde le fond d'écran noir, mais on remplace tout le reste par une fonction de dessin de cube et une autre pour les informations.

```
void draw()
{
  //Couleur du fond d'écran
  background(0);  //Noir dans notre cas

  drawInfo();
  drawCube();
}
```

Pour la fonction **drawInfo**() on doit commencer avec une condition : on actualise les informations qui seront écrites à l'écran seulement une fois par dixième de seconde ou si le programme vient juste de s'ouvrir pour qu'on ait au moins le temps de lire la donnée. Si le programme n'est pas sur pause, on le sait avec la variable infoPause qui est appelé hors de la fonction, on n'actualise pas les données d'information non plus. Une fois rentré dans la condition, l'intervalle est remis à jour avec le temps total que l'application est ouverte.

**Update : On a ajouté l'information de la distance du gant par rapport au sol dans les informations.

```
void drawInfo()
{
  //Met l'actualisation des infos moins rapides afin de mieux apercevoir les données.
  //Si le temps d'intervalle est écoulé et que les informations ne sont pas sur pause
  if((millis() - fIntervalleInfo >= 100 || millis() < 100) && !infoPause)
  { //Actualisation des données
    fVxI = fVx; fVyI = fVy; fVzI = fVz;
    fAxI = fAx; fAyI = fAy; fAzI = fAz;
    roulementXI = roulementXT; rotationYI = rotationYT; profondeurZI = profondeurZT;
    distanceSolI = fDistanceSol;
    fIntervalleInfo = millis();  //Remet le temps écoulé à 0
  }
```

Aussi, pour cette fonction, il fallu ajouter des floats hors fonction pour les données d'information à l'écran :

```
float fAxI, fAyI, fAzI;   //Accélérations mesurées en x, y et z pour les infos à l'écran
float fVxI, fVyI, fVzI;   //Vitesses angulaires mesurées en x, y et z pour les infos à l'écran
float distanceSolI;       //Distance du sol pour les infos à l'écran
float roulementXI, rotationYI, profondeurZI;   //Angles Euler pour les infos à l'écran
```

Après, on ajuste le texte avec la police de caractère et sa grosseur souhaité. On l'aligne aussi en commençant en haut à gauche de l'interface. Une fois les ajustements faits, on peut écrire le texte.

```
  textFont(police, 20);  //Données de la police de caractère
  textAlign(LEFT, TOP);  //Aligné le texte pour qu'il commence en haut à gauche
  //Écriture des informations
  text(" Angle Euler x (roulement) = " + roulementXI + "\n Angle Euler y (rotation)
       + rotationYI + "\n Angle Euler z (profondeur) = " + profondeurZI +
       "\n\n Vx = " + fVxI + "\n Vy = " + fVyI + "\n Vz = " + fVzI +
       "\n\n Ax = " + fAxI + "\n Ay = " + fAyI + "\n Az = " + fAzI +
       "\n\n Distance du sol = " + distanceSolI, 20, 20);
}
```

Maintenant, on peut faire la fonction **drawCube**(). Tout d'abord, Puisque nous allons avoir qu'une série d'images qui bougent tous ensemble dans notre espace graphique, nous n'allons pas avoir besoin des fonctions pushMatrix() ni popMatrix(). Cependant, si d'autres images s'ajouteraient, on les placerait aux deux extrémités de la fonction. On commence alors en mettant les images à un point d'origine, qui sera le milieu dans notre cas, et on décide la grosseur des images, qui nous donnera la grosseur du MPU à l'écran.

```
void drawCube()
{
  //PushMatrix() doit être utilisé avec PopMatrix()
  //Écrire un code entre ces deux fonctions bouge les images sans bouger le reste
  //Pour notre code, ce n'est pas vraiment nécessaire, puisqu'on n'a pas d'autres images dans l'interface
  //pushMatrix();  //Enregistre le point d'origine du plan

  translate(width/2, height/2 + 50, 0);  //Mettre le MPU à l'endroit souhaité au départ à l'écran
  scale(10);  //Grosseur du MPU à l'écran
```

Ensuite, nous n'avons qu'à recoller le code que nous avions déjà sur les quaternions, les angles d'axe et la rotation de l'image qui était dans le draw avant le changement de l'aspect graphique.

```
//Création du quaternion avec les angles de Euler afin de pouvoir avoir les angles d'axe
quat1 = Quaternion.createFromEuler(-roulementXT, -rotationYT, profondeurZT);

float[] axis = quat1.toAxisAngle();  //Stocke les angles d'axe
rotate(axis[0], axis[1], axis[3], axis[2]);  //Rotation du cube avec les angles d'axe
```

**Update : Après finalisation du code nous avons fait tourner le MPU graphique pour qu'il soit du même côté que nous l'utilisons en vrai afin d'améliorer le réalisme.

```
rotateY(-PI/2);  //Rotation afin d'ajuster pour avoir la bonne direction de départ
```

Ensuite, on appelle les fonctions de dessins des quatre images, soit les six côtés de notre cube. Ils prennent en paramètre les images que l'on souhaitent qu'ils dessinent. (Les côtés avant et arrière prennent la même image et sont donc dans la même fonction. De même pour les côtés gauche et droit.)

```
  drawDessus(imageDessusMPU);
  drawDessous(imageDessousMPU);
  drawCotes(imageDevantMPU);
  drawAvantArriere(imageCoteMPU);

  //popMatrix();  //Remet le point d'origine comme avant pushMatrix()
}
```

La fonction **drawDessus**(), est la première fonction de dessin des côtés du cube. On commence en y mettant beginShape(), qui lui dit quelle forme doit être dessinée. Nous voulons un quadrilatère. On lui dit aussi quelle texture il doit prendre, soit l'image qu'on prend en paramètre.

```
void drawDessus(PImage image1)
{
  beginShape(QUADS); //Commence à dessiner la forme d'un quadrilatère
  texture(image1);  //Dessine l'image dans le quadrilatère
```

Après, on lui dit quel sont les coordonnées qu'il doit prendre. On lui donne quatre points qui seront les coins du quadrilatère. Finalement, on lui mentionne que le dessin est terminé avec endShape().

```
  // Prend les coordonnées du quadrilatère, (x,y,z,u,v).
  // u et v sont les coordonnées horizontales et verticales
  // pour le mappage de texture.
  vertex(-20, -1, -15, 0, 0);
  vertex( 20, -1, -15, 1, 0);
  vertex( 20, -1, 15, 1, 1);
  vertex(-20, -1, 15, 0, 1);

  endShape();
}
```

On doit refaire ces fonctions avec des coordonnées différentes pour : **drawDessous**();

```
void drawDessous(PImage image1)
{
  beginShape(QUADS); //Dessine la forme d'un quadrilatère
  texture(image1);  //Dessine l'image dans le quadrilatère
  // Prend les côtés du quadrilatère, (x,y,z,u,v).
  // u et v sont les coordonnées horizontales et verticales
  // pour le mappage de texture.
  vertex(-20, 1, 15, 0, 0);
  vertex( 20, 1, 15, 1, 0);
  vertex( 20, 1, -15, 1, 1);
  vertex(-20, 1, -15, 0, 1);

  endShape();
}
```

---

drawCotes() ;

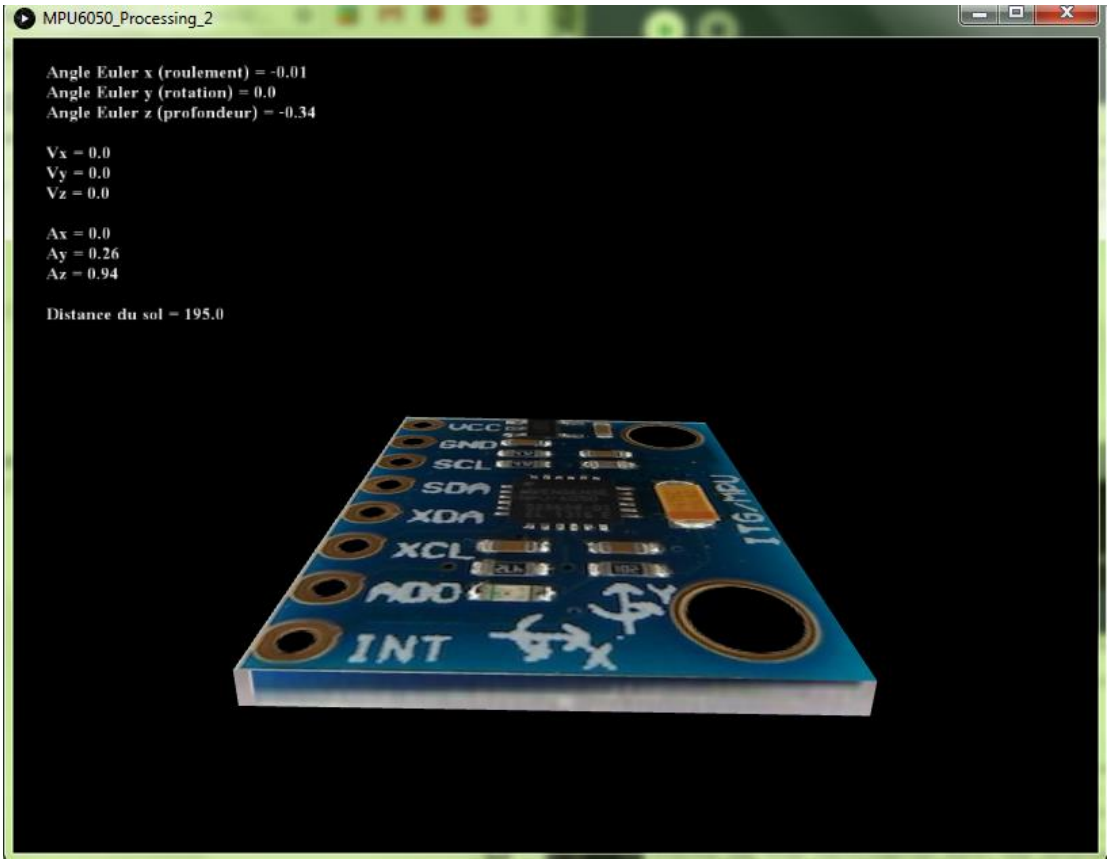
```
void drawCotes(PImage image1)
{
  beginShape(QUADS); //Dessine la forme d'un quadrilatère
  texture(image1);  //Dessine l'image dans le quadrilatère
  // Prend les côtés du quadrilatère, (x,y,z,u,v).
  // u et v sont les coordonnées horizontales et verticales
  // pour le mappage de texture.
  // Côté gauche :
  vertex(-20, -1, 15, 0, 0);
  vertex( 20, -1, 15, 1, 0);
  vertex( 20, 1, 15, 1, 1);
  vertex(-20, 1, 15, 0, 1);

  // Côté droit :
  vertex( 20, -1, -15, 0, 0);
  vertex(-20, -1, -15, 1, 0);
  vertex(-20, 1, -15, 1, 1);
  vertex( 20, 1, -15, 0, 1);

  endShape();
}
```

---

et **drawAvantArriere**().

```
void drawAvantArriere(PImage image1)
{
  beginShape(QUADS); //Dessine la forme d'un quadrilatère
  texture(image1);  //Dessine l'image dans le quadrilatère
  // Prend les côtés du quadrilatère, (x,y,z,u,v).
  // u et v sont les coordonnées horizontales et verticales
  // pour le mappage de texture.
  // Arrière :
  //vertex( 20, -1, 15, 0, 0);
  //vertex( 20, -1, -15, 1, 0);
  //vertex( 20, 1, -15, 1, 1);
  //vertex( 20, 1, 15, 0, 1);

  // Devant :
  vertex(-20, -1, -15, 0, 0);
  vertex(-20, -1, 15, 1, 0);
  vertex(-20, 1, 15, 1, 1);
  vertex(-20, 1, -15, 0, 1);

  endShape();
}
```

Avec ces fonctions, on peut maintenant avoir un environnement graphique de loin supérieur à ce que l'on avait avant avec le MPU Teapot.


MPU6050_Processing_2
Angle Euler x (roulement) = -0.01
Angle Euler y (rotation) = 0.0
Angle Euler z (profondeur) = -0.34
Vx = 0.0
Vy = 0.0
Vz = 0.0
Ax = 0.0
Ay = 0.26
Az = 0.94
Distance du sol = 195.0
VCC
GND
SCL
SDA
XDA
XCL
ADO
INT

Communication avec le drone et commandes

Utilisation d'un projet GitHub pour contrôler un drone Holy Stone par wifi

Voici le répertoire GitHub duquel nous vient le mode de transmission de données vers le drone :

https://github.com/lancecaraccioli/holystone-hs110w

Les drones Holy Stone peuvent être contrôlés à partir d'un téléphone cellulaire avec une application. Ce que l'auteur de ce répertoire a réussi à accomplir est "d'attraper" les paquets de *bytes* qui sortaient de son téléphone par wifi vers le drone. Ainsi, sachant quels paquets étaient envoyés lorsqu'il envoyait différentes commandes, il était en mesure de savoir ce que chaque byte contrôlait. Il est alors venu à la conclusion que les valeurs envoyées pour l'envol correspondaient à ces commandes :

```
var commandSettings = [
  0xff,//unknown header sent with apparently constant value
  0x04,//unknown header sent with apparently constant value
  0x7f,//vertical lift up/down
  0x3f,//rotation rate left/right
  0xc0,//advance forward / backward
  0x3f,//strafe left / right
  0x90,//yaw (used as a setting to trim the yaw of the uav)
  0x10,//pitch (used as a setting to trim the pitch of the uav)
  0x10,//roll (used as a setting to trim the roll of the uav)
  0x40,//throttle
  0x00,//this is a sanity check; 255 - ((sum of flight controls from index 1 to 9) % 256)
];
```

Puisque l'auteur, a décidé de faire son interface sur une page web, il devait utiliser node, qui permet la création de serveurs. Avec node, il pouvait alors envoyer les entrées de touches depuis la page demo.html vers le script api.js, qui faisait le relai vers le drone. Cet interface nous est inutile, car nous voulons transmettre les mouvements de notre senseur MPU-6050 vers le drone à partir de son protocole d'envoi et non de touches de clavier. C'est pourquoi

nous avons choisi d'utiliser Processing, d'autant plus que nous y affichons déjà à l'écran le mouvement du senseur, ce qui signifie que nous n'aurions qu'à utiliser un unique sketch Processing.

Migration de fichiers javascript vers processing

Pour commencer, il fallait trouver où la gestion des différentes commandes était faite. En suivant le nom des fonctions dans le fichier, il a fallu remonter jusqu'à la "fonction " stopSendingCommands dans le fichier *api.js* (le même fichier où le serveur est lancé). Après avoir examiné le code, j'ai tenté d'insérer une ligne qui écrit dans le command prompt avec la fonction console.log(). Cette ligne se situe devant ce qui paraissait être l'assignation de la nouvelle valeur qui est associée à la touche sur laquelle on appuie:

```
if(commandSettings[COMMANDS[KEY]] != commands[key] ){
    //Écriture dans le cmd: Changement de valeur
    console.log(`Changement ${KEY}: ${commandSettings[COMMANDS[KEY]]} -> ${commands[key]}`);
    commandSettings[COMMANDS[KEY]] = commands[key];
}
```

La condition insérée restreint l'écriture dans le command prompt seulement au moment où la valeur d'une commande est modifiée:

```
start flight commands
Changement LIFT: 127 -> 63
Changement LIFT: 63 -> 127
Changement STRAFE: 63 -> 0
Changement ADVANCE: 192 -> 127
Changement STRAFE: 0 -> 127
Changement ADVANCE: 127 -> 192
Changement STRAFE: 127 -> 63
Changement ADVANCE: 192 -> 0
Changement ADVANCE: 0 -> 192
Changement ADVANCE: 192 -> 0
Changement ADVANCE: 0 -> 192
Changement STRAFE: 63 -> 0
Changement STRAFE: 0 -> 63
Changement STRAFE: 63 -> 127
Changement STRAFE: 127 -> 63
Changement LIFT: 127 -> 63
Changement LIFT: 63 -> 127
Changement LIFT: 127 -> 255
Changement LIFT: 255 -> 127
```

Ensuite, ce qui permettait réellement d'interagir avec le serveur était le fichier html *demo* qui servait “d’interface” de contrôle. Cependant, c’est dans le fichier api.js que les lignes permettant d’envoyer les *bytes* nécessaires au drone se trouvaient. Puisqu’il n’y avait pratiquement aucune documentation dans ce fichier, j’ai dû y aller de logique et mettre en commentaires les lignes susceptibles d’être responsables du départ du drone (mon seul indicateur étant le clignotement des lumières et l’envol du drone). Ainsi, comme je le suspectais, les serveur tcp et websocket implémentés dans le fichier ne nous seraient pas utiles, puisqu’ils ne servaient qu’à faire le pont entre la page web et le serveur Node. Donc, le seul serveur indispensable à notre but de lier le drone à un programme en Processing se trouvait être le serveur **UDP**. Le ***U**ser **D**atagram **P**rotocol* est un protocole simple de communication à travers un port vers une adresse IP. Ce protocole est utile pour envoyer rapidement des petites quantités, soit des datagrammes (blocs de données en réseau), ce qui est parfait pour nos besoins, soit l’envoi d’un tableau de *bytes* par WiFi. Après avoir compris quel protocole utiliser pour envoyer les données vers le drone, il a fallu trouver comment implémenter le dit-protocole dans un programme Processing. Deux librairies pouvant m’aider à atteindre cet objectif semblaient déjà exister, mais après plusieurs tests, c’est la librairie oscP5 qui m’a permis de faire démarrer le drone. Puisque je ne voulais pas que le drone se mette à voler dès que je démarrais le programme, j’avais spécifié noLoop() dans le setup et puis loop() à l’appui sur la touche ENTER, mais c’est au départ que je devais laisser la boucle s’exécuter un certain nombre de fois pour que le drone puisse s’envoler. Pour ce qui est des valeurs dans le tableauS de commandes, les seuls inconnus étaient les deux premiers. Ces deux valeurs constantes semblent constituer l’entête du paquet UDP nécessaire à la lecture du côté du drone.

Changement des commandes selon les angles de rotation

Il était évidemment de bonne pratique de concevoir une fonction qui est valide pour tous les angles et donc pour toutes les directions. Ainsi, la façon la plus logique nous apparaissait être de convertir les angles en pourcentages de l’angle maximum pouvant être atteint en bougeant la main. Ainsi, l’angle d’inclinaison correspondant à une direction donnée serait proportionnel à

la vitesse à laquelle le drone se déplace dans ladite direction. Ainsi, pour les valeurs d'angles correspondant à un mouvement latéral vers la gauche, à une rotation vers la gauche ou à un mouvement longitudinal vers l'arrière, il fallait s'approcher de la valeur de 0. Pour accomplir cela, il fallait donc que la valeur envoyée au drone soit inversement proportionnelle à la valeur de l'angle selon la valeur milieu (défaut/immobilité):

```
parametres[index] = byte(abs((1 -(angle/limite)) * milieu));
```

Pour les valeurs d'angles correspondant à un mouvement latéral vers la droite, à une rotation vers la droite ou à un mouvement longitudinal vers l'avant, il fallait s'approcher de la valeur maximale, alors la valeur envoyée devait être proportionnelle à l'angle. Pour ce faire, il fallait ajouter à la valeur milieu un pourcentage de la différence entre la valeur maximale et la valeur milieu :

```
parametres[index] = byte(abs(angle/limite) * difference + ajout);
```

Changement des commandes d'altitude selon la distance par rapport au sol

Nous allons changer la hauteur dans la fonction changerHauteur(), que l'on appelle dans la fonction inputMPU().

Tout d'abord, il faut savoir la position de départ par rapport au sol pour pouvoir interpréter les positions futures comme une volonté de descendre ou de monter. Alors nous allons initialiser la valeur de la hauteur milieu avec la première valeur qui de distance qui est lue.

```
boolean hauteurInitialise = false;  //Dit si on a déjà déclaré le milieuHauteur ou non
float fDistanceSol; //Distance entre le gant et le sol
float milieuHauteur = 0.0f;  //La hauteur par rapport au sol du milieu - soit la hauteur de départ
```

```
// Fonction qui prend la distance par rapport au sol et la compar avec celle de départ
// pour savoir si elle est plus basse ou plus haute et, ainsi, changer la valeur de lift du drone
void changerHauteur()
{
  if(hauteurInitialise == false)
  {
    milieuHauteur = fDistanceSol;
    hauteurInitialise = true;
  }
```

Si milieuHauteur a déjà été initialisé, on peut donc passer à l'autre étape. Cette étape est le changement de hauteur qui est fait seulement si le gant est positionné relativement à l'horizontale, pour que le HC-SR04 puisse prendre la distance par rapport au sol et non par rapport à un mur ou un autre objet qui n'est pas situé vers le bas et qui pourrait fausser les données. Pour changer la hauteur, on prend un pourcentage de différence entre la valeur de distance du sol initiale et celle du présent. Si la valeur est plus haute que la valeur initiale, le pourcentage sera plus haut que 100% et augmentera plus la distance grandira. Si la valeur est plus basse que la valeur initiale, le pourcentage sera plus bas que 100% et diminuera plus que la distance grandira. Le pourcentage est multiplié par la valeur par défaut qui est le milieu, soit 100% de la valeur.

```
  else if(roulementXT <= 0.25f && roulementXT >= -0.25f && profondeurZT <= 0.25f && profondeurZT >= -0.25f)
  { //Variation du LIFT proportionnellement à la hauteur du gant
    parametres[PARAMETRES_INDEXES.LIFT] = byte(abs((1 -((milieuHauteur - fDistanceSol)* 3f/milieuHauteur))
    * parametresDefaut[PARAMETRES_INDEXES.LIFT]));
  }
}
```

Étape d'envoi des données

Pour que le drone accepte les changements de commande, il faut actualiser la somme de contrôle (checksum, dernier élément du tableau de commandes) après chaque changement. Ceci assure que le paquet d'octets est valide à l'arrivée au drone et qu'il correspond ainsi à celui envoyé au départ.