



NAUTILUS - RAPPORT

Dylan Bideau, Julien Turpin, Pierre Bogrand, Guillaume Vincenti

21 Avril 2018

Sommaire

1	Introduction	3
2	Présentation du projet	4
3	Cahier des charges	5
3.1	Analyse Fonctionnelle	5
3.1.1	Structure	5
3.1.2	Commandabilité	5
3.1.3	Milieu d'utilisation	5
3.1.4	Energie	6
3.1.5	Motorisation	6
3.1.6	Acquisitions	6
4	Outils de gestion	7
4.1	Trello	7
4.2	Github	8
4.3	Gestion de Budget	9
5	Motorisation et énergie	10
5.1	Moteurs brushless et ESC	10
5.2	Calibration et commande des ESC	12
5.3	Alimentation et montage	15
6	Acquisition et Commandabilité	16
6.1	Raspberry	16
6.1.1	Cameras	16
6.1.2	Capteur de Pression/Temperature	19
6.1.3	Central Inertielle	19
6.2	PC	23
6.2.1	Interface	23
6.2.2	Tunnel SSH	25
6.2.3	Videos	26
6.2.4	Capteurs et Moteurs	29
6.2.5	Affichage 3D	30
6.2.6	Cartographie	32

7 Structure	34
7.1 Design	34
7.1.1 Squelette interne	36
7.1.2 Tube	37
7.1.3 Coque	37
7.2 Matériaux, méthode de construction	38
7.3 Simulations poids et aqua-dynamisme	40
7.4 Test en eau	41
8 Conclusion	42
Bibliographie	43
9 Références	44
Table des figures	45

Chapitre 1

Introduction

Les fonds marins réunissent aujourd’hui de nombreux secteurs et enjeux, tant professionnels que particuliers. On y retrouve entre autre l’exploration sous-marine, la surveillance et maintenance d’installations professionnelles, ainsi que la cartographie des fonds marins. Tout ces domaines demandent le développement de solutions techniques plus rentables et pratiques qu’une intervention humaine. Notre projet propose ainsi un ROV (Remotely Operated Vehicle) polyvalent et simple d’utilisation à cet effet.

Chapitre 2

Présentation du projet

Un ROV est un robot sous-marin contrôlé à distance et permettant une acquisition d'informations, visuelles ou à partir de capteurs. Notre projet de ROV filoguidé, Nautilus, sera transportable et pilotable à l'aide d'un ordinateur portable. Il permettra d'observer facilement des installations ou des fonds marins à l'aide de caméras. Disposant également de fonctions avancées, le Nautilus sera en mesure de recréer le fond marin d'une zone géographique déterminée par l'utilisateur à partir d'une batterie de photographies prises lors de la phase d'exploration. Les différentes fonctionnalités du Nautilus en font ainsi un outil polyvalent, permettant exploration, maintenance et cartographie des fonds.

Chapitre 3

Cahier des charges

3.1 Analyse Fonctionnelle

3.1.1 Structure

Facilement transportable et peu emcombrant.

Contraintes :

- Poids : 2-3kg
- Dimension : 300*200*150mm
- Etanche de norme IP 68

3.1.2 Commandabilité

Commandé à distance par une liaison filaire.

Contraintes :

- Câble : 15m
- Carte intégrée dans le ROV
- FPV (First Person View)
- Piloté au clavier

3.1.3 Milieu d'utilisation

Adapté aux contraintes imposées par son environnement.

Contraintes :

- Eau non salé (moins de 1 g de sels dissous par kilogramme d'eau)
- Eau translucide (transmittance de la lumière entre 75% et 95%)
- Lieu : Piscine, lac
- Ecoulement laminaire
- Courant marin inférieur à 2 noeuds
- Profondeur de 10m (résistant à 2 bars)

3.1.4 Energie

Etre entièrement autonome.

Contraintes :

- Autonomie de 20 minutes

3.1.5 Motorisation

Etre mobile une fois immergé.

Contraintes :

- Propulsion électrique
- Déplacement horizontal (Vitesse maximale de 1m/s)
- Déplacement vertical (Vitesse maximale de 0.5m/s)
- Direction droite/gauche à 360 degrés

3.1.6 Acquisitions

Acquérir et transmettre l'information.

Contraintes :

- Acquisition et retransmission d'un signal vidéo
- Acquisition et stockage de photographies
- Mesure de la pression
- Mesure de la position relative avec signaux GPS

Chapitre 4

Outils de gestion

4.1 Trello

Notre premier outil de gestion est Trello. Il nous permet de gerer notre projet en terme de planification des tâches. Nous utilisons un code couleur pour chaque membre du projet :

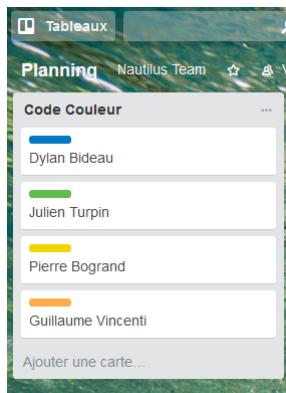


FIGURE 4.1 – Trello - Code Couleur

Voici actuellement ce que nous devons faire (Lors du premier livrable) :

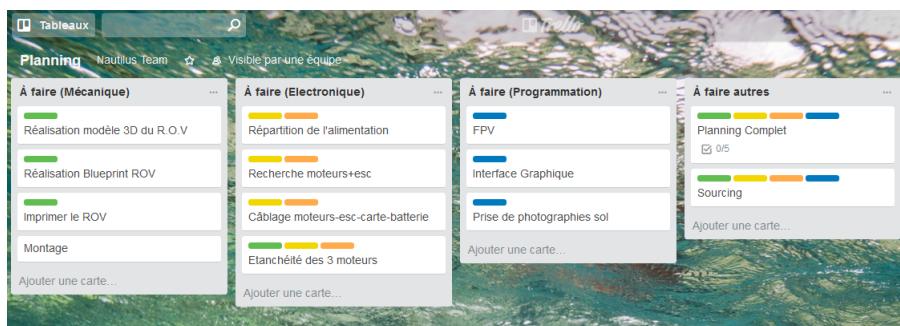


FIGURE 4.2 – Trello - Planning A faire

Et ce que nous sommes entrain de faire ainsi que les tâches terminées :

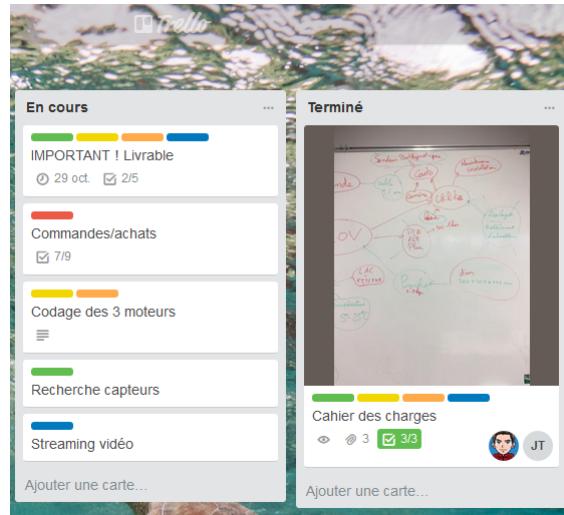


FIGURE 4.3 – Trello - Planning En cours et Terminé

4.2 Github

Tout notre code, que cela soit pour les documents en latex ou les codes liés au projet, est sur Github : <https://github.com/ROV-Nautilus/Nautilus>
En voici un aperçu :

Commit	Description	Date
dydydu86 Merge remote-tracking branch 'origin/master'	Latest commit 12ba12d 3 days ago	
Cahier des Charges	Debut Livrable	3 days ago
Premier Livrable	Correctif	3 days ago
MS5803-14BA_Breakout_v10.pdf	Schéma capteur pression	10 days ago
README.md	Correctif	3 days ago

FIGURE 4.4 – Github - Nautilus

4.3 Gestion de Budget

La gestion de notre budget est réalisé à l'aide d'Excel et de Trello, en effet Trello nous permet de mettre en commun les propositions de dépenses et avec Excel nous stockons l'ensemble des commandes par fournisseurs

Les voici ci-dessous :

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Fournisseur: Conrad												
2	Pos	I	P	Art	Désignation	Qté commandée	UA	Date Livrais	Prix unitai	Devise	Q	Grpe	Div
3	10	Z			1429711 - 62 Hélice de bateau 2 pales droite 52.5 mm	1	PC	31.10.2017	3.2	EUR	PC	TA.01	ESEA
4	20	Z			1429712 - 62 Hélice de bateau 2 pales gauche 52.5 mm	1	PC	31.10.2017	3.2	EUR	PC	TA.01	ESEA
5	30	Z			231891 - 62 Moteur brushless ROXXY 315079	3	PC	31.10.2017	28.8	EUR	PC	TA.01	ESEA
6	40	Z			238883 - 62 Batterie d'accu (NiMh) 7.2 V 3000 mAh	1	PC	31.10.2017	20	EUR	PC	TA.01	ESEA
7	50	Z			1429714 - 62 Hélice de bateau 2 pales droite 57 mm	1	PC	31.10.2017	3.2	EUR	PC	TA.01	ESEA
8	60	Z			1419716 - 62 Raspberry Pi 3 Modèle B 1 Go	1	PC	31.10.2017	36	EUR	PC	TA.01	ESEA
9	70	Z			417941 - 62 Carte microSD 16Go & adaptateur	1	PC	31.10.2017	9.6	EUR	PC	TA.01	ESEA
10													
11					Frais de port : 7€								

FIGURE 4.5 – Commande Conrad

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Fournisseur: RobotShop												
2	Pos	I	P	Art	Désignation	Qté commandée	UA	Date Livrais	Prix unitai	Devise	Q	Grpe	Div
3	10	Z			RB-Sup-14 ESC Multirotor 20A M20A	3	PC	31.10.2017	10.25	EUR	PC	TA.01	ESEA
4													
5					Frais de port : 7€								
6													

FIGURE 4.6 – Commande Robotshop

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Fournisseur: Exptech												
2	Pos	I	P	Art	Désignation	Qté commandée	UA	Date Livrais	Prix unitai	Devise	Q	Grpe	Div
3	10	Z			EXP-R25-547 IMU 9 v5 Gyro/Accelerometer/Compass	1	PC	31.10.2017	15.95	EUR	PC	TA.01	ESEA
4	20	Z			EXP-R05-663 Sparkfun Pressure Sensor MS5803-14BA	1	PC	31.10.2017	64.95	EUR	PC	TA.01	ESEA
5					Frais de port :	9.90	€						
6													
7													
8													

FIGURE 4.7 – Commande Exptch

Chapitre 5

Motorisation et énergie

5.1 Moteurs brushless et ESC

Dans un premier temps, il a été question de la technologie des moteurs à utiliser. Après une étude des différentes solutions disponibles, nous avons finalement choisi des moteurs brushless [1]. En effet, les moteurs brushless sont des machines synchrones auto-pilotées à aimants permanents et donc sans balais.

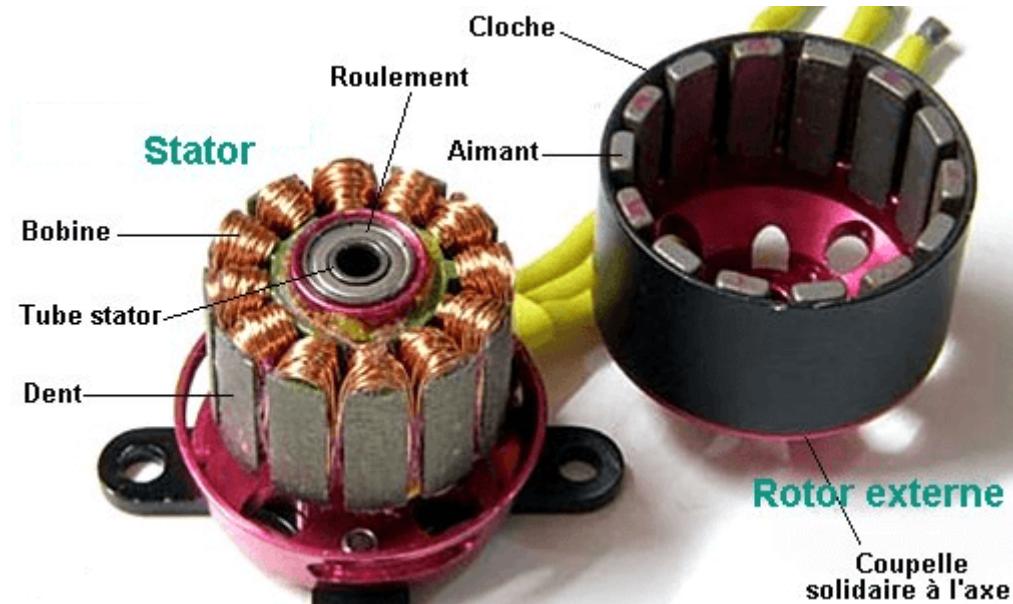


FIGURE 5.1 – Moteur Brushless

Le principal avantage de ces moteurs est qu'ils peuvent être utilisés immersés dans l'eau sans aucun traitement particulier au préalable. En revanche, un système électronique de commande doit assurer la commutation du courant dans les enroulements statoriques : les ESC, ou Electronic Speed Controllers. Un ESC transforme un signal d'alimentation continu, dans notre cas issu d'une batterie, en un signal triphasé envoyé ensuite au moteur brushless. Pour contrôler la vitesse de rotation du moteur, on envoie à l'ESC un signal de commande, généralement crêteau, et dont le rapport cyclique définit la vitesse du moteur.

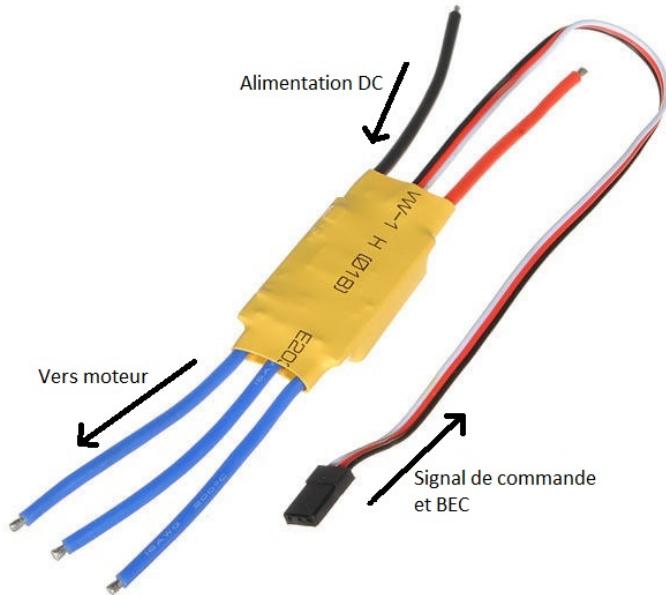


FIGURE 5.2 – ESC

Les trois ESC utilisés dans un premier temps pour nos moteurs [2] sont également équipés d'un circuit éliminateur de batterie, ou BEC, permettant de générer un signal d'alimentation constant de 5V et 3A maximum. Ce dernier permet d'alimenter un autre composant, comme une carte Raspberry Pi dans notre cas, sans avoir à recourir à une seconde batterie.

Cependant, ces ESC ne permettent la rotation du moteur que dans un seul sens. Le seul moyen de modifier le sens de rotation du moteur dans ce cas est d'échanger deux des trois signaux déphasés envoyés au moteur. La direction droite/gauche étant assurée par les deux moteurs de propulsion arrière, cette particularité n'est pas problématique : la rotation plus rapide d'un des deux moteurs arrière par rapport à l'autre permet de diriger le ROV à gauche ou à droite. En revanche, le moteur vertical devant assurer la propulsion verticale doit pouvoir tourner dans les deux sens. Un second modèle d'ESC a donc été nécessaire pour permettre au moteur de tourner dans les deux sens. Ce dernier [3] possède ainsi un mode "reverse" permettant au moteur de tourner dans les deux sens, ainsi qu'un BEC, et sera donc attribué au moteur vertical du ROV.

5.2 Calibration et commande des ESC

Traditionnellement, le signal de contrôle envoyé à l'ESC est un signal PWM de fréquence 50 Hz environ, un certain écart de cette valeur étant accepté. Dans notre cas, le signal est généré par une Raspberry Pi 3, et l'amplitude du signal est donc de 3,3 V environ.

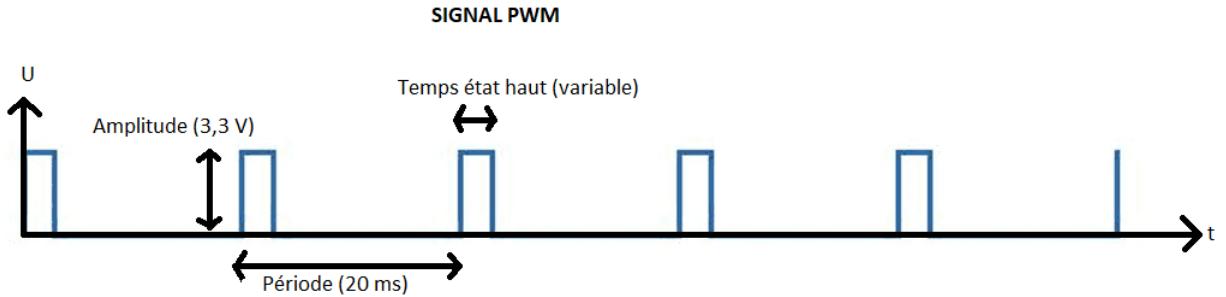


FIGURE 5.3 – Signal PWM

La caractéristique la plus importante de ce signal est la largeur de chaque impulsion. C'est cette dernière, généralement entre 0.5 ms et 2.5 ms, qui commande directement l'amplitude du signal envoyé aux moteurs et donc leur vitesse de rotation. Notre carte doit donc être capable de générer un signal créneau à 50 Hz, et de rapport cyclique variable sur commande. Pour générer un tel signal, on utilise le programme [4], disponible sur Github. En effet ce dernier a été conçu pour piloter des servomoteurs à partir des pins GPIO de la Raspberry Pi, et est donc le plus adapté pour générer un tel signal. Pour installer ServoBlaster sur la carte, on rentre les commandes suivantes sur le terminal de Raspbian :

```

1 sudo apt-get install git
2 git clone https://github.com/richardghirst/PiBits.git
3 cd PiBits/ServoBlaster/user
4 nano init-script
5 # Supprimer la valeur --idle-timeout=2000 des options par défaut
6 nano servod.c
7 # Modifier la ligne 960 pour avoir "else if (strstr(modelstr, "BCM2709") ||
8 strstr(modelstr, "BCM2835"))"
9 make
9 sudo make install

```

FIGURE 5.4 – Programme Servoblaster

Une fois cette installation effectuée, on peut générer le signal à l'aide de la commande, toujours dans le terminal de Raspbian, "echo x=y > /dev/servoblaster", où x représente le GPIO que l'on utilise et y la valeur du temps d'état haut, en dizaine de us (par exemple y=100 correspond à un temps d'état haut de 1 ms). Les valeurs de y peuvent aller de 50 à 250, correspondant respectivement à un temps haut de 0.5 ms et 2.5 ms.

Les valeurs de x désignent les GPIO suivant :

Servo number	GPIO number
0	4
1	17
2	18
3	21/27
4	22
5	23
6	24
7	25

FIGURE 5.5 – Valeurs GPIO

On obtient ainsi des commandes comme ci-dessous :

```
1 | echo 0=200 > /dev/servoblaster
2 | echo 0=100 > /dev/servoblaster
```

FIGURE 5.6 – Commandes moteurs

Avec chaque nouvelle valeur de y modifiant la valeur du temps haut du signal sortant.

Toutefois, avant leur utilisation, les ESC nécessitent d'être calibrés. En effet, en fonction des radiocommandes ou autre dispositifs de commande utilisés, l'intervalle des valeurs du temps d'état haut du signal de commande envoyé à l'ESC peut différer. L'ESC doit donc être calibré pour que la valeur maximale de temps d'état haut corresponde à la vitesse maximale de rotation du moteur. De même pour une valeur neutre (moteur à l'arrêt), et une valeur minimale correspondant à la vitesse de rotation inverse maximale, disponible uniquement pour l'ESC "Reverse", et donc le moteur vertical associé.

La calibration ne peut pas être effectuée à l'aide des commandes issues de ServoBlaster, l'ESC nécessitant une variation douce du temps d'état haut pour cela. C'est le cas d'un joystick de télécommande, mais pas du programme ne permettant que des variations abruptes de la valeur du temps d'état haut. La calibration des ESC a donc été effectuée à l'aide d'une radiocommande disponible à l'ENSEA [5]. Pour celà, on place le joystick de la manette au point neutre, on branche l'ESC à l'émetteur de la radiocommande et on allume dans l'ordre la radiocommande, puis l'ESC. Lorsque ce dernier s'allume, il associe ainsi automatiquement le signal en train d'être reçu au point neutre, donc moteur immobile. On effectue la même opération avec le joystick au maximum pour calibrer la valeur maximale de temps d'état haut et l'associer à la vitesse maximale de rotation du moteur. Une fois cette calibration effectuée, l'ESC en marche ne fonctionnera qu'après avoir reçu le signal correspondant à l'état neutre pendant un certain temps. Celà permet d'éviter entre autres un démarrage intempestif des moteurs.

Une fois la calibration effectuée, on observe à l'oscilloscope le signal de commande envoyé par la manette pour déterminer le temps d'état haut correspondant au point neutre, à la vitesse

maximale, et à la vitesse maximale en rotation inverse pour le moteur vertical.
On obtient les résultats suivants :

Type d'ESC	Temps état haut	Point neutre (valeur de y)	Vitesse maximale (valeur de y)	Vitesse maximale inverse (valeur de y)
ESC 20A (propulsion)	1,2 ms (120)		1,8 ms (180)	Pas de mode Reverse
ESC 30A (moteur vertical)	1,5 ms (150)		1,9 ms (190)	1,4 ms (140)

FIGURE 5.7 – Tableau des temps

Après calibration et relevé de ces valeurs, on peut bien commander les 3 moteurs à l'aide des commandes de ServoBlaster.

5.3 Alimentation et montage

On utilise pour l'alimentation des ESC et des moteurs une batterie NiMh, de capacité 3000 mAh et délivrant une tension de 7,2 V [6]. L'alimentation de la Raspberry est elle assurée par les BEC des ESC : ces derniers, mis en parallèle, délivrent une alimentation constante d'environ 5V et 3A, suffisante à alimenter la carte. On obtient donc le schéma fonctionnel suivant :

Architecture de la motorisation du ROV

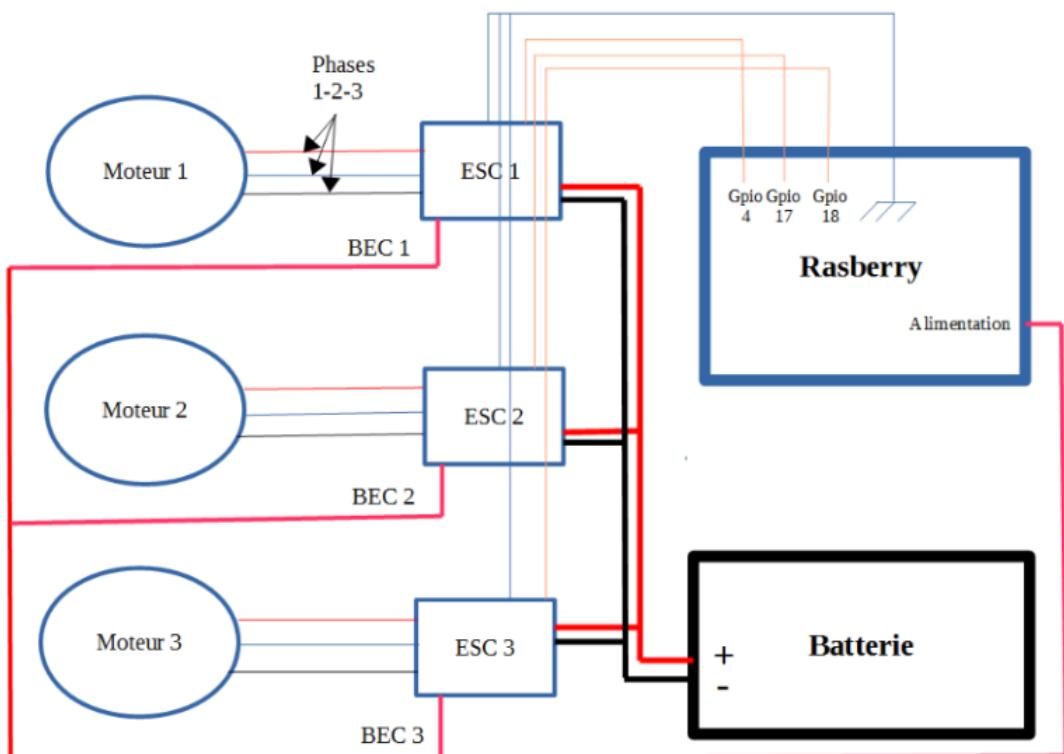


FIGURE 5.8 – Architecture de la motorisation du ROV

Lorsque les 3 moteurs sont à leur vitesse maximale, avec la carte fonctionnelle, le courant prélevé sur la batterie est de 2A maximum. L'autonomie théorique dans cette configuration est donc d'environ 90 minutes. Toutefois cette dernière risque de diminuer avec l'utilisation et donc l'alimentation des différents capteurs, ainsi que la résistance de l'eau en condition.

Chapitre 6

Acquisition et Commandabilité

Dans un second temps, nous devions relier les différents éléments de notre ROV sur une carte et ensuite traiter les informations reçues pour pouvoir agir sur les moteurs vus précédemment. Nous avons choisi la Raspberry.

6.1 Raspberry

Une partie de la programmation et des calculs est effectué sur une Raspberry PI 3 qui supportait tous les types de connexions que l'on voulait, en voici la description.



FIGURE 6.1 – Raspberry

6.1.1 Cameras

Nous avons 2 caméras qui permettent, l'une la direction (vision frontale) et l'autre la cartographie (vision par dessous). Dans un premier temps, détaillons leur connexion entre la Raspberry et le traitement effectué par celle-ci.

Logitech C170

La première est une webcam Logitech C170, que nous avons démonté pour l'assemblage, relié en USB à la Raspberry.

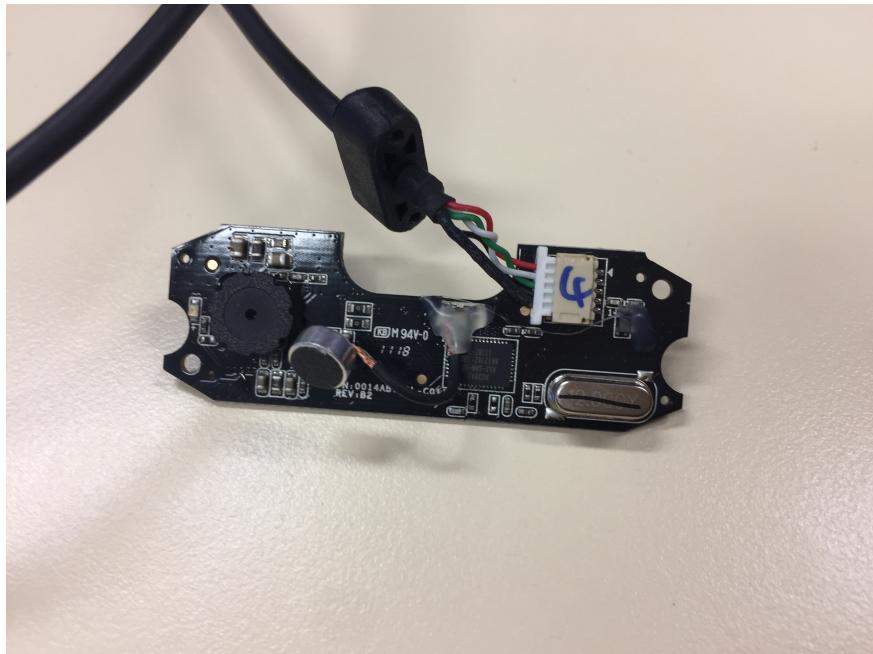


FIGURE 6.2 – Logitech C170

Nous l'avons choisi car le pilote de celle ci est déjà installé nativement sur la Raspberry. Nous utilisons motion qui permet d'envoyer le flux vidéo venant de la caméra et de la diffuser en ligne sur notre adresse local [7]. En voici le résultat sur un navigateur :

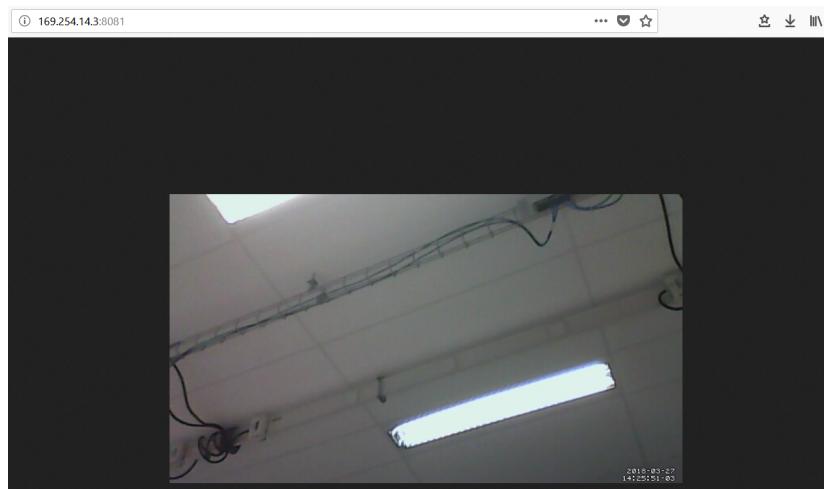


FIGURE 6.3 – Photo Caméra Frontale

Cette vidéo sera récupérée par l'interface (expliquer dans le chapitre associé) en 640*360.

Caméra V2

La deuxième est un module caméra pour raspberry [8] qui se raccorde directement par une nappe (un bus de type CSI-2).

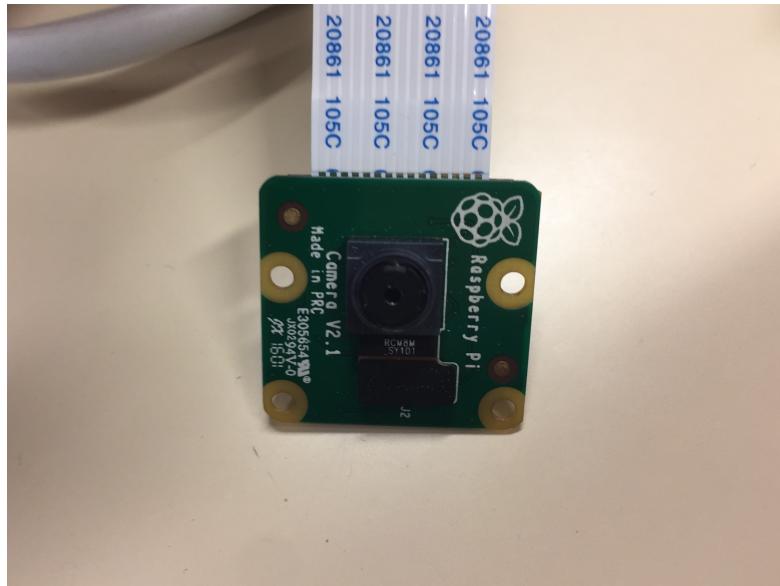


FIGURE 6.4 – Module Camera V2 Raspberry

Elle se paramètre en python avec les librairies données par le constructeur. De même que l'autre caméra, nous renvoyons un flux video en ligne sur notre adresse local [9] mais cette fois-ci sur un autre port.

Le résultat sur un navigateur :

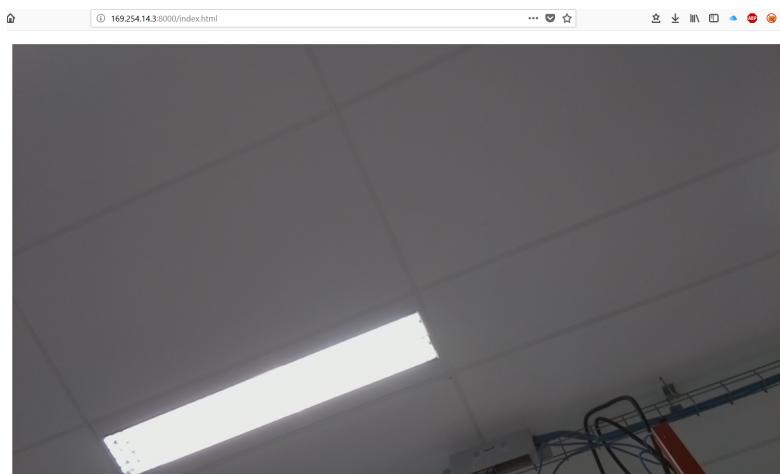


FIGURE 6.5 – Photo Caméra du dessous

La vidéo est diffusée en 1920*1080 et affichée par l'interface.

6.1.2 Capteur de Pression/Temperature

Le Nautilus est équipé d'un capteur lui permettant d'obtenir la température et la pression. Pour l'instant le ROV renvoie directement ces données mais il est envisageable d'utiliser par exemple la pression pour déterminer avec précision la profondeur du Nautilus. Le module utilisé pour notre ROV est le MS5803-14BA de Sparkfun.

MS5803-14BA

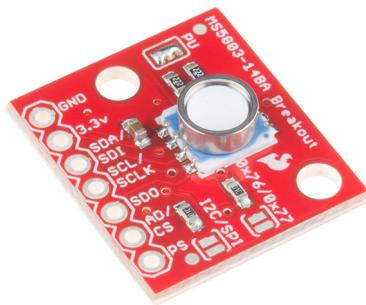


FIGURE 6.6 – Capteur de pression et de température MS5803-14BA

Ce capteur de pression et de température dispose d'une membrane ,pour des mesures de pressions de gaz ou de liquides, résistant à une pression de 30 Bar. De plus, il possède aussi des interfaces SPI et I2C. Nous utiliserons l'interface I2C car celle-ci utilise moins de connexions et la fréquence de communication est basse. De plus bien que les deux modes de communication permettent de placer plusieurs modules sur un même câble, le SPI semble être limité sur la Raspberry. Dans nos conditions de connexion l'adresse esclave est 0x1E. Pour utiliser ce capteur on utilise directement un script python qui sera lancé à chaque fois que l'on veut obtenir la pression. Le script est disponible sur le site suivant <https://github.com/ControlEverythingCommunity/MS5803-14BA>.

Comme ce capteur doit se trouver dans l'eau, nous l'avons recouvert, à l'exception de la membrane de résine Epoxy.

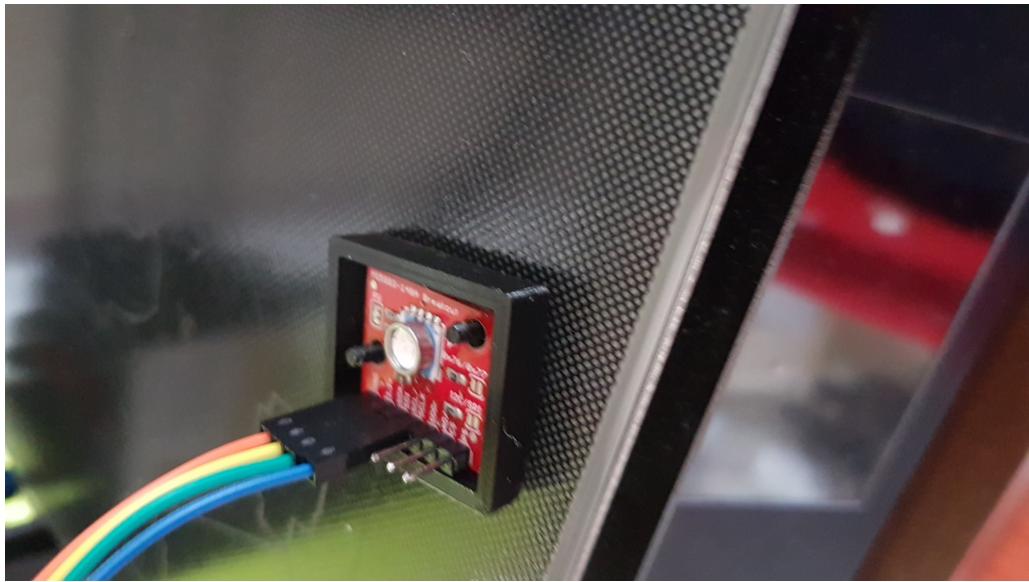


FIGURE 6.7 – Capteur MS5803-14BA avant le coulage dans la résine

6.1.3 Central Inertielle

Le Nautilus dispose aussi d'une centrale inertielle. Avec celle-ci, le Rov peut déterminer son assiette ainsi que sa vitesse. Il est bien-sûr possible d'extrapoler ces données pour obtenir la position et suivre le déplacement du drone. Le module utilisé pour notre ROV est le MinIMU-9 v2 de Pololu.

MinIMU-9 v2

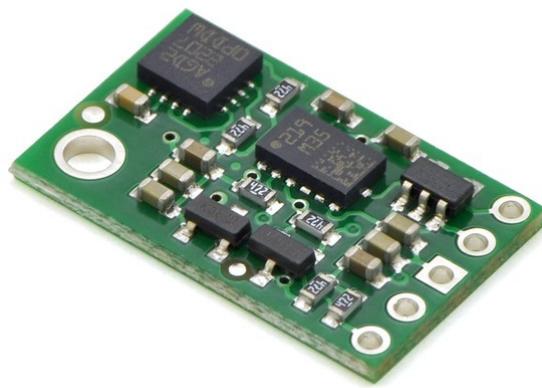


FIGURE 6.8 – Matrice inertielle MinIMU-9 v2

Cette matrice inertielle se compose d'un gyroscope, d'un acceleromètre et d'un compas. De

même ce module dispose des interfaces I2C et SPI mais nous utiliserons une fois encore le I2C. La complexité de la mise en place de cette matrice nous a poussé à aussi utiliser un code trouvé sur internet. Le script est disponible sur le site suivant <https://github.com/DavidEGrayson/minimu9-ahrs>. Ce code est accompagné d'un tutoriel permettant de mieux le comprendre et le mettre en place. Le code reconnaît la matrice inertuelle utilisée et donc s'adapte en connaissance, pas besoin de connaître l'adresse esclave. Nous avons quand même réalisé quelques modifications pour obtenir plus facilement les données en continue. En effet nous avons dans un premier temps modifier le fichier "minimu9-ahrs.cpp". Celui-ci ne faisait qu'afficher les valeurs d'angles d'euler dans le terminal. De plus la vitesse de retour des données était trop rapide, ces même donnée n'étaient pas stockées, et le système ne retournait pas uniquement les angles d'eulers. C'est pour cela que nous avons décidé de stocker les valeurs d'angles d'euler dans un fichier texte qui se réinitialise toutes les 100 valeurs.

On remplace alors la ligne d'affichage 63-64 :

```

1 void output_euler( quaternion & rotation )
2 {
3     std::cout << ( vector )( rotation . toRotationMatrix() . eulerAngles( 2 , 1 , 0
4                                     * ( 180 / M_PI ) );
5 }
6

```

Par le code suivant :

```

1
2 void output_euler( quaternion & rotation )
3 {
4     if ( c==100 )
5     {
6
7         std::ofstream fichier( "euler.txt" , std::ios::out | std::ios::trunc );
8         c=0;
9
10        if( fichier )
11        {
12
13            fichier << (( vector )( rotation . toRotationMatrix() . eulerAngles( 2 , 1 , 0
14             * ( 180 / M_PI )) [ 0 ] << ' ' << (( vector )( rotation . toRotationMatrix() . eulerAngles
15             ( 2 , 1 , 0 ) * ( 180 / M_PI )) [ 1 ] << ' ' << (( vector )( rotation . toRotationMatrix() .
16             eulerAngles( 2 , 1 , 0 ) * ( 180 / M_PI )) [ 2 ] << std::endl ;
17             c++;
18             fichier . close ();
19         }
20         else
21     {
22         std::cerr << " Impossible d'ouvrir le fichier ! " << std::endl ;
23     }
24     else
25     {
26         std::ofstream fichier( "euler.txt" , std::ios::out | std::ios::app );

```

```

26         if( fichier )
27         {
28
29             fichier << (( vector )( rotation . toRotationMatrix () . eulerAngles ( 2 , 1 ,
0 ) * ( 180 / M_PI )) [ 0 ] << ' / ' << (( vector )( rotation . toRotationMatrix () .
eulerAngles ( 2 , 1 , 0 ) * ( 180 / M_PI )) [ 1 ] << ' / ' << (( vector )( rotation .
toRotationMatrix () . eulerAngles ( 2 , 1 , 0 ) * ( 180 / M_PI )) [ 2 ] << std :: endl ;
30             c++;
31             fichier . close ();
32         }
33         else
34         {
35             std :: cerr << " Impossible d'ouvrir le fichier ! " << std :: endl ;
36         }
37     }
38     std :: cout << c ;
39
40 }
41

```

On écrit une ligne 'std : :cout « c ;' pour voir si le compteur de valeurs présent dans le fichier text s'incrémente bien.

6.2 PC

Maintenant que nous avons relié tous nos moteurs, caméras et capteurs à la raspberry ainsi qu'un premier traitement des informations, nous allons voir comment nous traitons cela sur le PC.

6.2.1 Interface

En premier lieu parlons de l'interface, celle ci à pris différentes formes au cours du temps, ici nous presenterons que la dernière version. Toute la partie PC a été programmé en JAVA [10], l'interface utilise la bibliothèque graphique Swing qui nous permet de gerer l'affichage facilement que ce soit pour la video ou pour les interactions. Lorsque l'application est lancé, nous arrivons donc dans un premier menu :

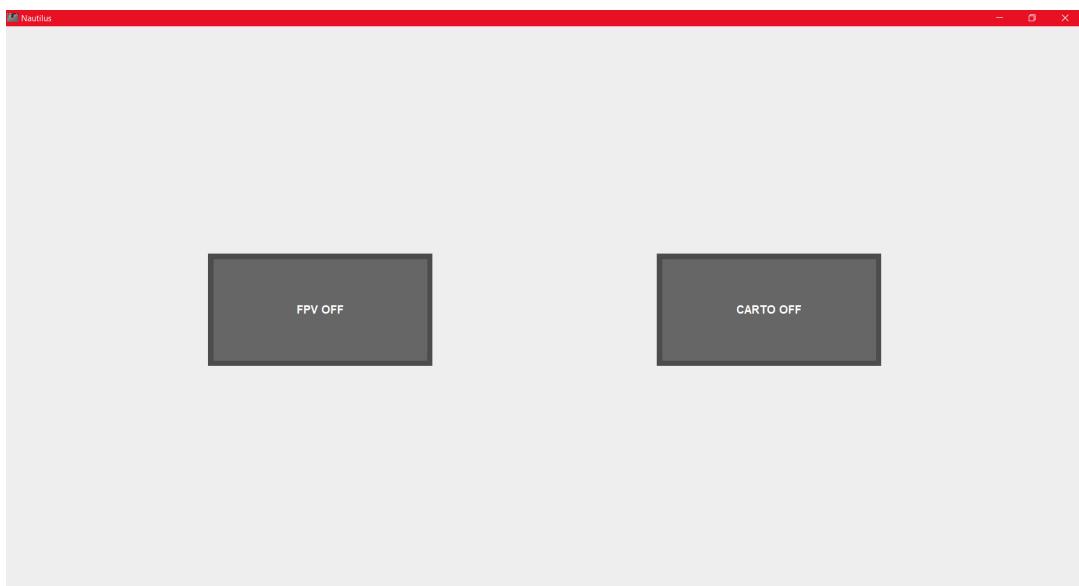


FIGURE 6.9 – Premier Menu Interface

Le bouton de Droite mène à une partie que nous développerons dans la partie 6.2.6 liée à la Cartographie.

Nous avons donc créé une fenêtre JFrame :

```
1 Interface inter = new Interface("Nautilus", 0, 0, 1920, 1080, true);  
2
```

Elle est paramétrée de façon à prendre tous l'écran, ici 1920*1080, les objets se trouvant dans cette fenêtre sont gérés pour se placer en fonction de la taille de l'écran.
Le manager s'appelle GridBagLayout, il nécessite de paramétriser chaque objet. Ce manager crée une grille qui se construit en fonction des paramètres de chaque objet qu'elle contient.

Prenons exemple du premier bouton FPV :

```
1 axPanel1.setMinimumSize(new Dimension(400,210));
2 axPanel1.setMaximumSize(new Dimension(400,210));
3 axPanel1.setPreferredSize(new Dimension(400,210));
4 c.fill = GridBagConstraints.BOTH;
5 c.anchor = GridBagConstraints.CENTER;
6 c.gridx = 0;
7 c.gridy = 0;
8 c.weighty = 0.0;
9 c.weightx = 0.0;
10 c.gridwidth = 1;
11 c.gridheight = 1;
12 c.insets = new Insets(0, 0, 0, 200);
13
```

Les 3 premières lignes correspondent à la taille du bouton, que nous avons choisis ici de garder fixe.

La ligne 4 n'est pas utile dans ce cas mais permet de correctement redimensionner l'objet lorsque la fenetre change de taille.

La ligne 5 fixe l'objet au centre de la partie qui lui a été alloué.

La ligne 6 et 7 donne la ligne et la colonne où doit se situé l'objet.

La ligne 8 et 9 définissent des poids en x et y qui sont utilisé lors d'un redimensionnement, cela permet de donner plus de poids à un objet plutot qu'à un autre. Nous ne l'utilisons pas d'où la valeur 0.

La ligne 10 et 11 permettent de definir combien de ligne et combien de colonne va prendre notre objet.

La ligne 12 insere une marge dans l'ordre suivant (margeSupérieure, margeGauche, margeInférieur, margeDroite).

Chaque objet de notre interface est defini de cette façon.

Ensuite il y a le bouton de Gauche qui lance le systeme complet. Un nouveau menu remplace le précédent :

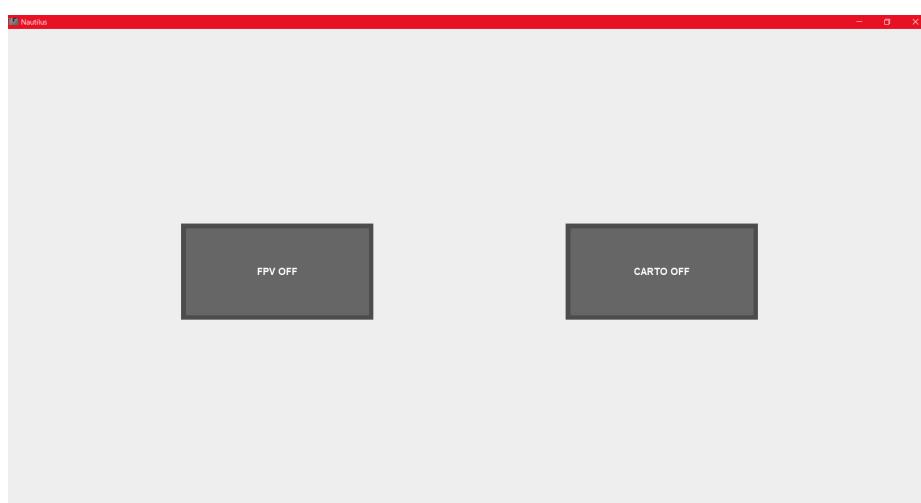


FIGURE 6.10 – Photo Deuxieme menu interface

Nous avons maintenant la FPV en haut à gauche, le bouton à droite est le même précédemment, les informations des capteurs sont affichées à droite, les commandes envoyées aux moteurs sont en bas et pour finir un affichage 3D (6.2.5) avec JAVA3D en bas à droite. Toutes les informations étant actualisées en temps réel avec la Raspberry. Nous allons voir comment.

6.2.2 Tunnel SSH

Nous avons choisi d'échanger les données avec la Raspberry par SSH, pour cela on utilise la bibliothèque Jcraft, plus précisément jsch. Au lancement de l'application, il y a 5 tunnels qui se créent (1 pour chaque caméras, 1 pour chaque capteurs et 1 pour les moteurs), ils sont créés au tout début pour ensuite permettre de transmettre directement les données sans refaire la procédure de connexion, le système gagne en rapidité.

Détaillons maintenant la procédure qui permet de créer ses tunnels. Comme nous sommes en JAVA, une seule classe permet de créer un tunnel et cette classe est ensuite instancié autant de fois que l'on veut.

Voici le code (Disponible dans /SSH/Exec.java) de création d'un tunnel :

```

1 try {
2     JSch jsch=new JSch();
3     this.session=jsch.getSession("pi", "169.254.14.03", 22);
4     UserInfo ui=new MyUserInfo();
5     this.session.setUserInfo(ui);
6     this.session.connect();
7     this.channel = this.session.openChannel("exec");
8     ((ChannelExec)this.channel).setCommand(this.Commande);
9     this.channel.setInputStream(null);
10    ((ChannelExec)this.channel).setErrStream(System.err);
11    this.in=this.channel.getInputStream();
12    this.channel.connect();
13    byte[] tmp=new byte[1024];
14    while(true){
15        while(this.in.available()>0){
16            int i=this.in.read(tmp, 0, 1024);
17            if(i<0)break;
18            System.out.print(new String(tmp, 0, i));
19            this.retour=new String(tmp, 0, i);
20        }
21        if(channel.isClosed()){
22            if(this.in.available()>0) continue;
23            System.out.println("exit-status: "+channel.getExitStatus());
24            break;
25        }
26    }
27 }
28 catch(Exception e){
29     System.out.println(e);
30 }
31

```

De la ligne 1 à 5, le tunnel est créé et les informations tel que le nom d'utilisateur, le mot de passe, l'adresse IP et le port sont ajouté aux couches correspondante du tunnel, puis ligne 6 la session est lancée.

A la ligne 7, nous ouvrons un canal d'exécution de commande et ligne 8 nous donnons à ce canal

la commande que nous voulons envoyée.

Les lignes 9,10 et 11 définissent où vont les données entrées, sorties et sorties erreur. Dans notre cas l'entrée ne nous intéresse pas car cela passe par une commande, la sortie normal sera stockée et la sortie erreur renvoyé sur l'afficheur d'erreur du système (la console). La ligne 12 lance la connection du canal, c'est ici que la commande est envoyée.

De la ligne 13 à 20 nous permet de récupérer les valeurs renvoyées par la Raspberry et de les stockées pour ensuite pouvoir les traiter.

Puis les dernières lignes, permettent de gerer le cas où le canal est coupé et nous renvoyer d'où vient l'erreur (par exceptions).

Ces tunnels nous permettent d'envoyer des commandes à la Raspberry et de pouvoir récupérer le retour de cette commande. C'est par ce moyen que nous récupérons presque tous. L'exception étant pour la video IP. Abordons ce sujet.

6.2.3 Videos

Nous avons donc 2 flux vidéos à récupérer depuis une adresse IP. Tout d'abord nous devons introduire quelque chose que nous utilisons partout dans notre code : le multi-Thread. Cela permet de lancer plusieurs actions en même temps, c'est ce qu'il se passe avec la récupération des flux vidéos. Chaque image des flux vidéos est récupéré puis traité puis affiché en temps réel sans interrompre le reste du programme, tout comme l'envoie de chaque commande.

Pour les flux vidéos, c'est une connection http qui est effectué, pour cela la méthode connect est appelée :

```
1 public void connect()
2 {
3     try
4     {
5         URL u = new URL(useMJPEGStream?mjpgURL:jpgURL);
6         huc = (HttpURLConnection) u.openConnection();
7         InputStream is = huc.getInputStream();
8         connected = true;
9         BufferedInputStream bis = new BufferedInputStream(is);
10        dis= new DataInputStream(bis);
11        if (!initCompleted) initDisplay();
12    }
13    catch(IOException e)
14    { //Relance la connection si pas de connection en attendant 60 sec
15        try
16        {
17            huc.disconnect();
18            Thread.sleep(60);
19        }
20        catch(InterruptedException ie)
21        {
22            System.out.println(ie);
23        }
24    }
25    catch(Exception e){;}
26}
27
```

De la ligne 5 à 10, la procédure de connection http est effectué.

Après la ligne 12, on gère les exceptions liées à la connection.

A la ligne 10, les données récupérées sont stockées dans une variable globale à Caméra.

Les informations vont être ensuite traité par initDisplay qui est appelé en ligne 11.

Observons cette méthode :

```
1 public void initDisplay()
2 {
3     if (useMJPEGStream) readMJPEGStream();
4     else
5     {
6         readJPG();
7         disconnect();
8     }
9     imageSize = new Dimension((image.getWidth(this)*2), image.getHeight(this)
10 *2);
11     setPreferredSize(imageSize);
12     parent.validate();
13     initCompleted = true;
14 }
```

Le premier If/Else permet de distinguer le cas où le flux serait vidéos ou juste photo. Dans notre cas c'est un flux vidéos composé d'image JPG.

Le reste de la méthode permet de préparer l'affichage des images.

Nous sommes toujours dans la procédure de connection, la lecture de la première image récupérée est fait. Pour cela la méthode readMJPEGStream qui appelle readJPG permet de décoder l'image et de la stocker dans la variable image. Voici les 2 méthodes :

```
1 public void readMJPEGStream()
2 {
3     readLine(4, dis); // enlève les 3 premières lignes
4     readJPG();
5     readLine(1, dis); // enlève les 2 dernières lignes
6 }
7
8 public void readJPG()
9 {
10     try {
11         JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(dis);
12         image = decoder.decodeAsBufferedImage();
13     } catch (Exception e) {
14         disconnect();
15     }
16 }
17
```

La procédure de décodage d'une image JPEG est faite par une bibliothèque externe.

Après cette procédure de connection, les 2 méthodes ci dessus sont appelé en boucle et l'affichage mis à jour :

```
1 public void readStream()
2 {
3     try {
4         if (useMJPEGStream) {
5             while(true) {
6                 readMJPEGStream();
7                 parent.repaint();
8             }
9         }
10    } catch (Exception e) {}
11 }
12
```

L'affichage étant geré par la bibliothèque Swing, c'est la méthode paint() qui affiche l'image et de plus trace une ligne qui en fonction des angles d'euler renvoyé par la centrale intertie :

```
1 public void paint(Graphics g)
2 {
3     if (image != null)
4         image=scale(image, 2);
5         g.drawImage(image, 0, 0, this);
6         Graphics2D g2 = (Graphics2D) g;
7         double alpha = Interface.rotZ* Math.PI/180f;
8         int a = image.getHeight();
9         int b = image.getWidth();
10        double S =(b/2)*(Math.sin(alpha))*(Math.cos((Math.PI/2)-alpha));
11        double T =(b/2)*(Math.sin(alpha))*(Math.sin((Math.PI/2)-alpha));
12        g2.draw(new Line2D.Double(S,(a/2)-T, b-S,(a/2)+T));
13    }
14
```

L'image est redimensionnée pour le bien de l'affichage et la ligne est tracé avec 2 points calculé par rapport à une rotation avec une origine au centre de l'image.

Nos 2 flux videos sont donc récupérés et peuvent donc être appellés par un JPanel n'importe où.

6.2.4 Capteurs et Moteurs

Capteurs

Nous avons déjà précédemment préparé la raspberry pour qu'elle renvoie les informations que l'on veut en fonction d'une commande. Il nous suffit maintenant d'envoyer en temps réel et en boucle infini les commandes (capteur de pression et centrale inertie) grâce au système de multi-thread.

Expliquons la méthode pour le capteur de pression. Tout d'abord il faut créer le thread lié à la classe qui sera exécuter en boucle, puis on lance le thread avec la méthode, ce qui donne :

```
1 Pression ex4 = new Pression();
2 Thread a4 = new Thread(ex4);
3 a4.start();
4
```

La classe Pression doit être héritière de Runnable pour permettre le lancement en thread.

Observons dans la classe Pression, la méthode start :

```
1 Interface.exPression.Commander("python MS5803_14BA.py");
2 String[] a = Interface.exPression.retour.split("/");
3 this.Pression=a[0];
4 this.TemperatureC=a[1];
5 this.TemperatureF=a[2];
6 Interface.pre=Pression;
7 Interface.tempC=TemperatureC;
8 Interface.tempF=TemperatureF;
9 Interface.pression.setText(" Pression = "+Interface.pre);
10 Interface.pression.repaint();
11 Interface.temperatureC.setText(" TemperatureC = "+Interface.tempC);
12 Interface.temperatureC.repaint();
13 Interface.temperatureF.setText(" TemperatureF = "+Interface.tempF);
14 Interface.temperatureF.repaint();
15
```

Nous commençons en ligne 1 par envoyer la commande grâce au tunnel précédemment créé pour le capteur de Pression puis nous récupérons les valeurs sous forme de chaînes de caractères. Enfin nous mettons à jour les valeurs globales et l'affichages.

Moteurs

Pour les moteurs, on envoie juste une commande à la Raspberry :

```
1 Interface.exMoteur.setCommande("echo 0="+m1+" > /dev/servoblaster & echo 1="+m2+
> /dev/servoblaster & echo 2="+m3+" > /dev/servoblaster");
2 Thread a1 = new Thread(Interface.exMoteur);
3 a1.start();
4 while( a1.isAlive()) {}
5 Interface.m1=this.m1;
6 Interface.m2=this.m2;
7 Interface.m3=this.m3;
8 Interface.moteur1.setText(""+Interface.m1+"      ");
9 Interface.moteur1.repaint();
```

```

10 Interface.moteur2.setText(" "+Interface.m2+" ");
11 Interface.moteur2.repaint();
12 Interface.moteur3.setText(" "+Interface.m3+" ");
13 Interface.moteur3.repaint();
14

```

On oublie pas de mettre à jour au passage l'affichage.

Ce code est donc exécuter à chaque fois que l'on appuie sur une des touches de directions.

Nous avons défini les touches suivantes :

Monter/Descente : Z/S

Gauche/Droite : Flèche Gauche/Droite

Monter/Descente : Z/S

Arrete d'urgence : Barre Espace

6.2.5 Affichage 3D

Nous utilisons JAVA3D pour afficher le modèle qui est extrait de Solidworks.

Tout d'abord nous devons créer un canvas 3D qui contiendra notre objet 3D et nous l'insérerons au milieu d'un JPanel :

```

1 Canvas3D canvas3D = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
2 this.add(canvas3D, BorderLayout.CENTER);
3

```

Puis nous créons un simple univers qui contient notre canvas3D.

Nous devons maintenant positionner notre point d'observation pour avoir une vue correcte :

```

1 OrbitBehavior orbit = new OrbitBehavior(canvas3D, OrbitBehavior.REVERSE_ROTATE);
2 orbit.setRotXFactor(0); //or any other value
3 orbit.setRotYFactor(0);
4 orbit.setTransXFactor(0);
5 orbit.setTransYFactor(0);
6 orbit.setSchedulingBounds(new BoundingSphere());
7 simpleU.getViewingPlatform().setViewPlatformBehavior(orbit);
8 ViewingPlatform vp = simpleU.getViewingPlatform();
9 TransformGroup steerTG = vp.getViewPlatformTransform();
10 Transform3D t3d = new Transform3D();
11 steerTG.getTransform(t3d);
12 t3d.lookAt(new Point3d(-1.2, 1.2, 1.2), new Point3d(0, 0, 0), new Vector3d(0, 1, 0));
13 t3d.invert();
14 steerTG.setTransform(t3d);
15

```

De la ligne 1 à 6, nous définissons un mouvement orbital de la caméra puis pour le moment nous fixons les rotations et les translations (on pourra les débloquer si on le veut)

Les lignes 9 à 14 permettent de définir la position initiale de notre point d'observation.

Nous devons ensuite créer ce qu'on appelle une scène, c'est dans cette scène que tous nos objets 3D se trouveront :

```

1 BranchGroup scene = createSceneGraph(simpleU);

```

```

2 scene.compile();
3 simpleU.addBranchGraph(scene);
4

```

Les 2 lignes suivant compile la scene et rattache notre scene à l'univers.

Maintenant nous devons crée cette scene 3d qui contiendra notre objet mais aussi la lumière, notre eau fictive et les mouvements possible.

Tous ce passe dans la méthode createSceneGraph.

```

1 public BranchGroup createSceneGraph(SimpleUniverse simpleU)
2 {
3     BranchGroup parent = new BranchGroup();
4     BoundingSphere bounds = new BoundingSphere(new Point3d(), 100);
5     Light ambientLight = new AmbientLight(new Color3f(Color.white));
6     ambientLight.setInfluencingBounds(bounds);
7     parent.addChild(ambientLight);
8
9     Light directionalLight = new DirectionalLight(
10         new Color3f(Color.white),
11         new Vector3f(1, -1, -1));
12     directionalLight.setInfluencingBounds(bounds);
13     parent.addChild(directionalLight);
14

```

La ligne 3 crée un objet parent qui contiendra tous nos objets.

De la ligne à 4 à 7, on crée une lumiere ambiante de couleur blanche

Puis de la ligne 9 à la ligne 13, une lumiere directionnel pour augmenter la luminosité dans la même direction que notre point d'observation.

Nous ne détaillerons pas ici le code de la création de l'eau, le principe étant qu'on a crée 4 plans positionnés correctement autour du ROV et on lui applique une textures bleu (que l'on peut changer facilement). Et on oublie pas de l'ajouter à l'objet parent.

Maintenant autorisons les mouvements de la souris pour notre objet :

```

1 TransformGroup mouseTransform = new TransformGroup();
2 mouseTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
3 mouseTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
4 mouseTransform.addChild(loadWavefrontObject());
5

```

Et en ligne 4 nous ajoutons notre ROV. La méthode loadWavefrontObject() permet de recuperer un objet 3D à partir d'un fichier .obj.

Pour la fin de cette méthode, les commentaires decrivent les opérations effectués

```

1 // Creation de l'homothetie (homothetie)
2 Transform3D homothetie = new Transform3D();
3 homothetie.setScale(Interface.scale);
4
5 // Creation de la transformation (translation)
6 Transform3D translation = new Transform3D();
7 translation.setTranslation(new Vector3f(0f, 0f, 0f));

```

```

8 translation.mul(homothetie);
9
10 // Creation de la rotation X(rotation)
11 Transform3D rotationX = new Transform3D();
12 rotationX.rotX(Interface.rotX * Math.PI/180f);
13 rotationX.mul(translation);
14
15 // Creation de la rotation Y(rotation)
16 Transform3D rotationY = new Transform3D();
17 rotationY.rotY(Interface.rotY * Math.PI/180f);
18 rotationY.mul(rotationX);
19
20 // Creation de la rotation Z(rotation)
21 Transform3D rotationZ = new Transform3D();
22 rotationZ.rotZ(Interface.rotZ * Math.PI/180f);
23 rotationZ.mul(rotationY);
24
25 this.rotationGroup = new TransformGroup(rotationZ);
26
27 // Autorisation de modifier les transformations
28 rotationGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
29 rotationGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
30
31 // Ajout au graphe de la scene
32 rotationGroup.addChild(mouseTransform);
33 parent.addChild(rotationGroup);
34
35 return parent;
36

```

Notre scene renvoie donc tous les objets qu'on a crée précédemment.

Maintenant nous voulons que notre ROV 3D soit capable de se mouvoir en fonction des angles d'euler renvoyer et stocker dans notre interface.

Pour cela on a crée une méthode mouvement :

```

1 public void mouvement(double rotX, double rotY, double rotZ, Vector3d trans,
2                      double scale)

```

Elle prend donc en argument, les 3 rotations, un vecteur des translations (sur tous les axes) et un paramètre multiplicateur d'echelle.

Cette méthode modifie la matrice 4*4 contenant la matrice de rotation, translation et de mise à l'echelle.

Puis effectue les transformations de cette matrice.

6.2.6 Cartographie

«««< Updated upstream Cette partie n'a pas été beaucoup développé mais un premier système a été créé mais pas intégré dans la dernière version de l'interface.

Grâce à l'appui sur une touche, une photo de la camera du dessous est enregistrer dans un dossier. Voici le code qui permet d'enregistrer notre photo en jpg :

```

1  BufferedImage img = new BufferedImage(axPanel.getWidth() , axPanel.getHeight() ,
2  	BufferedImage.TYPE_INT_RGB) ;
3  axPanel.print(img.getGraphics()) ;
4  try {
5      ImageIO.write(img, "jpg", new File(nomPhoto+".jpg")) ;
6  }
7  catch (IOException e5) {
8      e5.printStackTrace() ;
9 }
```

Le systeme n'est pas fini mais dans l'absolu chaque photo sera enregistrer avec un nom different avec les donnees de localisation dans les meta-donnees de la photo.

Ensuite il y a un autre bouton (non affiche sur la derniere interface) qui ouvre une autre fenetre avec toute nos photos positionnees, en fonction des coordonnees GPS, sur une carte.

On peut cliquer sur chaque photo pour les agrandir dans une nouvelle fenetre.

Tous les codes se trouvent dans le sous-dossier /Carto.

Chapitre 7

Structure

L'objectif suivant, et non des moindres dans la construction d'un véhicule sous-marin, était l'établissement de la structure. Comme on peut le voir dans le cahier des charges, la structure a été pensée dans 6 optiques différentes : étanchéité, transportabilité, durabilité, aqua-dynamisme, esthétisme et personnalisation.

7.1 Design

Tout d'abord, il fallait donner un premier aspect à notre R.O.V. pour en suite le modifier après différents tests pour mieux correspondre aux 6 optiques explicitées au début. Pour obtenir des idées sur différentes architectures et aussi nous faire gagner du temps sur la conception du design, on a étudié plusieurs R.O.V. du marché. Voici les 3 R.O.V qui ont eu la plus grosse influence sur notre design :



(a) Aquarov de Rov developpe-
ment



(b) SeaBotix d'Advanced Ma-
rine



(c) Trident d'Open ROV

Avec le design de ces drones en tête, on a alors réalisé le notre qui se veut plus épuré dans la même lignée que le Trident. En effet cet aspect se destine mieux à un marché dit 'grand public'. Très vite, après les premiers croquis, nous sommes passé à une conception sous SolidWorks du R.O.V.. Voici donc son aspect final sous Solidworks ainsi qu'un rendu 3D :

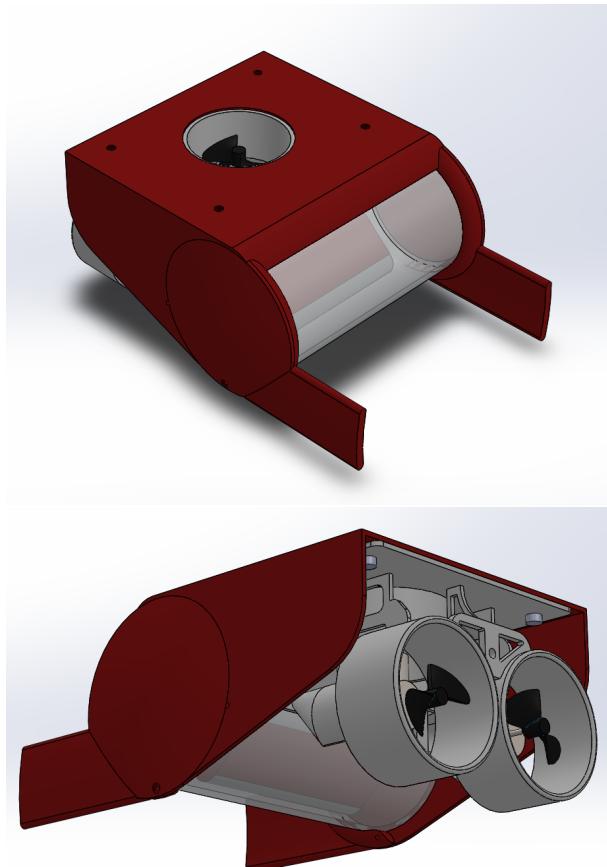


FIGURE 7.2 – modélisation du R.O.V. Nautilus sous Solidworks

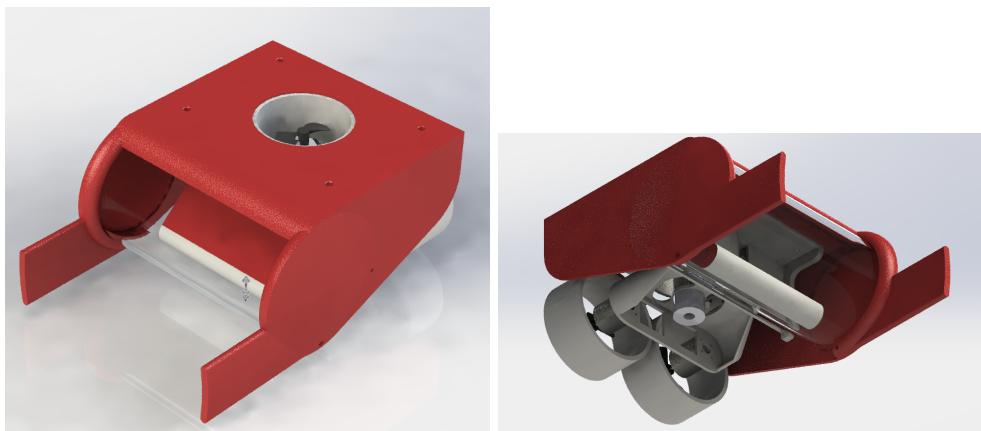


FIGURE 7.3 – rendu 3D du R.O.V. Nautilus sous Solidworks

Comme on peut l'observer le R.O.V. va se composer de 3 grandes parties : un squelette interne, un tube transparent ainsi qu'une coque. Cela va faciliter l'étanchéification du R.O.V mais aussi la modification ou le remplacement d'une partie sans toucher aux autres. On colle alors parfaitement aux sections esthétisme et personnalisation. Niveau dimensionnement, le R.O.V. a un emplacement total de 300*200*150 mm.

7.1.1 Squelette interne

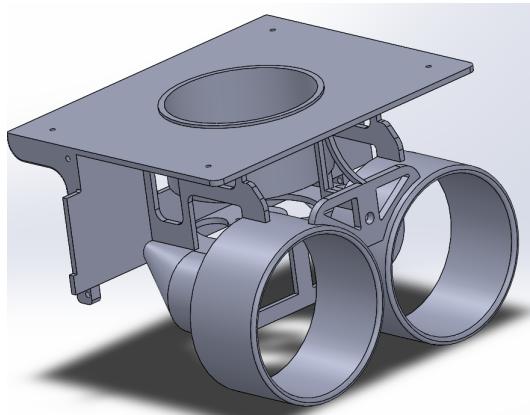


FIGURE 7.4 – squelette interne

Le squelette interne du R.O.V. joue, comme son nom l'indique, le rôle d'une structure de soutien et de rigidification du Nautilus. C'est donc la partie du R.O.V. la plus résistante à la déformation. L'ensemble des autres parties, comme la coque, les câbles, ainsi que les trois moteurs brushless viennent se fixer dessus bénéficiant ainsi de cette solidité. De plus comme l'eau va circuler énormément dans cette zone, le squelette interne a été conçu pour opposer très peu de résistance au passage de l'eau avec de nombreux trous et même concentrer et diriger les flux aux niveaux des moteurs.

7.1.2 Tube

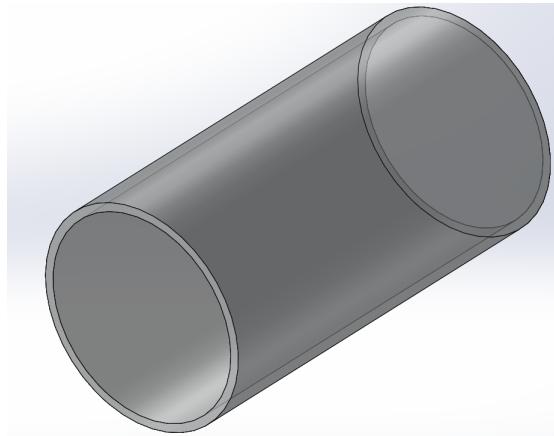


FIGURE 7.5 – Tube

Le tube est la partie étanche du Nautilus. En effet, ce tube regroupera l'ensemble de l'électronique. Celui-ci est entièrement transparent pour d'une part permettre de filmer ainsi que de photographier à l'aide de nos deux caméras et d'autre part de permettre à l'utilisateur un contrôle visuel de l'électronique et de l'étanchéité.

7.1.3 Coque

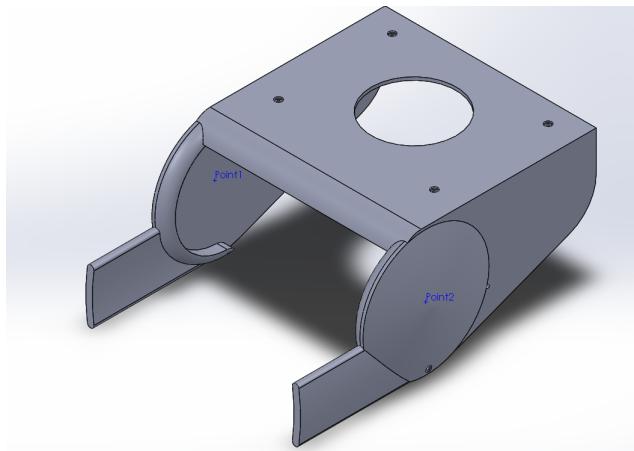


FIGURE 7.6 – Coque

La coque, quand à elle, va apporter le profil nécessaire à un bon aqua-dynamisme ainsi qu'une protection à l'environnement. Elle est aussi au coeur de la problématique de personnalisation car en plus de pouvoir avoir une forme ou une couleur différente, elle peut aussi accueillir des modules supplémentaires par exemple sur les bras à l'avant de sa coque.

7.2 Matériaux, méthode de construction

Une fois le design établit, nous sommes passé à la réalisation. Pour un premier prototypage rapide, nous avons choisi l'impression 3D de type 'fused filament fabrication'. Ainsi la coque a été imprimée en PLA et le squelette en ABS, tout deux avec un support en PVA qui est un plastique qui se dissous dans l'eau.

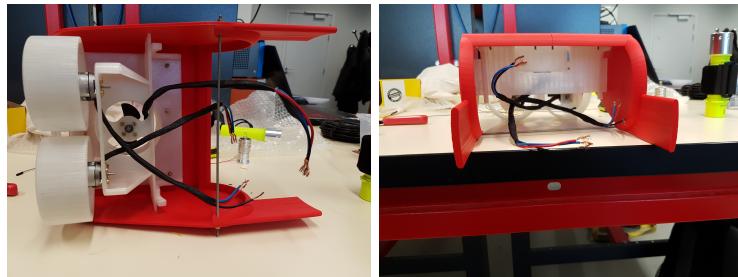


FIGURE 7.7 – impressions 3D du Nautilus

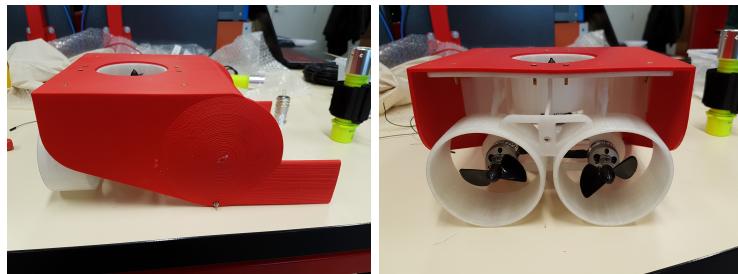


FIGURE 7.8 – impressions 3D du Nautilus

Cependant pour de prochaines impressions de la coque, il sera nécessaire de revoir le positionnement de celle-ci dans la zone d'impression. En effet, les efforts qui s'exercent sur la coque coïncident avec l'agencement des couches de plastique. Ceci explique l'apparition de fissure sur la coque.

Le tube est réalisé à l'aide d'un tube en acrylique dont une extrémité est un bouchon en acrylique scellé à la résine Epoxy et l'autre est aussi un bouchon avec un joint torrique qui permet d'ouvrir le tube.



FIGURE 7.9 – réalisation du tube

Pour que l'ensemble des câbles rentrent dans le tube, un petit accès a été réalisé puis scellé à la résine Epoxy.

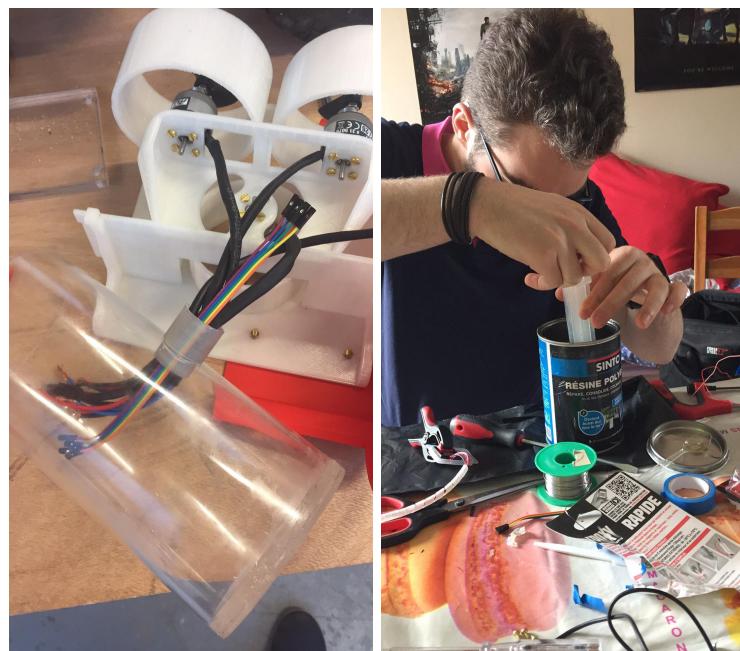


FIGURE 7.10 – accès des câbles et coulage de la résine

7.3 Simulations poids et aqua-dynamisme

Une fois la structure réalisée avec Solidworks, on a attribué les matériaux ou des densités quand le matériau n'était pas disponible, aux différents composants.

Pour un déplacement correct dans l'eau sans modification de l'assiette il faut que le centre d'inertie (en prenant en compte la poussée d'archimède) soit confondu avec les axes des moteurs. On a donc rajouté des poids en plomb sur le drone en modifiant leurs poids jusqu'à obtenir la positions du centre d'inertie voulue :

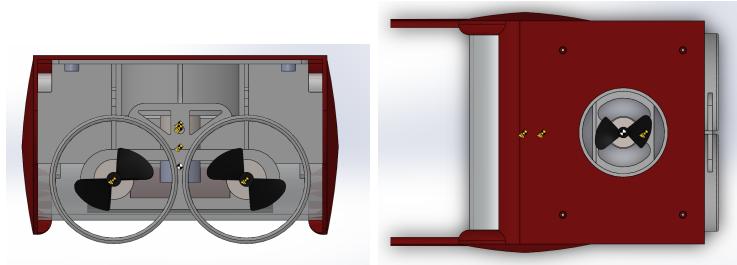


FIGURE 7.11 – répartition du poids

Avec une assiette correcte, on peut s'intéresser à l'aqua-dynamisme du ROV. En effet à l'aide du module flow-simulation de Solidworks, on place le Nautilus dans un cube de fluide s'écoulant à 2m/s (la vitesse maximale du ROV) et on regarde le comportement de certaines lignes de courants :

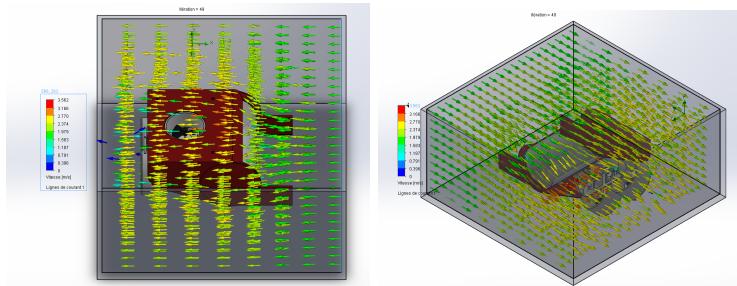


FIGURE 7.12 – test d'aqua-dynamisme

Comme on peut le voir, le fluide qui rencontre l'avant du ROV suit la paroi en accélérant à une vitesse semblable tout autour du Nautilus. Cet écoulement est bien caractéristique d'une bonne pénétration du ROV dans le fluide.

7.4 Test en eau

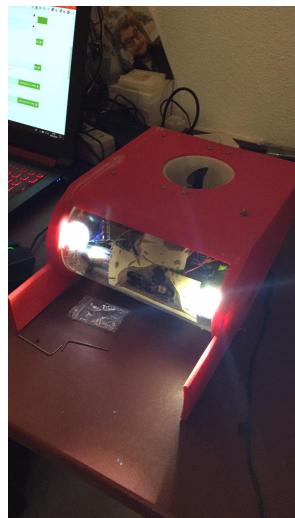


FIGURE 7.13 – ROV entièrement monté

Une fois la structure entièrement réalisée et testée en simulation, nous avons réalisé les tests en eau. Cependant durant ces tests, nous nous sommes aperçus que le ROV n'était pas parfaitement étanche. En effet au bout d'une dizaine de minutes nous avons pu observer de l'humidité dans le tube. Bien que le reste de la structure se comporte parfaitement bien il faut revoir l'étanchéité du ROV.

Chapitre 8

Conclusion

Bibliographie

- [1] Conrad. Moteur d'avion électrique brushless roxxy 315079 chez conrad (x3). Conrad, 2018.
- [2] RobotShop. Esc suppo multirotor 20a m20a chez robotshop (x3). RobotShop, 2018.
- [3] HobbyKing. Esc hobbyking 30a avec reverse (x1). Hobbyking, 2018.
- [4] Github. Programme servoblaster. ServoBlaster, 2018.
- [5] TopModel. Radiocommande graupner mx-20 (disponible à l'ensea). TopModel, 2018.
- [6] Conrad. Batterie d'accumulateurs (nimh) 7.2 v 3000 mah conrad (x1). Conrad, 2018.
- [7] Nautilus. Adresse video caméra frontale. 169.254.14.03:8081, 2018.
- [8] Mouse Electronics. Module camera v2 raspberry. Mouse Electronics, 2018.
- [9] Nautilus. Adresse video caméra du dessous. 169.254.14.03:8000, 2018.
- [10] GitHub. Code java de l'interface. Github, 2018.

Chapitre 9

Références

»»»> Stashed changes

Table des figures

4.1	Trello - Code Couleur	7
4.2	Trello - Planning A faire	7
4.3	Trello - Planning En cours et Terminé	8
4.4	Github - Nautilus	8
4.5	Commande Conrad	9
4.6	Commande Robotshop	9
4.7	Commande Exptch	9
5.1	Moteur Brushless	10
5.2	ESC	11
5.3	Signal PWM	12
5.4	Programme Servoblaster	12
5.5	Valeurs GPIO	13
5.6	Commandes moteurs	13
5.7	Tableau des temps	14
5.8	Architecture de la motorisation du ROV	15
6.1	Raspberry	16
6.2	Logitech C170	17
6.3	Photo Caméra Frontale	17
6.4	Module Camera V2 Raspberry	18
6.5	Photo Caméra du dessous	18
6.6	Capteur de pression et de température MS5803-14BA	19
6.7	Capteur MS5803-14BA avant le coulage dans la résine	20
6.8	Matrice inertuelle MinIMU-9 v2	20
6.9	Premier Menu Interface	23
6.10	Photo Deuxieme menu interface	24
7.2	modélisation du R.O.V. Nautilus sous Solidworks	35
7.3	rendu 3D du R.O.V. Nautilus sous Solidworks	35
7.4	squelette interne	36
7.5	Tube	37
7.6	Coque	37
7.7	impressions 3D du Nautilus	38
7.8	impressions 3D du Nautilus	38
7.9	réalisation du tube	39
7.10	accès des câbles et coulage de la résine	39

7.11 répartition du poids	40
7.12 test d'aqua-dynamisme	40
7.13 ROV entièrement monté	41